

*CHAPTER FOUR*

Register Transfer  
and Microoperations

|  |  |
| --- | --- |
|  | IN THIS CHAPTER |
| 4-1 | Register Transfer Language |
| 4-2 | Register Transfer |
| 4-3 | Bus and Memory Transfers |
| 4-4 | Arithmetic Microoperations |
| 4-5 | Logic Microoperations |
| 4-6 | Shift Microoperations |
| 4-7 | Arithmetic Logic Shift Unit |

**4-1 Register Transfer Language**

A digital system is an interconnection of digital hardware modules that accom­plish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various mod­ules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoper­ation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of the digital com­ponents introduced in Chap. 2 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the micro-

*microoperation*

operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Sim­ilarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

*register transfer language*

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subse­quent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4-2 Register Transfer

— ■— ■

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name **MAR.** Other designations for registers are PC (for program counter), **IR** (for instruction register, and **Rl** (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through **n -** 1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol **H** (for high byte). The name of the 16-bit register is PC. The symbol PC(0-7) or PC(L) refers to the low-order byte and PC(8-15) or **PC(H)** to the high-order byte. .

*registers*

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

*register transfer*

**R2** PI

denotes a transfer of the content of register PI into register P2. It designates a replacement of the content of P2 by the content of PI. By definition, the content of the source register PI does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Nor-

Figure 4-1 Block diagram of register.

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

(a) Register **R** (b) Showing individual bits

15 0 15 8 7 0

|  |  |  |  |
| --- | --- | --- | --- |
| **R2** |  | **PC (H)** | **PC(L)** |

(c) Numbering of bits

(d) Divided into two parts

mally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an **if-then** statement.

**If(P** = 1) **then** (P2 <- **Rl)**

where **P** is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a **control function.** A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

*control function*

P: R2 **«-** Rl

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

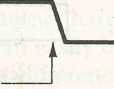
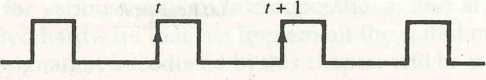
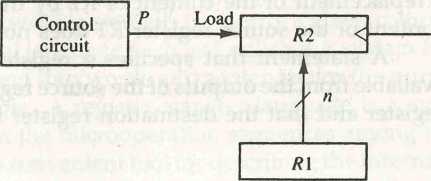
Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block dia­gram that depicts the transfer from PI to **R2.** The **n** outputs of register PI are connected to the **n** inputs of register P2. The letter **n** will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register P2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As show n

Figure 4-2 Transfer from Rl to R2 when P = 1.

**Clock**

**(a) Block diagram**

**Clock**

**Load**

/

**Transfer occurs here**

**(b) Timing diagram**

in the timing diagram, **P** is activated in the control section by the rising edge of a clock pulse at time **t**. The next positive transition of the clock at time **t** 4- 1 finds the load input active and the data inputs of **R2** are then loaded into the register in parallel. P may go back to 0 at time **t -1-1;** otherwise, the transfer will occur with every clock pulse transition while **P** remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as **P** becomes active just after time **t**, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time **t** + 1.

The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

**T:** R2 \*- **Rl,** R1 R2

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

TABLE 4-1 Basic Symbols for Register Transfers

|  |  |  |
| --- | --- | --- |
| Symbol | Description | Examples |
| Letters | Denotes a register | MAR, R2 |
| (and numerals) |  |  |
| Parentheses ( ) | Denotes a part of a register | R2(0-7), R2(L) |
| Arrow <— | Denotes transfer of information | R2 R1 |
| Comma , | Separates two microoperations | R2 Rl, R1 R2 |
| **4-3 Bus and Memory Transfers** | | |

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

***common bus***

determine which register is selected by the bus during each particular register transfer.

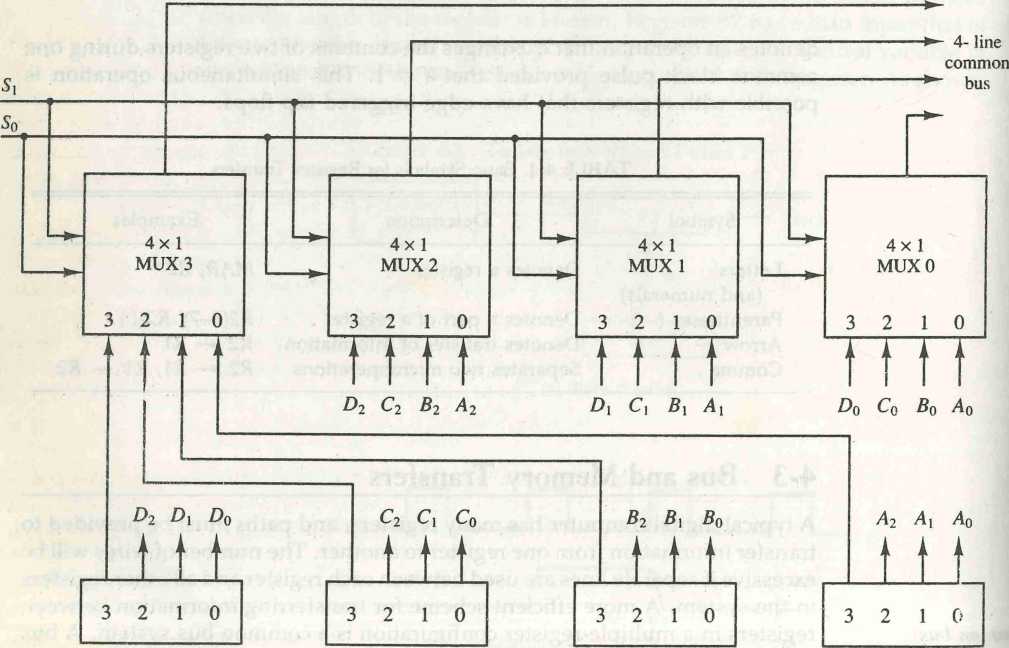
One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 4-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4x1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, Si and S0. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register **A** is connected to input 0 of MUX 1 because this input is labeled **A\.** The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

Figure 4-3 Bus system for four registers.

**Register D**

**Register C**

**Register B**

**Register A**

SECTION 4-3 Bus and Memory Transfers 99

The two selection lines Si and S0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When SiS0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if SiS0 = 01, and so on. Table 4-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

*bus selection*

TABLE 4-2 Function Table for Bus of Fig. 4-3

|  |  |  |
| --- | --- | --- |
| s, | So | Register selected |
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

In general, a bus system will multiplex k registers of n bits each to produce an w-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multi­plexer must be k x 1 since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destina­tion register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

BUS +- C, Rl <— BUS

The content of register C is placed on the bus, and the content of the bus is loaded into register Rl by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

Rl «- C

I

From this statement the designer knows which control signals must be acti­vated to produce the transfer through the bus.

state buffers, we need **n** circuits with four buffers in each as shown in Fig. 4-5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of **n** lines. Only one decoder is necessary to select between the four registers.

Memory Transfer

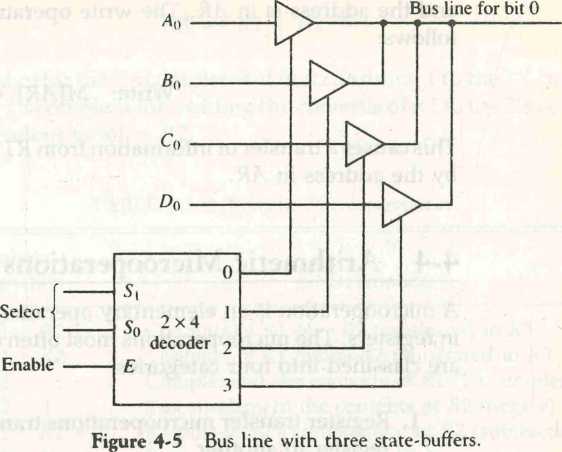
The operation of a memory unit was described in Sec. 2-7. The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter **M.**

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

***memory read***

**Read:** DR \*- M[AR]

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

The write operation transfers the content of a data register to a memory word **M** selected by the address. Assume that the input data are in register **R1**

***memory write***

and the address is in AR. The write operation can be stated symbolically as follows:

*add microoperation*

*subtract*

*microoperation*

Write: M[AR] Rl

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

4-**4 Arithmetic Microoperations**

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

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

The register transfer microoperation was introduced in Sec. 4-2. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperations. In the next two sections we present the logic and shift microoperations.

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

R3 jRI + R2

specifies an add microoperation. It states that the contents of register Rl are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arith­metic microoperations are listed in Table 4-3, Subtraction is most often imple­mented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

R3 Rl + R2 + 1

R2 is the symbol for the l's complement of R2. Adding 1 to the l's complement produces the 2's complement. Adding the contents of Rl to the 2's complement of R2 is equivalent to Rl — R2.

TABLE 4-3 Arithmetic Microoperations

Symbolic

designation Description

Contents of Rl plus R2 transferred to R3 Contents of Rl minus R2 transferred to R3 Complement the contents of R2 {l's complement) 2's complement the contents of R2 (negate)

Rl plus the 2's complement of R2 (subtraction) Increment the contents of Rl by one Decrement the contents of Rl by one

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

The arithmetic operations of multiply and divide are not listed in Table 4-

1. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are imple­mented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the oper­ation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most com­puters, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits con-

>inary adder

*R3 <- Rl + R2 R3 Rl\_~ R2 R2* «- R2 R2 «- R2 + 1\_

R3 Rl + R2 + 1 Rl <r- Rl + 1 Rl «- Rl - 1

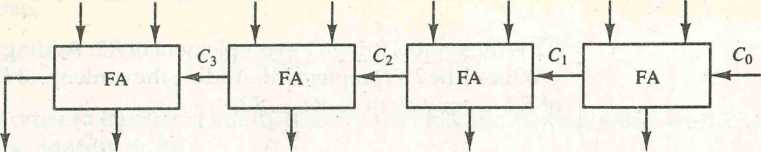
nected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of **A** and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C0 and the output carry is C4. The S outputs of the full-adders generate the required sum bits.

*full-adder*

An n-bit binary adder requires **n** full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The **n** data bits for the **A** inputs come from one register (such as Rl), and the **n** data bits for the **B** inputs come from another register (such as **R2).** The sum can be transferred to a third register or to one of the source registers (Rl or R2), replacing its previous content.

Binary Adcler-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction **A - B** can be done by taking the 2's complement of **B** and adding it to A. The 2's complement can be obtained by taking the l's complement and adding one to the least significant pair of bits. The l's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one com­mon circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7. The mode input M controls the operation. When **M** = 0 the circuit is an adder and when **M =** 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When **M =** 0, we have **B** ® 0 **- B.** The full-adders receive the value of B, the input carry is 0, and the circuit performs **A** plus B. When M = 1, we have B ® 1 = B' and C0 = 1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation **A** plus the

*adder-subtractor*

**By** A3 **B2** A2 **B**1 **A\ Bq Aq**

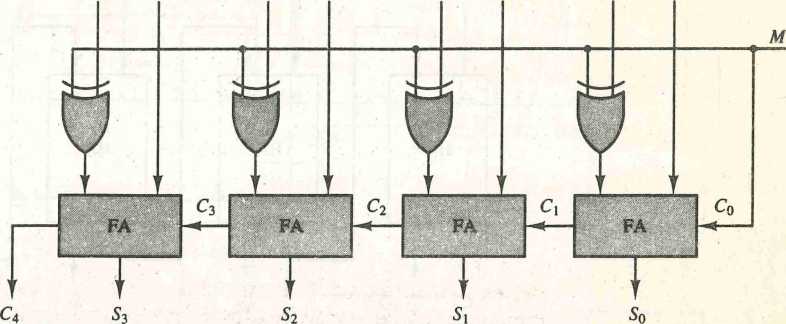
C4 S3 S2 S, S0

Figure 4-6 4-bit binary adder.

2's complement of B. For un°:y«ea numbers, this gives **A — Bi( A** > B or the 2's complement of (B - **A)** if **A** < B. For signed numbers, the result is **A - B** provided that there is no overflow.

Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter (see Fig. 2-10). Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders (see Fig. 1-16) connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 4-8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from **A0** through **A3/** adds one to it, and generates the incremented output in S0 through S3. The output carry C4 will be 1 only after incrementing binary **1111.** This also causes outputs S0 through S3 to go to 0.

*iticrementer*

***Bt, A3*** #2 A*2* ***B\ A\ Bq Aq***

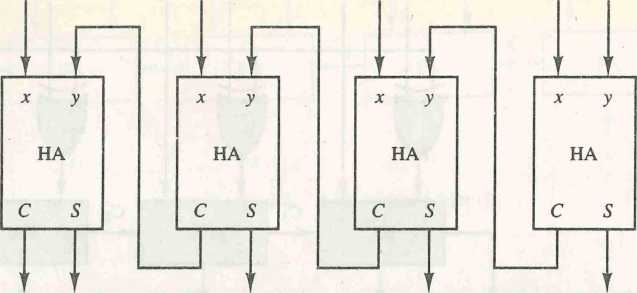
Figure 4-7 4-bit adder-subtractor.

The circuit of Fig. 4-8 can be extended to an n-bit binary incrementer by extending the diagram to include **n** half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

Arithmetic Circuit

The arithmetic microoperations listed in Table 4-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

***arithmetic circuit***

The diagram of a 4-bit arithmetic circuit is shown in Fig. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs **A** and **B** and a 4-bit output D. The four inputs from **A** go directly to the X inputs of the binary adder. Each of the four inputs from **B** are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, Si and S0. The input carry Qn goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

***input carry***

***At,*** A2 ***A\ Aq*** **1**

C 4 S3 S2 S1 5o

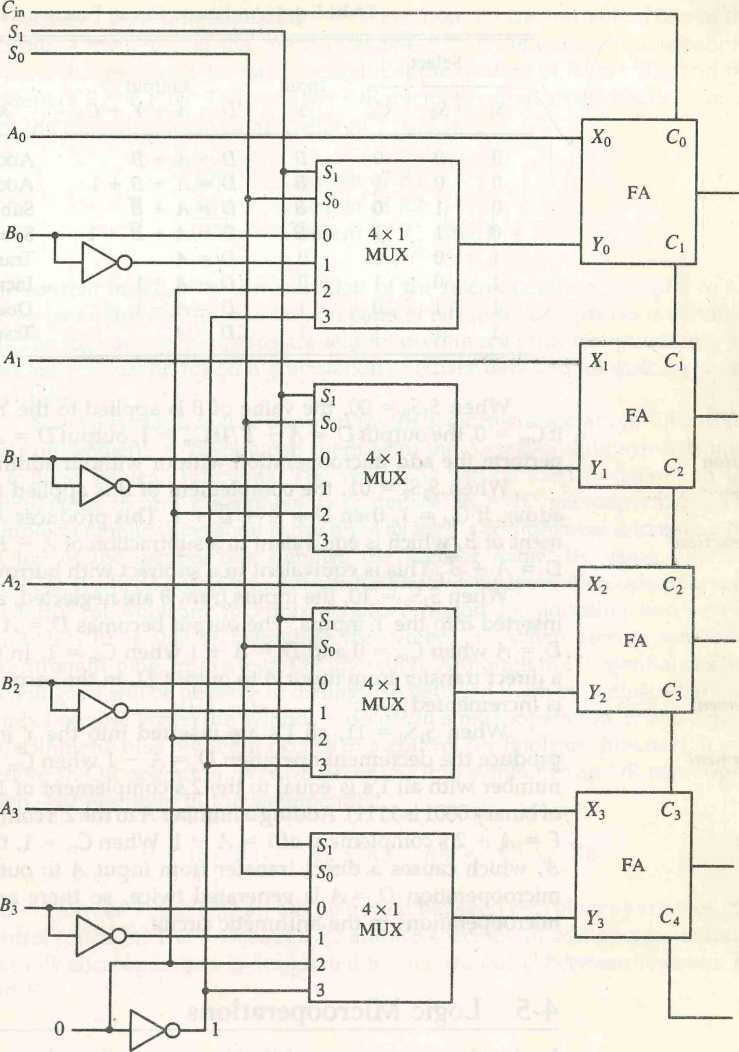
Figure 4-8 4-bit binary7 incrementer.

The output of the binary adder is calculated from the following arithmetic

sum:

D = **A +** Y + Cjn

where **A** is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs Si and S0 and making equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4-4.



Do

D2

out

Figure 4-9 4-bit arithmetic circuit.

# **D**,

***d3***

C

When SiS0 = 00, the value of B is applied to the **Y** inputs of the adder. If On = 0, the output D = A + B. If = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

addition

subtraction

increment

decrement

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Select |  | Input  Y | Output  D = A + Y + Cin Microoperation | |
| Si | So | Cin |
| 0 | 0 | 0 | B | D =A + B | Add |
| 0 | 0 | 1 | B | D =A + B + 1 | Add with carry |
| 0 | 1 | 0 | B | D = A + B | Subtract with borrow |
| 0 | 1 | 1 | B | D = A + B +1 | Subtract |
| 1 | 0 | 0 | 0 | D = A | Transfer A |
| 1 | 0 | 1 | 0 | D = A + 1 | Increment A |
| 1 | 1 | 0 | 1 | D = A - 1 | Decrement A |
| 1 | 1 | 1 | 1 | D = A | Transfer A |

TABLE 4-4 Arithmetic Circuit Function Table

When SiS0 = 01, the complement of **B** is applied to the **Y** inputs of the adder. If Cm = 1, then D = **A** + B +1. This produces **A** plus the 2's comple­ment of B, which is equivalent to a subtraction of A - B. When Cin = 0, then D = A + B . This is equivalent to a subtract with borrow, that is, A - B — 1.

When SiS0 = 10, the inputs from B are neglected, and instead, all 0's are inserted into the **Y** inputs. The output becomes D = A + 0 + C^. This gives D = A when C^, = 0 and D = A + 1 when Cin = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

When SiS0 = 11, all Ts are inserted into the **Y** inputs of the adder to produce the decrement operation D = A - 1 when = 0. This is because a number with all l's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces **F = A +** 2's complement of 1 = A - 1. When = 1, then D = A — 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation **D - A** is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

4-5 **Logic Microoperations**

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and **R2** is symbolized by the statement

P: Rl 4- Rl © R2

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable **P =** 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbol­izes the following logic computation:

1010 Content of Rl

**J**

**1100** Content of R2

0110 Content of Rl after **P** = 1

The content of Rl, after the execution of the microoperation, is equal to the

bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of Rl. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol V will be used to denote an OR microoperation and the symbol A to denote an AND microoperation. The complement microoperation is the same as the l's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol + , when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoper­ation. For example, in the statement

***special symbols***

**P + Q:** Rl <— R2 + R3, R4 ^ R5 V K6

the + between **P** and **Q** is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol V between registers R5 and R6.

List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5. In this table, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **X** | y | **F0** | **F!** | f2 | **F3** | **Fa** | **Fs** | f6 | f7 | **Fa** | f9 | **Fro** | **Fn** | **Fn** | **F13** | **FrA** | **Frs** |
| 0 | **0** | **0** | **0** | **0** | 0 | 0 | 0 | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| 0 | **i** | 0 | 0 | 0 | 0 | **1** | **1** | **1** | **1** | **0** | 0 | 0 | 0 | **1** | **1** | **1** | **1** |
| **1** | 0 | 0 | 0 | **1** | **1** | 0 | 0 | **1** | **1** | **0** | 0 | **1** | **1** | 0 | 0 | **1** | **1** |
| **1** | **i** | 0 | **1** | 0 | **1** | 0 | **1** | 0 | **1** | **0** | **1** | 0 | **1** | 0 | **1** | 0 | **1** |

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

two variables **x** and y. Note that the functions are determined from the 16 binary combinations that can be assigned to **F.**

The 16 Boolean functions of two variables **x** and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are derived from these functions by replacing variable **x** by the binary content of register **A** and variable y by the binary content of register **B.** It is important to realize that the Boolean functions listed in the first column of Table 4-6 repre­sent a relationship between two binary variables **x** and y. The logic micro­operations listed in the second column represent a relationship between the binary content of two registers **A** and **B**. Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

TABLE 4-6 Sixteen Logic Microoperations

Boolean function Microoperation Name

F <—0

F<—A AS F\*-A A B *F \*-A\_* F\*—A AS F<~B F<~A ©2? F\*-*A* y *B* F \*r-A V B

*f\*-aWb*

F<r~B F<r-Ay B *F<—A\_* F<r-A V B F<^A~/\B *F\*-* all l’s

*F0* = 0 *Fi* = *xy F2* = *xy' F3=x Fa, = x'y Fs = y F6 =* x©y *F7 = x + y Fs = (x + y)' F9* = (\*©y)' Fio = *y' Fn=x+ y' Fn* = x'

F**13** = x' + y Fu **=** (xy)'

**F15 = 1**

Clear

AND

Transfer A

Transfer B Exclusive-OR OR NOR

Exclusive-NOR Complement B

Complement A

NAND Set to all l’s

Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four—AND, OR, XOR (exclusive-OR), and complement— from which all others can be derived.

Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs Si and S0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0,1, 2,..., n — 1. The selection variables are applied to all stages. The function table in Fig. 4-10(b) lists the logic microoperations obtained for each combination of the selection variables.

Some Applications

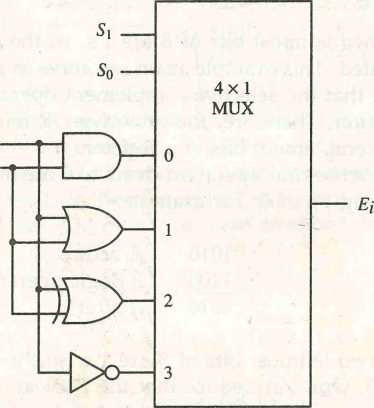
Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated

Figure 4-10 One stage of logic circuit.

A;

***Bi***

**(a) Logic diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| **5,** | So | **Output** | **Operation** |
| **0** | **0** | **E - A^B** | **AND** |
| **0** | **1** | **E = Aw B** | **OR** |
| **1** | **0** | **E = A®B** | **XOR** |
| **1** | **1** | **E = A** | **Complement** |

(b) Function table

by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The selective-set operation sets to 1 the bits in register A where there are corresponding Ts in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

selective-set

1010 A before

1100 B (logic operand)

1110 A after

The two leftmost bits of B are l's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previ­ous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

The selective-complement operation complements bits in A where there are corresponding Ts in B. It does not affect bit positions that have 0's in B. For example:

selective-complement

1010 A before

1100 B (logic operand)

0110 A after

Again the two leftmost bits of B are l's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The selective-clear operation clears to 0 the bits in A only where there are corresponding l's in B. For example:

selective-clear

1010 A before

1100 B (logic operand)

0010 A after

Again the two leftmost bits of B are l's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'. The corresponding logic microoperation is

A <— A *A* B

The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

1010 A before

1100 B (logic operand)

1000 A after masking

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are l's. The mask operation is more convenient to use than the selective- clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010 A before

0000 1111 B (mask)

0000 1010 A after masking

and then insert the new value:

0000 1010 A before

1001 0000 B (insert)

1001 1010 A after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The dear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010 A 1010 B 0000 A<—A ©B

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

4-6 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

A logical shift is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right microoperations. For example:

*logical shift*

Rlshl R1 £2^ shr R2

are two microoperations that specify a 1-bit shift to the left of the content of register Rl and a 1-bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accom­plished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respec­tively. The symbolic notation for the shift microoperations is shown in Ta­ble 4-7.

*circular shift*

TABLE 4-7 Shift Microoperations

Symbolic designation Description

|  |  |  |
| --- | --- | --- |
| R | ^shl R | Shift-left register R |
| R | <-shr R | Shift-right register R |
| R | <—cil R | Circular shift-left register R |
| R | <—cir R | Circular shift-right register R |
| R | <H-ashl R | Arithmetic shift-left R |
| R | <—ashr R | Arithmetic shift-right R |

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

*arithmetic shift*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  |  |
|  | Rn - 1 | Rn -2 |  | R\ |
| —►- |  |

Sign

bit

Figure 4'11 Arithmetic shift right.

when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 4-11 shows a typical register of n bits. Bit Rn \_ x in the leftmost position holds the sign bit. Rn-2 is the most significant bit of the number and R0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus Rn \_ x remains the same, Rn \_ 2 receives the bit from R„ \_ lf and so on for the other bits in the register. The bit in R0 is lost.

The arithmetic shift-left inserts a 0 into ,R0/ and shifts all other bits to the left. The initial bit of Rn \_ x is lost and replaced by the bit from R„ \_ 2. A sign reversal occurs if the bit in Rv \_ x changes in value after the shift. This happens if the multiplication by 2 causes an overflow7. An overflow occurs after an arithmetic shift left if initially, before the shift, Rn \_ x is not equal to Rn \_ 2. An overflow flip-flop Vs can be used to detect an arithmetic shift-left overflow.

Vs = Rn -1 ®Rn - 2

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

Hardware Implementation

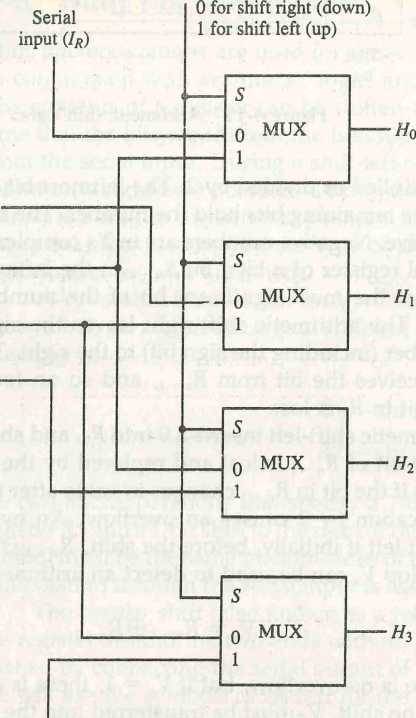
A possible choice for a shift unit would be a bidirectional shift register with parallel load (see Fig. 2-9). Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A0 through A3/ and four data outputs, H0 through H3. There are two serial inputs, one for shift left

*shifter*

(**k)** and the other for shift right (**I**L). When the selection input **S =** 0, the input data are shifted right (down in the diagram). When **S =** 1, the input data are shifted left (up in the diagram). The function table in Fig. 4-12 shows which input goes to each output after the shift. A shifter with **n** data inputs and outputs requires **n** multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

**Select**

****

**Serial input (4)**

Figure 4-12 4-bit combinational circuit shifter.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Select** |  | **Output** | |  |
| **S** | **H0** | **Hi** | **h2** | **h3** |
| **0** | **1r** | **^0** |  |  |
| **1** |  |  | **A?,** | **II** |

**Function table**

**4\*7** Arithmetic Logic Shift Unit

*ALU*

****

computer systems employ a number of storage registers connected to a com­mon operational unit called an arithmetic logic unit, abbreviated ALU. To

CHAPTER FIVE

Basic Computer  
Organization  
and Design

IN THIS CHAPTER

5-1 Instruction Codes

5-2 Computer Registers

5-3 Computer Instructions

5-4 Timing and Control

5-5 Instruction Cycle

5-6 Memory-Reference Instructions

5-7 Input-Output and Interrupt

5-8 Complete Computer Description

5-9 Design of Basic Computer

5-10 Design of Accumulator Logic

**5-1 Ins**truction Codes

In this chapter we introduce a basic computer and show how its operation can be specified with register transfer statements. The organization of the com­puter is defined by its internal registers, the timing and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. Although the basic computer presented in this chapter is very small compared to commercial computers, it has the advantage of being simple enough so we can demonstrate the design process without too many complications.

The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers. The general- purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations,

operands, and the sequence by which processing has to occur. The data- processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

***instruction code***

***operation code***

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microopera­tions. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2" (or less) distinct operations. As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configu­ration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

At this point we must recognize the relationship between a computer operation and a microoperation. An operation is part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation code, the control issues a sequence of microoperations needed for the hard­ware implementation of the specified operation. For this reason, an operation code is sometimes called a macrooperation because it specifies a set of micro­operations.

The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in proces­sor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their ad­dress. Processor registers can be specified by assigning to the instruction another binary code of **k** bits that specifies one of **2k** registers. There are many variations for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats are con-

SECTION 5-1 Instruction Codes 125

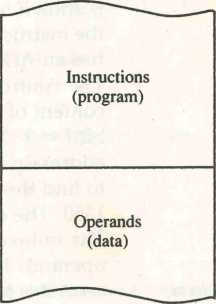
1

**'**

opcode

**Memory**

4096 x 16 J



ceived by computer designers who specify the architecture of the computer. In this chapter we choose a particular instruction code to explain the basic organization and design of digital computers.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Figure 5-1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since **2n** = 4096. If we store each instruc­tion code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible opera­tions, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 5-1 Stored program organization.

**Processor register  
(accumulator or AC)**

15 12 11 0

|  |  |
| --- | --- |
| **Opcode** | **Address** |
|  | **Instruction format** |
| **15** | **0** |
| **Binary operand** | |

Computers that have a single-processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

*accumulator (AC)*

Indirect Address

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

*immediate*

*instruction*

As an illustration of this configuration, consider the instruction code format shown in Fig. 5-2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Fig. 5-2(b). It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode speci­fies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in Fig. 5-2(c) has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself. We define the effective address to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction. Thus the effective address in the instruction of Fig. 5-2(b) is 457 and in the instruction of Fig 5-2(c) is 1350.

*effective address*

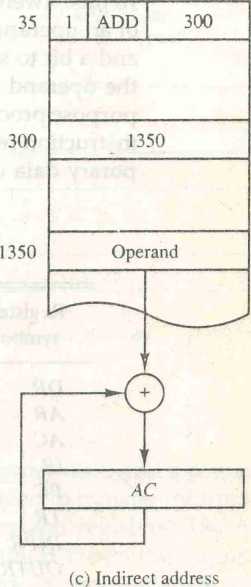
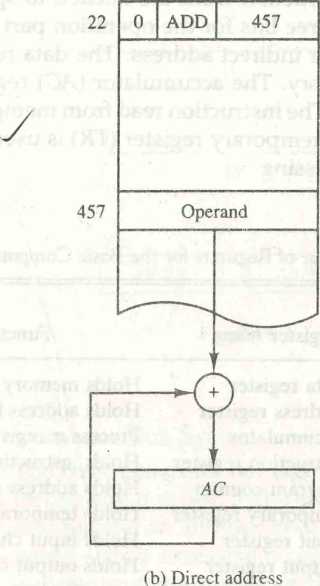
The direct and indirect addressing modes are used in the computer presented in this chapter. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of

15 14 12 11

**0**

**I Opcode**

**Address**

(a) Instruction format

**Memory Memory**

Figure 5-2 Demonstration of direct and indirect address.

data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

**5-2 Computer Registers**

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instructioncode after it is read from memory. The computer needs processor registers for manipulating data and a register for holding a memory address. These require­ments dictate the register configuration shown in Fig. 5-3. The registers are also listed in Table 5-1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation part of the instruction and a bit to specify a direct or indirect address. The data register **(DR)** holds the operand read from memory. The accumulator **(AC)** register is a general- purpose processing register. The instruction read from memory is placed in the instruction register **(IR).** The temporary register **(TR)** is used for holding tem­porary data during the processing.

TABLE 5-1 List of Registers for the Basic Computer

|  |  |  |  |
| --- | --- | --- | --- |
| Register  symbol | Number of bits | Register name | Function |
| DR | 16 | Data register | Holds memory operand |
| AR | 12 | Address register | Holds address for memory |
| AC | 16 | Accumulator | Processor register |
| IR | 16 | Instruction register | Holds instruction code |
| PC | 12 | Program counter | Holds address of instruction |
| TR | 16 | Temporary register | Holds temporary data |
| INPR | 8 | Input register | Holds input character |
| OUTR | 8 | Output register | Holds output character |

The memory address register **(AR)** has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The **PC** goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruc­tion is encountered. A branch instruction calls for a transfer to a nonconsecu- tive instruction in the program. The address part of a branch instruction is transferred to **PC** to become the address of the next instruction. To read an instruction, the content of **PC** is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

*program counter (PC)*

Two registers are used for input and output. The input register **(INPR)** receives an 8-bit character from an input device. The output register **(OUTR.)** holds an 8-bit character for an output device.

**n** o

y

PC

|  |  |  |  |
| --- | --- | --- | --- |
|  | **11** |  | **0** |
|  | **AR** | | |
| **15** |  |  | **0** |
| **IR** | | | |
| **15** |  |  | **0** |
| **TR** | | | |
| **7** | **0** | | **7 0** |
| **OUTR** | |  | **INPR** |

|  |  |  |
| --- | --- | --- |
|  | **Memory 4096 words** |  |
|  | **16 bits per word** |  |
| **15** |  | **0** |
| **DR** | | |
| **15** |  | **0** |
| **AC** | | |

Figure 5-3 Basic computer registers and memory.

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit (to be presented in Sec. 5-4). Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We have shown in Sec. 4-3 how to construct a bus system using multiplex­ers or three-state buffer gates. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 5-4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, Si, and S0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of **DR** is 3. The 16-bit outputs of **DR** are placed on the bus lines when S2SiS0 = 011 since this is the binary value of decimal 3. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2SiS0 = **111.**

*load (LD)*

Four registers, **DR, AC, IR,** and **TR,** have 16 bits each. Two registers, **AR**

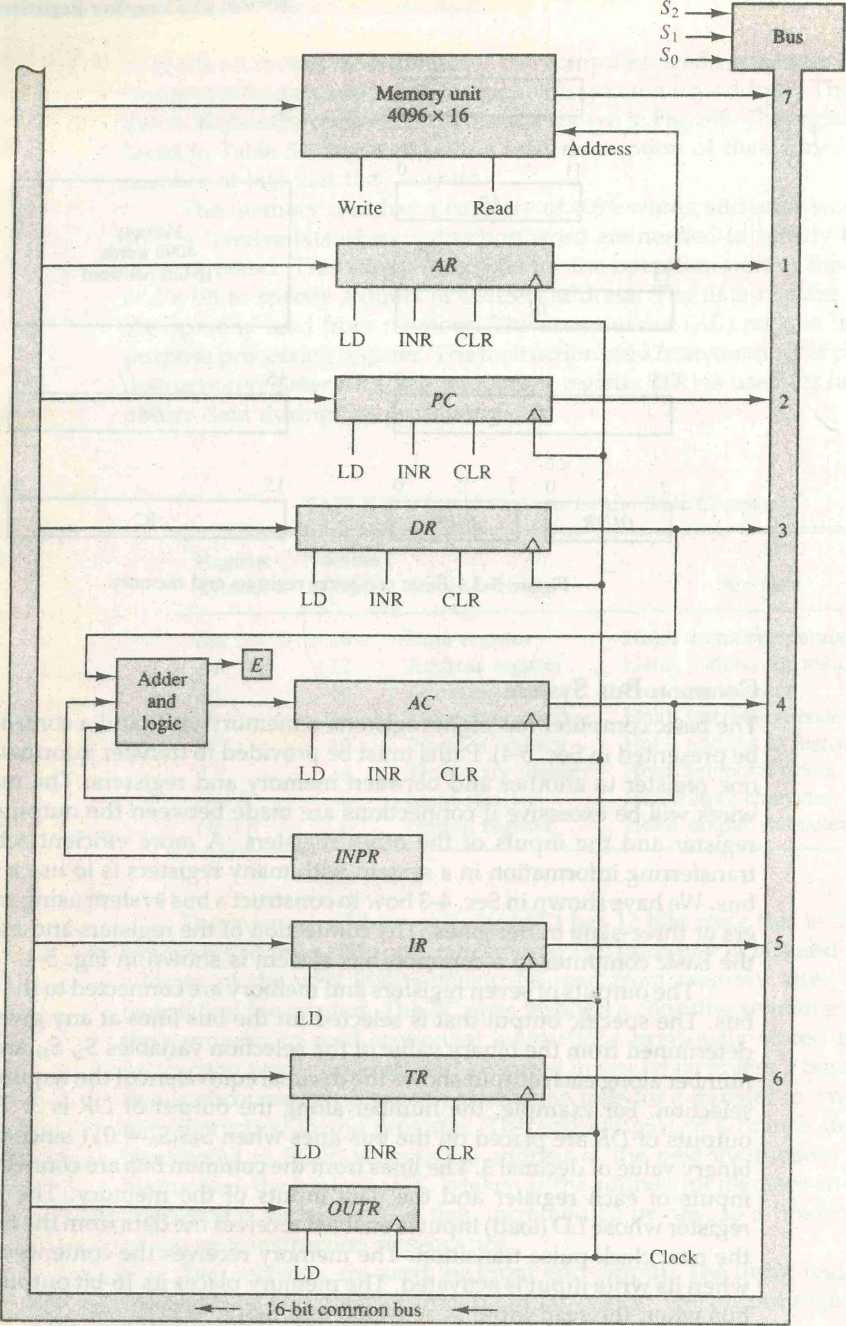


Figure 5-4 Basic computer registers connected to a common bus.



and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive infor­mation from the bus. This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (incre­ment), and CLR (clear). This type of register is equivalent to a binary counter with parallel load and synchronous clear similar to the one shown in Fig. 2-11. The increment operation is achieved by enabling the count input of the coun­ter. Two registers have only a LD input. This type of register is shown in Fig. 2-7.

The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC.

memory address

The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC. They are used to implement register microoperations such as complement AC and shift AC. Another set of 16-bit inputs come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microoperations, such as add DR to AC or AND DR to AC. The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (ex­tended AC bit). A third set of 8-bit inputs come from the input register INPR. The operation of INPR and OUTR is explained in Sec. 5-7.

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC. For example, the two microoperations

DR <- AC and AC DR

can be executed at the same time. This can be done by placing the content of AC on the bus (with S2S]S0 = 100), enabling the LD (load) input of DR, trans­ferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

**5-3 Computer Instructions**

The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode **1.1** is equal to 0 for direct address and to 1 for indirect address (see Fig. 5-2). The register- reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the **AC** register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

***Instruction format***

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode **1.** If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the

J

|  |  |  |  |
| --- | --- | --- | --- |
| 15 14 12 | | 11 | 0 |
| I | Opcode | Address | |
|  | (a) Memory | - reference instruction |  |
| 15 | 12 | 11 | 0 |
| 0 | 1 1 1 | Register operation | |
|  | (b) Register | - reference instruction |  |
| 15 | 12 | 11 | 0 |
| 1 | 1 1 1 | 1/0 operation | |

Figure 5-5 Basic computer instruction formats.

(Opcode = 000 through 110)

(Opcode =111, / = 0)

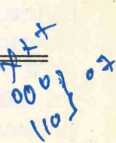
“(Opcode = 111, / = 1)

(c) Input - output instruction

instruction is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol **I** but is not used as a mode bit when the operation code is equal to 111.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register-reference and input-output instructions use the remain­ing 12 bits as part of the operation code, the total number of instructions can exceed eight. In fact, the total number of instructions chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 5-2. The symbol designation is a three-letter word and represents an abbreviation intended for

TABLE 5-2 Basic Computer Instructions

Hexadecimal code

Symbol 7 = 0 7=1 Description

|  |  |  |
| --- | --- | --- |
| AND  ADD  LDA  STA  BUN  BSA  ISZ | Oxxx 8xxx lxxx 9xxx 2xxx Axxx 3xxx Bxxx 4xxx Cxxx 5xxx Dxxx 6xxx Exxx | AND memory word to **AC** Add memory word to **AC** Load memory word to **AC** Store content of **AC** in memory Branch unconditionally Branch and save return address Increment and skip if zero |
| CLA | 7800 | Clear **AC** |
| CLE | 7400 | Clear **E** |
| CMA | 7200 | Complement **AC** |
| CME | 7100 | Complement **E** |
| CIR | 7080 | Circulate right **AC** and **E** |
| CIL | 7040 | Circulate left **AC** and **E** |
| INC | 7020 | Increment **AC** |
| SPA | 7010 | Skip next instruction if **AC** positive |
| SNA | 7008 | Skip next instruction if **AC** negative |
| SZA | 7004 | Skip next instruction if **AC** zero |
| SZE | 7002 | Skip next instruction if £ is 0 |
| HLT | 7001 | Halt computer |
| INP | F800 | Input character to **AC** |
| OUT | F400 | Output character from **AC** |
| SKI | F200 | Skip on input flag |
| SKO | F100 | Skip on output flag |
| ION | F080 | Interrupt on |
| IOF | F040 | Interrupt off |

programmers and users. The hexadecimal code is equal to the equivalent hexa­decimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x’s and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol 1. When 1 = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When 1 = 1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits. The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

hexadecimal code

Instruction Set Completeness

Before investigating the operations performed by the instructions, let us dis­cuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabil­ities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision­making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the

user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

The instructions listed in Table 5-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC (INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC (CM A), and clear AC(CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables (listed in Table 4-6). Moving information from memory to AC is accomplished with the load AC(LDA) instruction. Storing information from AC into memory is done with the store AC(STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. The programs are presented in Chap. 6 together with other pro­gramming examples for the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way we can demonstrate the basic principles of computer organization and design without going into excessive complex details. In Chap. 8 we present a complete list of computer instructions that are included in most commercial computers.

The function of each instruction listed in Table 5-2 and the microopera­tions needed for their execution are presented in Secs. 5-5 through 5-7. We delay this discussion because we must first consider the control unit and understand its internal organization.

**5 A Timing and Control**

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by

*clock pulses*

a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

hardwired control

microprogrammed

control

control unit

timing signals

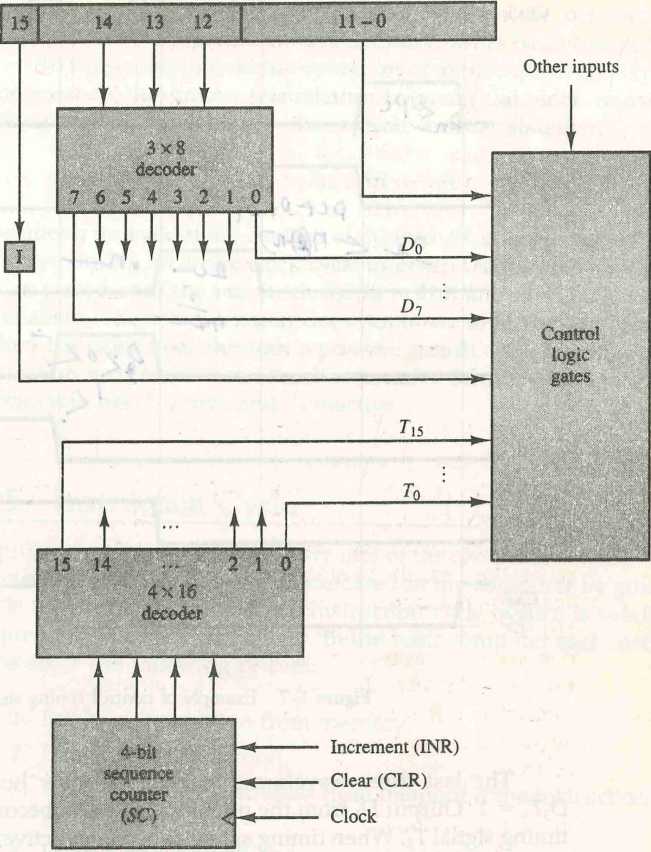
There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, re­quires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section. A microprogrammed control unit for a similar computer is presented in Chap. 7.

The block diagram of the control unit is shown in Fig. 5-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Fig. 5-4. The instruction register is shown again in Fig. 5-6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7. The subscripted decimal number is equivalent to the binary value of the corresponding opera­tion code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T0 through T15. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

The sequence counter SC can be incremented or cleared synchronously (see the counter of Fig. 2-11). Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T0. As an example, consider the case where SC is incremented to provide timing signals T0, Tu t2> T3, and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

D3T4: SC ^ 0

The timing diagram of Fig. 5-7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the

clock clears SC to 0, which in turn activates the timing signal T0 out of the decoder. T0 is active during one clock cycle. The positive clock transition labeled T0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T0. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T0/ Ti, T2, T3/ T4, and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.) If SC is not cleared, the timing signals will continue with T5, T6, up to T15 and back to T0.

Instruction register (//?)

**Control outputs ►-**

Figure 5-6 Control unit of basic computer.

T0 Tx T2 T3 T4 T0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Clock  To | n i | n\_j  i | d | n i | d | n i | n\_  i |
|  | ! |
| Tx |  | L |  |  |  |
|  |
| t2 | j> - |  |  |  |  | - |
| j  i |
| t3 |  |  |  |  |  |  |
| • | r |
| T4 |  |  | KT: |  |  |  |
|  |
| d3 |  |  | | |  |  |  |
| — |  |  |  |
|  | . |  |  |
| CLR  SC |  |

Figure 5-7 Example of control timing signals.

The last three waveforms in Fig. 5-7 show how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T2. When timing signal T4 becomes active, the output of the AND gate that implements the control function D3T4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T4 in the diagram) the counter is cleared to 0. This causes the timing signal T0 to become active instead of T5 that would have been active if SC were incremented instead of cleared.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle ini­tiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

**T0:** AR PC

specifies a transfer of the content of **PC** into **AR** if timing signal **T**0is active. **T0** is active during an entire clock cycle interval. During this time the content of **PC** is placed onto the bus (with **S**2**SiS**0= 010) and the LD (load) input of **AR** is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter **SC** from 0000 to 0001. The next clock cycle has Ta active and T0 inactive.

5'5 Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indi­rect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode

Initially, the program counter **PC** is loaded with the address of the first instruc­tion in the program. The sequence counter **SC** is cleared to 0, providing a decoded timing signal **T0.** After each clock pulse, **SC** is incremented by one, so that the timing signals go through a sequence **T0, Tu T2,** and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

**T0:** AR+-PC

**7V *IR\*-M[AR],*** PC+-PC + **1**

**T2: D0/..., D7Decode** /R(12-14), AR <-/R(0-ll), **7<-/R(** 15)

Since only AR is connected to the address inputs of memory, it is neces­sary to transfer the address from **PC** to **AR** during the clock transition associ­ated with timing signal T0. The instruction read from memory is then placed in the instruction register **1R** with the clock transition associated with timing

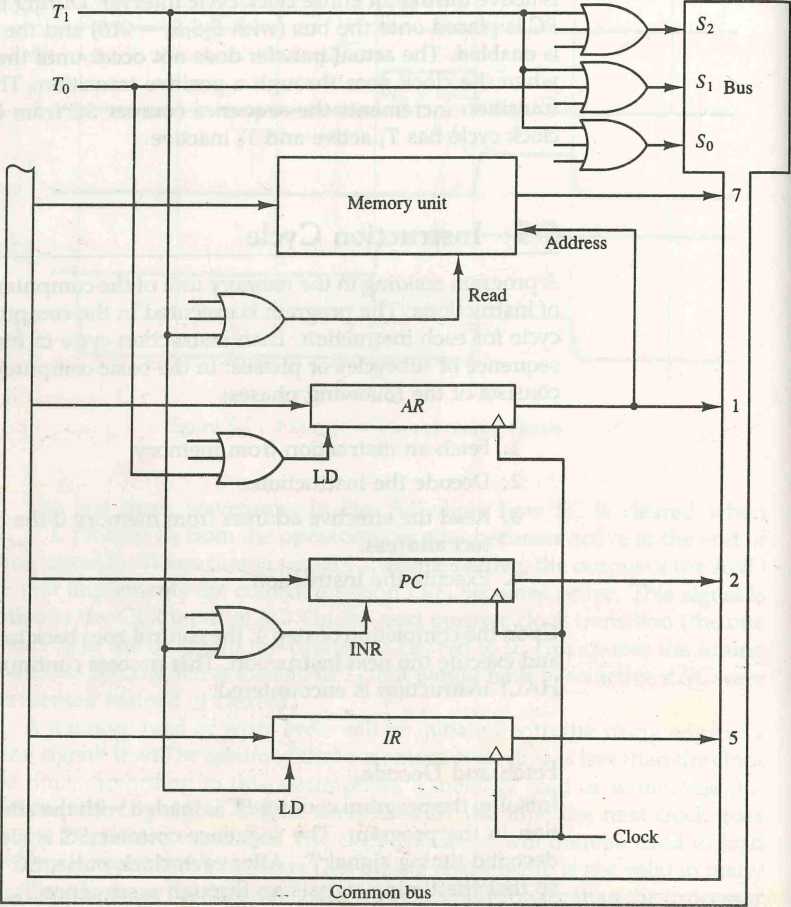


Figure 5-8 Register transfers for the fetch phase.

signal T1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time P2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T0, Tu and T2.

Figure 5-8 shows how the first two register transfer statements are imple­mented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs S2SiS0 equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

The next clock transition initiates the transfer from PC to AR since T0 = 1. In order to implement the second statement

Ti. IR<-M[AR], PC\*-PC + 1

it is necessary to use timing signal Ta to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making S^So = 111.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since Tx = 1.

Figure 5-8 duplicates a portion of the bus system and shows how T0 and Ti are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. 5-9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. 5-5.

Decoder output D7 is equal to 1 if the operation code is equal to binary 111. From Fig. 5-5 we determine that if D7 = 1, the instruction must be a

register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop 7. If D7 = 0 and 7 = 1, we have a memory- reference instruction with an indirect address. It is then necessary to read the

CHAPTER FIVE Basic Computer Organization and Design

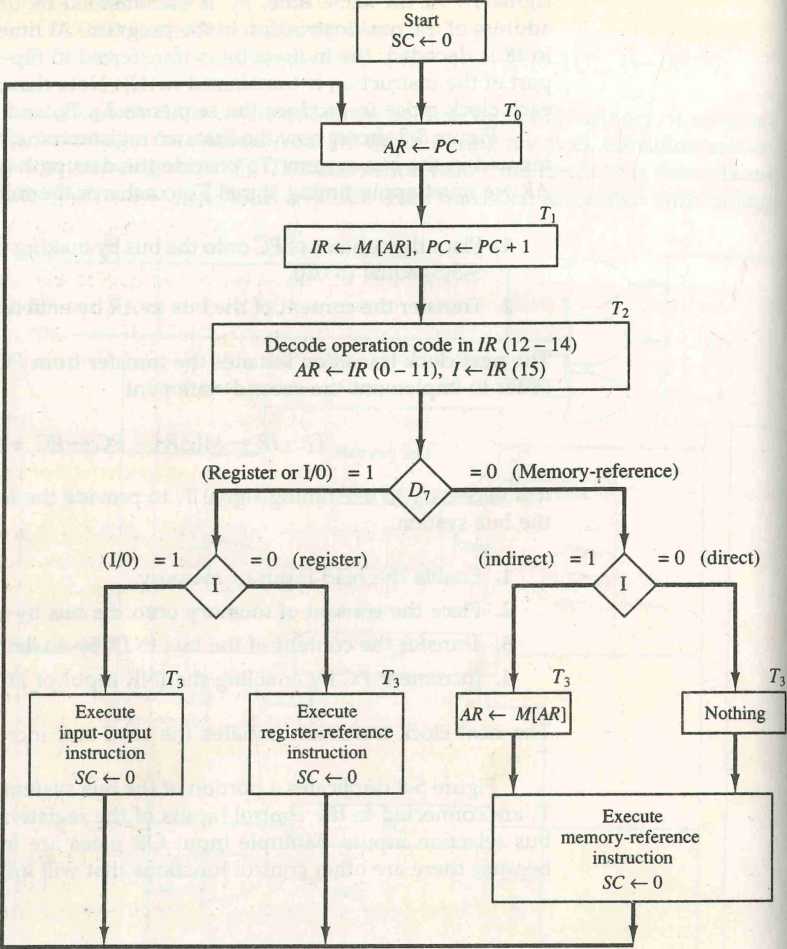


Figure 5-9 Flowchart for instruction cycle (initial configuration).

a

|  |  |
| --- | --- |
| indirect address | effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement  AR <—M[AR]  Initially/ AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.  The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3. This can be symbolized as follows:  D7 /T3: AR±-M[AR]  D7ZT3: Nothing  D7/T3: Execute a register-reference instruction D7IT3: Execute an input-output instruction  When a memory-reference instruction with I = 0 is encountered, it is not necessary7 to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when D7T3 = 1, so that the execution of the memory-reference instruction can be continued with timing variable T4. A register-reference or input-output instruction can be executed with the clock associated with timing signal T3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T0 = 1.  Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC 0.  The register transfers needed for the execution of the register-reference instructions are presented in this section. The memory-reference instructions are explained in the next section. The input-output instructions are included in Sec. 5-7. |
|  | Register-Reference Instructions  Register-reference instructions are recognized by the control when D7 = 1 and 1 = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(O-ll). They were also transferred to AR during time T2.  The control functions and microoperations for the register-reference in- |

structions are listed in Table 5-3. These instructions are executed with the clock transition associated with timing variable T3. Each control function needs the Boolean relation D71'T3, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR(O-ll). By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rB,. For example, the instruction CLA has the hexadecimal code 7800 (see Table 5-2), which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to The next three bits constitute the operation code and are recognized from decoder output D7. Bit 11 in IR is 1 and is recognized from Bn. The control function that initiates the microoperation for this instruction is D7rT3Bn = rBn. The execution of a register-reference instruction is com­pleted at time T3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T0.

The first seven register-reference instructions perform clear, comple­ment, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time Ti). The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

TABLE 5-3 Execution of Register-Reference Instructions

|  |  |  |  |
| --- | --- | --- | --- |
| D7I'T3 | = r (common to all register-reference instructions) | |  |
| IR(i) | = Bi [bit in /R(Q-ll) that specifies the operation] | |  |
|  | r: | SC «-0 | Clear SC |
| CLA | rBn: AC 0 | | Clear AC |
| CLE | rB 1( | £<-0 | Clear E |
| CMA | rB9 | AC\*-AC | Complement AC |
| CME | rB8 | E\*~E | Complement E |
| CIR | rB7 | AC \*-shr AC, AC(15) \*-E, E \*-AC{{)) | Circulate right |
| CIL | rB6 | AC\*-shl AC, AC(0) \*-E, E \*-AC{ 15) | Circulate left |
| INC | rB5 | AC\*-AC + 1 | Increment AC |
| SPA | rZ?4 | If (AC(15) = 0) then {PC\*-PC + 1) | Skip if positive |
| SNA | rB3 | If (AC(15) = 1) then {PC\*-PC + 1) | Skip if negative |
| SZA | rB2 | If {AC = 0) then PC\*-PC + 1) | Skip if AC zero |
| SZE | rB, | If (E = 0) then {PC\*-PC + 1) | Skip if E zero |
| HLT | rB0 | S 0 {S is a start-stop flip-flop) | Halt computer |

**5\*6 Memory ^Reference Instructions**

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 5 2, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

Table 5-4 lists the seven memory-reference instructions. The decoded output D, for i = 0,1, 2, 3, 4, 5, and 6 from the operation decoder that belongs effective address to each instruction is included in the table. The effective address of the instruc­tion is in the address register AR and was placed there during timing signal T2 when 7 = 0, or during timing signal T3 when 7 = 1. The execution of the memory-reference instructions starts with timing signal T4. The symbolic de­scription of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is pre- , sented at the end of this section.

TABLE 5-4 Memory-Reference Instructions

|  |  |  |
| --- | --- | --- |
| Symbol | Operation  decoder | Symbolic description |
| AND | Do | AC\*-AC A M[AR] |
| ADD | Dx | AC <-AC + M[AR], E \*- Cout |
| LDA | d2 | AC\*-M[AR] |
| STA | z>3 | M[AR] \*-AC |
| BUN | Z)4 | PC \*~AR |
| BSA | Ds | M[AR]\*~PCPC\*-AR + 1 |
| ISZ | d6 | M[AR]\*-M[AR] + 1,  If M[AR] + 1 = 0 then PC\*-PC + 1 |

AND to **AC**

\ \

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of

the operation is transferred to AC. The microoperations that execute this instruction are:

D0T4: DR^M[AR]

D0Ts: AC +-AC A DR, SC \*-0

The control function for this instruction uses the operation decoder D0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T4 transfers the operand from memory into DR. The clock transition associated with the next timing signal T5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transfer­ring control to timing signal T0 to start a new instruction cycle.

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry Cout is transferred to the £ (extended accumulator) flip-flop. The microopera­tions needed to execute this instruction are

D,T,: DR <- M[AR]

DiT5: AC <— AC + DR, E \*— Cout, SC <■— 0

The same two timing signals, T4 and T5, are used again but with operation decoder D1 instead of D0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-ref­erence instruction.

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

D2T4: DR <- M[AR]

D2T5: AC DR, SC <- 0

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into AC. The adder and logic circuit receive information from DR which can be transferred into AC. Therefore, it is necessary to read the

memory word into DR first and then transfer the content of DR into AC. The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of AC we can maintain one clock cycle per microoperation.

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

D3T4: M[AR] <- AC, SC 0

BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time Ti to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out ofh sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

D4T4: PC <- AR, SC «- 0

The effective address from AR is transferred through the common bus to PC. Resetting SC to 0 transfers control to T0. The next instruction is then fetched and executed from the memory address given by the new value in PC.

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the ad­dress of the next instruction in sequence (which is available in PC) info a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified in Table 5-4 with the following register transfer:

M[AR] <- PC, PC AR + 1

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10. The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as return address the return address). AR holds the effective address 135. This is shown in part

(a) of the figure. The BSA instruction performs the following numerical oper­ation:

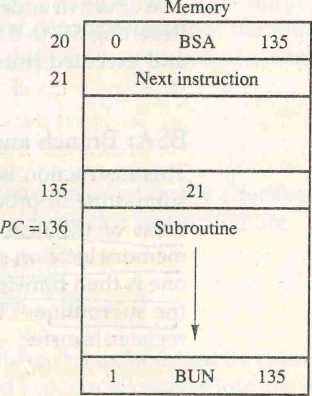
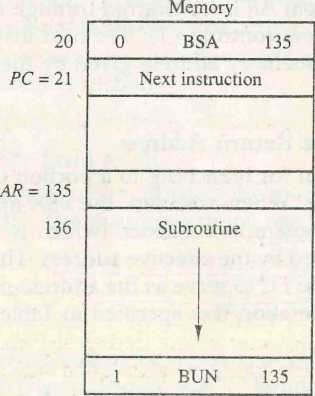
M[135] <- 21, PC <— 135 + 1 = 136

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

subroutine call The BSA instruction performs the function usually referred to as a sub­

routine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor

Figure 5-10 Example of BSA instruction execution.



register or in a portion of memory called a stack. This is discussed in more detail in Sec. 8-7.

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D5T4: M[AR] \*- PC, AR <— AR + 1 D5T5: PC +- AR, SC <- 0

Timing signal T4 initiates a memory write operation, places the content of PC onto the bus, arid enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T5 to transfer the content of AR to PC.

ISZ: Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

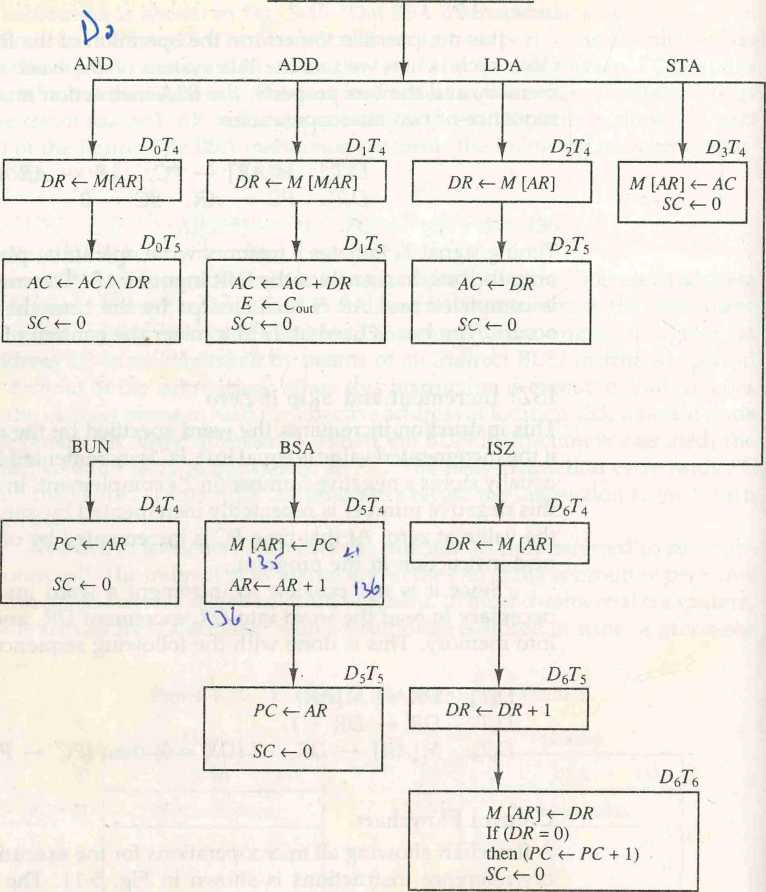
D6T4: DR M[AR]

D6T5: DR DR + 1

D6T6: M[A.R] <- DR, if (DR = 0) then (PC <- PC + 1), SC <- 0 Control Flowchart

A flowchart showing all microoperations for the execution of the seven mem­ory-reference instructions is shown in Fig. 5-11. The control functions are indicated on top of each box. The microoperations that are performed during time T4, T5, or T6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T0 to start the next instruction cycle.

Note that we need only seven timing signals to execute the longest instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

**5^7 Input-Output and Interrupt**

**Memory - reference instruction**

Figure 5-11 Flowchart for memory-reference instructions.

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types ofinput and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer. Input-output organization is dicsussed further in Chap. 11.

Input—Output Configuration

The terminal sends and receives serial information. Each quantity of informa­tion has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. 5-12. The transmitter interface re­ceives serial information from the keyboard and transmits it to INPR. The re­ceiver interface receives information from OUTR and sends it to the printer serially. The operation of the serial communication interface is explained in Sec. 11-3.

The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is

input register

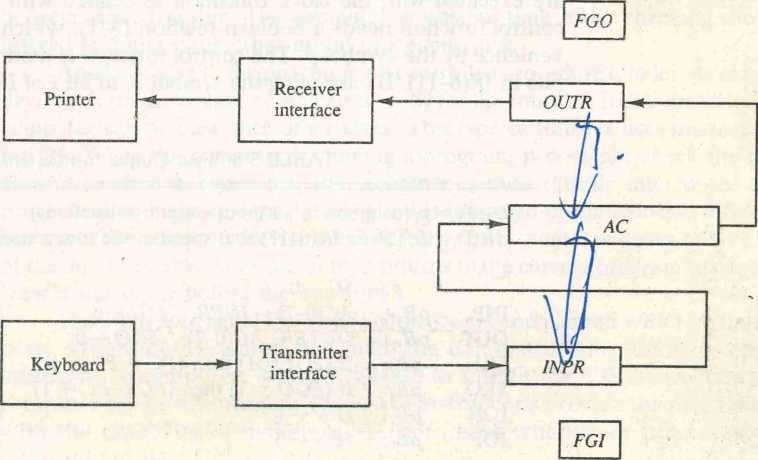
Figure 5-12 Input-output configuration.

y

Input - output Serial Computer

terminal communication registers and

interface flip-flops



set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanu­meric code is shifted into IN PR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR. is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

V

*output register*

The output register OUTR works similarly but the direction of informa­tion flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded infor­mation, prints the corresponding character, and when the operation is com­pleted, it sets FGO to 1. The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

Input-Output Instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recog­nized by the control when D7 = 1 and 7 = 1. The remaining bits of the instruc­tion specify the particular operation. The control functions and microopera­tions for the input-output instructions are listed in Table 5-5. These instructions are executed with the clock transition associated with timing signal T3. Each control function needs a Boolean relation D7IT3, which we designate for con­venience by the symbol p. The control function is distinguished by one of the bits in ZjR(6-11). By assigning the symbol B, to bit i of IR, all control functions

TABLE 5-5 Input-Output Instructions

D~IT3 = p (common to all input-output instructions) IR(\) = Bi [bit in /i?(6—11) that specifies the instruction]

|  |  |  |  |
| --- | --- | --- | --- |
|  | **P** | **SC+-** 0 | Clear **SC** |
| INP | **pBn** | **AC(Q-1)\*-INFR, FGI ^-0** | Input character |
| OUT | **pBio** | **OUTR \*-AC((N**7), **FGO \*-** 0 | Output character |
| SKI | **pB9** | If (**FGI =** l) then **(PC\*-PC** + l) | Skip on input flag |
| SKO | **pBs** | If **(FGO =** 1) then **(PC\*-PC +** 1) | Skip on output flag |
| ION | **pB7** | **IEN \*-l** | Interrupt enable on |
| IOF | **pB6** | **IEN \*~** 0 | Interrupt enable off |

cart be denoted by pB, tor i = 6 though 11. The sequence counter SC is cleared to 0 when p = D7IT3 = 1.

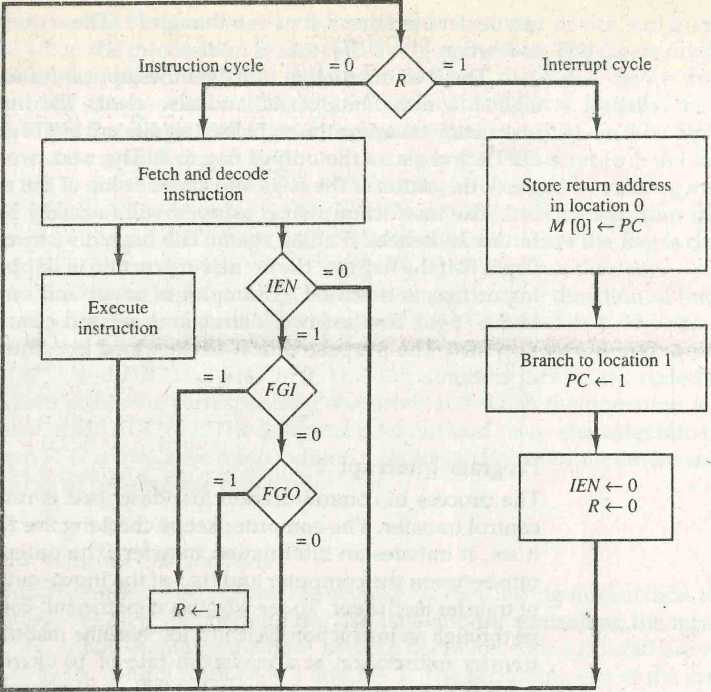
The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0. The next two instructions in Table 5-5 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is'O. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. (Examples of input and output programs are giver in Sec. 6-8.) The last two instructions set and clear an interrupt enable/' , flop IEN. The purpose of IEN is explained in conjunction with the interr'a^ operation.

Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 ps. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 ps. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can foe set and cleared with two instruc­tions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The wav that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-13. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

*interrupt cycle*

Figure 5-13 Flowchart for interrupt cycle.

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor

register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Fig. 5-14. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 5-14{a).

When control reaches timing signal T0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is sec to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 544(b).

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruc­tion is placed at the end of the I/O service program. After this instruction is

Figure 5-14 Demonstration of the interrupt cycle. Memory Memory

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **0** |  | **0** | **256** | |
| **1** | **0 BUN 1120** | **PC = 1** | **0** | **BUN 1120** |
| **255 PC = 256** | **Main**  **program** | **255**  **256** | **Main**  **program** | |
| **1120** | **I/O** | **1120** |  | **I/O** |
|  | **program** |  |  | **program** |
|  | **1 BUN 0** |  | **1 BUN 0** | |

(a) Before interrupt

(b) After interrupt cycle

read from memory during the fetch phase, control goes to the indirect phase (because I - 1) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Interrupt **Cycle**

We are now ready to list the register transfer statements for the interrupt cycle, The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T0/ Tif or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

*TqT[T2(IEN)(FGI + FGO): R* <- 1

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and TqT{T2. modified fetch phase We now modify the fetch and decode phases of the instruction cycle.

Instead of using only timing signals T0, Tlf and T2 (as shown in Fig. 5-9) we will AND the three timing signals with R' so that the fetch and decode phases will be recognized from the three control functions R'T0, R'TU and R'T2. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R — 0. Otherwise, if R — 1, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of microoperations:

RT0: AR «—0, TR \*-PC

RTii M[AR]\*-TR, PC<- 0

RT2: PC+-PC + 1, IEN <—0, R <- 0, SC<-0

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T0 by clearing SC to 0. The beginning of the next instruction cycle has the condition R'T0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

5-8 Complete Computer Description \_\_\_\_\_

The final flowchart of the instruction cycle, including the interrupt cycle for the **flowchart for basic** basic computer, is shown in Fig. 545 . The interrupt flip-flop **R** may be set at

**computer** any time during the indirect or execute phases. Control returns to timing signal

T0 after SC is cleared to 0. If **R** = 1, the computer goes through an interrupt cycle. If **R** = 0, the computer goes through an instruction cycle. If the instruc­tion is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruc­tion according to the flowchart of Fig. 5-11. If the instruction is one of the register-reference instructions, it is executed with one of the microoperations listed in Table 5-3. If it is an input-output instruction, It is executed with one the inicrooperations listed in Table 5-5.

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table. The entries in the table are taken from Figs. 5-11 and 5-15, and Tables 5-3 and 5-5.

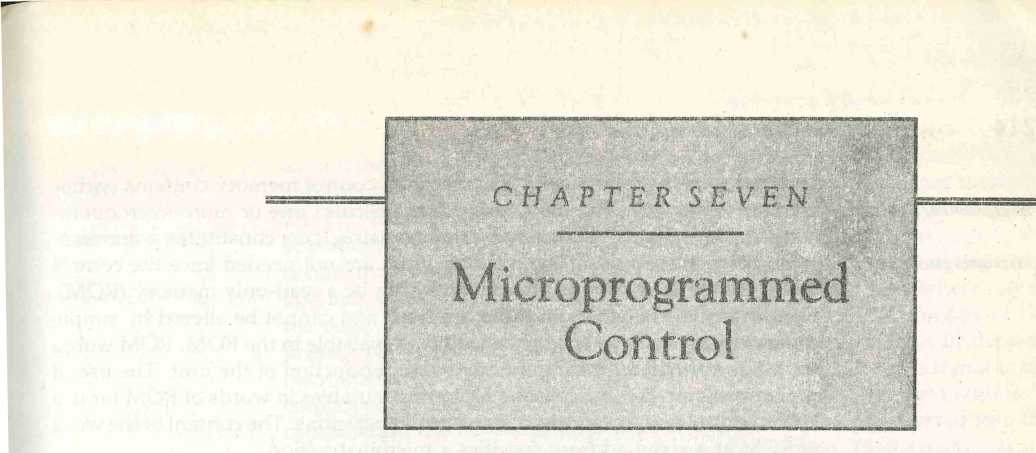
The control functions and microoperations for the entire computer are summarized in Table 5-6. The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer. The control functions and conditional control statements listed in the table formulate the Boolean functions for the gates in the control unit. The list of microoperations specifies the type of control inputs needed for the registers and memory. A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

Design of Basic Computer \_

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each
2. **Nine registers:** AR, PC, DR, AC, IR, TR, OUTR, INPR, **and** SC
3. Seven flip-flops: **I, S, E, R, IEN, FGI**, and **FGO**
4. Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder
5. A 16-bit common bus
6. Control logic gates
7. Adder and logic circuit connected to the input of AC

The memory unit is a standard component that can be obtained readily from a commercial source. The registers are of the type shown in Fig. 2-11 and



IN THIS CHAPTER

7-1 Control Memory

7-2 Address Sequencing

7-3 Microprogram Example

' 7-4 Design of Control Unit

7-1 Control Memory

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for con­trolling the microoperation sequences in a digital computer.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. A control variable in the opposite binary state does not change the state of the registers in the system. The active state of a control variable may be either the I state or the 0 state, depending on the application. In a bus-organized system,, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units,

The control unit initiates a series of sequential steps of micrcoperations. During any given time, certain microoperations are to be iniliated, while others remain idle. The control variables at any given time can be represented by a control word string of l's and 0's called a control word. As such, control words can be

programmed to perform various operations on the components of the system, A control unit whose binary- control variables are stored in memory is called

a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microopera­tions for the system. A sequence of microinstructions constitutes a micropro­gram . Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM. ROM words are made permanent during the hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

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

microins truction microprogram

*control memory*

A computer that employ s a microprogrammed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of mi­croinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initiates a series of microin­structions in control memory. These microinstructions generate the microop­erations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The generakfonfiguratiort of a microprogrammed control unit is demon­strated in the block diagram of Fig. 7-1. The control memory is assumed to be a ROM, within which all control information is permanently stored. The

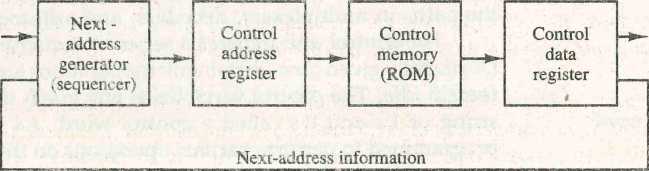


Figure 7-1 Microprogrammed control organization.

**External**

**input**

**Control**

**word**

control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more micro­operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruc­tion may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruc­tion contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

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

control address register

sequencer

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

pipeline register

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

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control se-

quence for the system, all we need to do is specify a different set of microin­structions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

It should be mentioned that most computers based on the reduced in­struction set computer (RISC) architecture concept (see Sec. 8-8) use hardwired control rather than a control memory with a microprogram. An example of a hardwired control for a simple computer is presented in Sec. 5-4.

hardwired control

**7-2 Address Sequencing**

Microinstructions are stored in control memory in groups, with each group specifying a **routine.** Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

routine

/An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index regis­ters. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruc­tion fetched from memory. The microoperation steps to be generated in proc­essor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a **mapping** process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by increment­ing the control address register, but sometimes the sequence of rricroopera-

mapping

SECTION 7-2 Address Sequencing 217

tions will depend on values of certain status bits in processor registers. Microprograms that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the unit has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure 7-2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address. The microin­struction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer incre­ments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

Conditional Branching

The branch logic of Fig. 7-2 provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

*special bits*

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

branch logic



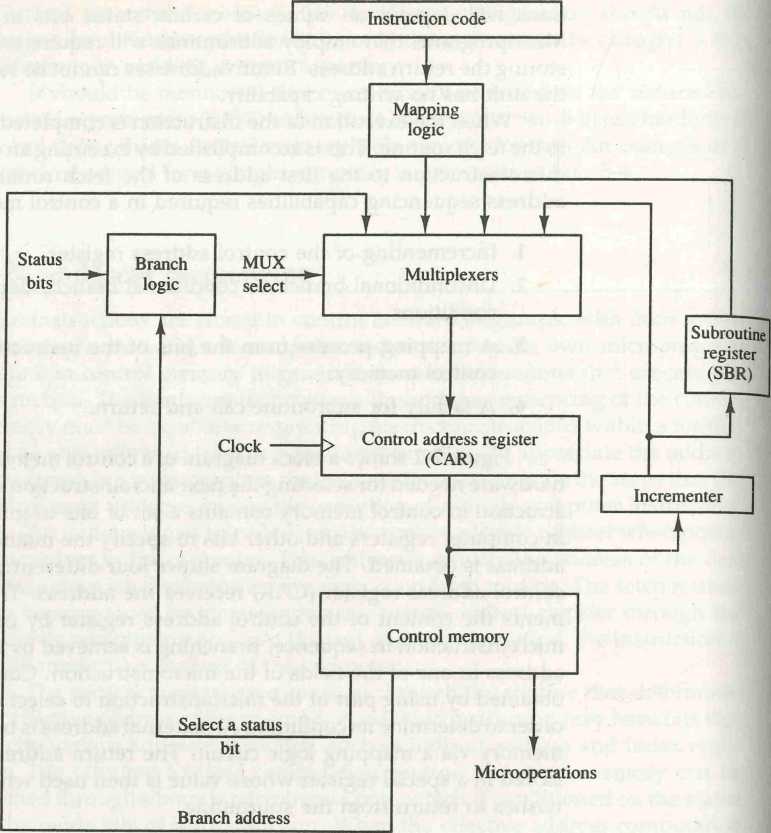


Figure 7-2 Selection of address for control memory.

This can be implemented with a multiplexer. Suppose that there are eight status bit conditions in the system. Three bits in the microinstruction are used to specify any one of eight status bit conditions. These three bits provide the selection variables for the multiplexer. If the selected status bit is in the 1 state, the output of the multiplexer is 1; otherwise, it is 0. A 1 output in the multi­plexer generates a control signal to transfer the branch address from the microinstruction into the control address register. A 0 output in the multiplexer causes the address register to be incremented; In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.

An unconditional branch microinstruction can be implemented by load­ing the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to be loaded into the control address register unconditionally.

Mapping of Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruc­tion is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 7-3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig. 7-3. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the map­ping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any de­sired location in control memory. The mapping concept provides flexibility for adding instructions for control memory as the need arises.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **10 11** | | **address** | |
| **0** | | **X X X X** | **0 0** |  |
| **0** | | **10 11** | **0 0** |  |

Figure 7-3 Mapping from instruction code to microinstruction address.

Opcode

Computer instruction:

Mapping bits:

Microinstruction address:



**220** **CHAPTER SEVEN** Microprogrammed Control

The mapping function is sometimes implemented by means of an inte­grated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently with a PLD.

Subroutines

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many microprograms contain identical sec­tions of code. Microinstructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoper­ations needed to generate the effective address of the operand for an instruc­tion is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented subroutine register output from the control address register into a subroutine register and branch­ing to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack. The use of a stack in subroutine calls and returns is explained in more detail in Sec. 8-7.

**7-3 Microprogram Example**

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is mi­croprogrammed. The computer used here is similar but not identical to the basic computer introduced in Chap. 5.

Computer Configuration

The block diagram of the computer is shown in Fig. 7-4. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are

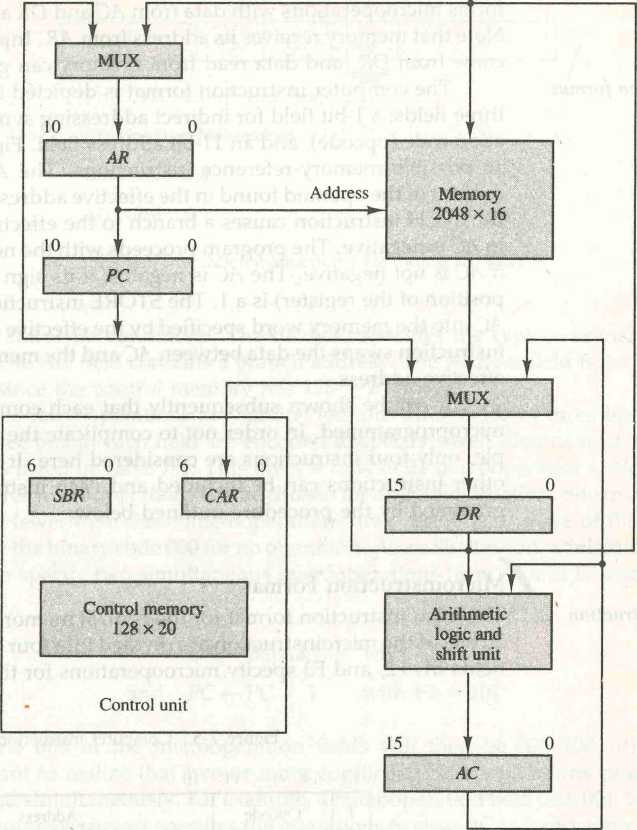


Figure 7-4 Computer hardware configuration.

program counter PC, address register AR, data register DR, and accumulator register AC. The function of these registers is similar to the basic computer introduced in Chap. 5 (see Fig. 5-3). The control unit has a control address register CAR and a subroutine register SBR. The control memory and its registers are organized as a microprogrammed control unit, as shown in Fig. 7-2.

The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic, logic, and shift unit per-

forms microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

The computer instruction format is depicted in Fig. 7-5(a). It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit oper­ation code (opcode), and an 11-bit address field. Figure 7-5(b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

***instruction format***

***microinstruction***

***format***

It will be shown subsequently that each computer instruction must be microprogrammed. In order not to complicate the microprogramming exam­ple, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be micropro­grammed by the procedure outlined below.

Microinstruction Format

The microinstruction format for the control memory is shown in Fig. 7-6. The 20 bits of the microinstruction are divided into four functional parts. The three fields FI, F2, and F3 specify microoperations for the computer. The CD field

Figure 7-5 Computer instructions.

15 14 11 10

0



**I Opcode**

**Address**

**(a) Instruction format**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Opcode** | **Description** |
| **ADD** | **0000** | **AC <-AC+ M [EA]** |
| **BRANCH** | **0001** | **If (AC < 0) then (PC <— EA)** |
| **STORE** | **0010** | **M [EA] <- AC** |
| **EXCHANGE** | **0011** | **AC <r- M[EA], M[EA] <- AC** |

**EA is the effective address**

**(b) Four computer instructions**

***microoperations***

***condition field***

*\*<4/*

SECTION 7-3 Microprogram Example 223

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **3** | **3** | **2** | **2** | **7** |
| **FI** | **F2** | **F3** | **CD** | **BR** | **AD** |

**FI, F2, F3: Microoperation fields CD: Condition for branching BR: Branch field AD: Address field**

Figure 7-6 Microinstruction code format (20 bits).

selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27 words.

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microopera­tions as listed in Table 7-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an illustration, a microinstruc­tion can specify two simultaneous microoperations from F2 and F3 and none from FI.

DR \*-M[AR] and PC\*-PC + 1

with F2 = 100 with F3 = 101

The nine bits of the microoperation fields will then be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and subtract DR from AC at the same time.

Each microoperation in Table 7-1 is defined with a register transfer state­ment and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC<—DR (FI = 100) has the symbol DRTAC, which stands for a transfer from DR to AC.

(The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 7-1. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit

|  |  |  |
| --- | --- | --- |
| FI | Microoperation | Symbol |
| 000 | None | NOP |
| 001 | AC \*—AC + DR | ADD |
| 010 | AC 0 | CLRAC |
| Oil | AC+-AC + 1 | INCAC |
| 100 | AC <—DR | DRTAC |
| 101 | AR <— DR(0-10) | DRTAR |
| 110 | AR+-PC | PCTAR |
| 111 | M[AR] <—DR | WRITE |



F2 Microoperation Symbol

|  |  |  |
| --- | --- | --- |
| 000 | None | NOP |
| 001 | AC <—AC - DR | SUB |
| 010 | AC <r-AC V DR | OR |
| Oil | AC <^-AC A DR | AND |
| 100 | DR \*-M[AR] | READ |
| 101 | DR <—AC | ACTDR |
| 110 | DR \*—DR + 1 | INCDR |
| 111 | DR(0-10)<-PC | PCTDR |
| F3 | Microoperation | Symbol |
| 000 | None | NOP |
| 001 | AC +- AC© DR | XOR |
| 010 | AC<r~AC | COM |
| Oil | AC «-shl AC | SHL |
| 100 | AC <— shr AC | SHR |
| 101 | PC^PC + 1 | INCPC |
| 110 | PC <—AR | ARTPC |
| 111 | Reserved |  |

CD Condition Symbol Comments

|  |  |  |  |
| --- | --- | --- | --- |
| 00 | Always = 1 | U | Unconditional branch |
| 01 | DR( 15) | I | Indirect address bit |
| 10 | AC(15) | S | Sign bit of AC |
| 11 | AC = 0 | **z** | Zero value in AC |

BR Symbol Function

|  |  |  |
| --- | --- | --- |
| 00 | JMP | CAR «— AD if condition = 1 CAR CAR + 1 if condition = 0 |
| 01 | CALL | CAR \*—AD, SBR <—CAR + 1 if condition CAR <^-CAR + 1 if condition = 0 |
| 10 | RET | CAR <— SBR (Return from subroutine) |
| 11 | MAP | CAR(2-5) <—DjR(11-14), CAR(0,1,6) <—0 |

I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit. The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero. We will use the symbols U, I, S, and Z for the four status bits when we write microprograms in symbolic form.

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in Table 7-1, when BR = 00, the control performs a jump (JMP) opera­tion (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register SBR. The jump and call operations depend on the value of the CD field. If the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

***branch*** *field*

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. This mapping is as depicted in Fig. 7-3. The bits of the operation code are in DR(11-14) after an instruction is read from memory. Note that the last two conditions in the BR field are independent of the values in the CD and AD fields.

Symbolic Microinstructions

The symbols defined in Table 7-1 can be used to specify microinstructions in symbolic form. A symbolic microprogram can be translated into its binary equivalent by means of an assembler. A microprogram assembler is similar in concept to a conventional computer assembler as defined in Sec. 6-3. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following information.

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2. The microoperations field consists of one, two, or three symbols, sep­arated by commas, from those defined in Table 7-1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3. The CD field has one of the letters U, I, S, or Z.
4. The BR field contains one of the four symbols defined in Table 7-1.
5. The AD field specifies a value for the address field of the microinstruc­tion in one of three possible ways:

*address field*

1. With a symbolic address, which must also appear as a label.
2. With the symbol NEXT to designate the next address in sequence.
3. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

We will use also the pseudoinstruction ORG to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

ORG

The Fetch Routine

The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64. The microinstructions needed for the fetch routine are

AR+-PC

DR<-M[AR], PC\*-PC + 1

AR <-DR (0-10), CAR(2-5) DR(11-14), CAR(0,1,6) <-0

The address of the instruction is transferred from PC to AR and the instruction is then read from memory into DR. Since no instruction register is available, the instruction code remains in DR. The address part is transferred to AR and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from DR into CAR.

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. Using the assembly language conventions defined previously, we can write the symbolic microprogram for the fetch routine as follows:

*fetch and decode*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FETCH:** | **ORG G4 PCTAR** | **U** | **JMP** | **NEXT** |
|  | **READ**, **INCPC** | **U** | **JMP** | **NEXT** |
|  | **DRTAR** | **U** | **MAP** |  |

The translation of the symbolic microprogram to binary produces the following binary microprogram. The bit values are obtained from Table 7-1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Binary  Address | FI | F2 | F3 | CD | BR | AD |
| **1000000** | **no** | **000** | **000** | **00** | **00** | **1000001** |
| **1000001** | **000** | **100** | **101** | **00** | **00** | **1000010** |
| **1000010** | **101** | **000** | **000** | **00** | **11** | **0000000** |

The three microinstructions that constitute the fetch routine have been listed in three different representations. The register transfer representation shows the internal register transfer operations that each microinstruction im­plements. The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory. It is customary to write microprograms in symbolic form and then use an assembler program to obtain a translation to binary.

Symbolic Microprogram

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address OxxxxOO, where xxxx are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to CAR the address

1. which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20,..., 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 7-2. The table also shows the symbolic microprogram for the fetch routine and the microin­struction routines that execute four computer instructions.

To see how the transfer and return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored. The first microinstruc­tion in the ADD routine calls subroutine INDRCT, conditioned on status bit

1. If I = 1, a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register SBR. The INDRCT subroutine has two microinstructions:

INDRCT: READ U JMP NEXT

DRTAR U R£T

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Label | Microoperations | CD | BR | AD |
| ADD: | ORG 0 NOP | I | CALL | INDRCT |
|  | READ | U | JMP | NEXT |
|  | ADD | U | JMP | FETCH |
| BRANCH: | ORG 4 NOP | S | JMP | OVER |
|  | NOP | U | JMP | FETCH |
| OVER: | NOP | I | CALL | INDRCT |
|  | ARTPC | u | JMP | FETCH |
| STORE: | ORG 8 NOP | I | CALL | INDRCT |
|  | ACTDR | u | JMP | NEXT |
|  | WRITE | u | JMP | FETCH |
| EXCHANGE: | ORG 12 NOP | I | CALL | INDRCT |
|  | READ | u | JMP | NEXT |
|  | ACTDR, DRTAC | u | JMP | NEXT |
|  | WRITE | u | JMP | FETCH |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ORG 64 |  |  |  |
| FETCH: | PCTAR | **U** | JMP | NEXT |
|  | READ, INCPC | **U** | JMP | NEXT |
|  | DRTAR | **u** | MAP |  |
| INDRCT: | READ | **u** | JMP | NEXT |
|  | DRTAR | **u** | RET |  |

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to AR. The return from subroutine (RET) transfers the address from SBR to CAR, thus returning to the second microinstruction of the ADD routine.

The execution of the ADD instruction is carried out by the microinstruc­tions at addresses 1 and 2. The first microinstruction reads the operand from memory into DR. The second microinstruction performs an add microopera­tion with the content of DR and AC and then jumps back to the beginning of the fetch routine.

***execution of instructions***

The BRANCH instruction should cause a branch to the effective address

if AC < 0. The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1. The BRANCH routine in Table 7-2 starts by checking the value of S. If S is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if 1 = 1. The effective address is then transferred from AR to PC and the mi­croprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if / = 1. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR. The contents of DR and AC are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type (see Fig. 1-23). The original content of AC that is now in DR is stored back in memory.

Note that Table 7-2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been microprogrammed. Also, control memory words at locations 69 to 127 have not been used. Instructions such as multiply, divide, and others that require a long sequence of micro­operations will need more than four microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

Binary Microprogram

The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example.

The equivalent binary form of the microprogram is listed in Table 7-3. The addresses for control memory are given in both decimal and binary. The binary content of each microinstruction is derived from the symbols and their equiv­alent binary values as defined in Table 7-1.

Note that address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0,1, and 2. The next routine starts at address 4. Even though address 3 is not used, some binary value must be specified for each word in control memory. We could have specified all 0's in the word since this location will never be used. However, if some unforeseen error occurs, or if a noise signal sets CAR to the value of 3, it will be wise to jump to address 64, which is the beginning of the fetch routine.

The binary microprogram listed in Table 7-3 specifies the word content of the control memory. When a ROM is used for the control memory, the

***control memory***

TABLE 7-3 Binary Microprogram for Control Memory (Partial)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Micro  Routine | Address | |  | Binary Microinstruction | | | | |
| Decimal | Binary | FI | F2 | F3 | CD | BR | AD |
| ADD | 0 | 0000000 | 000 | 000 | 000 | 01 | 01 | 1000011 |
|  | 1 | 0000001 | 000 | 100 | 000 | 00 | 00 | 0000010 |
|  | 2 | 0000010 | 001 | 000 | 000 | 00 | 00 | 1000000 |
|  | 3 | 0000011 | 000 | 000 | 000 | 00 | 00 | 1000000 |
| BRANCH | 4 | 0000100 | 000 | 000 | 000 | 10 | 00 | 0000110 |
|  | 5 | 0000101 | 000 | 000 | 000 | 00 | 00 | 1000000 |
|  | 6 | 0000110 | 000 | 000 | 000 | 01 | 01 | 1000011 |
|  | 7 | 0000111 | 000 | 000 | 110 | 00 | 00 | 1000000 |
| STORE | 8 | 0001000 | 000 | 000 | 000 | 01 | 01 | 1000011 |
|  | 9 | 0001001 | 000 | 101 | 000 | 00 | 00 | 0001010 |
|  | 10 | 0001010 | 111 | 000 | 000 | 00 | 00 | 1000000 |
|  | 11 | 0001011 | 000 | 000 | 000 | 00 | 00 | 1000000 |
| EXCHANGE | 12 | 0001100 | 000 | 000 | 000 | 01 | 01 | 1000011 |
|  | 13 | 0001101 | 001 | 000 | 000 | 00 | 00 | 0001110 |
|  | • 14 | 0001110 | 100 | 101 | 000 | 00 | 00 | 0001111 |
|  | 15 | 0001111 | 111 | 000 | 000 | 00 | 00 | 1000000 |
| FETCH | 64 | 1000000 | 110 | 000 | 000 | 00 | 00 | 1000001 |
|  | 65 | 1000001 | 000 | 100 | 101 | 00 | 00 | 1000010 |
|  | 66 | 1000010 | 101 | 000 | 000 | 00 | 11 | 0000000 |
| INDRCT | 67 | 1000011 | 000 | 100 | 000 | 00 | 00 | 1000100 |
|  | 68 | 1000100 | 101 | 000 | 000 | 00 | 10 | 0000000 |

microprogram binary list provides the truth table for fabricating the unit. This fabrication is a hardware process and consists of creating a mask for the ROM so as to produce the l's and 0's for each word. The bits of ROM are fixed once the internal links are fused during the hardware production. The ROM is made of IC packages that can be removed if necessary and replaced by other pack­ages. To modify the instruction set of the computer, it is necessary to generate a new microprogram and mask a new ROM. The old one can be removed and the new one inserted in its place.

If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing a RAM for the control memory is that the mi­croprogram can be altered simply by writing a new pattern of l's and 0's without resorting to hardware procedures. A writable control memory pos­sesses the flexibility of choosing the instruction set of a computer dynamically by changing the microprogram under processor control. However, most mi­croprogrammed systems use a ROM for the control memory because it ischeaper and faster than a RAM and also to prevent the occasional user from changing the architecture of the system.

7-4 Design of Control Unit

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in in­struction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate microop­erations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2k microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the proces­sor unit.

Figure 7-7 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruc­tion presently available in the output of control memory are decoded with a 3x8 decoder to provide eight outputs. Each of these outputs must be con­nected to the proper circuit to initiate the corresponding microoperation as specified in Table 7-1. For example, when FI = 101 (binary 5), the next clock pulse transition transfers the content of DR (0-10) to AR (symbolized by DRTAR in Table 7-1). Similarly, when FI = 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 7-7, outputs 5 and 6 of decoder FI are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of the decoders that initiate transfers between registers must be con­nected in a similar fashion.

decoding of F fields

The arithmetic logic shift unit can be designed as in Figs. 5-19 and 5-20. Instead of using gates to generate the control signals marked by the symbols AND, ADD, and DR in Fig. 5-19, these inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC, respec-

arithmetic logic shift unit

FI **F2** F3

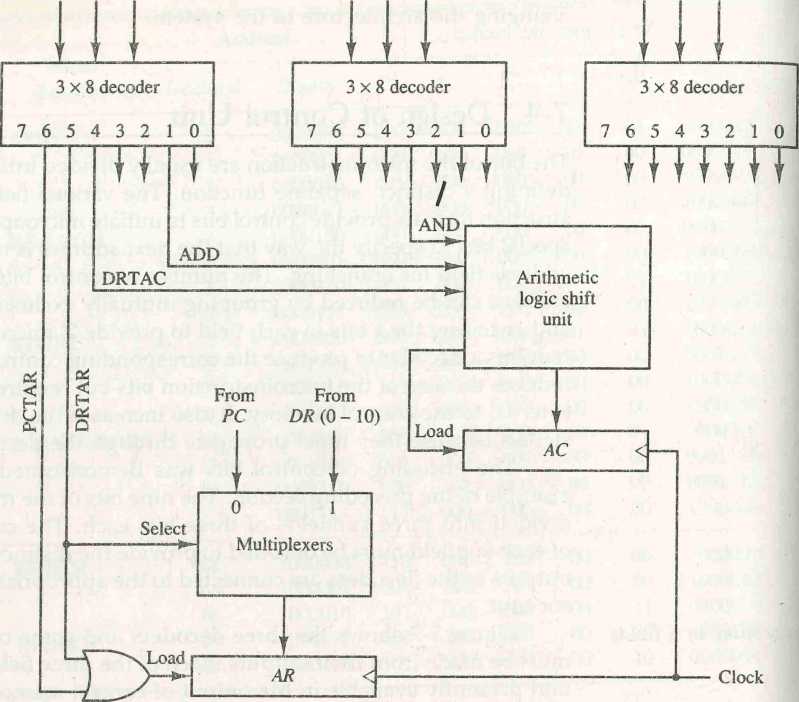


Figure 7-7 Decoding of microoperation fields.

tively, as shown in Fig. 7-7. The other outputs of the decoders that are asso­ciated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

Microprogram **Sequencer**

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be con­structed with digital functions to suit a particular application. However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram con­trol units. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applicalions.

SECTION 7-4 Design of Control Unit 233

The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 7-8. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from CAR provides the address for the control memory. The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction. Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subrou­tines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selectgs one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

The input logic circuit in Fig. 7-8 has three inputs, I0, h, and T, and three outputs, S0, Si, and L. Variables S0 and Sa select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with Si S0 = 10, multiplexer input number 2 is selected and establishes a transfer

*design of input logic*

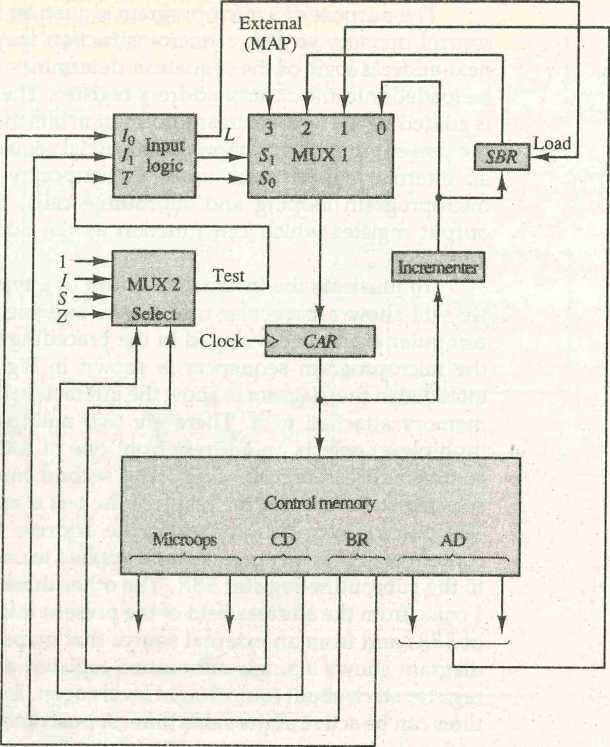


Figure 7-8 Microprogram sequencer for a control memory.

path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

The truth table for the input logic circuit is shown in Table 7-4. Inputs h and I0 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 7-1. The bit values for Si and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1). The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

S,=/i

So = hio + I[T l = IikT

The circuit can be constructed with three AND gates, an OR gate, and an inverter.

BR Input MUX 1 Load SBR

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Field | | h | h | T | s, | So | L |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | X | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | X | 1 | 1 | 0 |

Note that the incrementer circuit in the sequencer of Fig. 7-8 is not a counter constructed with flip-flops but rather a combinational circuit con­structed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits (see Fig. 4-8). The output carry from one stage must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

7-1. What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all mi­croprogrammed computers also microprocessors?

PROBLEMS

7-2. Explain the difference between hardwired control and microprogrammed control. Is it possible to have a hardwired control associated with a control memory?

7-3. Define the following: (a) microoperation; (b) microinstruction; (c) micro­program; (d) microcode.

7-4. The microprogrammed control organization shown in Fig. 7-1 has the fol­lowing propagation delay times. 40 ns to generate the next address, 10 ns to transfer the address into the control address register, 40 ns to access the control memory ROM, 10 ns to transfer the microinstruction into the control data register, and 40 ns to perform the required microoperations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?

7-5. The system shown in Fig. 7-2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The microoperations field has 16 bits.

a. How many bits are there in the branch address field and the select field?

-

of all 0's (making the B field 000). A register can be cleared to 0 with an exclusive-OR operation. This is because = 0.

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as micropro­grammed control. A microprogrammed control unit is shown in Fig. 7-8. The binary control word for the CPU will come from the outputs of the control memory marked "micro-ops."

8-3 Stack Organization

A useful feature that is included in the CPU of most computers is a stack or LIFO last-in, first-out (LIFO) list. A stack is a storage device that stores information

in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The stack pointer register that holds the address for the stack is called a stack pointer (SP) because

its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word ^ that is inserted or deleted.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a com­puter stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word

**Address**

|  |  |  |
| --- | --- | --- |
|  | |  |
|  |
| **FULL EMTY** | |
|  | |
|  |  |  |
| **SP** |  | **C** |
|  |  | **B** |
|  | | **A** |
|  |

**4**

**3**

2

1

0

**j**

***DR***

Figure 8-3 Block diagram of a 64-word stack.

at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because 26 = 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push push operation. The push operation is implemented with the following sequence of

microoperations:

Increment stack pointer Write item on top of the stack

*SP+-SP* + 1 *M[SP]^DR*

If (SP = 0) then (FULL <— 1) Check if stack is full

EMTY 0 Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that M[SP] denotes the memory word spq^ified by the address presently available in SP. The first item stored in the stack is at address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of micro­operations:

*pop*

DR^M[SP]

SP^SP - 1

If (SP = 0) then (EMTY <—1) FULL^O

Read item from the top of stack

Decrement stack pointer

/

Check if stack is empty Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY = 1.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 8-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer

Address

Memory unit

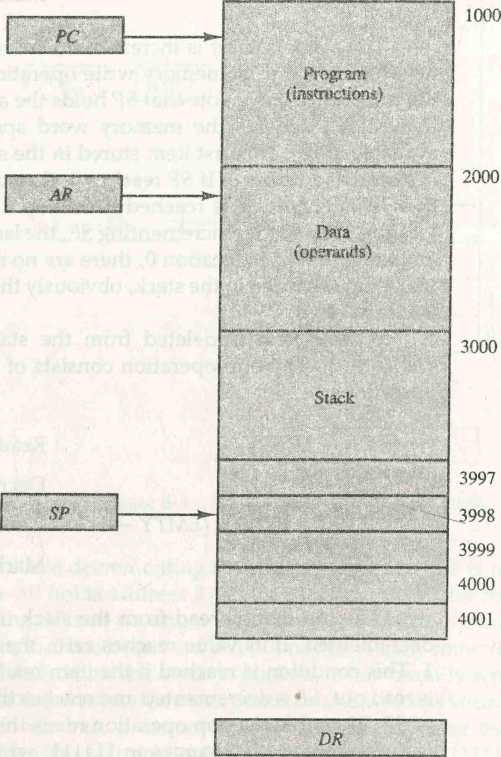


Figure 8-4 Computer memory with program, data, and stack segments.

SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

As shown in Fig. 8-4, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second irem is stored at address 3999, and the last address tha\*t can be used for the stack is 3000. No provisions are available for stack limit checks.

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

SP \*—SP - 1 M[SP)^-DR

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a pop operation as follows:

DR <—M[SP]

SP <— SP + 1

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

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is com­pared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two micro­operations is done first and whether SP is updated by incrementing or decre­menting depends on the organization of the stack. In Fig. 8-4 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in Fig. 8-3. In such a case, SP is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack. In this case the sequence of microoperations must be interchanged.

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

Reverse Polish Notation

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

A\*B + C \* D

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product A\*B, store this product while computing C \* D, and then sum the two products. From this example we see that to evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the next operation to be performed.

The Polish mathematician Lukasiewicz showed that arithmetic expres­sions can be represented in prefix notation. This representation, often referred to as Polish notation, places the operator before the operands. The postfix notation, referred to as reverse Polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

RPN

A + B Infix notation

+AB Prefix or Polish notation

AB + Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation. The expression

A \* B + C \* D

is written in reverse Polish notation as

AB \* CD \* +

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

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

(.*A\*B)CD\* +*

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

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

The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

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

conversion to RPN

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

To evaluate the expression we must first perform the arithmetic inside the parentheses (A + B) and (D + E). Next we must calculate the expression inside the square brackets. The multiplication of C\*(D + E) must be done prior to the addition off since multiplication has precedence over addition. The last operation is the multiplication of the two terms between the parentheses and brackets. The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

AB + DE + *C\*F* + \*

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

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

where (A + B) and (D + E) are each a single number obtained from the sum. The two operands for the next \* are C and (D+E). These two numbers are multiplied and the product added to E. The final \* causes the multiplication of the two terms.

Evaluation of Arithmetic Expressions

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

The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation. The operands are pushed into the stack in the order in which they appear. The initiation of an operation depends on whether we have a calculator or a computer. In a calculator, the operators are entered through the keyboard. In a computer, they must be initiated by instructions that contain an operation field (no address field is required). The following microoperations are executed with the stack when an operation is entered in a calculator or issued by the control in a computer: (1) the two topmost operands in the stack are used for the operation, and (2) the stack is popped and the result of the operation replaces the lower operand. By pushing the operands into the stack continuously and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on top of the stack.

The following numerical example may clarify this procedure. Consider the arithmetic expression

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

In reverse Polish notation, it is expressed as

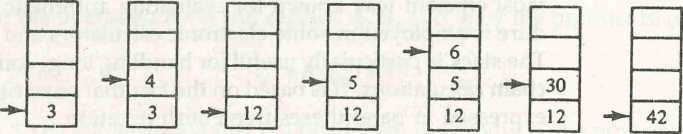
34 \* 56 \* +

Now consider the stack operations shown in Fig. 8-5. Each box represents one stack operation and the arrow always points to the top of the stack. Scanning- the expression from left to right, we encounter two operands. First the number 3 is pushed into the stack, then the number 4. The next symbol is the multi­plication operator \* . This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next \* replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

*stack operations*

Scientific calculators that employ an internal stack require that the user convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



3 4\*56

conversion for the user. Most compilers, irrespective of their CPU organiza­tion, convex all arithmetic expressions into Polish notation anyway because this is the most efficient method for translating arithmetic expressions into machine language instructions. So in essence, a stack-organized CPU may be more efficient in some applications than a CPU without a stack.

8-4 Instruction Formats

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

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

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

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The most common operations available in computer instructions are enumer­ated and discussed in Sec. 8-6. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. The various addressing modes that have been formu­lated for digital computers are presented in Sec. 8-5. In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in memory are specified by their memory address. Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2k registers in the CPU. Thus a CPU with 16 processor

*register address*

registers XO through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

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

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

An example of an accumulator-type organization is the basic computer presented in Chap. 5. All operations are performed with an implied accumu­lator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

ADD X

where X is the address of the operand. The ADD instruction in this case results in the operation AC AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

An example of a general register type of organization was presented in Fig. 7-1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD Rl, RE, R3

to denote the operation Rl<—R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD Rl, RE

would denote the operation XI «-Xl -I- R2. Only register addresses for XI and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruc­tion

MOV Rl, RE

denotes the tr nsfer XI X2 (or X2<—XI, depending on the particular com­puter). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD Rl, X

would specify the operation Rl «-Rl + M[X]. It has two address fields, one for register Rl and the other for the memory address X.

The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

ADD

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Most computers fall into one of the three types of organizations that have just been described. Some computers combine features from more than one organizational structure. For example, the Intel 8080 microprocessor has seven CPU registers, one of which is an accumulator register. As a consequence, the processor has some of the characteristics of a general register type and some of the characteristics of an accumulator type. All arithmetic and logic instruc­tions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instruc­tions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack-organized CPU.

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

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

using zero, one, two, or three address instructions. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to andfrom memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) \* (C + D) is shown below, to­gether with comments that explain the register transfer operation of each instruction.

ADD Rl, A, B R1<-M[A] + M[B]

ADD RE , C f D RE <— M [ C ] + M [ D ]

MUL X, Rl, RE M [ X ] <— Rl \* RE

It is assumed that the computer has two processor registers, Rl and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

Two-Address Instructions

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

MOV

|  |  |
| --- | --- |
| Rl, A | Rl <-M[A] |
| Rl, B | R1R1 + M [ B ] |
| RE, C | RE<-M[C] |
| RE, D | RE <—RE + M[D] |
| Rl, RE | R1«-R1\*RE |
| X, Rl | M [X] <— Rl |

ADD

MOV

ADD

MUL

MOV

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate X = (A + B)\*(C + D) is

|  |  |  |  |
| --- | --- | --- | --- |
| LOAD | A | AC | M [ A] |
| ADD | B | AC <— | A[C] + M[ |
| STORE | T | M[T] | <— AC |
| LOAD | C | AC \*— | M[C] |
| ADD | D | AC <— | AC + M[D] |
| MUL | T | AC •\*— | AC \* M[T] |
| STORE | X | M[X] | <—AC |

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

Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B)\*(C + D) will be written for a stack- organized computer. (TOS stands for top of stack.)

|  |  |  |
| --- | --- | --- |
| PUSH | A | TOS <— A |
| PUSH | B | TOS B |
| ADD |  | TOS <— ( A + B ) |
| PUSH | C | TOS C |
| PUSH | D | TOS^D |
| ADD |  | TOS <— ( C + D) |
| MUL |  | TOS <— ( C + D)\*(A + B) |
| POP | X | M [X] <~TOS |

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

RISC Instructions

The advantages of a reduced instruction set computer (RISC) architecture are explained in Sec. 8-8. The instruction set of a typical RISC processor is restricted

to the use of load and store instructions when communicating between mem­ory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate X = (A + B) \* (C + D).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| LOAD | Rl, | A |  | Rl | <— | M[ A] |
| LOAD | **R2,** | B |  | RE | **<—** | M[B] |
| LOAD | R3, | C |  | R3 |  | M[C] |
| LOAD | **R4** r | D |  | R4 | **<-** | M[D] |
| ADD | Rl, | Rl, | RE | Rl | **\*-** | Rl + RE |
| ADD | R3, | R3, | RE | R3 | <— | R3 + R2 |
| MUL | Rl, | Rl, | R3 | Rl |  | R1\*R3 |
| STORE | X, Rl | |  | M[: | **K]** | <— R1 |

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

8-5 Addressing Modes

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

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

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the

computer. The control unit of a computer is designed to go through an instruc­tion cycle that is divided into three major phases:

***program counter (PC)***

***mode field***

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

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

An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6. The operation code specifies the operation to be per­formed. The mode field is used to locate the operands needed for the opera­tion, There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruc­tion, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accu­mulator" is an implied-mode instruction because the operand in the accumu­lator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Figure 8-6 Instruction format with mode field.

Opcode

Mode

Address

Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode: In this mode the operand is specified in the instruction

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

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within

the CPU. The particular register is selected from a register field in the instruc­tion. A k-bit field can specify any one of 2k registers.

Register Indirect Mode: In this mode the instruction specifies a register in the

CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the pro­grammer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode: This is similar to the register in­

direct mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that auto­matically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-

*effective address*

■

type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in Chap. 5. They are summarized here for reference.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction

gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access mem­ory again to read the effective address. The indirect address mode is also explained in Sec. 5-1 in conjunction with Fig. 5-2.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained fro™ fhe following computation:

effective address = address part of instruction + content of CPU register » “ —-

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result pro­duces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instruc­tions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is

added to the address part of the instruction to obtain the effective address. The

264 CHAPTER EIGHT Central Processing Unit

index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index- type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

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

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register Rl is 400, and the content of an index register XR is 100. AC receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

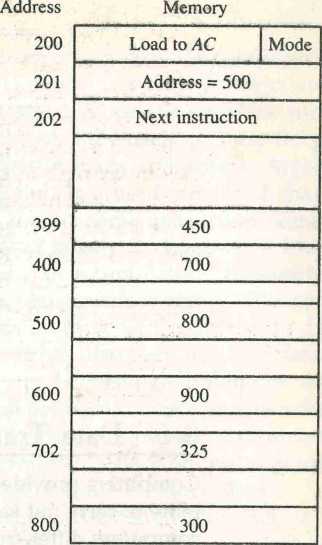
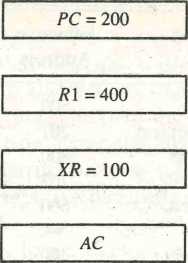


Figure 8-7 Numerical example for addressing modes.

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.) In the indirect mode the effective address is stored in memory at address 500. Therefore,, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900. In the register mode the operand is in PI and 400 is loaded into AC. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of PI and the operand loaded into AC is 700. The autoincrement mode is the same as the register indirect mode except that PI is incremented to 401 after the execution of the instruction. The autodecrement mode decre­ments PI to 399 prior to the execution of the instruction. The operand loaded into AC is now 450. Table 8-4 lists the values of the effective address, and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

|  |  |  |
| --- | --- | --- |
| Addressing  Mode | Effective  Address | Content of AC |
| Direct address | 500 | 800 |
| Immediate operand | 201 | 500 |
| Indirect address | 800 | 300 |
| Relative address | 702 | 325 |
| Indexed address | 600 | 900 |
| Register | — | 400 |
| Register indirect | 400 | 700 |
| Autoincrement | 400 | 700 |
| Autodecrement | 399 | 450 |

**8-6 Data Transfer and Manipulation**

Computers provide an extensive set of instructions to give the user the flexi­bility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are deter­mined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field is differ­ent in different computers, even for the same operation. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction. Nevertheless, there is a set of basic operations that most, if not all, computers include in their instruction repertoire. The basic set of operations available in a typical com­puter is the subject covered in this and the next section.

set of

basic operations

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instruc­tions are those that perform arithmetic, logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table 8-5 gives a list of eight data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The store instruction designates a transfer from a processor register into memory. The move instruc­tion has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The exchange instruction swaps information between two registers or a register and a memory word. The input and output instructions transfer data among proces­sor registers and input or output terminals. The push and pop instructions transfer data between processor registers and a memory stack.

It must be realized that the instructions listed in Table 8-5, as well as in subsequent tables in this section, are often associated with a variety of address­ing modes. Some assembly language conventions modify the mnemonic sym­bol to differentiate between the different addressing modes. For example, the mnemonic for load immediate becomes LDI. Other assembly language conven­tions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the load to accumulator instruction when used with eight different addressing modes.

TABLE 8-5 Typical Data Transfer

Mnemonic

Instructions

Name

Load

LD

ST

MOV

XCH

IN

OUT

PUSH

POP

Store

Move

Exchange

Input

Output

Push

Pop

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Assembly

Mode Convention Register Transfer

|  |  |  |  |
| --- | --- | --- | --- |
| Direct address | LD ADR | AC\*— | M[ADR] |
| Indirect address | LD @ADR | AC\*— | M[M[ADR]] |
| Relative address | LD SADR | AC\*— | M[PC + ADR] |
| Immediate operand | LD #NBR | AC \*— | NBR |
| Index addressing | LD ADR(X) | AC\*- | M[ADR + XR] |
| Register | LD R1 | AC \*— | Rl |
| Register indirect | LD (Rl) | AC\*— | **M[R1]** |
| Autoincrement | LD (Rl) + | AC \*— | M[R\], R\ <— Rl + 1 |

Table 8-6 shows the recommended assembly language convention and the actual transfer accomplished in each case. ADR stands for an address, NBR is a number or operand, X is an index register, XI is a processor register, and AC is the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the program counter PC. The # character precedes the operand in an immediate-mode instruction. An indexed mode instruction is recognized by a register that is placed in parentheses after the symbolic address. The register mode is symbol­ized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in paren­theses. The autoincrement mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The autodecrement mode would use a minus instead. To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the partic­ular computer.

Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instruc­tions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

A list of data manipulation instructions will look very much like the list of microoperations given in Chap. 4. It must be realized, however, that each instruction when executed in the computer must go through the fetch phase

to read its binary code value from memory. The operands must also be brought into processor registers according to the rules of the instruction addressing mode. The last step is to execute the instruction in the processor. This last step is implemented by means of microoperations as explained in Chap. 4 or through an ALU and shifter as shown in Fig. 8-2. Some of the arithmetic instructions need special circuits for their implementation.

Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instruc­tions. The multiplication and division must then be generated by means of software subroutines. The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of nu­merical analysis methods.

A list of typical arithmetic instructions is given in Table 8-7. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all l's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all l's.

The add, subtract, multiply, and divide instructions may be available for data type different types of data. The data type assumed to be in processor registers

during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-preci­sion data. The various data types are presented in Chap. 3.

It is not uncommon to find computers with three or more add instruc-

TABLE 8-7 Typical Arithmetic Instructions

|  |  |
| --- | --- |
| Name | Mnemonic |
| Increment | INC |
| Decrement | DEC |
| Add | ADD |
| Subtract | SUB |
| Multiply | MUL |
| Divide | DIV |
| Add with carry | ADDC |
| Subtract with borrow | SUBB |
| Negate (2’s complement) | NEG |

tions: one for binary integers, one for floating-point operands, and one for decimal operands. The mnemonics for three add instructions that specify different data types are shown below.

ADDI Add two binary integer numbers

ADDF Add two floating-point numbers

ADDD Add two decimal numbers in BCD

Algorithms for integer, floating-point, and decimal arithmetic operations are developed in Chap. 10.

The number of bits in any register is of finite length and therefore the results of arithmetic operations are of finite precision. Some computers provide hardware double-precision operations where the length of each operand is taken to be the length of two memory words. Most small computers provide special instructions to facilitate double-precision arithmetic. A special carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable. By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some typical logical and bit manipulation instructions are listed in Table 8-8. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the l's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corre­sponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. The three logical instructions are usually applied to do just that.

The AND instruction is used to clear a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x bO = 0 and x bl = x dictate that a binary variable ANDed with a 0 produces a 0; but the variable

clear selected bits

shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 8-9 lists four types of shift instructions. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually con- | form with the rales for signed-2's complement numbers. These rales are given in Sec. 4-6. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left in­struction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arith­metic shift-left instiuction when the logical shift-left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire Register to the left.

Some computers have a multiple-field format for the shift instructions. One field contains the operation code and the others specify the type of shift and the number of times that an operand is to be shifted. A possible instruction code format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

Here OP is the operation code field; REG is a register address that specifies the location of the operand; TYPE is a 2-bit field specifying the four different types of shifts; RL is a 1-bit field specifying a shift right or left; and COUNT is a k-bit field specifying up to 2k — 1 shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

TABLE 8-9 Typical Shift Instructions

|  |  |
| --- | --- |
| Name | Mnemonic |
| Logical shift right | SHR |
| Logical shift left | SHL |
| Arithmetic shift right | SHRA |
| Arithmetic shift left | SHLA |
| Rotate right | ROR |
| Rotate left | ROL |
| Rotate right through carry | RORC |
| Rotate left through carry | ROLC |

8-7 Program Control

Instructions are always stored in successive memory locations. When proc­essed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence. After the execution of a data transfer or data manip­ulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation in­structions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program con­trol instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Some typical program control instructions are listed in Table 8-10. The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote! different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address with­out any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken

TABLE 8-10 Typical Program Control Instructions

|  |  |
| --- | --- |
| Name | Mnemonic |
| Branch | BR |
| Jump | JMP |
| Skip | SKP |
| Call | CALL |
| Return | RET |
| Compare (by subtraction) | CMP |
| Test (by ANDing) | TST |

from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

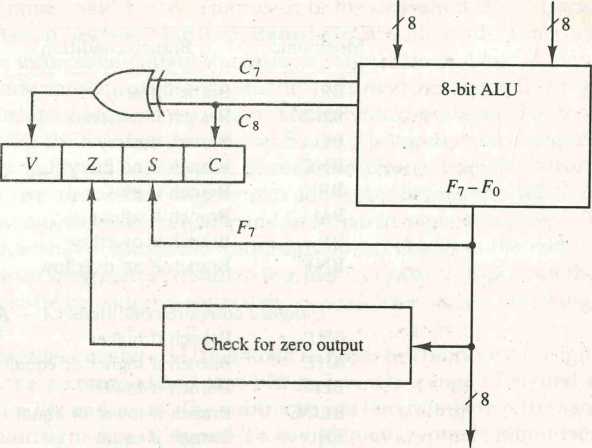
The call and return instructions are used in conjunction with subroutines. Their performance and implementation are discussed later in this section. The compare and test instructions do not change the program sequence directly. They are listed in Table 8-10 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction per­forms a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or chang­ing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure 8-8 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an

A B



Output **F**

Figure 8-8 Status register bits.

overflow when negative numbers are in 2's complement (see Sec. 3-3). For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because x © x = 0, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101x1100, where x is the bit to be checked. The AND operation of A with B = 00010000 produces a result 000x0000. If x = 0, the Z status bit is set, but if x = 1, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction listed in Table 8-10 if the original content of A must be preserved.

Conditional Branch Instructions

Table 8-11 gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is

inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

276 CHAPTER EIGHT Central Processing Unit

TABLE 8-11 Conditional Branch Instructions

|  |  |  |
| --- | --- | --- |
| Mnemonic | Branch condition | Tested condition |
| BZ | Branch if zero | Z - 1 |
| BNZ | Branch if not zero | Z = 0 |
| BC | Branch if carry | C - 1 |
| BNC | Branch if no carry | **C** = 0 |
| BP | Branch if plus | £ = 0 |
| BM | Branch if minus | **S** = 1 |
| BV | Branch if overflow | **V =** 1 |
| BNV | Branch if no overflow | **V = 0** |
| **Unsigned** compare conditions **{A** — **B)** | |  |
| BHI | Branch if higher | **A > B** |
| BHE | Branch if higher or equal | **A > B** |
| BLO | Branch if lower | **A < B** |
| BLOE | Branch if lower or equal | **A < B** |
| BE | Branch if equal | **A — B** |
| BNE | Branch if not equal | **A^B** |
| **Signed** compare conditions **(A** — **B)** | | « |
| BGT | Branch if greater than | **A > B** |
| BGE | Branch if greater or equal | **A > B** |
| BLT | Branch if less than | **A < B** |
| BLE | Branch if less or equal | **A < B** |
| BE | Branch if equal | **A = B** |
| BNE | Branch if not equal | **A^B** |

The zero status bit is used for testing if the result of an ALU operation is equal to zero or not. The cariy bit is used to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. S = 0 denotes a positive sign and S = 1, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's comple­ment representation.

As stated previously, the compare instruction performs a subtraction of two operands, say A ~ B. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of A and B. Some comput­ers provide conditional branch instructions that can be applied right after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers. Table 8-11 gives a list of such conditional branch instructions. Note that we use the words higher and lower to denote the relations between unsigned numbers, and greater and less than for signed numbers. The relative magnitude shown under the tested condition column in the table seems to be the same for unsigned and signed numbers. However, this is not the case since each must be considered separately as explained in the following numerical example.

Consider an 8-bit ALU as shown in Fig. 8-8. The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and -128. The subtraction of two numbers is the same whether they are unsigned or in signed-2's complement representation (see Chap, 3). Let A = 11110000 and B — 00010100. To perform A — B, the ALU takes the 2's complement of B and adds it to A.

numerical example

A: 11110000

B + 1: +11101100

A - B: 11011100 C = 1 S = 1 V = 0 Z = 0

The compare instruction updates the status bits as shown. C = 1 because there is a carry out of the last stage. S = 1 because the leftmost bit is 1. V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

If we assume unsigned numbers, the decimal equivalent of A is 240 and that of B is 20. The subtraction in decimal is 240 - 20 ^ 220. The binary result 11011100 is indeed the equivalent of decimal 220. Since 240 > 20, we have that A > B and A =£ B. These two relations can also be derived from the fact that status bit C is equal to 1 and bit Z is equal to 0. The instructions that will cause a branch after this comparison are BHI (branch if higher), BHE (branch if higher or equal), and BNE (branch if not equal).

If we assume signed numbers, the decimal equivalent of A is -16. This is because the sign of A is negative and 11110000 is the 2's complement of 00010000, which is the decimal equivalent of +16. The decimal equivalent of B is +20. The subtraction in decimal is (-16) - (+20) = -36. The binary result 11011100 (the 2's complement of 00100100) is indeed the equivalent of decimal -36. Since (-16) < (+20) we have that A < B and A + B. These two relations can also be derived from the fact that status bits S = 1 (negative), V = 0 (no overflow), and Z = 0 (not zero). The instructions that will cause a branch after this comparison are BLT (branch if less than), BLE (branch if less or equal), and BNE (branch if not equal).

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical. Each is repeated three times in Table 8-11 for the purpose of clarity' and completeness.

It should be obvious from the example that the relative magnitude of two unsigned numbers can be determined (after a compare instruction) from the values of status bits C and Z (see Prob. 8-26). The relative magnitude of two signed numbers can be determined from the values of S, V, and Z (see Prob. 8-27).

Some computers consider the C bit to be a borrow bit after a subtraction operation A - B. A borrow does not occur if A 2= B, but a bit must be borrowed from the next most significant position if A < B. The condition for a borrow is the complement of the carry', obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after adding the 2's complement of the subtrahend and denote this bit a borrow.

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main pro­gram. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address. A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations: (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and (2) control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

Different computers use a different temporary location for storing the return address. Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from

subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following microoperations:

SP \*—SP — 1 Decrement stack pointer

M[SP] <— PC Push content of PC onto the stack

PC <—effective address Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

PC <—M[SP] Pop stack and transfer to PC SP <—SP + 1 Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

A recursive subroutine is a subroutine that calls itself. If only one register or memory location is used to store the return address, and the recursive subroutine calls itself, it destroys the previous return address. This is undesir­able because vital information is destroyed. This problem can be solved if different storage locations are employed for each use of the subroutine while another lighter-level use is still active. When a stack is used, each return address can be pushed into the stack without destroying any previous values. This solves the problem of recursive subroutines because the next subroutine to exit is always the last subroutine that was called.

Program Interrupt

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence. Program interrupt refers to the transfer of program control from a currently running program to another service pro­gram as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations: (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and (3) an interrupt procedure usually stores all the information

necessary to define the state of the CPU rather than storing only the program counter. These three procedural concepts are clarified further below.

After a program has been interrupted and the service routine been exe­cuted, the CPU must return to exactly the same state that it was when the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU. Typically, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode. Many computers have a resident operating system that controls and supervises all other programs in the computer. When the CPU is executing a program that is part of the operating system, it is said to be in the supervisor or system, mode. Certain instructions are privileged and can be executed in this mode only. The CPU is normally in the user mode when executing user programs. The mode that the CPU is operating at any given time is determined from special status bits in the PSW.

*program status word*

*supervisor mode*

Some computers store only the program counter when responding to an interrupt. The service program must then include instructions to store status and register content before these resources are used. Only a few computers store both program counter and all status and register content in response to an interrupt. Most computers just store the program counter and the PSW. In some cases, there exist two sets of processor registers within the computer, one for each CPU mode. In this way, when the program switches from the user to the supervisor mode (or vice versa) in response to an interrupt, it is not necessary to store the contents of processor registers as each mode uses its own set of registers.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction. The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the program counter. The beginning address of the service routine is deter­mined by the hardware rather than the address field of an instruction. Some computers assign one memory location where interrupts are always trans­ferred. The service routine must then determine what caused the interrupt and proceed to service it. Some computers assign a memory location for each possible interrupt. Sometimes, the hardware interrupt provides its own ad­dress that directs the CPU to the desired service routine. In any case, the CPU

must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a hardware inter­rupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then transferred to PC arid a new PSW is loaded into the status register. The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.

The last instruction in the service program is a retun2 from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution ot a program. They can be classified as:

1. External interrupts
2. internal interrupts
3. ‘•'•ofruare interrupts

External interrupts come from input-output (I/O) devices, from a timing devil, e from a circuit monitoring the power supply, or Irom any other external source Examples that cause externa! interrupts are I/O device requesting transfer ot data I O dec ?<:\*.• finished transier of data, elapsed time of an event, or power failure Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memorv in the few milliseconds before power ceases.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruc­tion execution. The service program that processes the internal interrupt deter­mines the corrective measure to be taken.

The difference between internal and external interrupts is that the inter­nal interrupt is initiated by some exceptional condition caused bv the program itself rather than by an external event. Internal interrupts arc synchronous with

]

the program while external interrupts are asynchronous. If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

282 CHAPTER EIGHT Central Processing Unit

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruc­tion. Software interrupt is a special call instruction that behaves like an inter­rupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the super­visor mode. Certain operations in the computer may be assigned to the super­visor mode only, as for example, a complex input or output transfer procedure.

software interrupt

CISC

RISC

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

8-8 Reduced Instruction Set

Computer (RISC) • j

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes. The trend into computer hardware complexity was influenced by various factors, such as upgrading existing models to provide more customer applications, adding instructions that facilitate the translation from high-level language into machine language programs, and striving to develop machines that move functions from software implementa­tion into hardware implementation. A computer with a large number of in­structions is classified as a complex instruction set computer, abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be exe­cuted much faster within the CPU without having to use memory as often. This type of computer is classified as a reduced instruction set computer or RISC, In

this section we introduce the major characteristics of CISC and RISC architec­tures and then present the instruction set and instruction format of a RISC processor.

CISC Characteristics

The design of an instruction set for a computer must take into consideration not only machine language constructs, but also the requirements imposed on the use of high-level programming languages. The translation from high-level to machine language programs is done by means of a compiler program. One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance. The task of a compiler is to generate a sequence of machine instructions for each high-level language statement. The task is simplified if there are machine instructions that implement the statements directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

Another characteristic of CISC architecture is the incorporation of vari­able-length instruction formats. Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code. If the computer has 32-bit words (four bytes), the first instruction occupies half a word, while the second instruction needs one word in addition to one byte in the next word. Packing variable instruction formats in a fixed-length memory word requires special decoding circuits that count bytes within, words and frame the instructions according to their byte length.

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct addressing. Another memory location may be in­cluded in the instruction to store the sum. This requires three memory refer­ences during execution of the instruction. Although CISC processors have instructions that use only processor registers., the availability of other modes of operations tend to simplify high-level language compilation. However, as more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. In summary, the major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infre­quently
3. A large variety of addressing modes—typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time bv simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relativelv few addressing modes
3. Memory' access limited to load and store instructions . 4. All operations done within the registers of the CPU
4. Fixed-length, easily decoded instruction format
5. Single-cycle instruction execution
6. Hardwired rather than microprograrnroed control

The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, with only simple load and store operations for memory access. Thus each operand is brought into a processor register with a load instruction. All computations are done among the data stored in proces sor registers. Results are transferred to memory by means ot store instructions. This architectural feature simplifies the instruction set and encourages the optimization of register manipulation. The use of only a few addressing modes results from the fact that almost all instructions have simple register address­ing. Other addressing modes may be included, such as immediate operands and relative mode.

By using a relatively simple instruction format, the instruction length can be fixed and aligned on word boundaries. An important aspect ot RISC instruc­tion format is that it is ea^y to decode. Thus the operation code and register fields of the instruction code can be accessed simultaneously bv the control. Bv simplifying the instructions and their format, it is possible to simplify the control iogic. For faster operations, a hardwired control is preferable over a microprogrammed control An example oi hardwired control is presented in Chap. 5 in conjunction with the control unit of the basic, computer. Examples of microprogrammed cento i are presented in Chap. 7

A characteristic of RISC processors is their ability to execute one instruc­tion per clock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelin­ing. A load or store instruction may require two clock cvcles because access to

memory takes more time than register operations. Efficient pipelining, as well as a few other characteristics, are sometimes attributed to RISC, although they may exist in non-RISC architectures as well. Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language pro­grams into machine language programs

A large number of registers is useful for storing intermediate results and for optimizing operand references. The advantage of register storage as op­posed to memory storage is that registers can transfer information to other registers much faster than the transfer of information to and from memory. Thus register-to-memory operations can be minimized by keeping the most frequent accessed operands in registers. Studies that show improved perform­ance for RISC architecture do not differentiate between the effects of the reduced instruction set and the effects of a large register file. For this reason a large number of registers in the processing unit are sometimes associated with RISC processors. The use of overlapped register windows when trans- lerrmg program control after a procedure call is explained below. Instruction pipeline in RISC is presented in Sec. 9-5 after we explain the concept of pipelining.

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming lan­guages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the proce­dure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time- consuming operations. Some computers provide multiple-register banks, and each procedure is allocated its own bank of registers. This eliminates the need for saving and restoring register values Some computers use the memory stack to store the parameters that are needed by the procedure, but this requires a memory access every time the stack is accessed.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of

a new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent proce­dures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlapped register windows is illustrated in Fig. 8-9. The system has a total of 74 registers. Registers RO through R9 are global registers that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results between adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

As an example, suppose that procedure A calls procedure B. Registers R26 through R31 are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers. Procedure B uses local registers R32 through R41 for local variable storage. If procedure B calls pro­cedure C, it will pass the parameters through registers R42 through R47. When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers R26 through R31 and transfers back to the register window of procedure A. Note that registers RIO through R15 are common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

As mentioned previously, the 10 global registers RO through R9 are available to all procedures. Each procedure in Fig. 8-9 has available a total of 32 registers while it is active. This includes 10 global registers, 10 local registers, six low overlapping registers, and six high overlapping registers. Other fixed- size register window schemes are possible, and each may differ in the size of the register window and the size of the total register file. In general, the organization of register windows will have the following relationships:

number of global registers = G number of local registers in each window = L number of registers common to two windows = C number of windows = W

SECTION 8-8 Reduced Instruction Set Computer (RISC) 287

**R**15

**R**10

R13

R64

Proc D

**Common to all procedures**

Global

registers

Common to **D** and **A**

Local to **D**

**R** 63

/?58

/?57

/?48

Proc C

Common to **C** and **D**

Local to **C**

R41

R42

R41

R32

Proc **B**

Common to **B** and **C**

Local to **B**

R31

R26

R25

**R**16

**R**15

R10

Proc **A**

Common to **A** and **B**

Local to **A**

Common to **A** and **D**

Figure 8-9 Overlapped register windows.

SECTION 11-2 Inpur-Output Interface 385

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

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity byte as a single unit called a byte. Therefore, ASCII characters most often are stored

one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font. When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded chatacter.

11-2 Input—Output Interface

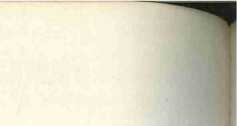
Input-output interface provides a method for transferring information be­tween internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize interface all input and output transfers. These components are called interface units

because they interface between the processor bus and the peripheral device.

386 CHAPTER ELEVEN Input-Output Organization

In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

I/O Bus and Interface Modules

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

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface

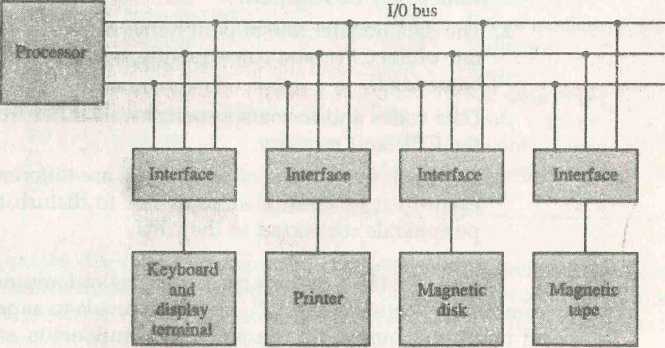


Figure 11-1 Connection of I/O bus to input-output devices.

**Data**

Address

**Control**

r-

HO command

control command

***status***

output data

input data

SECTION 11-2 Input-Output interface 387

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

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

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

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

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

**I/O versus Memory Bus**

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

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

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

Isolated versus Memory-Mapped I/O

*IOP*

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

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

*isolated I/O*

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

*memory-mapped*

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

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

Example of I/O Interface

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

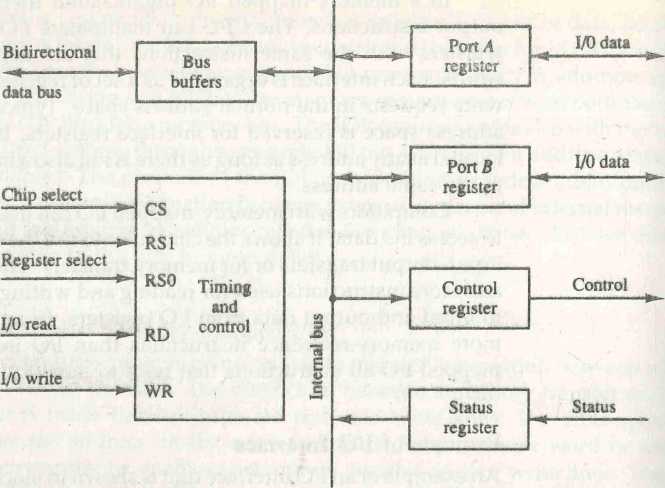
The I/O data to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **CS** | **RSI** | **RSO** | **Register selected** |
| **0** | **X** | **X** | **None: data bus in high-impedance** |
| **1** | **0** | **0** | **Port A register** |
| **1** | **0** | **1** | **Port B register** |
| **1** | **1** | **0** | **Control register** |
| **1** | **1** | **1** | **Status register** |

Figure 11\*2 Example of I/O interface unit.

the forward direction. The bits m the status register are used for status condi­tions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port A has received a new data item from the I/O device. Another bit in the status register may indicate that a parity error has occurred during the transfer.



**—— —■— To CPU To I/O device**

The interface registers communicate with the CPU through the bidirec­tional data bus. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RSI and RSO are usually connected to the two least significant lines of the address bus. These two inputs

■

select one of the four registers in the interface as specified in the table accom­panying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

***strobe***

*handshaking*

*timing diagram*

**11-3** As**ynchronous** Data Transfer

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

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indi­cate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agree­ment between two independent units is referred to as handshaking.

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

Strobe Control

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

T

392 CHAPTER ELEVEN Input-Output Organization

Data bus

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ti** | |  |
| **Source** |  | | **Destination** |
| **unit** | **Strobe** | | **unit** |
|  | **(a) Block diagram** | |  |
| **Data** | **-\* Valid data — ►** | |  |
| **Strobe** | J |  | |

(b) Timing diagram

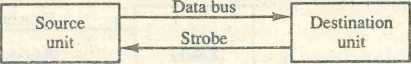
Figure 11-3 Source-initiated strobe for data transfer.

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

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

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

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



**(a) Block diagram**

Data Valid data —

Strobe

(b) Timing diagram

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

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

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

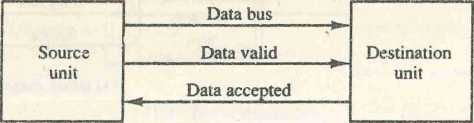
Handshaking

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

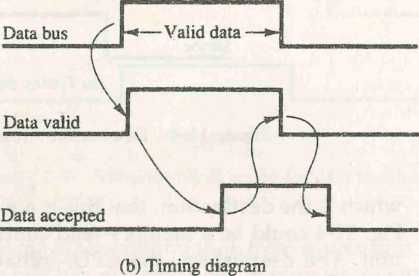
*wire control*

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

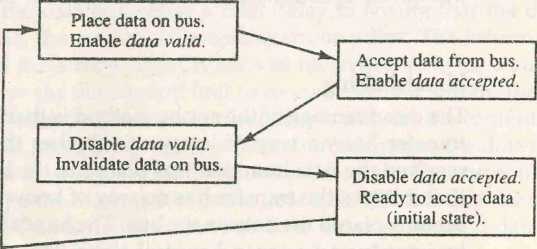
394 CHAPTER ELEVEN Input-Output Organization



(a) Block diagram



**Source unit Destination unit**



**(c) Sequence of events**

Figure 11-5 Source-initiated transfer using handshaking.

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

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

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

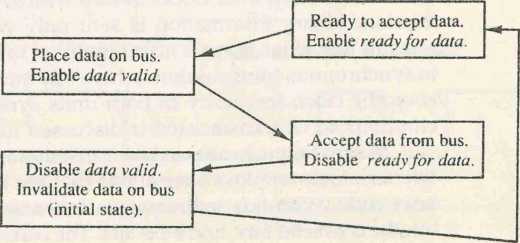
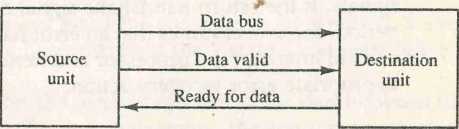
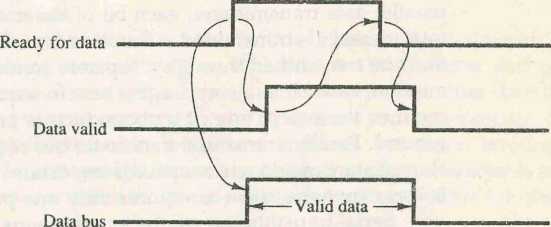


Figure 11-6 Destination-initiated transfer using handshaking.



**(a) Block diagram**



**(b) Timing diagram**

**Source unit**

**Destination unit**

**(c) Sequence of events**

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

The handshaking scheme provides a high degree of flexibility and reliabil­ity because the successful completion of a data transfer relies on active partic­ipation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a prede­termined time. The timeout is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred. The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.

*timeout*

Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through n separate conductor paths. In serial data trans­mission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be synchronous or asynchronous. In synchron­ous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long- distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other. In asynchronous trans­mission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other. Syn­chronous serial transmission is discussed further in Sec. 11-8.

*synchronous*

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

*asynchronous*

bit, the character bits, and stop bits. The convention is that the transmitter rests

\ |

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

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

***start bit***

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Using these rules, the receiver can detect the start bit when the line goes from 1 to 0. A dock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.

***stop bit***

At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

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

Start

bit

■\*»-

-\*■

Stop ^

bits

Figure 11-7 Asynchronous serial transmission.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 0 | 1 | 0 | 1 |
|  |  | - |  |  |  |

Character bits

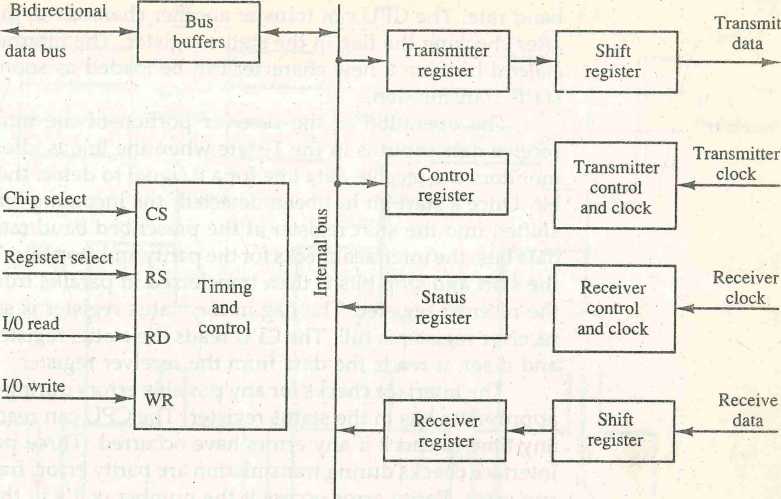
start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten .naracters per second means that each character takes 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

The terminal has a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially along a wire. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver. The transmitter accepts an 8-bit character from the computer and proceeds to send a serial 11-bit message into the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer. Inte­grated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver- transmitter (UART),

Asynchronous Communication Interface

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

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



|  |  |  |  |
| --- | --- | --- | --- |
| **cs** | **RS** | **Operation** | **Register selected** |
| **0** | **X** | **X** | **None: data bus in high-impedance** |
| **1** | **0** | **WR** | **Transmitter register** |
| **1** | **1** | **WR** | **Control register** |
| **1** | **0** | **RD** | **Receiver register** |
| **1** | **1** | **RD** | **Status register** |

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

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

The operation of the transmitter portion of the interface is as follows. The CPU reads the status register and checks the flag to see if the transmitter register is empty. If it is empty, the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit. The character is transferred in parallel from the transmitter register to the shift register and the appropriate number of stop bits are appended into the shift register. The transmitter register is then marked empty. The character can now7 be transmit­ted one bit at a time by shifting the data in the shift register at the specifiedbaud rate. The CPU can transfer another character to the transmitter register after checking the flag in the status register. The interface is said to be double buffered because a new character can be loaded as soon as the previous one starts transmission.

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

*receiver*

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

First'In, First-Out Buffer

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

FIFO

The logic diagram of a typical 4x4 FIFO buffer is shown in Fig. 11-9. It consists of four 4-bit registers RI, I = 1,2,3,4, and a control register with

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

**Data ' input**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| **4-bit** |  |  |  |  |  |  |
|  | **4-bit** |  | **4-bit** |  | **4-bit** |
| **register** |  | **register** |  | **register** |  | **register** |
|  |  |  |  |  |
| \_y\ |  | -A-u |  | \_ A . |  | u A |

R]

R 2

SECTION 11-3 Asynchronous Data Transfer 401 R3 RA

**Clock**

**Clock**

**Clock**

**Clock**

**Data**

**output**

r\

A

**Insert**

►

|  |  |
| --- | --- |
| **s** | **Fx** |
| **R** |  |
| **11** |

|  |  |
| --- | --- |
| **5** | **f2** |
| **R** | **f'2** |

|  |  |
| --- | --- |
| **s** | **L?** |
|  |
| **R** |  |

|  |  |
| --- | --- |
| **s** |  |
|  |
| **R** | **F\** |

**Output**

**ready**

**Input ready**

**Delete —«?—**

**Master clear**

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

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

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

Data are inserted into the buffer provided that the input ready signal is enabled. This occurs when the first control flip-flop Fj is reset, indicating that register Rl is empty. Data are loaded from the input lines by enabling the clock in Rl through the insert control line. The same clock sets Fi, which disables the input ready control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that R2 is empty. The data in Rl are transferred into R2 and F4 is cleared. This enables the input ready line, indicating that the inputs are now available for another data word. If the FIFO is full, Fj remains set and the input ready line stays in the 0 state. Note that the two control lines input ready and insert constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The output ready control line is enabled when the last control flip-flop F4 is set, indicating that there are valid data in the output register R4. The output data from R4 are accepted by a destination unit, which then enables the delete control signal. This resets F4, causing output ready to disable, indicating that the data on the output are no longer valid. Only after the delete signal goes back to 0 can the data from R3 move into R4. If the FIFO is empty, there will be no data in R3 and F4 will remain in the reset state. Note that the two control lines output ready and delete constitute a source-initiated pair of handshake lines.

**11-4** Modes of Transfer

Binary information received from an external device is usually stored in mem­ory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit,'Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

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

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant mon­itoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

programmed I/O

interrupt

DMA

IOP

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory. DMA transfer is discussed in more detail in Sec. 11-6.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in Sec. 11-7.

Example of Programmed I/O

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

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

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

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

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

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

/

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **„ Data bus v** | **Interface** | |  | | **^ I/O bus** |  |
|  | **^— —7**  **Address bus /** |  | **Data register** | |  |  |  |
|  |  |  | |  | | **Data valid** |  |
| **CPU** | **I/O read** |  | |  | |  | **1/0**  **device** |
|  | **I/O write** |  | **Status** | **F** |  | **Data accepted** |
|  |  |  | **register** |  |  | —— —-fe-t |  |

**F = Flag bit**

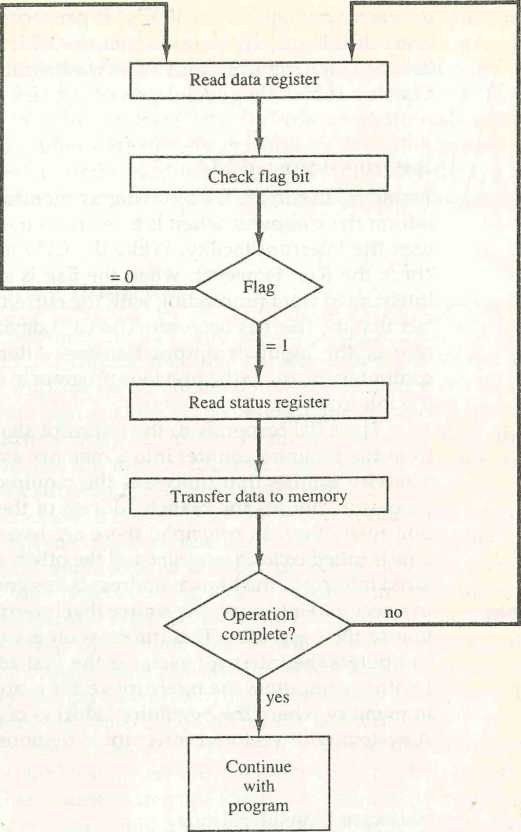


Figure 11-11 Flowchart for CPU program to input data.

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

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 ps. Assume that the input device transfers its

data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 |xs. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

Interrupt-Initiated I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, nonvectored interrupt. In a non vec­tored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch informa­tion to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored. A system with vectored interrupt is demonstrated in Sec. 11-5.

vectored interrupt

Software Considerations

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

I/O routines

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

**11-5** P**riority Interrupt**

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

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which condi­tions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high­speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

*priority interrupt*

Establishing the priority of simultaneous interrupts can be done by soft­ware or hardware. A polling procedure is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and

*polling*

so on. Thus the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer. The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy- chaining method.

Daisy-Chaining Priority

The daisy-chaining method of establishing priority consists of a serial connec- ^ tion of all devices that request an interrupt. The device with the highest priority

is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Fig. 11-12. The interrupt request line is common to all devices and forms a wired logic connec­tion. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative- logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) vector address (VAD) into the data bus for the CPU to use.during the interrupt cycle.

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device

Processor data bus

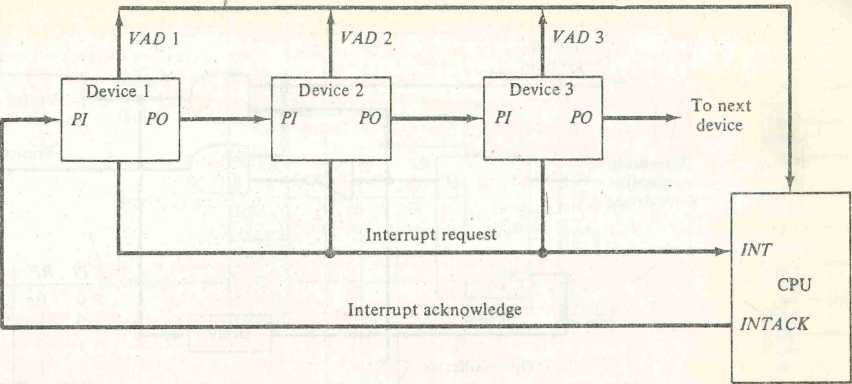


Figure 11-12 Daisy-chain priority interrupt.

by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting art interrupt, and this device places its VAD on the data bus. The daisy\;ham arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 11-13 shows the internal logic that must be included within each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when Pi = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

Parallel Priority Interrupt

> The parallel priority interrupt method uses a register whose bits are set sepa­rately by the interrupt signal from each device. Priority' is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable

VAD

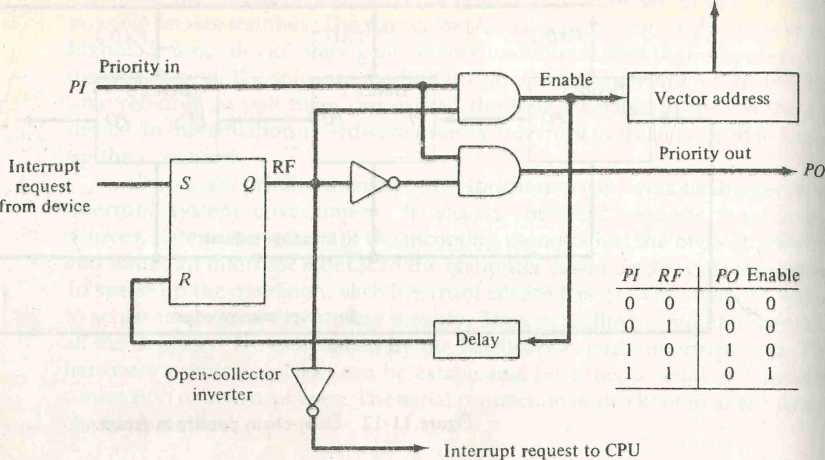


Figure 11-13

One stage of the daisy-chain priority arrangement.

lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

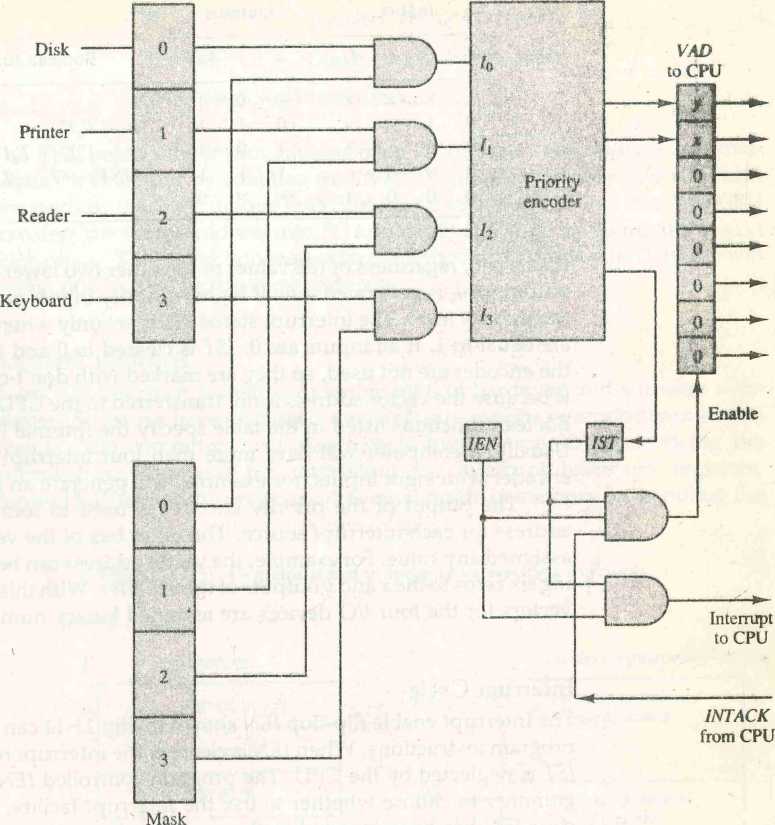
The priority logic for a system of four interrupt sources is shown in Fig. 11-14. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

*priority logic*

Another output from the encoder sets an interrupt status flip-flop 1ST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of 1ST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt con­troller and the CPU.

interrupt

register



register

Figure 11-14 Priority interrupt hardware.

Priority Encoder

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 11-2. The x's in the table designate don't-care conditions. Input I0 has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output xy — 00. /i has the next priority level. The output is 01 if L = 1 provided

TABLE 11-2 Priority Encoder Truth Table

Inputs Outputs

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| h | h | h | h | **X** | y | 1ST | Boolean functions |
| 1 | **X** | **X** | **X** | 0 | 0 | 1 |  |
| 0 | 1 | **X** | **X** | 0 | l | 1 | \* - I'ol'i |
| 0 | 0 | 1 | **X** | 1 | 0 | 1 | y = /'/, + r0r2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | (1ST) = I0 +I, + J2 + 13 |
| 0 | 0 | 0 | 0 | **X** | **X** | 0 |  |

that I0 ~ 0, regardless of the values of the other two lower-priority inputs. The output for I2 is generated only if higher-priority inputs are 0, and so on down the priority level. The interrupt status 1ST is set only when one or more inputs are equal to L If all inputs are 0, 1ST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when 1ST = 0. The Boolean functions listed in the table specify the internal logic of the encoder. Usually, a computer will have more than four interrupt sources. A priority encoder with eight inputs, for example, will generate an output of three bits.

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

Interrupt **Cycle**

The interrupt enable flip-flop IEN shown in Fig. 11-14 can be set or cleared by program instructions. When IEN is cleared, the interrupt request coming from 1ST is neglected by the CPU. The program-controlled IEN bit allows the pro­grammer to choose whether to use the interrupt facility. If an instruction to dear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set IEN indicates that the interrupt facility will be used while the current program is running. Most computers include internal hardware that clears IEN to 0 every time an inter­rupt is acknowledged by the processor.

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

SP \*— SP - 1 Decrement stack pointer

M[SP]«- PC Push PC into stack

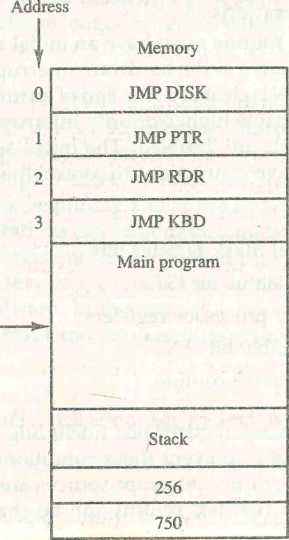
INTACK \*- 1 Enable interrupt acknowledge PC <— VAD Transfer vector address to PC IEN <— 0 Disable further interrupts

Go to fetch next instruction

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

Software Routines

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

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

**I/O service programs**

**DISK**

**PTR \*-**

**RDR ►**

**KBD**

**256 ►**

**Program to service  
magnetic disk**

**Program to service  
line printer**

**Program to service  
character reader**

**Program to service  
keyboard**

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

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

*service program*

Initial and Final Operations

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

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

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

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

r

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

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

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

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

**11-6** Direct Memory **Access (DMA)**

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly

would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

bus request

bus grant

burst transfer

cycle stealing

|  |  |
| --- | --- |
| **DBUS** | **—\*■—Address bus** |
| **BR** |  |
| **ABUS** | **——Data bus** |
| **CPU** |  |
| **RD** | **Read** |
| **BG** |  |
| **WR** | **+~- Write** |

Figure 11-16 CPU bus signals for DMA transfer.

**High-impedance (disable) when BG is enabled**

**Bus grant**

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 11-16 shows two control signals in the CPU that facilitate the DMA transfer. The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic signif­icance (see Sec. 4-3). The CPU activates the bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA burst transfer, a block sequence consisting of a number of memory words is trans­ferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and TO device. In addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines

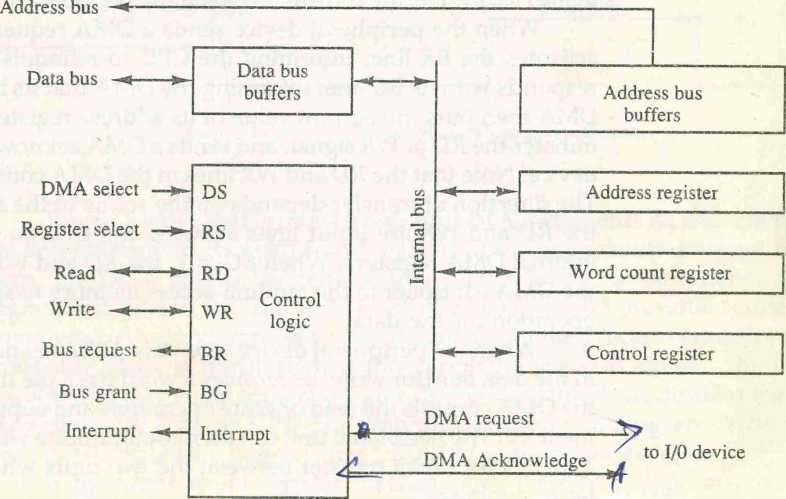
Bus request

are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 11-17 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by speci­fying an address in the address bus and activating the RD or WR control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

Figure 11-17 Block diagram of DMA controller.



The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consist­ing of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Fig. 11-18. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control com­mand, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG - 1, the RD and WR are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory7 through the data bus for direct transfer between the two units while the CPU is momen­tarily disabled.

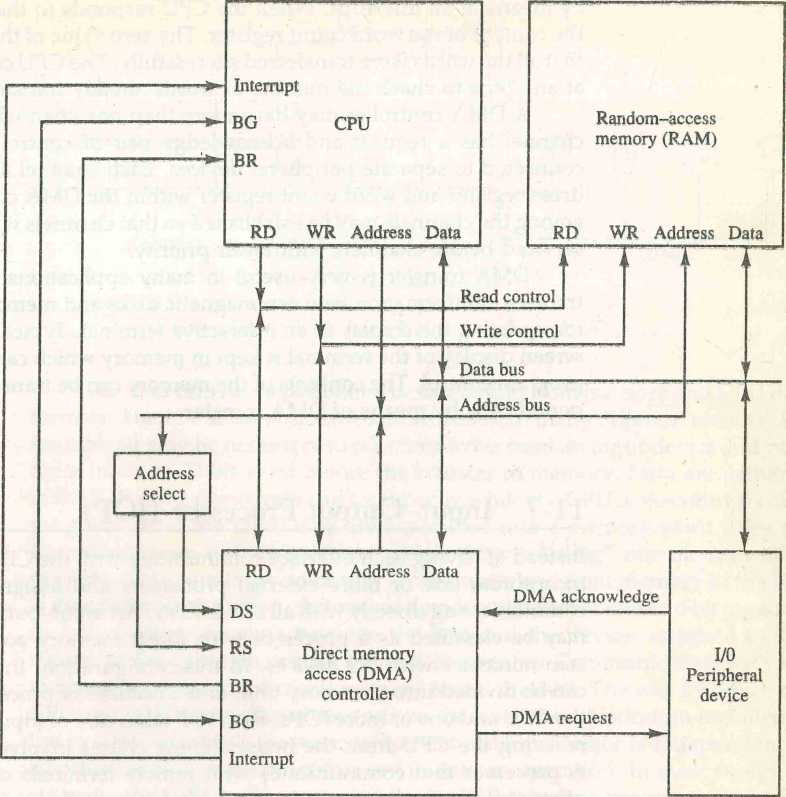


Figure 11-18 DMA transfer in a computer system.

For each word that is transferred, the DMA increments its address regis­ter and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination

by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than one channel. In this case, each channel has a request and acknowledge pair of control signals which are connected to separate peripheral devices. Each channel also has its own ad­dress register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

**11-7** Input-Output Processor (iOP)

Instead of having each interface communicate with the CPU, a computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An input-output processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit, and a number of processors comprised of the CPU and one or more lOPs. Each IOP takes care of input and output tasks, relieving the CPU from the housekeeping chores involved in I/O transfers. A processor that communicates with remote terminals over telephone and other communication media in a serial fashion is called a data communication processor (DCP).

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

*1JO processing*

The block diagram of a computer with two processors is shown in Fig. 11-19. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP pro­vides a path for transfer of data between various peripheral devices and the memory unit. The CPU is usually assigned the task of initiating the I/O program. From then on the IOP operates independent of the CPU and contin­ues to transfer data from external devices and memory.