CHAPTER 11

Computer Design

11.1 Introduction

This chapter presents a small general-purpose digital computer starting from its functional specifications and culminating in its design. Although the computer is small, it is far from useless. Its scope is quite limited when compared with commercial electronic data-processing systems, yet it encompasses enough functional capabilities to demonstrate the design process. It is suitable for construction in the laboratory with ICs, and the finished product can be a useful system capable of processing digital data.*

The computer consists of a central processor unit, a memory unit, and a teletypewriter input-output unit. The logic design of the central processor unit will be derived here. The other two units are assumed to be available as finished products with known external characteristics.

The hardware design of a digital computer may be divided into three interrelated phases: system design, logic design, and circuit design. System design is concerned with the specifications and general properties of the system. This task includes the establishment of design objective and design philosophy, the formulation of computer instructions, and the investigation of its economic feasibility. The specifications of the computer structure are translated by the logic designer to provide the hardware implementation of the system. The circuit design specifies the components for the various logic circuits, memory circuits, electromechanical equipment, and power supplies. The computer hardware design is greatly influenced by the software system, which is normally developed concurrently and which constitutes an integral part of the total computer system.

The design of a digital computer is a complicated task. One cannot expect to cover all aspects of the design in one chapter. Here we are concerned with the system and logic design of a small digital computer whose specifications are formulated somewhat arbitrarily in order to establish a minimum configuration for a very small, yet practical machine. The procedure outlined in this chapter can be useful in the logic design of more complicated systems.

The design process is divided into six phases:

1. The decomposition of the digital computer into registers which specify the general configuration of the system.

*The instructions for the computer are a subset of the instructions in the PDP -8 computer.

2. The specification of computer instructions.
3. The formulation of a timing and control network.
4. The listing of the register-transfer operations needed to execute all computer instructions.
5. The design of the processor section.
6. The design of the control section.

The design process is carried out by means of tabular listings that summarize the specifications and register-transfer operations in compact form. The processor section is defined by means of a block diagram consisting of registers and multiplexers. It is assumed that the reader has sufficient information to replace the blocks in the diagram with MSI circuits. The control section is designed by each of the three methods outlined in Chapter 10.

11.2 System Configuration

The configuration of the computer is shown in Fig. 11-1. Each block represents a register, except for the memory unit, the master-clock generator, and the control logic. This configuration is assumed to satisfy the final system structure. In a practical situation, the designer starts with a tentative system configuration and constantly modifies it during the design process. The name of each register is written inside the block, together with a symbolic designation in parentheses.

The master-clock generator is a common clock-pulse source, usually an oscillator, which generates a periodic train of pulses. These pulses are fanned out by means of amplifiers and

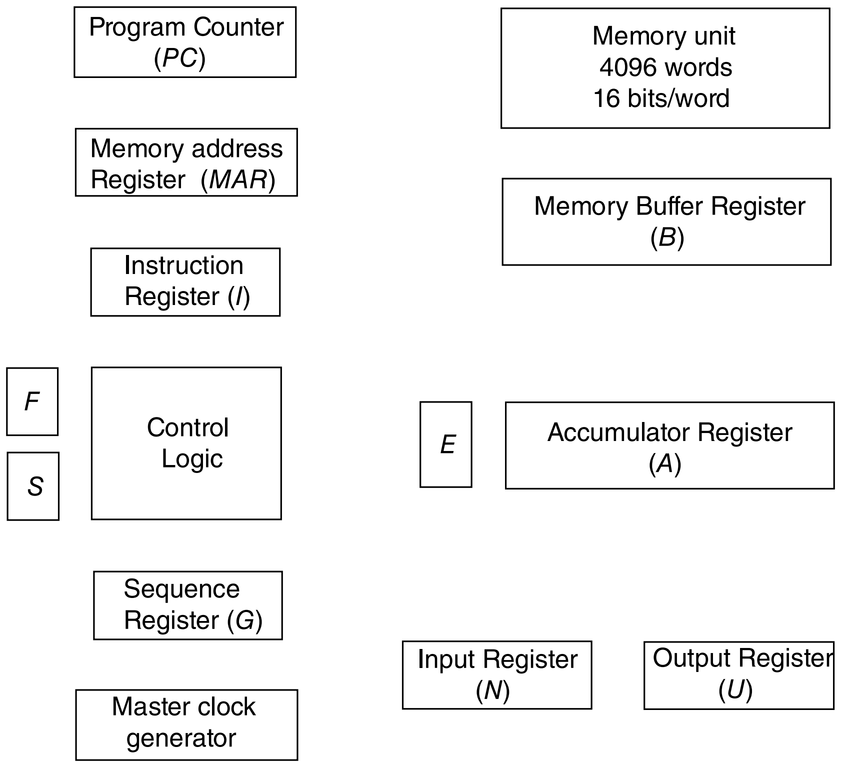


Figure 11.1 Block diagram of digital computer

Table 11-1 List of registers tor computer

Symbolic designation	Name	Number of bits	Function
A	Accumulator register	16	Processor register
B	Memory buffer register	16	Holds contents of memory word
PC	Program counter	12	Holds address of next instruction
MAR	Memory address register	12	Holds address of memory word
I	Instruction register	4	Holds current operation-code
E	Extension flip-flop	1	Accumulator extension
F	Fetch flip-flop	1	Controls fetch and execute cycles
S	Start-stop flip-flop	1	Starts and stops computer
G	Sequence register	2	Provides timing signals
N	Input register	9	Holds information from input device
U	Output register	9	Holds information for output device

distributed over the entire system. Each pulse must reach every flip-flop and register at the same time. Phasing delays may be needed intermittently so that the difference in transmission delays is uniform throughout. The frequency of the pulses is a function of the speed with which the system operates. We shall assume a frequency of 1 megahertz, which gives one pulse every microsecond. This pulse frequency is chosen for the sake of having a round number and to avoid problems of circuit propagation delays.

The memory unit has a capacity of 4096 words of 16 bits each. This capacity is large enough for meaningful processing. A smaller size may be used if the computer is to be constructed in the laboratory under economic restrictions. Twelve bits of an instruction are needed to specify the address of an operand, which leaves four bits for the operation part of the instruction. The access time of the memory is assumed to be less than 1 microsecond so that a word can be read or written during the interval between two clock pulses.

The part of the digital computer to be designed is decomposed into register subunits. The following paragraphs explain why each register is needed and what function it performs. A list of the registers and a brief description of their functions is presented in Table 11-1. Registers that hold memory words are 16 bits long. Those that hold an address are 12 bits long. Other registers have different numbers of bits, depending on their function.

11.2.1 Memory Address and Memory Buffer Registers

The memory address register, MAR, is used to address specific memory locations. MAR is loaded from PC when an instruction is to be read from memory, and from the 12 least significant bits of the B register when an operand is to be read from memory. Memory buffer register B holds the word read from or written into memory. The operation part of an instruction word placed in B is transferred into the I register, and the address part is left in the B register for transfer to MAR. An operand word placed in the B register is accessible for operation with the A register. A word to be stored in memory must be loaded into the B register before a write operation is initiated.

11.2.2 Program Counter

Program counter *PC* holds the address of the next instruction to be read from memory. This register goes through a step-by-step counting sequence and causes the computer to read successive instructions previously stored in memory. When the program calls for a transfer to another location or for skipping the next instruction in sequence, the *PC* is modified accordingly, causing the program to continue from a memory location out of the counting sequence. To read an instruction, the contents of *PC* are transferred to *MAR* and a read operation is initiated. The program counter is always incremented by 1 while a memory write operation reads the present instruction. Therefore, the address of the next instruction, one higher than the one presently being executed in the processor, is always available in *PC*.

11.2.3 Accumulator Register

Accumulator register *A* is a processor register that operates on data previously stored in memory. This register is used to execute most instructions and for accepting data from the input device or transferring data to the output device. The *A* register, together with the *B* register, makes up the bulk of the processor unit for the computer. Although most data processing systems include more registers for the processor unit, we have chosen to include only one accumulator here in order not to complicate the design. With a single accumulator as the arithmetic element, it is possible to implement only the add operation. Other arithmetic operations such as subtraction, multiplication, and division must be implemented with a sequence of instructions that form a subroutine.

11.2.4 Instruction Register

Instruction register *I* holds the operation-code bits of the current instruction. This register has only four bits since the operation-code of instructions is four bits long. The operation-code bits are transferred to the *I* register from the *B* register, while the address part of the instruction is left in *B*. The operation-code part must be taken out of the *B* register because an operand read from memory into the *B* register will destroy the previously held instruction. The operation part of the instruction is needed by the control to determine what is to be done to the operand just read.

11.2.5 Sequence Register

Sequence register *G* is a counter that produces the timing signals for the computer. The *G* register is decoded to supply four timing variables for the control unit. The timing variables, together with other control variables, produce the control functions that initiate all the microoperations for the computer.

11.2.6 *E*, *F*, and *S* Flip-flops

Each of these flip-flops is considered a one-bit register. The *E* flip-flop is an extension of the *A* register. It is used during shifting operations, receives the end carry during addition, and otherwise is a useful flip-flop that can simplify the data processing capabilities of the computer. The *F* flip-flop distinguishes between the fetch and execute cycles. When *F* is 0, the word read from memory is treated as an instruction. When *F* is 1, the word is treated as an operand. *S* is a start-stop flip-flop that can be cleared by program control and manipulated manually. When *S* is 1, the computer runs according to a sequence determined by the program stored in memory. When S is 0, the computer stops its operation.

11.2.7 Input and Output Registers

The input-output (I/0) device is not shown in the block diagram of Fig. 11-1. It is assumed to be a teletypewriter unit with a keyboard and a printer. The teletypewriter sends and receives serial information. Each quantity of information has 8 bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register. The serial information for the printer is stored in the output register. These two registers communicate with the teletypewriter serially and with the accumulator register in parallel.

Input register N consists of nine bits. Bits 1 through 8 hold alphanumeric input information; bit 9 is a control bit called an input *flag.* The flag bit is set when a new character is available from the input device and cleared when the character is accepted by the computer. The flag bit is needed to synchronize the slow rate by which the input device operates compared to the high-speed circuits in the computer. The process of information transfer is as follows. Initially, the flag bit in N_9 is cleared. When a key is struck on the keyboard, an 8-bit code is shifted into the input register ($N_1 - N_8$). As soon as the shift operation is completed, the flag bit in N_9 is set to 1. The computer checks the flag bit; if it is 1, the character code from the N register is transferred in parallel into the A register and the flag bit is cleared. Once the flag is cleared, a new character can be shifted into the N register by striking another key.

Output register U works in a similar fashion, but the direction of information flow is reversed. Initially, the output flag in U_9 is set to 1. The computer checks the flag bit; if it is I, a character code from the A register is transferred in parallel to the output register ($U_1 - U_8$) and the flag bit U_9 is cleared to 0. The output device accepts the coded information and prints the corresponding character; when the operation is completed, it sets the flag bit to 1. The computer does not load a new character into the output register when the flag is 0, because this condition indicates that the output device is in the process of printing the previous character.

11.3 Computer Instructions

The number of instructions available in a computer and their efficiency in solving the problem at hand are a good indication of how well the system designer foresaw the intended application of the machine. Medium- to large-scale computing systems may have hundreds of instructions, while most small computers limit the list to less than 100. The instructions must be chosen carefully to supply sufficient capabilities to the system for solving a wide range of data processing problems. The minimum requirements of such a list should include a capability for storing and loading words from memory, a sufficient set of arithmetic and logic operations, some address-modification capabilities, unconditional branching and branching under test conditions, register manipulation capabilities, and I/O instructions. The instruction list chosen for our computer is believed to be close to the absolute minimum required for a restricted but practical data processor.

The formulation of a set of instructions for the computer goes hand in hand with the formulation of the formats for data and instruction words. A memory word consists of 16 bits. A word may represent either a unit of data or an instruction. The formats of data words are shown in Fig. 11-2. Data for arithmetic operations are represented by a 15-bit binary number, with the sign in

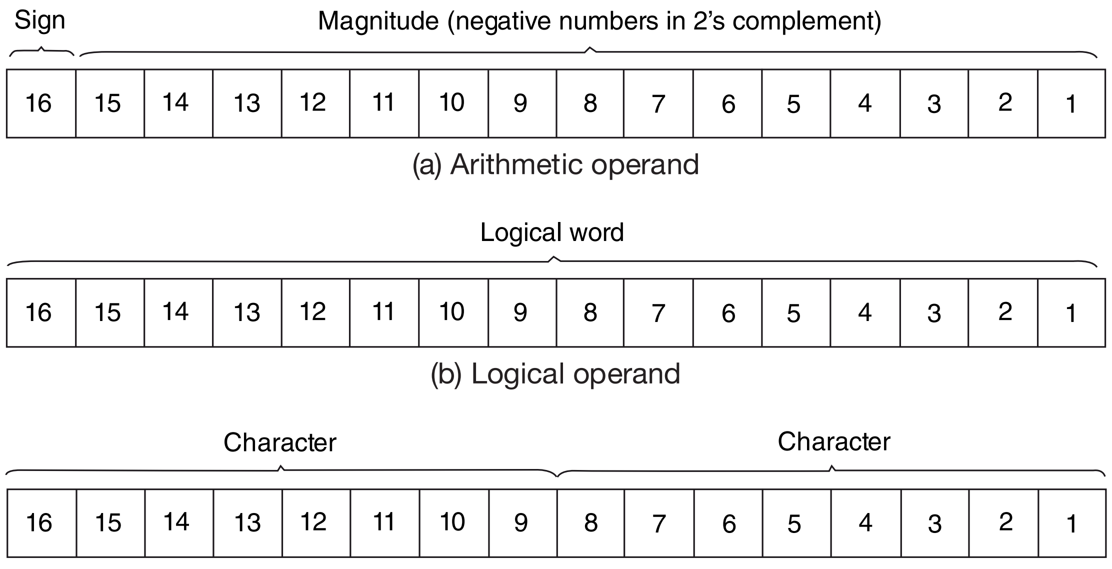
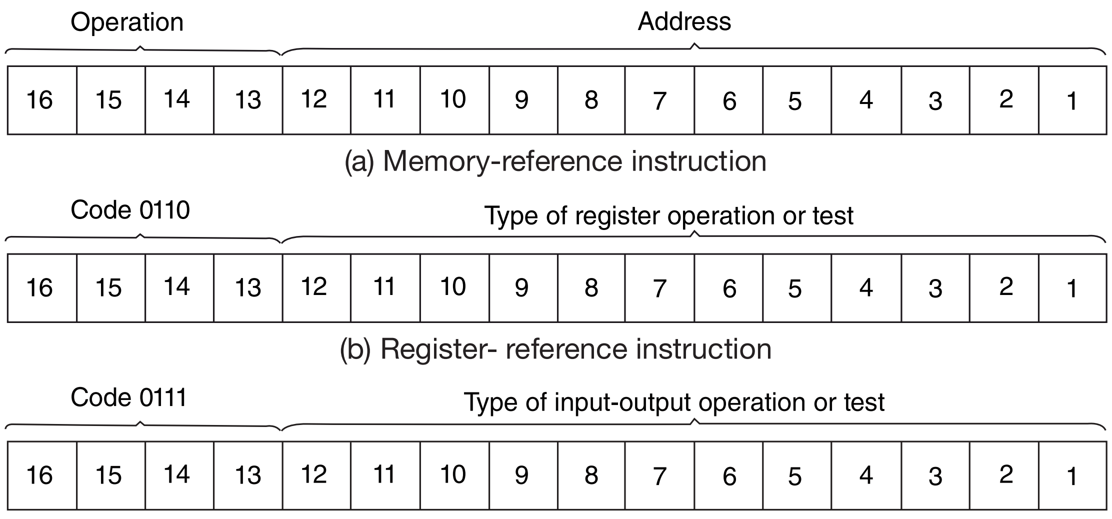


Figure 11.2 Data formats

the 16th bit position. Negative numbers are assumed to be in their 2's-complement equivalent. Logical operations are performed on individual bits of the word, with bit 16 treated as any other bit. When the computer communicates with the I/O device, the information transferred is considered to be 8-bit alphanumeric characters. Two such characters can be accommodated in one computer word.

The formats of instruction words are shown in Fig. 11-3. The operation part of the instruction contains four bits; the meaning of the remaining 12 bits depends on the operation-code encountered. A *memory-reference* instruction uses the remaining 12 bits to specify an address. A *register-reference* instruction implies an operation on, or a test of, the *A* or *E* register. An operand



Operation
Address
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
(a) Memory-reference instruction
Code 0110
Type of register operation or test
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
(b) Register- reference instruction
Code 0111
Type of input-output operation or test
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
(c) Input/output instruction

Figure 11.3 Instruction formats

Table 11-2 Memory-reference instructions

Symbol	Hexa-decimal code	Description	Function
AND	0 m*	AND to A	$A \leftarrow A \wedge M$*
ADD	1 m	Add to A	$A \leftarrow A + M, E \leftarrow$ Carry
STO	2 m	Store in A	$M \leftarrow A$
ISZ	3 m	Increment and skip if zero	$M \leftarrow M + 1$, if $(M + 1 = 0)$ then $(PC \leftarrow PC + 1)$
BSB	4 m	Branch to subroutine	$M \leftarrow PC + 5000, PC \leftarrow m + 1$
BUN	5 m	Branch unconditionally	$PC \leftarrow m$

* m is the address part of the instruction. M is the memory word addressed by m.

from memory is not needed; therefore, the 12 least significant bits are used to specify the operation or test to be executed. A register-reference instruction is recognized by the code 0110 in the operation part. Similarly, an *input-output* instruction does not need a reference to memory and is recognized by the operation code 0111. The remaining 12 bits are used to specify the particular device and the type of operation or test performed.

Only four bits of the instruction are available for the operation code. It would seem, then, that the computer is restricted to a maximum of 16 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 16. In fact, the total number of instructions chosen for the computer is 22.

Of the 16 distinct operations that can be formulated with four bits, only eight have been utilized by the computer because the leftmost bit of all instructions (bit 16) is always a 0. This leaves open the possibility of adding new instructions and extending the computer capabilities if desired.

The six memory-reference instructions for the computer are listed in Table 11-2. The symbolic designation is a three-letter word and represents an abbreviation intended for use by programmers and users when writing symbolic programs for the computer. The hexadecimal code listed is an equivalent hexadecimal number of the binary code adopted for the operation-code. A memory-reference instruction uses one hexadecimal digit (4 bits) for the operation-code; the remaining three hexadecimal digits (12 bits) of the instruction represent an address designated by die letter m. Each instruction has a brief word description and is specified more precisely in the function column with a-macrooperation statement. A further clarification of each instruction is given below, together with an explanation of its use.

11.3.1 AND to *A*

This is a logic operation that performs the AND operation on corresponding pairs of bits in A, with the memory word M specified by the address part of the instruction. The result of the operation is left in register A, replacing its previous contents. Any computer must have a basic set of logic operations for manipulating nonnumerical data. The most common logic operations found in computer instructions are AND, OR, exclusive-OR, and complement. Here we use only the AND and complement. The latter is included as a register-reference instruction. These two logic operations constitute a minimal set from which all other logic operations can be derived, because together the AND and the complement perform a NAND operation. In Section 4-7 we saw that this is a universal operation from which any other logic operation can be obtained.

11.3.2 ADD to *A*

This instruction adds the contents of the memory word *M*, specified by the address part of the instruction, to the present contents of register *A*. The addition is done assuming that negative numbers are in their 2's-complement form. This requires that the sign bit be added in the same way as all other bits are added. The end-carry out of the sign-bit position is transferred to the *E* flip-flop. This instruction, together with the register-reference instructions, is sufficient for writing programs to implement all other arithmetic operations. Subtraction is achieved by complementing and incrementing the subtrahend. Multiplication is achieved by adding and shifting. The increment and shift are register-reference instructions.

The ADD instruction must be used for loading a word from memory into the *A* register. This is done by first clearing the *A* register with the register-reference instruction CLA (defined in Table 11-3). The required word is then loaded from memory by adding it to the cleared *A* register.

11.3.3 STORE in *A*

This instruction stores the contents of the *A* register into the memory word specified by the instruction address. The first three memory-reference instructions are used to manipulate data between memory words and the *A* register. The next three instructions are control instructions that cause a change in normal program sequence.

11.3.4 Increment and Skip if Zero (ISZ)

The increment-and-skip instruction is useful for address modification and for counting the number of times a program loop is executed. A negative number previously stored in memory at address *m* is read by the ISZ instruction. This number is incremented by 1 and stored back into memory. If, after it is incremented, the number reaches 0, the next instruction is skipped. Thus, at the end of a program loop, one inserts an ISZ instruction followed by a branch unconditionally (BUN) instruction to the beginning of the program loop. If the stored number does not reach 0, the program returns to execute the loop again. If it reaches 0, the next instruction (BUN) is skipped and the program continues to execute instructions after the program loop.

11.3.5 Branch Unconditionally (BUN)

This instruction transfers control unconditionally to the instruction at the location specified by the address part *m*. Remember that the program counter holds the address of the next instruction to be read and executed. Normally, the *PC* is incremented to give the address of the next instruction in sequence. The programmer has the prerogative of specifying any other instruction out of sequence by using the BUN instruction. This instruction tells the computer to take the address part *m* and transfer it into *PC*. The address of the next instruction to be executed is now in *PC* and is the one which was previously the address part of the BUN instruction.

The BUN instruction is listed with the memory-reference instructions because it needs an address part *m*. However, it does not need a reference to memory to access a memory word (designated by the symbol *M*), as is required by the other memory-reference instructions.

11.3.6 Branch to Subroutine (BSB)

This instruction is useful for branching to a subroutine portion of a program. When executed, the instruction stores the address of the next instruction in sequence which is presently held in *PC* (called the *return address)* into the memory word specified by the address part of the instruction. It also stores the operation code of BUN (hexadecimal 5) in the same memory location. The contents of the address part *m* plus 1 are transferred into *PC* to start executing the subroutine program at this location. After the subroutine is executed, control is transferred back to the calling program by means of a BUN instruction placed at the end of the subroutine.

The process of branching to a subroutine and the return to the calling program is demonstrated in Fig. 11-4 by means of a specific numerical example. The calling program is now in location 32. The subroutine program starts at location 65. The BSB instruction causes a transfer to the subroutine, and the last instruction in the subroutine causes a branch back to location 33 in the calling program. The numerical example in Fig. 11-4 shows a BSB instruction in location 32 with an address part *m* equal to binary 64. While this instruction is being executed, *PC* holds the address of the next instruction in sequence, which is 33. The BSB instruction performs the macrooperation (see Table 11-2):

$$M \leftarrow PC + 5000,\ PC \leftarrow m + 1$$

The contents of *PC* plus hexadecimal 5000 (code for BUN) are transferred into location 64. This transfer produces an instruction BUN 33. The address part of the instruction is incremented and

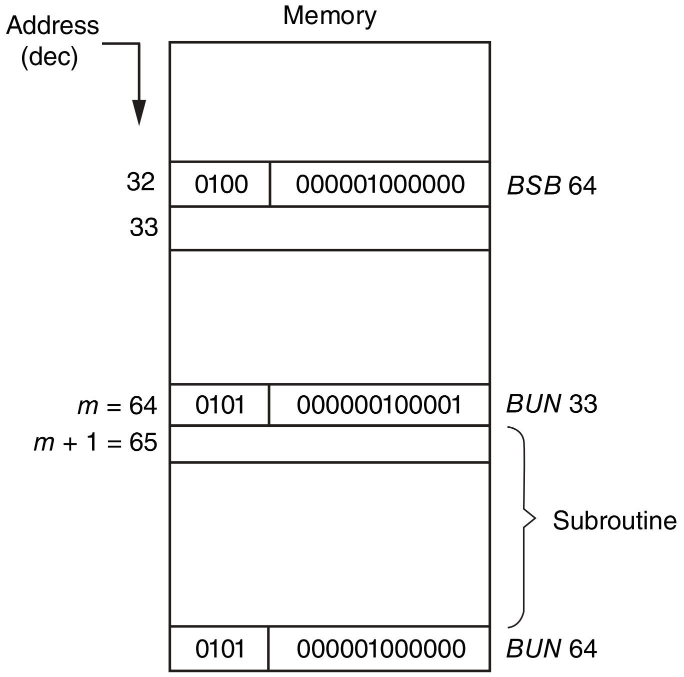


Figure 11.4 Demonstration of branch-to-subroutine instruction

placed in *PC*. *PC* now holds the binary equivalent of 65, so the computer starts executing the subroutine at this location. The last instruction in the subroutine is BUN 64. When this instruction is executed, control is transferred to the instruction in location 64. But in address 64, there is now an instruction that branches back to address 33. The address stored in location 64 by the BSB instruction will always have the proper return address no matter where the BSB instruction is located. In this way, the subroutine return is always to a location one higher than the location of the BSB instruction. Note that the address number of the BUN instruction placed at the end of the subroutine must always be equal to the address number where the return address is temporarily stored, which is 64 in this case.

11.3.7 Register-reference Instructions

The 12 register-reference instructions for the computer are listed in Table 11-3. Each register-reference instruction has an operation code 0110 (hexadecimal 6) and contains a single 1 in one of the remaining 12 bits of the instruction. These instructions are specified with four hexadecimal digits which represent all 16 bits of an instruction word. The first seven instructions perform an operation on the *A* or *E* register and are self-explanatory. The next four are skip instructions used for program control, conditioned on certain status bits. To skip the next instruction, the *PC* is incremented by 1 once again. The first increment occurs when the present instruction is read. In this way, the next instruction read from memory is two locations up from the location of the present (skip) instruction.

The status bits for the skip instructions are the sign bit in *A*, which is in flip-flop A_{16}, and a zero condition for *A* or *E*. If the designated status condition is present, the next instruction in sequence is skipped; otherwise, the computer continues from the next instruction in sequence because *PC* is not incremented.

Table 11-3 Register-reference instructions

Symbol	Hexa-decimal code	Description	Function
CLA	6800	Clear *A*	$A \leftarrow 0$
CLE	6400	Clear *E*	$E \leftarrow 0$
CMA	6200	Complement *A*	$A \leftarrow \overline{A}$
CME	6100	Complement *E*	$E \leftarrow \overline{E}$
SHR	6080	Shift-rights *A* and *E*	$A \leftarrow \text{shr}\, A, A_{16} \leftarrow E, E \leftarrow A_1$
SHL	6040	Shift-left *A* and *E*	$A \leftarrow \text{shl}\, A, A_1 \leftarrow E, E \leftarrow A_{16}$
INC	6020	Increment *A*	$A \leftarrow A + 1$
SPA	6010	Skip on positive *A*	If $(A_{16} = 0)$ then $(PC \leftarrow PC + 1)$
SNA	6008	Skip on negative *A*	If $(A_{16} = 1)$ then $(PC \leftarrow PC + 1)$
SZA	6004	Skip on zero *A*	If $(A = 0)$ then $(PC \leftarrow PC + 1)$
SZE	6002	Skip on zero *E*	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	6001	Halt computer	$S \leftarrow 0$

Table 11-4 Input-output instructions

Symbol	Hexadecimal code	Description	Function
SKI	7800	Skip on input flag	If ($N_9 = 1$) then ($PC \leftarrow PC + 1$)
INP	7400	Input to A	$A_{1-8} \leftarrow N_{1-8}, N_9 \leftarrow 0$
SKO	7200	Skip on output flag	If ($U_9 = 1$) then ($PC \leftarrow PC + 1$)
OUT	7100	Output from A	$U_{1-8} \leftarrow A_{1-8}, U_9 \leftarrow 0$

The halt instruction is usually placed at the end of a program if one wishes to stop the computer. Its execution clears the start-stop flip-flop, which prevents further operations.

11.3.8 Input-Output Instructions

The computer has four input-output instructions and they are listed In Table 11-4. These instructions have an operation code 0111 (hexadecimal 7), and each contains a 1 in only one of the remaining 12 bits of the instruction word. The input-output instructions are specified with four hexadecimal digits starting with 7.

The INP instruction transfers the input character from N to A and also clears the input flag in N_9. The OUT instruction transfers an 8-bit character code from A into the output register and also clears the output flag in U_9. The two skip instructions check the corresponding status flags and cause a skip of the next instruction if the flag bit is 1. The instruction that is skipped is normally a BUN instruction. The BUN instruction is not skipped if the flag bit is 0; this causes a branch back to the skip instruction to check the flag again. If the flag bit is 1, the BUN instruction is skipped and an input or output operation is executed. Thus, the computer stays in a two-instruction loop (skip on flag and branch back to previous instruction) until the flag bit is set by the external device. The next instruction in sequence must be an input or output instruction.

11.4 Timing and Control

All operations in the computer are synchronized by the master-clock generator whose clock pulses are applied to all flip-flops in the system. In addition, a certain number of timing variables are available in the control unit to sequence the operation in the proper order. These timing variables are designated t_0, t_1, t_2, and t_3 and are shown in Fig. 11-5. The clock pulses occur once every microsecond (μs). Each timing variable is 1 μs long and occurs once every 4 μs. We assume that the triggering of flip-flops occurs during the negative edge of the clock pulses. By applying one of the timing variables to the enable input of a given register, we can control the specific clock pulse that triggers the register. The timing variables repeat continuously in such a way that t_0 always appears after t_3. Four timing variables are sufficient for the execution of any instruction in the computer we are considering here. In other situations, it may be necessary to employ a different number of timing variables.

We assume that the memory access time is less than 1 μs. A memory read or write operation can be initiated with one of the timing variables when it goes high. The memory operation will be completed by the time the next clock pulse arrives.

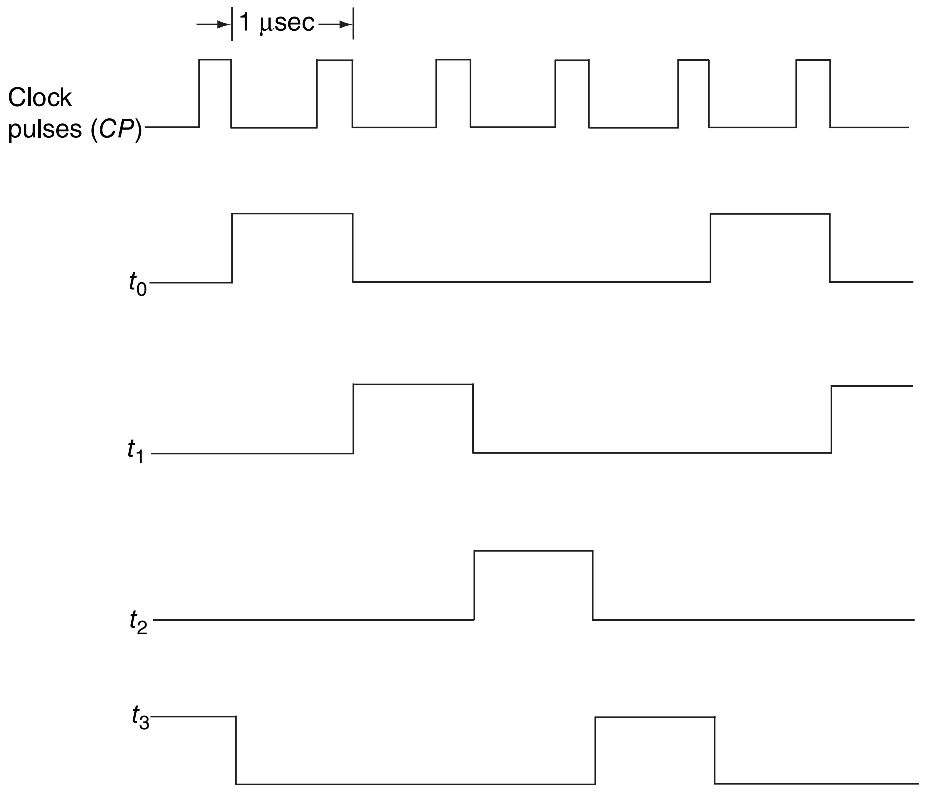


Figure 11 5 Computer timing signals

The digital computer operates in discrete steps controlled by the timing signals. An instruction is read from memory and executed in registers by a sequence of microoperations. When the control receives an instruction, it generates the appropriate control functions for the required microoperations. A block diagram of the control logic is shown in Fig. 11-6. An instruction read from memory is placed in the memory buffer register *B.* The instruction has an operation code of 4 bits, designated by the symbol *OP.* If it is a memory-reference instruction, it has an address part designated by the symbol *AD.* The operation code is always transferred to the instruction register *I*. The operation code in *I* is decoded into eight outputs q_0–q_7, the subscript number being equal to the hexadecimal code for the operation. The *G* register is a 2-bit counter that continuously counts the clock pulses as long as start-stop flip-flop *S* is set. The outputs of the *G* register are decoded into the four timing variables t_0–t_3. The *F* flip-flop distinguishes between the fetch and execute cycles. Other status conditions are sometimes needed to determine the sequence of control. The outputs of the control logic network initiate all microoperations for the computer. The block diagram of the control logic is helpful in visualizing the control unit of the computer when the register-transfer operations are derived during the logic design process.

The control logic network is a combinational circuit consisting of a random connection of gates. Its implementation constitutes a hard-wired control. We shall see in Section 11-7 that the control part of the computer can also be implemented with programmable logic arrays. The PLA configuration will replace the control logic network as well as the operation and timing decoders. It will also be shown in Section 11-7 that the control can be partially implemented with a microprogram unit. The microprogram control configuration will replace the control logic network, the two decoders, and the *I* and *G* registers.

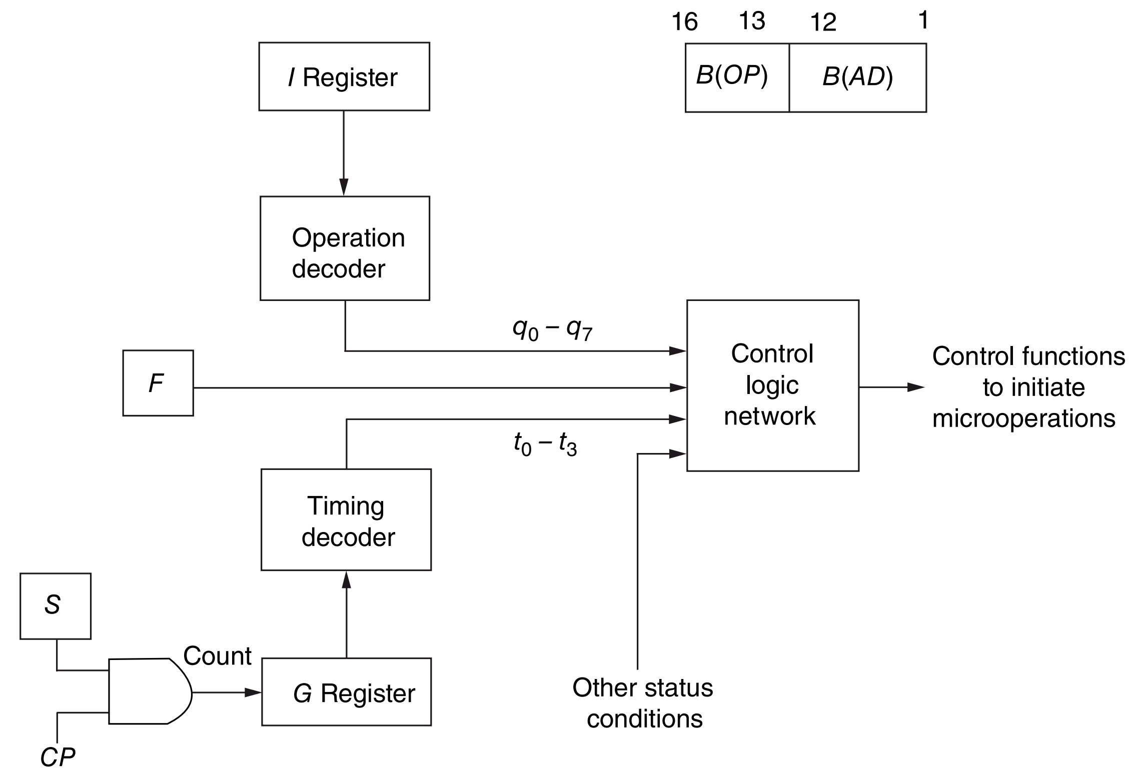


Figure 11.6 Block diagram of control logic

11.5 Execution of Instructions

Up to this point, we have considered the system design of the computer. We have specified the register configuration, the set of computer instructions, a timing sequence, and the configuration of the control unit. In this section, we start with the logic design phase of the computer. The first step is to specify the microoperations, together with the control functions, needed to execute each machine instruction.

The register-transfer operations describe in concise form the process of information transfer within the registers of the computer. Each statement in the description consists of a control function, followed by a colon, followed by one or more microoperations in symbolic notation. The control function is a Boolean function whose variables are the timing signals t_0–t_3, the decoded operation q_0–q_7, and certain status-bit conditions. The microoperations are specified in accordance with the symbolic notation defined in the register-transfer method.

Once a "start" switch is activated, the computer sequence follows a basic pattern. An instruction whose address is in *PC* is read from memory. Us operation part is transferred to register *I*, and *PC* is incremented by 1 to prepare it for the address of the next instruction. If the instruction is a memory-reference type, it may be necessary to access the memory again to read an operand. Thus, words read from memory into the *B* register can be either instructions or data. The *F* flip-flop is used to distinguish between the two. When *F* = 0, the word read from memory is interpreted to be an instruction and the computer is said to be in an instruction *fetch* cycle, When *F* = 1, the word read from memory is taken as an operand and the computer is said to be in a data *execute* cycle.

11.5.1 Fetch Cycle

An instruction is read from memory during the fetch cycle. The register-transfer relations that specify this process are:

$$\begin{array}{ll} F't_0: & MAR \leftarrow PC \\ F't_1: & B \leftarrow M, PC \leftarrow PC + 1 \\ F't_2: & I \leftarrow B\ (OP) \end{array}$$

When $F = 0$, liming signals t_0, t_1, and t_2 initiate a sequence of operations that transfer the contents of *PC* into *MAR*, initiate a memory read, increment *PC*, and transfer the operation code of the instruction to the *I* register. All microoperations are executed when the control function is logic 1 and when a clock pulse occurs. The microoperations in registers and the transfer of the memory word into *B* are executed during the negative edge of a clock pulse. This occurs just prior to the time that the specified timing variable goes to 0.

The operation code in the *I* register is decoded at time t_3. The next step depends on the value of q_i, $i = 0, 1, ..., 7$, that produces a 1 in the output of the decoder. If the decoded output is a memory-reference instruction, an operand may be needed. If not, the instruction can be executed during time t_3.

The BUN instruction and the register-reference and input-output instructions do not need a second access to memory. When an operation code 0, 1, 2, 3, or 4 is encountered, the computer has to go to an execute cycle to access the memory again. This condition is detected from the operation decoder which causes a transfer to the execute cycle by selling *F* to 1:

$$F'(q_0 + q_1 + q_2 + q_3 + q_4)\ t_3: F \leftarrow 1$$

The register-transfer operations common to all instructions during the fetch cycle are listed in Table 11-5.

The BUN instruction has an operation code 5, and its corresponding output from the operation decoder is q_5. This instruction does not need an operand from memory, even though it is listed as a memory-reference instruction. It merely specifies that the next instruction be taken from a location given by the address part *m.* The address part of the instruction is in *B*(*AD*) at time t_3 of the fetch cycle. The instruction can be executed during the fetch cycle at this time:

$$q_5 t_3: PC \leftarrow B(AD)$$

There is no need to include *F* in the control function because the only time that q_5 can be 1 is during the fetch cycle. The microoperation that executes the instruction specifies a transfer of bits 1 through 12 of register *B* into the *PC*. The next timing variable after t_3 is always t_0. Since *F* remains 0 for this instruction, the computer returns to the beginning of the fetch cycle to read the instruction given by *PC*.

The register-reference instructions are recognized from the decoder output q_6, and the input-output instructions from q_7. Since these instructions require only one more microoperation for their execution, they can be terminated at time t_3 during the fetch cycle. This fact is indicated in Table 11-5. The specific microoperations are listed in later tables.

Table 11-5 Register-transfer operations during fetch cycle

$F't_0$:	$MAR \leftarrow PC$	Transfer instruction address
$F't_1$:	$B \leftarrow M, PC \leftarrow PC + 1$	Read instruction, increment PC
$F't_2$:	$I \leftarrow B(OP)$	Transfer operation code
$F'(q_0 + q_1 + q_2 + q_3 + q_4)\, t_3$:	$F \leftarrow 1$	Go to execute cycle
$q_5 t_3$:	$PC \leftarrow B(AD)$	Branch unconditionally (BUN)
$Q_6 t_3$:	See Table 11-8	Register-reference instruction
$Q_7 t_3$:	See Table 11-9	Input-output instruction

11.5.2 Execute Cycle

Flip-flop F is equal to 1 during the execute cycle. The four timing variables that occur during this cycle perform the microoperations for executing one of the memory-reference instructions. The instruction to be executed is specified by variable q_i, $i = 0, 1,2, 3, 4$, available from the operation decoder. At the end of the fetch cycle, the address part of the instruction is in bits 1 through 12 of register B, symbolized by $B(AD)$. This address is transferred to MAR at the beginning of the execute cycle to serve as the memory address for the subsequent memory word:

$$Ft_0\text{: } MAR \leftarrow B(AD)$$

The instructions that need an operand from memory are the AND (q_0) ADD (q_1), and ISZ (q_3). The other two instructions, STO (q_2) and BSB (q_4), store a value into memory and are excluded during the next memory read operation:

$$F(q_0 + q_1 + q_3)t_1\text{: } B \leftarrow M$$

The particular decoded instruction is executed with timing variables t_2 and t_3. At time t_3, the F flip-flop is cleared for the computer to return to the fetch cycle:

$$Ft_3\text{: } F \leftarrow 0$$

The next timing variable after t_3 is t_0. But now F is equal to 0, so the next control function is $F't_0$. This is the first control function in the fetch cycle. Thus, after executing the current instruction, control always returns to the fetch cycle to read the next instruction whose address is in PC. The common operations performed during the execute cycle are listed in Table 11-6.

Table 11-6 Common operations for execute cycle

Ft_0:	$MAR \leftarrow B(AD)$	Transfer address part
$F(q_0 + q_1 + q_3)t_1$:	$B \leftarrow M$	Read operand
$F(t_2 + t_3)$:	See Table 11-7	Execute memory-reference instruction
Ft_3:	$F \leftarrow 0$	Return to fetch cycle

Table 11-7 Execution of memory-reference instructions

AND	Fq_0t_3:	$A \leftarrow A \wedge B$	AND microoperation
ADD	Fq_1t_3:	$A \leftarrow A \wedge B, E \leftarrow$ carry	Add microoperation
STO	Fq_2t_2:	$B \leftarrow A$	Transfer A to B
	Fq_2t_3:	$M \leftarrow B$	Store in memory
ISZ	Fq_3t_2:	$B \leftarrow B + 1$	Increment memory word
	Fq_3t_3:	$M \leftarrow B$	Store back in memory
	$Fq_3B_zt_3$:	$PC \leftarrow PC + 1$	Skip if $B_z = 1$ ($B = 0$)
BSB	Fq_4t_2:	$B(AD) \leftarrow PC$, $B(OP) \leftarrow 0101$, $PC \leftarrow MAR$	Transfer return address, transfer address to PC
	Fq_4t_3:	$M \leftarrow B$, $PC \leftarrow PC + 1$	Store return address, increment address in PC

The five memory-reference instructions and their corresponding register-transfer operations are listed in Table 11-7. These instructions are executed when $F = 1$ and with timing variables t_2 and t_3 The decoded operation q_i determines the particular instruction that is executed.

The AND and ADD instructions are executed with timing variable t_3, although they could use timing variable t_2 instead. The operand from memory has been transferred to B with timing variable t_1. The corresponding operation can be performed now between the B and A registers.

The STO instruction specifies a transfer of the contents of A into the memory word whose address was transferred to MAR with timing variable t_0. The contents of A are first transferred into B, and a write operation transfers the contents of B into the memory word specified by MAR:

$$Fq_2t_2: B \leftarrow A$$
$$Fq_2t_3: M \leftarrow B$$

The ISZ instruction is executed with the following microoperations:

$$Fq_3t_2: B \leftarrow B + 1$$
$$Fq_3t_3: M \leftarrow B$$
$$Fq_3B_zt_3: PC \leftarrow PC + 1 \qquad B_z = 1 \text{ if } B = 0$$

The word from location M was placed in B during time t_1 (see Table 11-6). The B register is incremented at time t_2 and the new value is stored back in memory. All this time MAR does not change, so it always specifies the address of M. Remember that a memory word cannot be incremented while residing in memory. It must be transferred to a processor register where the counting can be implemented. While the incremented number is being stored in memory, its value in B is checked; if it is 0, PC is incremented to cause a skip of one instruction. Variable B_z used in the last statement above is a zero-detect variable and is equal to binary 1 if register B contains an all-0's number.

The BSB instruction is the most complicated instruction available in the computer. A possible way to execute this instruction is as follows:

$$Fq_4t_2\text{: } B(AD) \leftarrow PC,\ B(OP) \leftarrow 0101,\ PC \leftarrow MAR$$
$$Fq_4t_3\text{: } M \leftarrow B,\ PC \leftarrow PC + 1$$

The return address available in *PC* is transferred to the address part of register *B* and the code 0101 (BUN) is transferred to the operation-code part of the same register. Remember that the address register *MAR* contains the address part of the instruction designated by *m*. The transfer from *MAR* to *PC* results in transferring *m* into *PC*. All this is done during timing variable t_2. The return address is stored in memory at time t_3. *PC* is also incremented at this time, so the instruction to be read during the next fetch cycle will be from location $m + 1$.

11.5.3 Register-reference Instructions

The register microoperations that execute the register-reference instructions are listed in Table 11-8. These instructions are recognized from operation decoder output q_6 and are executed during time t_3 of the fetch cycle. For convenience, we define a new variable $r = q_6t_3$ and use it in all register-reference control functions. The rest of the control function is determined from one of the bits in the *B* register, where the rest of the instruction resides at this time. For example, the instruction CLA has the hexadecimal code 6800, which corresponds to a binary code 0110 1000 0000 0000. The operation code is decoded from the *I* register and is equal to q_6. Bit 12 in the *B* register is 1; so the control function that executes this instruction is $q_6t_3B_{12} = rB_{12}$.

The first seven register-reference instructions perform the clear, complement, shift, and increment operations on the *A* or *E* register. The next four instructions are skip instructions executed only if the stated condition is satisfied. The skipping of the instruction is achieved by incrementing *PC* again, in addition to the incrementing at time t_1 (see Table 11-5). The status-bit condition for skipping becomes part of the control function. Thus the accumulator is positive if

Table 11-8 Execution of register-reference instructions

	$r = q_6t_3$		
CLA	rB_{12}:	$A \leftarrow 0$	Clear *A*
CLE	rB_{11}:	$E \leftarrow 0$	Clear *E*
CMA	rB_{10}:	$A \leftarrow \overline{A}$	Complement *A*
CME	rB_9:	$E \leftarrow \overline{E}$	Complement *E*
SHR	rB_8:	$A \leftarrow \text{shr } A,\ A_{16} \leftarrow E,\ E \leftarrow A_1$	Shift-right *A* and *E*
SHL	rB_7:	$A \leftarrow \text{shl } A,\ A_1 \leftarrow E,\ E \leftarrow A_{16}$	Shift-left *A* and *E*
INC	rB_6:	$A \leftarrow A + 1$	Increment *A*
SPA	$rB_5A'_{16}$:	$PC \leftarrow PC + 1$	Increment *PC* if *A* is positive
SNA	rB_4A_{16}:	$PC \leftarrow PC + 1$	Increment *PC* if *A* is negative
SZA	rB_3A_z:	$PC \leftarrow PC + 1$	Increment *PC* if *A* is zero
SZE	rB_2E':	$PC \leftarrow PC + 1$	Increment *PC* if *E* is zero
HLT	rB_1:	$S \leftarrow 0$	Gear start-stop flip-flop

Table 11-9 Execution of input-output instructions

	$p = q_7t_3$		
SKI	$pB_{12}N_9$:	$PC \leftarrow PC + 1$	Increment PC if input flag $N_9 = 1$
INP	pB_{11}:	$A_{1-8} \leftarrow N_{1-8}, N_9 \leftarrow 0$	Input to A, clear flag
SKO	$pB_{10}U_9$:	$PC \leftarrow PC + 1$	Increment PC if output flag $U_9 = 1$
OUT	pB_9:	$U_{1-8} \leftarrow A_{1-8}, U_9 \leftarrow 0$	Output from A, clear flag

$A_{16} = 0$ and negative if $A_{16} = 1$. The symbol $A_{\cdot}$ is a binary variable equal to I when the A register contains all 0's. E' is equal to 1 when the E flip-flop contains 0.

The halt instruction clears the start-stop flip-flop S and stops the timing sequence. The sequence register G stops counting while its value is 0. This causes the computer to idle, with t_0 always being at the output of the timing decoder. Since F is also 0, the control function $F't_0$ is the only one produced while the computer is halted. This control function transfers the contents of PC to MAR continuously (see Table 11-5). We could tolerate this continuous transfer when the computer halts. If this is undesirable, we can remove the clock pulses from MAR as well to prevent this transfer from occurring when $S = 0$. The computer can resume when the "start" switch is activated, which sets flip-flop S. This causes die clock pulses to reach the sequence register G and start producing the other timing variables.

11.5.4 Input-Output Instructions

The register-transfer microoperations that execute the four input-output instructions are listed in Table 11-9. These instructions are recognized from operation-decoder output q_7 and are executed during time t_3. We define a new variable $p = q_7t_3$ and use it in all input-output control functions. The control functions for these instructions contain a single bit from the B register which is part of the instruction-code definition. The two skip instructions depend on the status condition of flag bits N_9 and U_9.

11.6 Design of Computer Registers

The design of a synchronous digital system follows a prescribed procedure. From a knowledge of the system requirements, one formulates a control network and obtains a list of register-transfer operations for the system. Once this list is derived, the rest of the design is straightforward. Some installations utilize computer design automation techniques for translating the register-transfer statements to a circuit diagram composed of integrated circuits.

Section 11-5 specified the register-transfer statements for the computer in five separate tables. The entries in the tables consist of control functions and microoperations. The list of control functions provides the Boolean functions for the gates in the control logic network. The list of microoperations gives an indication of the types of registers that must be chosen for the computer. Although these tables are sufficient to complete the logic design of the system, it may be convenient to rearrange the information in the tables in a more convenient way during the actual implementation .process.