CHAPTER SEVEN

Microprogrammed Control

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 1 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 microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a control word. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called

control word

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.

microinstruction

microprogram

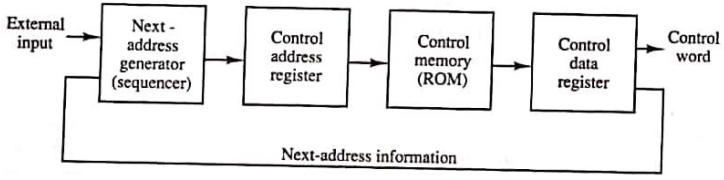
A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the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

control memory

A computer that employs 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 general configuration 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.



control address register

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.

sequencer

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.

pipeline register

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.

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 sequence 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

routine

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.

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 microopera-

mapping

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

special bits

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.

branch logic

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.

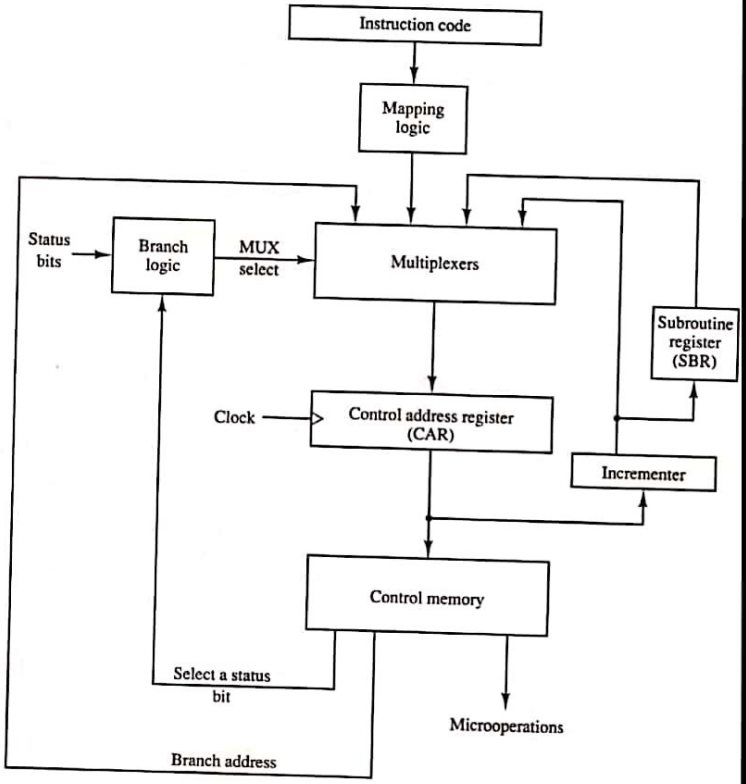


Figure 7-2 Selection of address for control memory.

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.

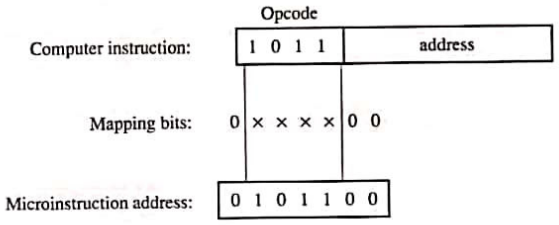
An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status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

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



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 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.

subroutine register

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

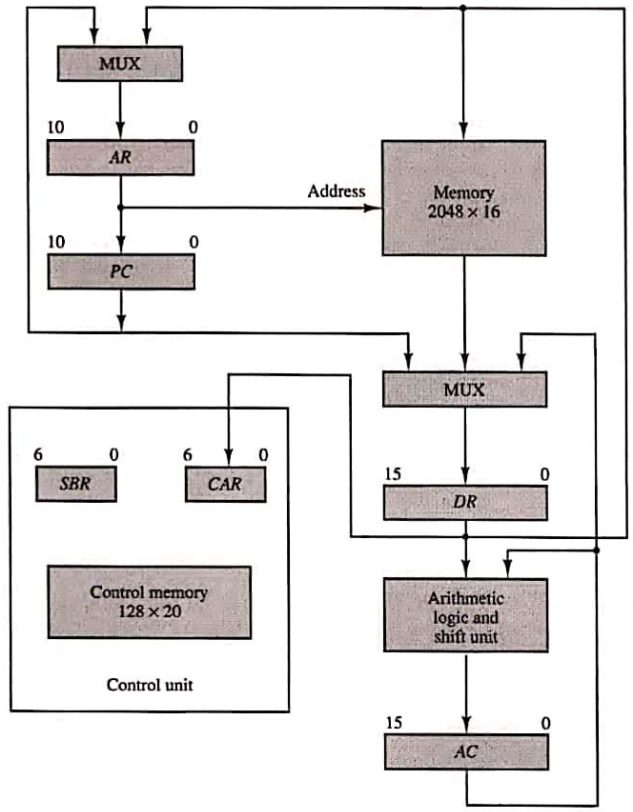


Figure 7-4 Computer hardware configuration.

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 performs 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*.

instruction format

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.

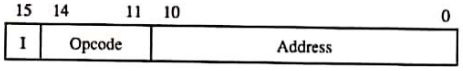
It will be shown subsequently that each computer instruction must be microprogrammed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be microprogrammed by the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

Microinstruction Format

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 F1, F2, and F3 specify microoperations for the computer. The CD field

Figure 7-5 Computer instructions.

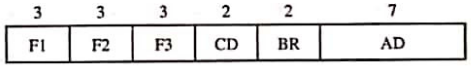


(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions



F1, 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 $128 = 2^7$ words.

microoperations

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 F1.

$$DR \leftarrow M[AR] \quad \text{with F2} = 100$$
$$\text{and} \quad PC \leftarrow PC + 1 \quad \text{with F3} = 101$$

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 $AC \leftarrow DR$ (F1 = 100) has the symbol DRTAC, which stands for a transfer from *DR* to *AC*.

condition field

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

TABLE 7-1 Symbols and Binary Code for Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	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	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

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.

branch field

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.

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.

address field

5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
   a. With a symbolic address, which must also appear as a label.
   b. With the symbol NEXT to designate the next address in sequence.
   c. 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.

ORG

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.

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

$$AR \leftarrow PC$$

$$DR \leftarrow \mathrm{M}[AR], \quad PC \leftarrow PC + 1$$

$$AR \leftarrow DR(0\text{–}10), \quad CAR(2\text{–}5) \leftarrow DR(11\text{–}14), \quad CAR(0,1,6) \leftarrow 0$$

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*.

fetch and decode

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:

```
        ORG 64
FETCH:  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	F1	F2	F3	CD	BR	AD
1000000	110	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 0xxxx00, 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 0000000, 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 *I*. 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    RET
```


TABLE 7-2 Symbolic Microprogram (Partial)

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

execution of instructions

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.

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 $I = 1$. 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 $I = 1$. 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.

control memory

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

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

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	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 1'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 1'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 is

cheaper 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 2^k 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.

decoding of F fields

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 3 × 8 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 $F1 = 101$ (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 $F1 = 110$ (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 $F1$ 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.

arithmetic logic shift unit

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, respectively, 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.

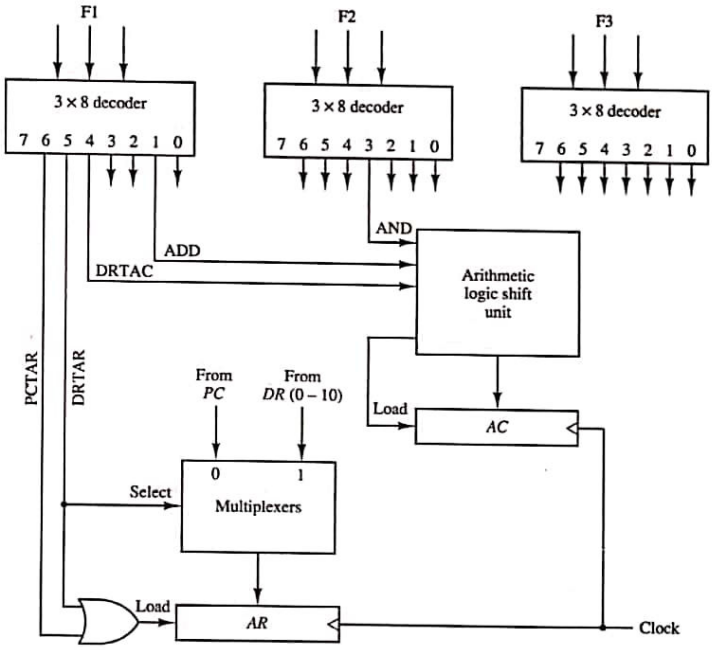


Figure 7-7 Decoding of microoperation fields.

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 applications.

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.

design of input logic

The input logic circuit in Fig. 7-8 has three inputs, I_0, I_1, and T, and three outputs, S_0, S_1, and L. Variables S_0 and S_1 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 $S_1S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer

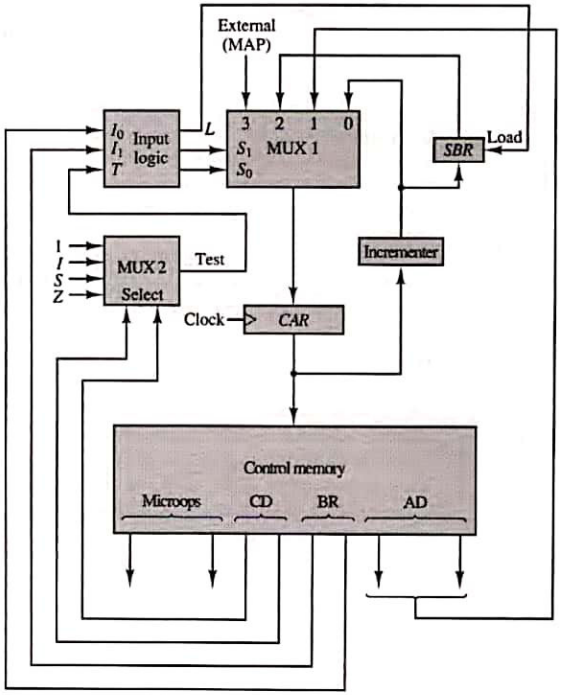


Figure 7-8 Microprogram sequencer for a control memory.

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 I_1 and I_0 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 S_1 and S_0 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:

$$S_1 = I_1$$
$$S_0 = I_1I_0 + I_1'T$$
$$L = I_1'I_0T$$

TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Field	Input			MUX 1		Load *SBR*
	I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	1
1	0	1	0	1	0	0
1	1	1	1	1	1	0

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

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.

PROBLEMS

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?

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?

b. If there are 16 status bits in the system, how many bits of the branch logic are used to select a status bit?

c. How many bits are left to select an input for the multiplexers?

7-6. The control memory in Fig. 7-2 has 4096 words of 24 bits each.

a. How many bits are there in the control address register?

b. How many bits are there in each of the four inputs shown going into the multiplexers?

c. What are the number of inputs in each multiplexer and how many multiplexers are needed?

7-7. Using the mapping procedure described in Fig. 7-3, give the first microinstruction address for the following operation code: (a) 0010; (b) 1011; (c) 1111.

7-8. Formulate a mapping procedure that provides eight consecutive microinstructions for each routine. The operation code has six bits and the control memory has 2048 words.

7-9. Explain how the mapping from an instruction code to a microinstruction address can be done by means of a read-only memory. What is the advantage of this method compared to the one in Fig. 7-3?

7-10. Why do we need the two multiplexers in the computer hardware configuration shown in Fig. 7-4? Is there another way that information from multiple sources can be transferred to a common destination?

7-11. Using Table 7-1, give the 9-bit microoperation field for the following microoperations:

a. $AC \leftarrow AC + 1$, $DR \leftarrow DR + 1$

b. $PC \leftarrow PC + 1$, $DR \leftarrow M[AR]$

c. $DR \leftarrow AC$, $AC \leftarrow DR$

7-12. Using Table 7-1, convert the following symbolic microoperations to register transfer statements and to binary.

a. READ, INCPC

b. ACTDR, DRTAC

c. ARTPC, DRTAC, WRITE

7-13. Suppose that we change the ADD routine listed in Table 7-2 to the following two microinstructions.

```
ADD:    READ    I    CALL    INDR2
        ADD     U    JMP     FETCH
```

What should be subroutine INDR2?

7-14. The following is a symbolic microprogram for an instruction in the computer defined in Sec. 7-3.

```
        ORG 40
        NOP         S    JMP     FETCH
        NOP         Z    JMP     FETCH
        NOP         I    CALL    INDRCT
        ARTPC       U    JMP     FETCH
```

a. Specify the operation performed when the instruction is executed.

b. Convert the four microinstructions into their equivalent binary form.

7-15. The computer of Sec. 7-3 has the following binary microprogram:

Address	Binary Microprogram
60	0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1
61	1 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0
62	0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1
63	1 0 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0

a. Translate it to a symbolic microprogram as in Table 7-2. (FETCH is in address 64 and INDRCT in address 67.)
b. List all the things that will be wrong when this microprogram is executed in the computer.

7-16. Add the following instructions to the computer of Sec 7-3 (*EA* is the effective address). Write the symbolic microprogram for each routine as in Table 7-2. (Note that *AC* must not change in value unless the instruction specifies a change in *AC*.)

Symbol	Opcode	Symbolic Function	Description
AND	0100	$AC \leftarrow AC \wedge M[EA]$	AND
SUB	0101	$AC \leftarrow AC - M[EA]$	Subtract
ADM	0110	$M[EA] \leftarrow M[EA] + AC$	Add to memory
BTCL	0111	$AC \leftarrow AC \wedge \overline{M[EA]}$	Bit clear
BZ	1000	If $(AC = 0)$ then $(PC \leftarrow EA)$	Branch if *AC* zero
SEQ	1001	If $(AC = M[EA])$ then $(PC \leftarrow PC + 1)$	Skip if equal
BPNZ	1010	If $(AC > 0)$ then $(PC \leftarrow EA)$	Branch if positive and nonzero

7-17. Write a symbolic microprogram routine for the ISZ (increment and skip if zero) instruction defined in Chap. 5 (Table 5-4). Use the microinstruction format of Sec. 7-3. Note that $DR = 0$ status condition is not available in the *CD* field of the computer defined in Sec. 7-3. However, you can exchange *AC* and *DR* and check if $AC = 0$ with the Z bit.

7-18. Write the symbolic microprogram routines for the BSA (branch and save address) instructions defined in Chap. 5 (Table 5-4). Use the microinstruction format of Sec. 7-3. Minimize the number of microinstructions.

7-19. Show how outputs 5 and 6 of decoder F3 in Fig. 7-7 are to be connected to the program counter *PC*.

7-20. Show how a 9-bit microoperation field in a microinstruction can be divided into subfields to specify 46 microoperations. How many microoperations can be specified in one microinstruction?

7-21. A computer has 16 registers, an ALU (arithmetic logic unit) with 32 operations, and a shifter with eight operations, all connected to a common bus system.
a. Formulate a control word for a microoperation.
b. Specify the number of bits in each field of the control word and give a general encoding scheme.
c. Show the bits of the control word that specify the microoperation $R4 \leftarrow R5 + R6$.

7-22. Assume that the input logic of the microprogram sequencer of Fig. 7-8 has four inputs, I_2, I_1, I_0, T (test), and three outputs, S_1, S_0, and L. The operations that are performed in the unit are listed in the following table. Design the input logic circuit using a minimum number of gates.

I_2	I_1	I_0	Operation
0	0	0	Increment *CAR* if $T = 1$, jump to *AD* if $T = 0$
×	0	1	Jump to *AD* unconditionally
1	0	0	Increment *CAR* unconditionally
0	1	0	Jump to *AD* if $T = 1$, increment *CAR* if $T = 0$
1	1	0	Call subroutine if $T = 1$, increment *CAR* if $T = 0$
0	1	1	Return from subroutine unconditionally
1	1	1	Map external address unconditionally

7-23. Design a 7-bit combinational circuit incrementer for the microprogram sequencer of Fig. 7-8 (see Fig. 4-8). Modify the incrementer by including a control input *D*. When $D = 0$, the circuit increments by one, but when $D = 1$, the circuit increments by two.

7-24. Insert an exclusive-OR gate between MUX 2 and the input logic of Fig. 7-8. One input to the gate comes from the test output of the multiplexer. The other input to the gate comes from a bit labeled *P* (for polarity) in the microinstruction from control memory. The output of the gate goes to the input *T* of the input logic. What does the polarity control *P* accomplish?

REFERENCES

1. Dasgupta, S., *Computer Architecture: A Modern Synthesis*. Vol. 1. New York: John Wiley, 1989.
2. Gorsline, G. W., *Computer Organization: Hardware/Software*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1986.
3. Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 3rd ed. New York: McGraw-Hill, 1990.

CHAPTER 7

7-1

A microprocessor is a small size CPU (computer on a chip)
Microprogram is a program for a sequence of microoperations.
The control unit of a microprocessor can be hardwired or microprogrammed, depending on the specific design.
A microprogrammed computer does not have to be a microprocessor.

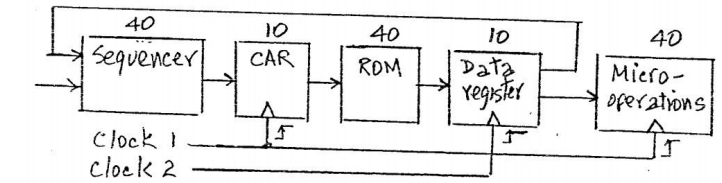
7-2

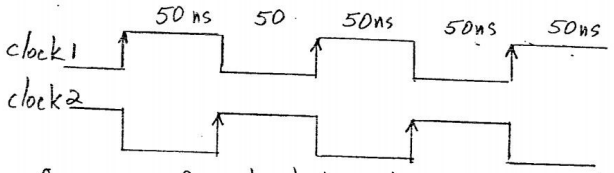
Hardwired control, by definition, does not contain a control memory.

7-3

Microoperation - an elementary digital computer operation.
Microinstruction - an instruction stored in control memory.
Microprogram - a sequence of microinstructions.
Microcode - same an microprogram.

7-4





frequency of each clock $= \frac{1}{100 \times 10^{-9}} = \frac{1000}{100} \times 10^6 = 10\ MHz$

If the data register is removed, we can use a single phase clock with a frequency of $\frac{1}{90 \times 10^{-9}} = 11.1\ MHz$

7-5

Control memory $= 2^{10} \times 32$

(a)

6	10	16	= 32 bits
Select	Address	Microoperations	

(b) 4 bits
(c) 2 bits

7-6

Control memory $= 2^{12} \times 24$

(a) 12 bits
(b) 12 bits
(c) 12 multiplexers, each of size 4-to-1 line.

7-7

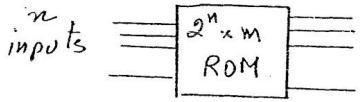
(a) 0001000 = 8
(b) 0101100 = 44
(c) 0111100 = 60

7-8

op code = 6 bits
control memory address = 11 bits

00	xxxxxx	000

7-9



The ROM can be programmed to provide any desired address for a given inputs from the instruction.

7-10

Either multiplexers, three-state gates, or gate logic (equivalent to a mux) are needed to transfer information from many sources to a common destination.

7-11

	F1	F2	F3			
(a)	011	110	000	INCAC	INCDR	NOP
(b)	000	100	101	NOP	READ	INCPC
(c)	100	101	000	DRTAC	ACTDR	NOP

7-12

				Binary
(a)	READ	DR ← M[AR]	F2 = 100	000 100 101
	DRTAC	AC ← DR	F3 = 101	
(b)	ACTDR	DR ← AC	F2 = 101	000 100 101
	DRTAC	AC ← DR	F1 = 100	
(c)	ARTPC	PC ← AR	F3 = 110	Impossible.
	DRTAC	AC ← DR	F1 = 100	Both use F1
	WRITE	M[AR] ← DR	F1 = 111	

7-13

If I=0, the operand is read in the first microinstruction and added to AC in the second.

If I=1, the effective address is read into DR and control goes to INDR2. The subroutine must read the operand into DR.

```
INDR2:  DRTAR   U   JMP   NEXT
        READ    U   RET   ---
```

7-14

(a) Branch if S=0 and Z=0 (positive and non-zero AC) - See last instruction in Problem 7-16.

(b)

```
40:  000 000 000   10 00   1000000
41:  000 000 000   11 00   1000000
42:  000 000 000   01 01   1000011
43:  000 000 110   00 00   1000000
```


7-15

(a)

```
60:  CLRAC, COM   U   JMP    INDRCT
61:  WRITE, READ  I   CALL   FETCH
62:  ADD, SUB     S   RET    63 (NEXT)
63:  DRTAC, INCDR Z   MAP    60
```

(b)

60: Cannot increment and complement AC at the same time. With a JMP to INDRCT, control does not return to 61.

61: Cannot read and write at the same time. The CALL behaves as a JMP since there is no return from FETCH.

62: Cannot add and subtract at the same time. The RET will be executed independent of S.

63: The MAP is executed irrespective of Z or 60.

7-16

```
        ORG 16
AND:    NOP          I   CALL   INDRCT
        READ         U   JMP    NEXT
ANDOP:  AND          U   JMP    FETCH
```

```
        ORG 20
SUB:    NOP          I   CALL   INDRCT
        READ         U   JMP    NEXT
        SUB          U   JMP    FETCH
```

```
        ORG 24
ADM:    NOP          I   CALL   INDRCT
        READ         U   JMP    NEXT
        DRTAC, ACTDR U   JMP    NEXT
        ADD          U   JMP    EXCHANGE+2
                                (Table 7-2)
```

7-16 (CONTINUED)

```
        ORG 28
BTCL:   NOP              I   CALL   INDRCT
        READ             U   JMP    NEXT
        DRTAC, ACTDR     U   JMP    NEXT
        COM              U   JMP    ANDOP
        ORG 32
BZ:     NOP              Z   JMP    ZERO
        NOP              U   JMP    FETCH
ZERO:   NOP              I   CALL   INDRCT
        ARTPC            U   JMP    FETCH
        ORG 36
SEQ:    NOP              I   CALL   INDRCT
        READ             U   JMP    NEXT
        DRTAC, ACTDR     U   JMP    NEXT
        XOR (or SUB)     U   JMP    BEQ1
        ORG 69
BEQ1:   DRTAC, ACTDR     Z   JMP    EQUAL
        NOP              U   JMP    FETCH
EQUAL:  INCPC            U   JMP    FETCH
        ORG 40
BPNZ:   NOP              S   JMP    FETCH
        NOP              Z   JMP    FETCH
        NOP              I   CALL   INDRCT
        ARTPC            U   JMP    FETCH
```


7-17

```
ISZ:    NOP              I   CALL   INDRCT
        READ             U   JMP    NEXT
        INCDR            U   JMP    NEXT
        DRTAC, ACTDR     U   JMP    NEXT (by fast INDRCT)
        DRTAC, ACTDR     Z   JMP    ZERO
        WRITE            U   JMP    FETCH
ZERO:   WRITE, INCPC     U   JMP    FETCH
```

7-18

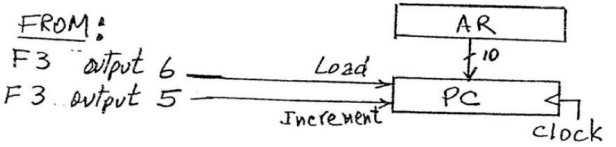
```
BSA:    NOP              I   CALL   INDRCT
        PCTDR, ARTPC     U   JMP    NEXT
        WRITE, INCPC     U   JMP    FETCH
```

7-19

From Table 7-1:

F3 = 101 (5) PC ← PC+1

F3 = 110 (6) PC ← AR



7-20

A field of 5 bits can specify $2^5 - 1 = 31$ microoperations

A field of 4 bits can specify $2^4 - 1 = 15$ microoperations

9 bits 46 microoperations

7-21

See Fig. 8-2(b) for control word example.

(a) 16 registers need 4 bits; ALU need 5 bits, and the shifter need 3 bits, to encode all operations.

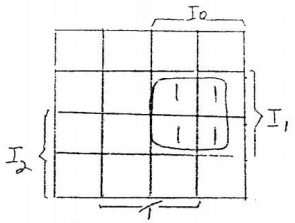
4	4	4	5	3	= 20 bits total
SRC1	SRC2	DEST	ALU	SHIFT	

(c)

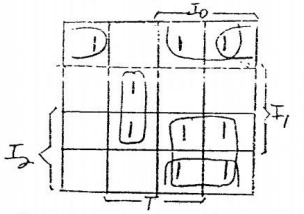
R5	R6	R4	ADD	SHIFT	
0101	0110	0100	00100	000	R4 ← R5+R6

7-22

I_2	I_1	I_0	T	S_1	S_0	L	
0	0	0	0	0	1	0	AD(1)
0	0	0	1	0	0	0	INC(0)
0	0	1	0	0	1	0	AD(1)
0	0	1	1	0	1	0	AD(1)
0	1	0	0	0	0	0	INC(0)
0	1	0	1	0	1	0	AD(1)
0	1	1	0	1	0	0	RET(2)
0	1	1	1	1	0	0	RET(2)
1	0	0	0	0	0	0	INC(0)
1	0	0	1	0	0	0	INC(0)
1	0	1	0	0	1	0	AD(1)
1	0	1	1	0	1	0	AD(1)
1	1	0	0	0	0	0	INC(0)
1	1	0	1	0	1	1	CALL(1)
1	1	1	0	1	1	0	MAP(3)
1	1	1	1	1	1	0	MAP(3)



$S_0 = I_1 I_0$



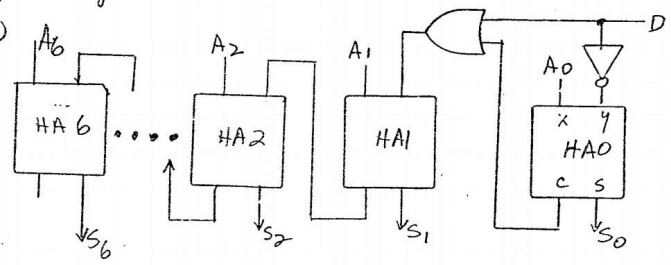
$S_0 = I_2 I_0 + I_1' I_0 + I_1 I_0' T + I_2' I_1' T'$

$L = I_2 I_1 I_0' T$

7-23

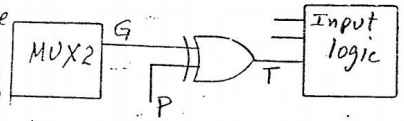
(2) See Fig. 4-8 (chapter 4)

(b)



7-24

P is used to determine the polarity of the selected status bit.



when P=0, T=G because $G \oplus 0 = G$,

When P=1, T=G' because $G \oplus 1 = G'$

Wher G is the value of the setected bit in MUX2

CHAPTER TEN

Computer Arithmetic

IN THIS CHAPTER

10-1 Introduction
10-2 Addition and Subtraction
10-3 Multiplication Algorithms
10-4 Division Algorithms
10-5 Floating-Point Arithmetic Operations
10-6 Decimal Arithmetic Unit
10-7 Decimal Arithmetic Operations

10-1 Introduction

Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems. These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer. The four basic arithmetic operations are addition, subtraction, multiplication, and division. From these four basic operations, it is possible to formulate other arithmetic functions and solve scientific problems by means of numerical analysis methods.

An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in processor registers during the execution of an arithmetic instruction is specified in the definition of the instruction. An arithmetic instruction may specify binary or decimal data, and in each case the data may be in fixed-point or floating-point form. Fixed-point numbers may represent integers or fractions. Negative numbers may be in signed-magnitude or signed-complement representation. The arithmetic processor is very simple if only a binary fixed-point *add* instruction is included. It would be more complicated if it includes all four arithmetic oper-

ations for binary and decimal data in fixed-point and floating-point representation.

At an early age we are taught how to perform the basic arithmetic operations in signed-magnitude representation. This knowledge is valuable when the operations are to be implemented by hardware. However, the designer must be thoroughly familiar with the sequence of steps to be followed in order to carry out the operation and achieve a correct result. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an *algorithm*. An algorithm was stated in Sec. 3-3 for the addition of two fixed-point binary numbers when negative numbers are in signed-2's complement representation. This is a simple algorithm since all it needs for its implementation is a parallel binary adder. When negative numbers are in signed-magnitude representation, the algorithm is slightly more complicated and its implementation requires circuits to add and subtract, and to compare the signs and the magnitudes of the numbers. Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting algorithms is a flowchart. The computational steps are specified in the flowchart inside rectangular boxes. The decision steps are indicated inside diamond-shaped boxes from which two or more alternate paths emerge.

algorithm

In this chapter we develop the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We consider addition, subtraction, multiplication, and division for the following types of data:

1. Fixed-point binary data in signed-magnitude representation
2. Fixed-point binary data in signed-2's complement representation
3. Floating-point binary data
4. Binary-coded decimal (BCD) data

10-2 Addition and Subtraction

As stated in Sec. 3-3, there are three ways of representing negative fixed-point binary numbers: signed-magnitude, signed-1's complement, or signed-2's complement. Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating-point operations, most computers use the signed-magnitude representation for the mantissa. In this section we develop the addition and subtraction algorithms for data represented in signed-magnitude and again for data represented in signed-2's complement.

It is important to realize that the adopted representation for negative numbers refers to the representation of numbers in the registers before and after the execution of the arithmetic operation. It does not mean that complement arithmetic may not be used in an intermediate step. For example, it is convenient to employ complement arithmetic when performing a subtraction operation with numbers in signed-magnitude representation. As long as the initial minuend and subtrahend, as well as the final difference, are in signed-magnitude form the fact that complements have been used in an intermediate step does not alter the fact that the representation is in signed-magnitude.

Addition and Subtraction with Signed-Magnitude Data

The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward. A review of this procedure will be helpful for deriving the hardware algorithm.

We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 10-1. The other columns in the table show the actual operation to be performed with the *magnitude* of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be $+0$ not -0.

magnitude

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words inside parentheses should be used for the subtraction algorithm):

Addition (subtraction) algorithm: when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (identical), compare the magnitudes and

addition (subtraction) algorithm

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

subtract the smaller number from the larger. Choose the sign of the result to be the same as A if $A > B$ or the complement of the sign of A if $A < B$. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm, and vice versa.

Hardware Implementation

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers. Let A and B be two registers that hold the magnitudes of the numbers, and A_s and B_s be two flip-flops that hold the corresponding signs. The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A_s. Thus A and A_s together form an accumulator register.

Consider now the hardware implementation of the algorithms above. First, a parallel-adder is needed to perform the microoperation $A + B$. Second, a comparator circuit is needed to establish if $A > B$, $A = B$, or $A < B$. Third, two parallel-subtractor circuits are needed to perform the microoperations $A - B$ and $B - A$. The sign relationship can be determined from an exclusive-OR gate with A_s and B_s as inputs.

This procedure requires a magnitude comparator, an adder, and two subtractors. However, a different procedure can be found that requires less equipment. First, we know that subtraction can be accomplished by means of complement and add. Second, the result of a comparison can be determined from the end carry after the subtraction. Careful investigation of the alternatives reveals that the use of 2's complement for subtraction and comparison is an efficient procedure that requires only an adder and a complementer.

Figure 10-1 shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_s and B_s. Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added. The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. 4-7 in Chap. 4. The M signal is also applied to the input carry of the adder. When $M = 0$, the output of B is transferred to the adder, the input carry is 0, and the output of

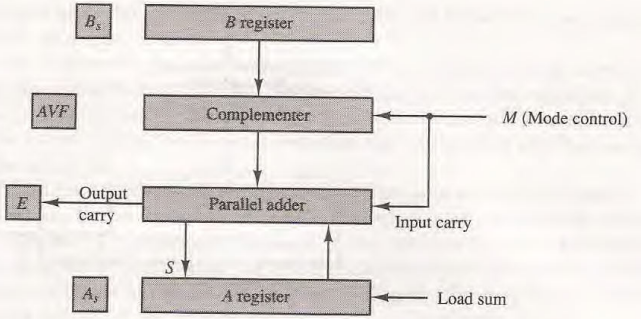


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

the adder is equal to the sum $A + B$. When $M = 1$, the 1's complement of B is applied to the adder, the input carry is 1, and output $S = A + \overline{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction $A - B$.

Hardware Algorithm

The flowchart for the hardware algorithm is presented in Fig. 10-2. The two signs A_s and B_s are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an *add* operation, identical signs dictate that the magnitudes be added. For a *subtract* operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation $EA \leftarrow A + B$, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

The two magnitudes are subtracted if the signs are different for an *add* operation or identical for a *subtract* operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that $A \geq B$ and the number in A is the correct result. If this number is zero, the sign A_s must be made positive to avoid a negative zero. A 0 in E indicates that $A < B$. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation $A \leftarrow \overline{A} + 1$. However, we assume that the A register has circuits for microoperations *complement* and *increment*, so the 2's complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A_s is required. However, when $A < B$, the sign of the result is the complement of the original sign of A. It is then necessary to complement A_s to obtain

complement and increment

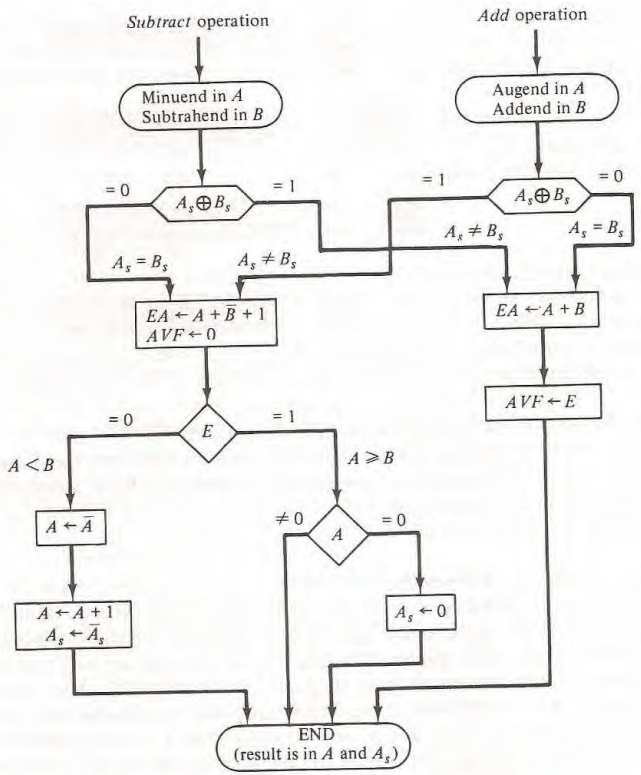


Figure 10-2 Flowchart for add and subtract operations.

the correct sign. The final result is found in register A and its sign in A_s. The value in AVF provides an overflow indication. The final value of E is immaterial.

Addition and Subtraction with Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are introduced in Sec. 3-3. They are summarized here for easy reference. The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative. If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented

as 00100001 and −33 as 11011111. Note that 11011111 is the 2's complement of 00100001, and vice versa.

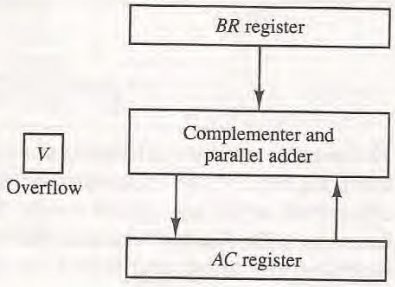
The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. The effect of an overflow on the sum of two signed-2's complement numbers is discussed in Sec. 3-3. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is shown in Fig. 10-3. This is the same configuration as in Fig. 10-1 except that the sign bits are not separated from the rest of the registers. We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.

The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart of Fig. 10-4. The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

Figure 10-3 Hardware for signed-2's complement addition and subtraction.



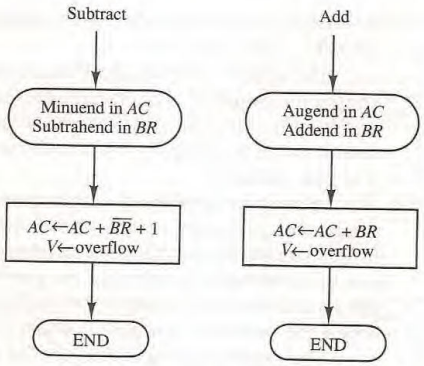


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2's complement representation. For this reason most computers adopt this representation over the more familiar signed-magnitude.

10-3 Multiplication Algorithms

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

```
23      10111     Multiplicand
19    × 10011     Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101     Product
```

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

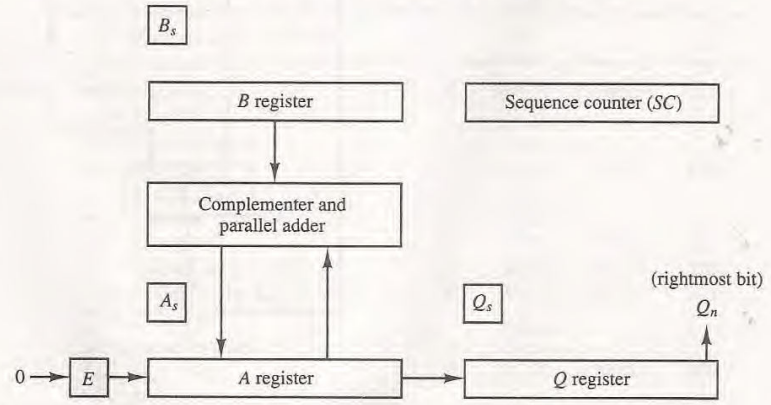
Hardware Implementation for Signed-Magnitude Data

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

The hardware for multiplication consists of the equipment shown in Fig. 10-1 plus two more registers. These registers together with registers A and B are shown in Fig. 10-5. The multiplier is stored in the Q register and its sign in Q_s. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 10-5. The

Figure 10-5 Hardware for multiply operation.

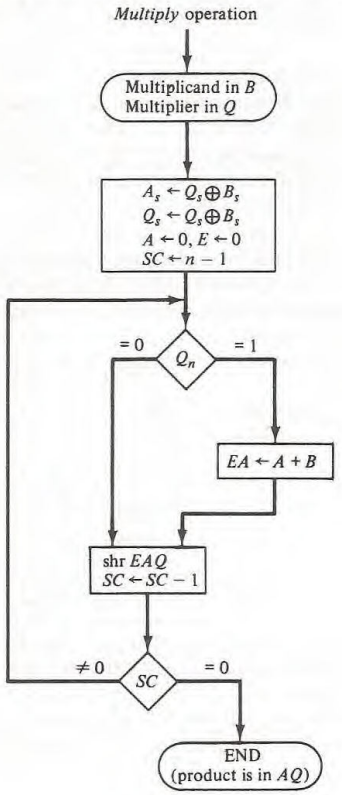


least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm

Figure 10-6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively. The signs are compared, and both A and Q are set to

Figure 10-6 Flowchart for multiply operation.



correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of $n - 1$ bits.

After the initialization, the low-order bit of the multiplier in Q_n is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when $SC = 0$. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

The previous numerical example is repeated in Table 10-2 to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart.

Booth Multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001110 (+14) has a string of 1's from 2^3 to 2^1

TABLE 10-2 Numerical Example for Binary Multiplier

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

($k = 3, m = 1$). The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

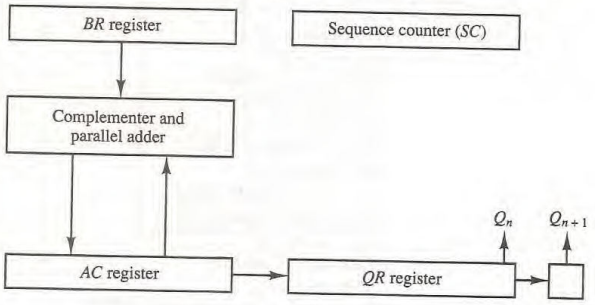
As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

The algorithm works for positive or negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 10-7. This is similar to Fig. 10-5 except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register QR. An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. 10-8. AC and the appended

Figure 10-7 Hardware for Booth algorithm.



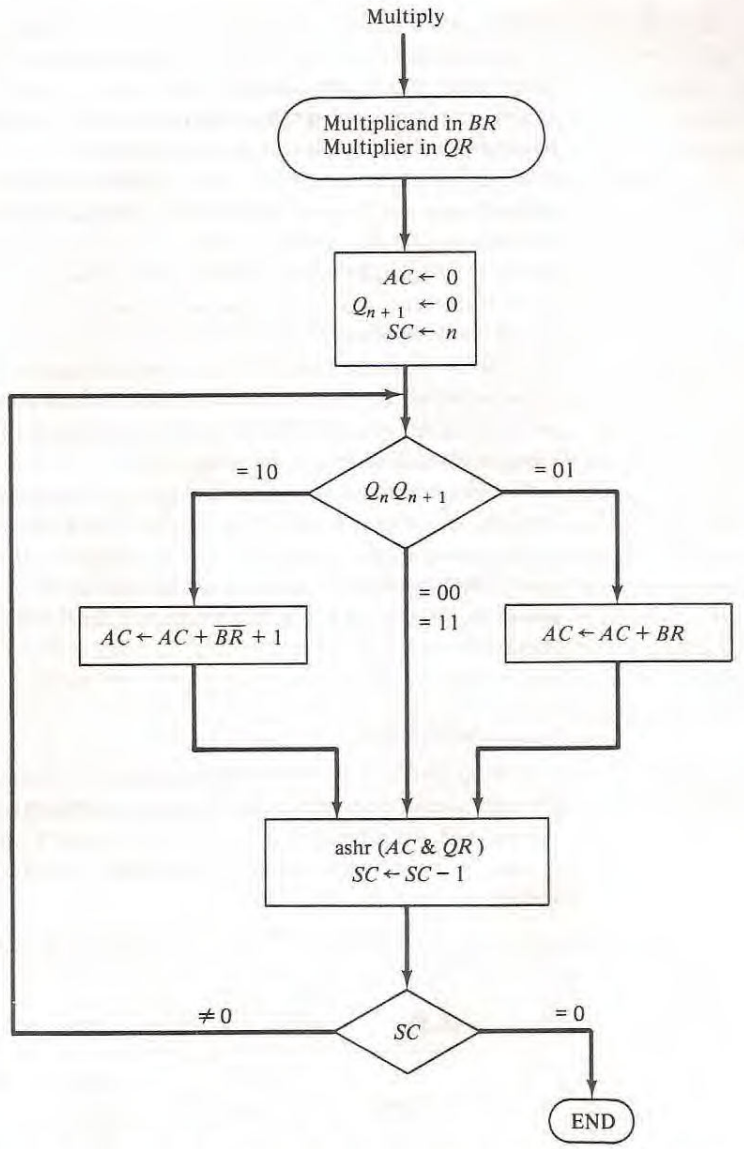


Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged (see Sec. 4-6). The sequence counter is decremented and the computational loop is repeated n times.

A numerical example of Booth algorithm is shown in Table 10-3 for $n = 5$. It shows the step-by-step multiplication of $(-9) \times (-13) = +117$. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive. The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations. The multiplication of two binary numbers can be done with one microoperation by means of a combinational circuit that forms the product bits all

TABLE 10-3 Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

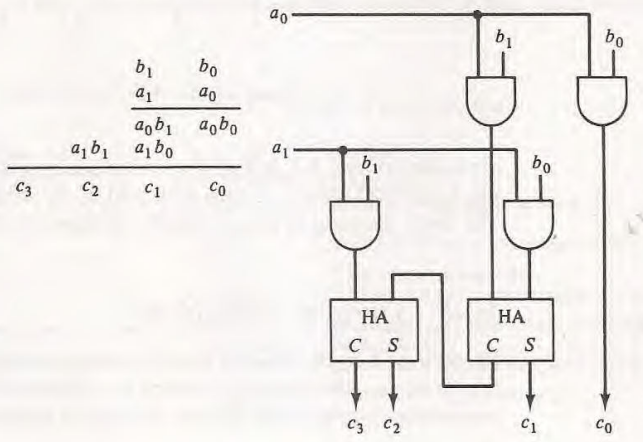
at once. This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 10-9. The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0, and the product is $c_3 c_2 c_1 c_0$. The first partial product is formed by multiplying a_0 by $b_1 b_0$. The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by $b_1 b_0$ and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need $j \times k$ AND gates and $(j - 1)$ k-bit adders to produce a product of $j + k$ bits.

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be

Figure 10-9 2-bit by 2-bit array multiplier.



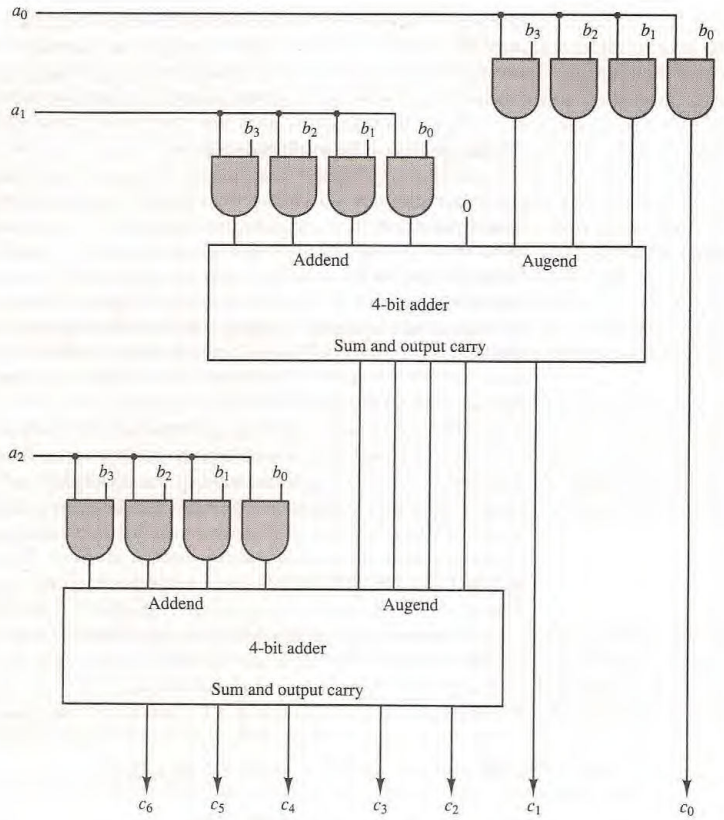


Figure 10-10 4-bit by 3-bit array multiplier.

represented by $b_3 b_2 b_1 b_0$ and the multiplier by $a_2 a_1 a_0$. Since $k = 4$ and $j = 3$, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig. 10-10.

10-4 Division Algorithms

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division be-

cause the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Fig. 10-11. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a *partial remainder* because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

partial remainder

Hardware Implementation for Signed-Magnitude Data

When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry.

The hardware for implementing the division operation is identical to that required for multiplication and consists of the components shown in Fig. 10-5. Register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E lost. The numerical example is repeated in Fig. 10-12 to clarify the

Figure 10-11 Example of binary division.

Divisor:	11010	Quotient = Q
B = 10001	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 10-12 Example of binary division with digital hardware.

proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If $E = 1$, it signifies that $A \geq B$. A quotient bit 1 is inserted into Q_n and the partial remainder is shifted to the left to repeat the process. If $E = 0$, it signifies that $A < B$ so the quotient in Q_n remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs

are alike, the sign of the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Fig. 10-11 we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

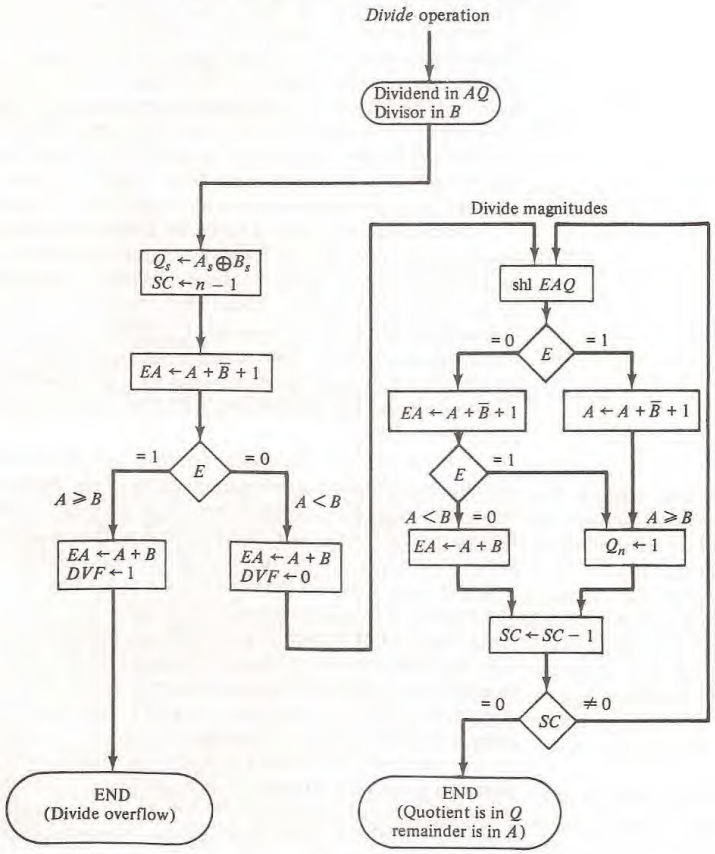
When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it *DVF*.

The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if *DVF* is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow. In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a *divide stop*. Stopping the operation of the computer is not recommended because it is time consuming. The procedure in most computers is to provide an interrupt request when *DVF* is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. It is then the responsibility of the user who wrote the program to rescale the data or take any other corrective measure. The best way to avoid a divide overflow is to use floating-point data. We will see in Sec. 10-5 that a divide overflow can be handled very simply if numbers are in floating-point representation.

Hardware Algorithm

The hardware divide algorithm is shown in the flowchart of Fig. 10-13. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Q_s to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has

Figure 10-13 Flowchart for divide operation.



words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of $n-1$ bits.

A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If $A \geq B$, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If $A < B$, no divide overflow occurs so the value of the dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that $EA > B$ because EA consists of a 1 followed by $n-1$ bits while B consists of only $n-1$ bits. In this case, B must be subtracted from EA and 1 inserted into Q_n for the quotient bit. Since register A is missing the high-order bit of the dividend (which is in E), its value is $EA - 2^{n-1}$. Adding to this value the 2's complement of B results in

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

The carry from this addition is not transferred to E if we want E to remain a 1.

If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If $E = 1$, it signifies that $A \geq B$; therefore, Q_n is set to 1. If $E = 0$, it signifies that $A < B$ and the original number is restored by adding B to A. In the latter case we leave a 0 in Q_n (0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After $n - 1$ times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Q_s and the sign of the remainder in A_s is the same as the original sign of the dividend.

Other Algorithms

restoring method

The hardware method just described is called the *restoring method*. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference. Two other methods are available for dividing numbers, the *comparison* method and the *nonrestoring* method. In the comparison method A and B are compared *prior* to the subtraction operation. Then if $A \geq B$, B is subtracted from A. If $A < B$ nothing is done. The partial remainder is shifted left and the numbers are compared again. The comparison can be determined prior to the subtraction by inspecting the end-carry out of the parallel-adder prior to its transfer to register E.

comparison and nonrestoring method

In the nonrestoring method, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. To see why this is possible consider the case when $A < B$. From the flowchart in Fig. 9-11 we note that the operations performed are $A - B + B$; that is, B is sub-

tracted and then added to restore A. The next time around the loop, this number is shifted left (or multiplied by 2) and B subtracted again. This gives $2(A - B + B) - B = 2A - B$. This result is obtained in the nonrestoring method by leaving $A - B$ as is. The next time around the loop, the number is shifted left and B *added* to give $2(A - B) + B = 2A - B$, which is the same as before. Thus, in the nonrestoring method, B is subtracted if the previous value of Q_n was a 1, but B is added if the previous value of Q_n was a 0 and no restoring of the partial remainder is required. This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.

10-5 Floating-Point Arithmetic Operations

integer declaration statement

Many high-level programming languages have a facility for specifying floating-point numbers. The most common way is to specify them by a *real* declaration statement as opposed to fixed-point numbers, which are specified by an *integer* declaration statement. Any computer that has a compiler for such high-level programming language must have a provision for handling floating-point arithmetic operations. The operations are quite often included in the internal hardware. If no hardware is available for the operations, the compiler must be designed with a package of floating-point software subroutines. Although the hardware method is more expensive, it is so much more efficient than the software method that floating-point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations

Floating-point representation of data was introduced in Sec. 3-4. A floating-point number in computer registers consists of two parts: a mantissa m and an exponent e. The two parts represent a number obtained from multiplying m times a radix r raised to the value of e; thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix r are assumed and are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with $m = 53725$ and $e = 3$ and is interpreted to represent the floating-point number

$$.53725 \times 10^3$$

A floating-point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of significant digits. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be $\pm(2^{47} - 1)$, which is approximately $\pm 10^{14}$. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be accommodated is

$$\pm(1 - 2^{-35}) \times 2^{2047}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and the fact that $2^{11} - 1 = 2047$. The largest number that can be accommodated is approximately 10^{615}, which is a very large number. The mantissa can accommodate 35 bits (excluding the sign) and if considered as an integer it can store a number as large as $(2^{35} - 1)$. This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer may use four words to represent one floating-point number. One word of 8 bits is reserved for the exponent and the 24 bits of the other three words are used for the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers and their execution takes longer and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

$$.5372400 \times 10^2$$
$$+ .1580000 \times 10^{-1}$$

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has

the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added:

$$\begin{array}{r} .5372400 \times 10^2 \\ +.0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -.56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

A floating-point number that has a 0 in the most significant position of the mantissa is said to have an *underflow*. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

The operations performed with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed-1's complement.

A fourth representation employed in many computers is known as a *biased* exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from −50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number $e + 50$, where e is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range from 49 to 00. The subtraction of 50 gives the negative values in the range of −1 to −50.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

In the examples above, we used decimal numbers to demonstrate some of the concepts that must be understood when dealing with floating-point numbers. Obviously, the same concepts apply to binary numbers as well. The algorithms developed in this section are for binary numbers. Decimal computer arithmetic is discussed in the next section.

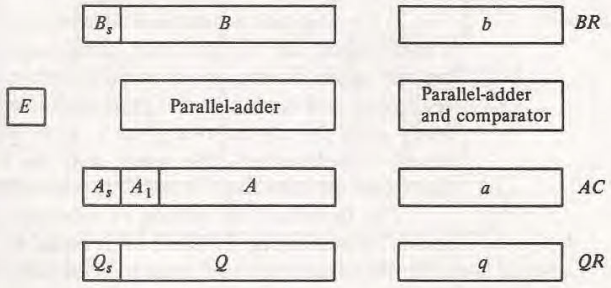
Register Configuration

The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 10-14. There are three registers, *BR*, *AC*, and *QR*. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.

It is assumed that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the *AC* has a mantissa

Figure 10-14 Registers for floating-point arithmetic operations.



whose sign is in A_s and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A_1. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A, and a.

Similarly, register BR is subdivided into B_s, B, and b, and QR into Q_s, Q, and q. A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a *fraction*, so the binary point is assumed to reside to the left of the magnitude part. Integer representation for floating-point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands coming from and going to the memory unit are always normalized.

Addition and Subtraction

During addition or subtraction, the two floating-point operands are in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

A floating-point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be unnormalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.

The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer

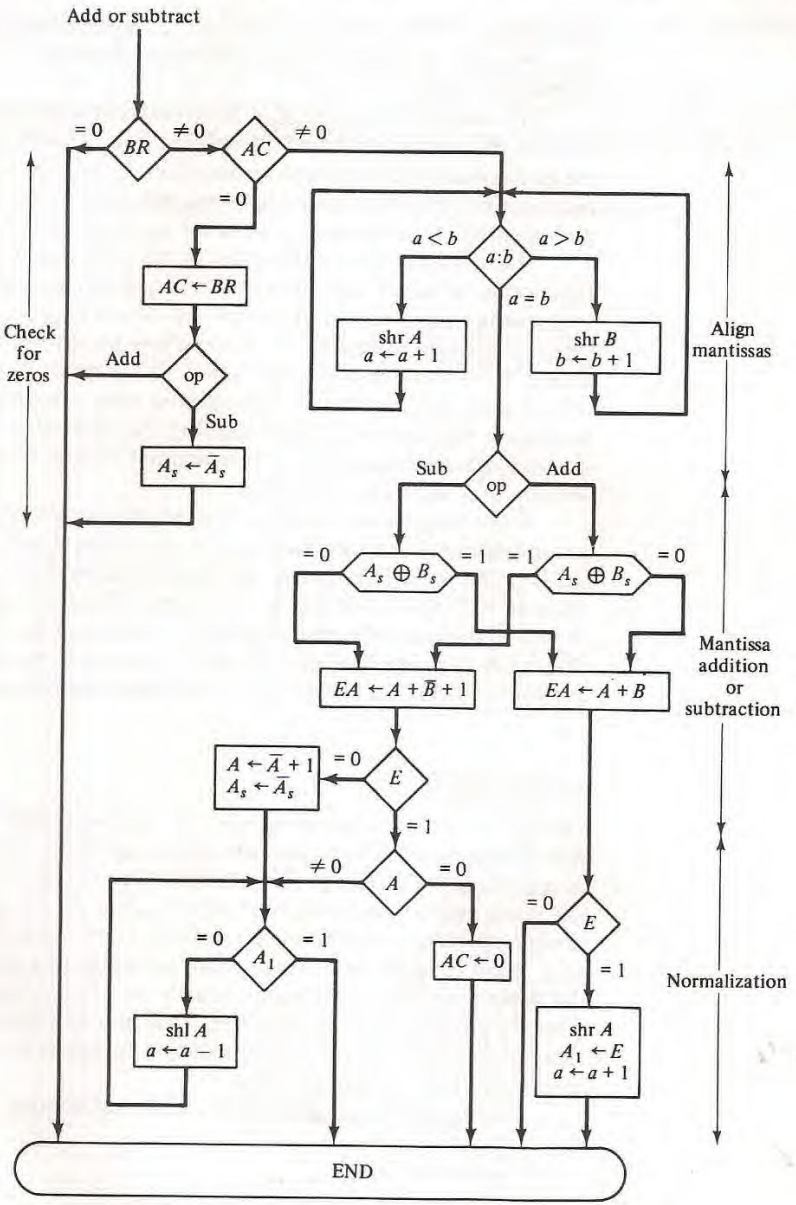


Figure 10-15 Addition and subtraction of floating-point numbers.

the content of *BR* into *AC* and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.

The magnitude comparator attached to exponents *a* and *b* provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in Fig. 10-2. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop *E*. If *E* is equal to 1, the bit is transferred into A_1 and all other bits of *A* are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the *AC* is made zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A_1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A_1 is checked again and the process is repeated until it is equal to 1. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

Multiplication

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product. The double-precision answer is used in fixed-point numbers to increase the accuracy of the product. In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.

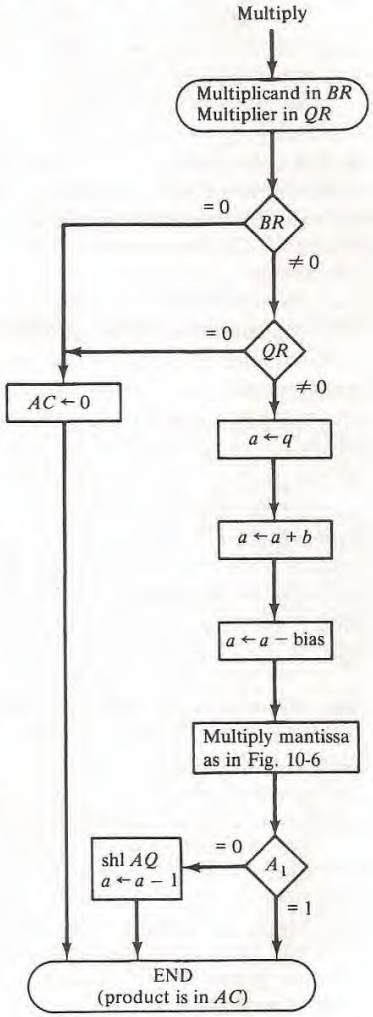
The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.

Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.

The flowchart for floating-point multiplication is shown in Fig. 10-16. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the *AC* is set to zero and the operation is

Figure 10-16 Multiplication of floating-point numbers.



terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q. Overflow cannot occur during multiplication, so there is no need to check for it.

The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur.

Although the low-order half of the mantissa is in Q, we do not use it for the floating-point product. Only the value in the AC is taken as the product.

Division

Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction and not an integer. For integer representation, a single-precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

The check for divide-overflow is the same as in fixed-point representation. However, with floating-point numbers the divide-overflow imposes no problems. If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. For normalized operands this is a sufficient operation to ensure that no mantissa divide-overflow will occur. The operation above is referred to as a *dividend alignment*.

dividend alignment

The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require a normalization.

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.

3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

The flowchart for floating-point division is shown in Fig. 10-17. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q_s. The sign of the dividend in A_s is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If $A \geq B$, it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that $A < B$.

Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into q because the quotient is formed in QR.

The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies $(n - 1)$ positions to the left of A_1. The remainder can be converted to a normalized fraction by subtracting $n - 1$ from the dividend exponent and by shift and decrement until the bit in A_1 is equal to 1. This is not shown in the flow chart and is left as an exercise.

10-6 Decimal Arithmetic Unit

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and

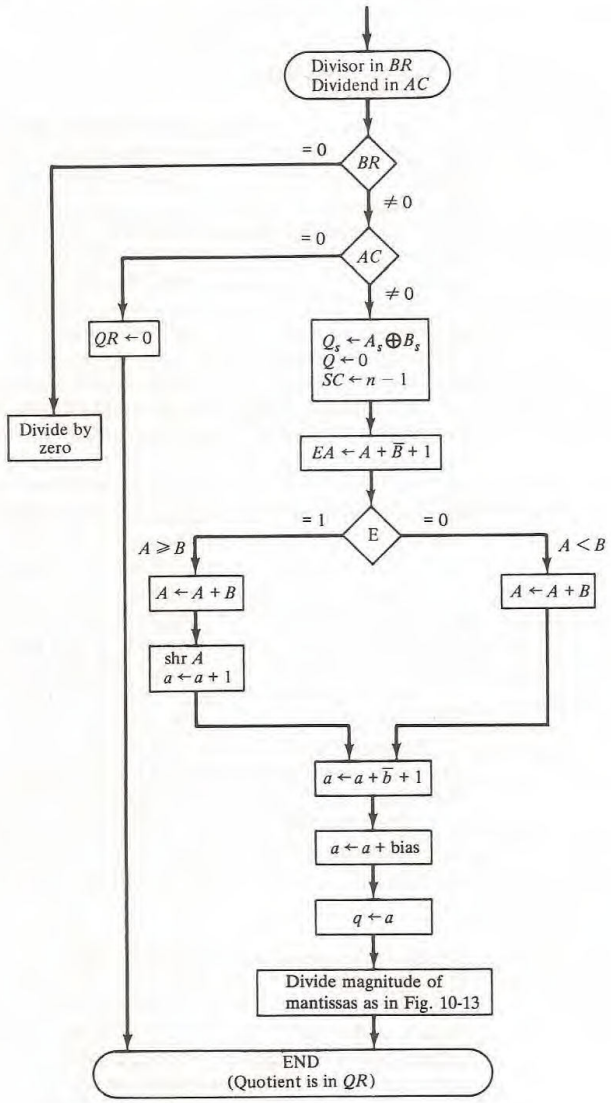


Figure 10-17 Division of floating-point numbers.

output data. When the application calls for a large amount of input–output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations.

Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, since this process requires special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data. Users can specify by programmed instructions whether they want the computer to perform calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code. A single-stage decimal arithmetic unit consists of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the augend digit, four inputs for the addend digit, and an input-carry. The outputs include four terminals for the sum digit and one for the output-carry. Of course, there is a wide variety of possible circuit configurations dependent on the code used to represent the decimal digits.

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal* numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical

TABLE 10-4 Derivation of BCD Adder

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001 which also have a 1 in position Z_8, we specify further that either Z_4

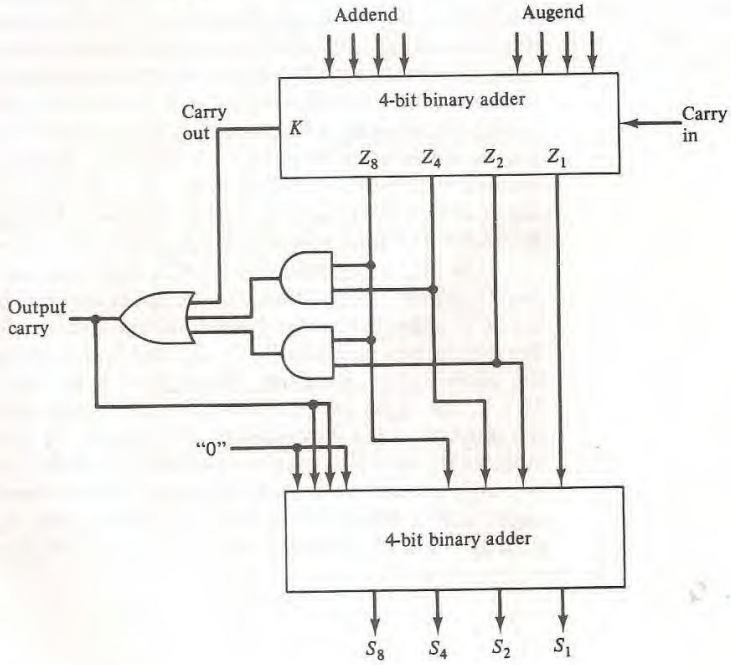
or Z_2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

Figure 10-18 Block diagram of BCD adder.



A decimal parallel-adder that adds n decimal digits needs n BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage. To achieve shorter propagation delays, BCD adders include the necessary circuits for carry look-ahead. Furthermore, the adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

BCD Subtraction

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, the 9's complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of a 4-bit binary number N is identical to the subtraction of the number from 1111 (decimal 15). Adding the binary equivalent of decimal 10 gives $15 - N + 10 = 9 - N + 16$. But 16 signifies the carry that is discarded, so the result is $9 - N$ as required. Adding the binary equivalent of decimal 6 and then complementing gives $15 - (N + 6) = 9 - N$ as required.

The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, and B_1. Let M be a mode bit that controls the add/subtract operation. When $M = 0$, the two digits are added; when $M = 1$, the digits are subtracted. Let the binary variables x_8, x_4, x_2, and x_1 be the outputs of the 9's complementer circuit. By an examination of the truth table for the circuit, it may be observed (see Prob. 10-30) that B_1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x_4 is 1 when the exclusive-OR of B_2 and B_4 is 1; and x_8 is 1 when $B_8 B_4 B_2 = 000$. The Boolean functions for the 9's complementer circuit are

$$x_1 = B_1 M' + B_1' M$$
$$x_2 = B_2$$

$$x_4 = B_4 M' + (B_4' B_2 + B_4 B_2')M$$
$$x_8 = B_8 M' + B_8' B_4' B_2' M$$

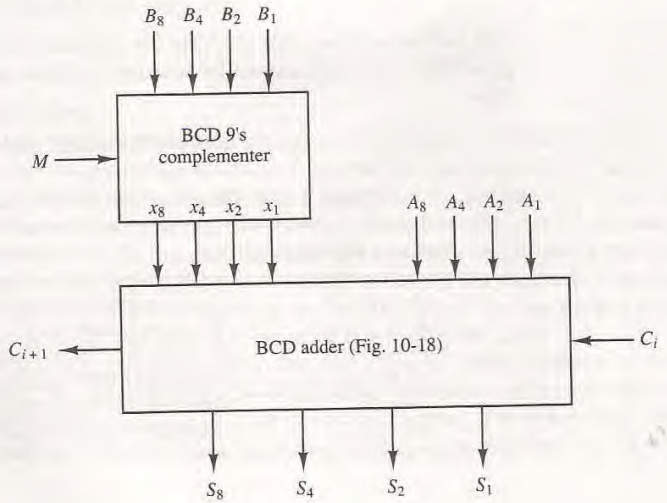
From these equations we see that $x = B$ when $M = 0$. When $M = 1$, the x outputs produce the 9's complement of B.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 10-19. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With $M = 0$, the S outputs form the sum of A and B. With $M = 1$, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carry C_{i+1} from one stage must be connected to the input carry C_i of the next-higher-order stage. The best way to subtract the two decimal numbers is to let $M = 1$ and apply a 1 to the input carry C_1 of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.

10-7 Decimal Arithmetic Operations

The algorithms for arithmetic operations with decimal data are similar to the algorithms for the corresponding operations with binary data. In fact, except for a slight modification in the multiplication and division algorithms, the same

Figure 10-19 One stage of a decimal arithmetic unit.



flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit and must be taken as a unit when performing decimal microoperations.

For convenience, we will use the same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in Table 10-5, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus, for decimal numbers, the symbol $A \leftarrow A + \overline{B} + 1$ denotes a transfer of the decimal sum formed by adding the original content A to the 10's complement of B. The use of identical symbols for the 9's complement and the 1's complement may be confusing if both types of data are employed in the same system. If this is the case, it may be better to adopt a different symbol for the 9's complement. If only one type of data is being considered, the symbol would apply to the type of data used.

Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 10 states from 0000 to 1001 and back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000.

A decimal shift right or left is preceded by the letter d to indicate a shift over the four bits that hold the decimal digits. As a numerical illustration consider a register A holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

0111 1000 0110 0000

The microoperation *dshr A* shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

0000 0111 1000 0110

TABLE 10-5 Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

Addition and Subtraction

The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the microoperation symbols in the proper manner. Similarly, the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9's complement of the subtrahend as shown in Fig. 10-19.

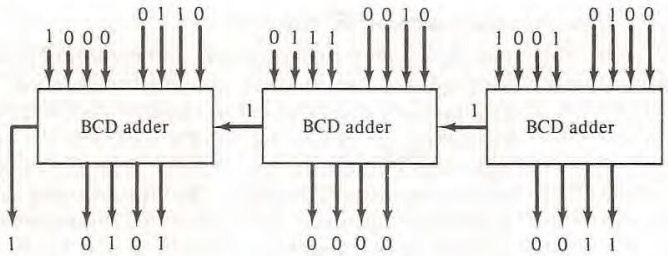
Decimal data can be added in three different ways, as shown in Fig. 10-20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time. For k decimal digits, this configuration requires k microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. This correction, discussed in Sec. 10-6, consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits. It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.

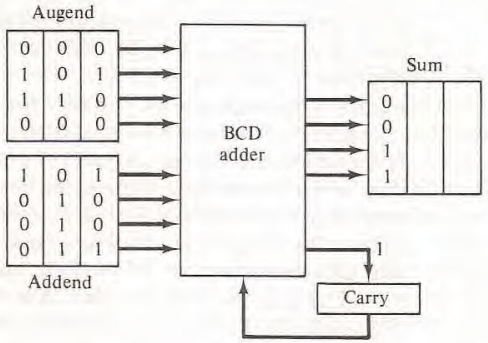
Multiplication

The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits. In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product. This operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit.

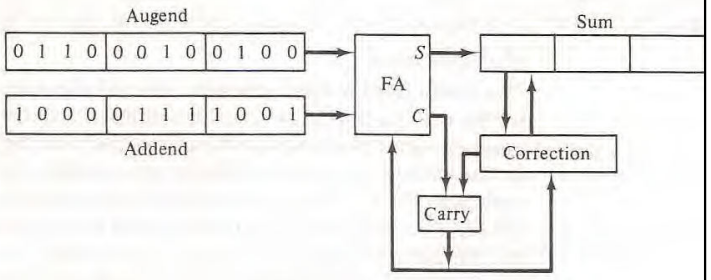
The registers organization for the decimal multiplication is shown in Fig. 10-21. We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, A, B, and Q, each having a corresponding sign flip-flop A_s, B_s, and Q_s.



(a) Parallel decimal addition: 624 + 879 = 1503



(b) Digit-serial, bit-parallel decimal addition



(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.

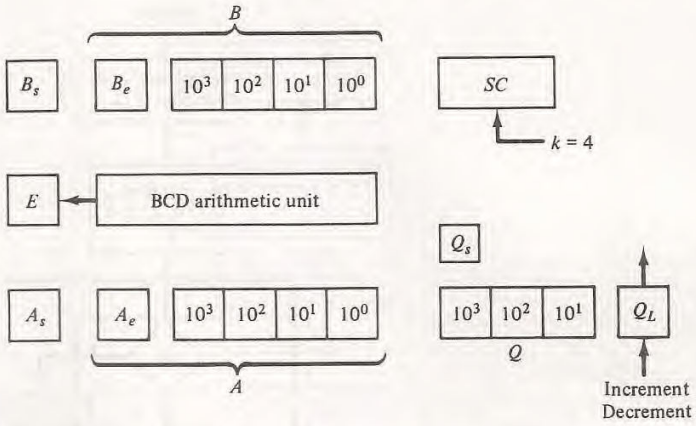


Figure 10-21 Registers for decimal arithmetic multiplication and division.

Registers A and B have four more bits designated by A_e and B_e that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit A register. The end-carry goes to flip-flop E. The purpose of digit A_e is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit B_e is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register Q is denoted by Q_L. This digit can be incremented or decremented.

A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to B_s and the magnitude of the operand is placed in the lower 16 bits of B. Both B_e and A_e are cleared initially. The result of the operation is also 17 bits long and does not use the A_e part of the A register.

The decimal multiplication algorithm is shown in Fig. 10-22. Initially, the entire A register and B_e are cleared and the sequence counter SC is set to a number k equal to the number of digits in the multiplier. The low-order digit of the multiplier in Q_L is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and Q_L is decremented. Q_L is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in A_e and can range in value from 0 to 9.

Next, the partial product and the multiplier are shifted once to the right. This places zero in A_e and transfers the next multiplier quotient into Q_L. The process is then repeated k times to form a double-length product in AQ.

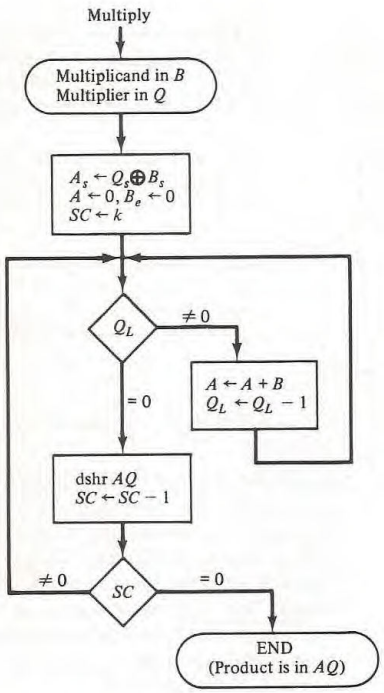


Figure 10-22 Flowchart for decimal multiplication.

Division

Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

The decimal division algorithm is shown in Fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in A_e. The divisor is then subtracted by adding its 10's complement value. Since B_e is initially cleared, its complement value is 9 as required. The carry in E determines the relative magnitude of A and B. If $E = 0$, it signifies

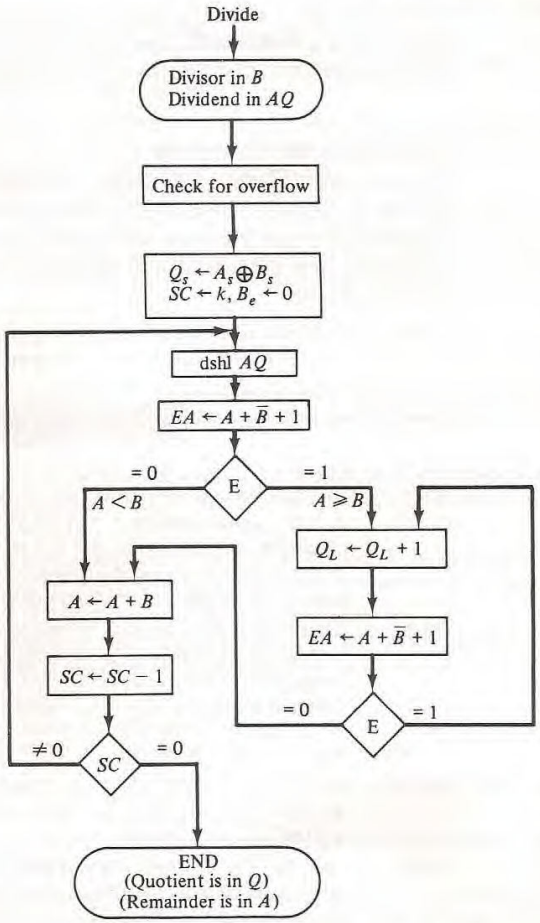


Figure 10-23 Flowchart for decimal division.

that $A < B$. In this case the divisor is added to restore the partial remainder and Q_L stays at 0 (inserted there during the shift). If $E = 1$, it signifies that $A \geq B$. The quotient digit in Q_L is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0. When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder "goes" into the divisor.

The partial remainder and the quotient bits are shifted once to the left and the process is repeated k times to form k quotient digits. The remainder is then found in register A and the quotient is in register Q. The value of E is neglected.

Floating-Point Operations

Decimal floating-point arithmetic operations follow the same procedures as binary operations. The algorithms in Sec. 10-5 can be adopted for decimal data provided that the microoperation symbols are interpreted correctly. The multiplication and division of the mantissas must be done by the methods described above.

PROBLEMS

10-1. The complementer shown in Fig. 10-1 is not needed if instead of performing $A + \overline{B} + 1$ we perform $B + \overline{A}$ (B plus the 1's complement of A). Derive an algorithm in flowchart form for addition and subtraction of fixed-point binary numbers in signed-magnitude representation with the magnitudes subtracted by the two microoperations $A \leftarrow \overline{A}$ and $EA \leftarrow A + B$.

10-2. Mark each individual path in the flowchart of Fig. 10-2 by a number and then indicate the overall path that the algorithm takes when the following signed-magnitude numbers are computed. In each case give the value of AVF. The leftmost bit in the following numbers represents the sign bit.

a. 0 101101 + 0 011111
b. 1 011111 + 1 101101
c. 0 101101 − 0 011111
d. 0 101101 − 0 101101
e. 1 011111 − 0 101101

10-3. Perform the arithmetic operations below with binary numbers and with negative numbers in signed-2's complement representation. Use seven bits to accommodate each number together with its sign. In each case, determine if there is an overflow by checking the carries into and out of the sign bit position.

a. (+35) + (+40)
b. (−35) + (−40)
c. (−35) − (+40)

10-4. Consider the binary numbers when they are in signed-2's complement representation. Each number has n bits: one for the sign and $k = n - 1$ for the magnitude. A negative number $-X$ is represented as $2^k + (2^k - X)$, where the first 2^k designates the sign bit and $(2^k - X)$ is the 2's complement of X. A positive number is represented as $0 + X$, where the 0 designates the sign bit, and X, the k-bit magnitude. Using these generalized symbols, prove that the sum $(\pm X) + (\pm Y)$ can be formed by adding the numbers including their sign bits and discarding the carry-out of the sign-bit position. In other words, prove the algorithm for adding two binary numbers in signed-2's complement representation.

10-5. Formulate a hardware procedure for detecting an overflow by comparing the sign of the sum with the signs of the augend and addend. The numbers are in signed-2's complement representation.

10-6. **a.** Perform the operation $(-9) + (-6) = -15$ with binary numbers in signed-1's complement representation using only five bits to represent each number (including the sign). Show that the overflow detection procedure of checking the inequality of the last two carries fails in this case.
b. Suggest a modified procedure for detecting an overflow when signed-1's complement numbers are used.

10-7. Derive an algorithm in flowchart form for adding and subtracting two fixed-point binary numbers when negative numbers are in signed-1's complement representation.

10-8. Prove that the multiplication of two n-digit numbers in base r gives a product no more than $2n$ digits in length. Show that this statement implies that no overflow can occur in the multiplication operation.

10-9. Show the contents of registers E, A, Q, and SC (as in Table 10-2) during the process of multiplication of two binary numbers, 11111 (multiplicand) and 10101 (multiplier). The signs are not included.

10-10. Show the contents of registers E, A, Q, and SC (as in Fig. 10-12) during the process of division of (a) 10100011 by 1011; (b) 00001111 by 0011. (Use a dividend of eight bits.)

10-11. Show that adding B after the operation $A + \overline{B} + 1$ restores the original value of A. What should be done with the end carry?

10-12. Why should the sign of the remainder after a division be the same as the sign of the dividend?

10-13. Design an array multiplier that multiplies two 4-bit numbers. Use AND gates and binary adders.

10-14. Show the step-by-step multiplication process using Booth algorithm (as in Table 10-3) when the following binary numbers are multiplied. Assume 5-bit registers that hold signed numbers. The multiplicand in both cases is +15.

a. (+15) × (+13)
b. (+15) × (−13)

10-15. Derive an algorithm in flowchart form for the nonrestoring method of fixed-point binary division.

10-16. Derive an algorithm for evaluating the square root of a binary fixed-point number.

10-17. A binary floating-point number has seven bits for a biased exponent. The constant used for the bias is 64.

a. List the biased representation of all exponents from −64 to +63.

b. Show that a 7-bit magnitude comparator can be used to compare the relative magnitude of the two exponents.
c. Show that after the addition of two biased exponents it is necessary to subtract 64 in order to have a biased exponents sum. How would you subtract 64 by adding its 2's complement value?
d. Show that after the subtraction of two biased exponents it is necessary to add 64 in order to have a biased exponent difference.

10-18. Derive an algorithm in flowchart form for the comparison of two signed binary numbers when negative numbers are in signed-2's complement representation:
a. By means of a subtraction operation with the signed-2's complement numbers.
b. By scanning and comparing pairs of bits from left to right.

10-19. Repeat Prob. 10-18 for signed-magnitude binary numbers.

10-20. Let n be the number of bits of the mantissa in a binary floating-point number. When the mantissas are aligned during the addition or subtraction, the exponent difference may be greater than $n - 1$. If this occurs, the mantissa with the smaller exponent is shifted entirely out of the register. Modify the mantissa alignment in Fig. 10-15 by including a sequence counter SC that counts the number of shifts. If the number of shifts is greater than $n - 1$, the larger number is then used to determine the result.

10-21. The procedure for aligning mantissas during addition or subtraction of floating-point numbers can be stated as follows: Subtract the smaller exponent from the larger and shift right the mantissa having the smaller exponent a number of places equal to the difference between the exponents. The exponent of the sum (or difference) is equal to the larger exponents. Without using a magnitude comparator, assuming biased exponents, and taking into account that only the AC can be shifted, derive an algorithm in flowchart form for aligning the mantissas and placing the larger exponent in the AC.

10-22. Show that there can be no mantissa overflow after a multiplication operation.

10-23. Show that the division of two normalized floating-point numbers with fractional mantissas will always result in a normalized quotient provided a dividend alignment is carried out prior to the division operation.

10-24. Extend the flowchart of Fig. 10-17 to provide a normalized floating-point remainder in the AC. The mantissa should be a fraction.

10-25. The algorithms for the floating-point arithmetic operations in Sec. 10-5 neglect the possibility of exponent overflow or underflow.
a. Go over the three flowcharts and find where an exponent overflow may occur.
b. Repeat (a) for exponent underflow. An exponent underflow occurs if the exponent is more negative than the smallest number that can be accommodated in the register.
c. Show how an exponent overflow or underflow can be detected by the hardware.

10-26. If we assume integer representation for the mantissa of floating-point numbers, we encounter certain scaling problems during multiplication and division. Let the number of bits in the magnitude part of the mantissa be $(n - 1)$. For integer representation:
a. Show that if a single-precision product is used, $(n - 1)$ must be added to the exponent product in the AC.
b. Show that if a single-precision mantissa dividend is used, $(n - 1)$ must be subtracted from the exponent dividend when Q is cleared.

10-27. Show the hardware to be used for the addition and subtraction of two decimal numbers in signed-magnitude representation. Indicate how an overflow is detected.

10-28. Show that 673 − 356 can be computed by adding 673 to the 10's complement of 356 and discarding the end carry. Draw the block diagram of a three-stage decimal arithmetic unit and show how this operation is implemented. List all input bits and output bits of the unit.

10-29. Show that the lower 4-bit binary adder in Fig. 10-1 can be replaced by one full-adder and two half-adders.

10-30. Using combinational circuit design techniques, derive the Boolean functions for the BCD 9's complementer of Fig. 10-19. Draw the logic diagram.

10-31. It is necessary to design an adder for two decimal digits represented in the excess-3 code (Table 3-6). Show that the correction after adding two digits with a 4-bit binary adder is as follows:
a. The output carry is equal to the uncorrected carry.
b. If output carry = 1, add 0011.
c. If output carry = 0, add 1101 and ignore the carry from this addition. Show that the excess-3 adder can be constructed with seven full-adders and two inverters.

10-32. Derive the circuit for a 9's complementer when decimal digits are represented in the excess-3 code (Table 3-6). A mode control input determines whether the digit is complemented or not. What is the advantage of using this code over BCD?

10-33. Show the hardware to be used for the addition and subtraction of two decimal numbers with negative numbers in signed-10's complement representation. Indicate how an overflow is detected. Derive the flowchart algorithm and try a few numbers to convince yourself that the algorithm produces correct results.

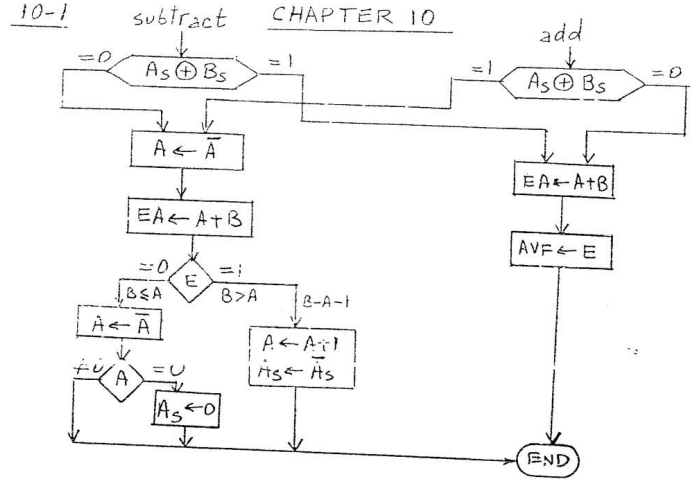
10-34. Show the content of registers A, B, Q, and SC during the decimal multiplication (Fig. 10-22) of (a) 470 × 152 and (b) 999 × 199. Assume three-digit registers and take the second number as the multiplier.

10-35. Show the content of registers A, E, Q, and SC during the decimal division (Fig. 10-23) of 1680/32. Assume two-digit registers.

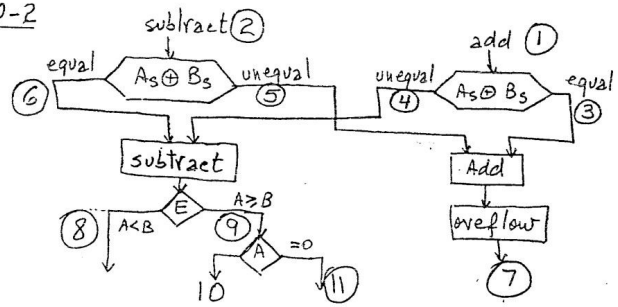
10-36. Show that subregister A_e in Fig. 10-21 is zero at the termination of (a) the decimal multiplication as specified in Fig. 10-22, and (b) the decimal division as specified in Fig. 10-23.

10-37. Change the floating-point arithmetic algorithms in Sec. 10-5 from binary to decimal data. In a table, list how each microoperation symbol should be interpreted.

10-1 subtract CHAPTER 10



10-2



$2^6 - 1 = 63$, Overflow if sum greater than |63|

(a) (+45) + (+31) = 76 ① ③ ⑦ ← Path. AVF = 1

(b) (-31) + (-45) = -76 ① ③ ⑦ AVF = 1

(c) (+45) - (+31) = 14 ② ⑥ ⑨ ⑩ AVF = 0

(d) (+45) - (+45) = 0 ② ⑥ ⑨ ⑪ AVF = 0

(e) (-31) - (+45) = -76 ② ⑤ ⑦ AVF = 1

10-3

(a) +35 0 100011 (b) -35 1 011101

+40 0 101000 -40 1 011000

+75 1 001011 -70 0 110101

F=0 E=1 ← Carries → F=1 E=0

$F \oplus E = 1$; overflow $F \oplus E = 1$; overflow

10-4

case	(a) operation in sign-magnitude	(b) operation in sign-2's complement	(c) required result in sign-2's complement
1.	(+X) + (+Y)	(0+X)+(0+Y)	0+(X+Y)
2.	(+X) + (-Y)	(0+X)+2^k+($\bar{2^k}$Y)	0+(X-Y) if X≥Y $2^k+2^k-(Y-X)$ if X<Y
3.	(-X) + (+Y)	2^k+($\bar{2^k}$-X)+(0+Y)	0+(Y-X) if Y≥X $2^k+2^k-(X-Y)$ if Y<X
4.	(-X)+(-Y)	(2^k+2^k-X)+(2^k+2^k-Y)	$2^k+2^k-(X+Y)$

It is necessary to show that the operations in column (b) produce the results listed in column (c).

case 1. column (b) = column (c)

case 2. If X≥Y then (X-Y)≥0 and consists of k bits. operation in column (b) gives: 2^{2k}+(X-Y). Discard carry $2^{2k} = 2^n$ to get 0+(X-Y) as in column (c)

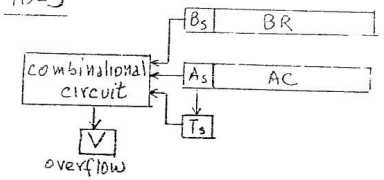
If X<Y then (Y-X)>0. Operation gives $2^k + 2^k - (Y-X)$ as in colum (c).

case 3. is the same as case 2 with X and Y reversed

case 4. Operation in column (b) gives: $2^{2k} + 2^k + 2^k - (X-Y)$. Discard carry $2^{2k} = 2^n$ to obtain result of (c):

$2^k + (\bar{2^k}$-X-Y)

10-5



Transfer Augend sign into T_s. Then add: $AC \leftarrow AC + BR$. A_s will have sign of sum.

Truth Table for combin. circuit

T_s	B_s	A_s	V	
0	0	0	0	
0	0	1	1	change of sign
0	1	0	0	quantities
0	1	1	0	subtracted
1	0	0	0	
1	0	1	0	
1	1	0	1	change of sign
1	1	1	0	

Boolean function for circuit:

$V = T_s' B_s' A_s + T_s B_s A_s'$

10-6 (a)

-9 1 0110
-6 1 1001
-15 0 1111

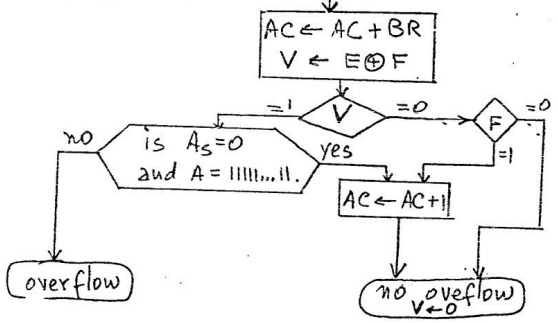
F=1 E=0 ← Carries

$E \oplus F = 1$ but there should be no overflow since result is -15

Add end around carry F as needed in signed-1's complement addition:

0 1111
+1
1 0000 $= -15$

(b) The procedure $V \leftarrow E \oplus F$ is valid for 1's complement numbers provided we check the result 0 1111...11 when $V = 1$. Algorithm → Add



10-7 Add algorithm flowchart is shown above (Prob. 10-6b)

10-8 Maximum value of numbers is $r^n - 1$. It is necessary to show that maximumum product is less than or equal to $r^{2n} - 1$. Maximum product is:

$$(r^n - 1)(r^n - 1) = r^{2n} - 2r^n + 1 \leq r^{2n} - 1$$

which gives: $2 \leq 2r^n$ or $1 \leq r^n$

This is always true since $r \geq 2$ and $n \geq 1$

10-9 Multiplicand $B = 11111 = (31)_{10}$ $31 \times 21 = 651$

Multiplier in Q — $Q = (21)_{10}$

	E	A	Q	sc
	0	00000	10101	101
Q_n=1, add B		11111		
	0	11111		
shr EAQ		01111	11010	100
Q_n=0, shr EAQ		00111	11101	011
Q_n=1, add B		11111		
	1	00110		
shr EAQ	0	10011	01110	010
Q_n=0, shr EAQ		01001	10111	001
Q_n=1, add B		11111		
	1	01000		
shr EAQ		1010001011		000
		$(651)_{10}$		

10-10 (a) $\frac{10100011}{1011} = 1110 + \frac{1001}{1011}$ $\frac{163}{11} = 14 + \frac{9}{11}$

B = 1011 $\bar{B}+1 = 0101$ DVF=0

	E	A	Q	sc
Dividend in AQ	0	1010	0011	100
shl EAQ	1	0100	0110	
add $\bar{B}+1$, suppress carry		0101		
E=1, set Q_n to 1	1	1001	0111	011
shl EAQ	1	0010	1110	
add $\bar{B}+1$, suppress carry		0101		
E=1, set Q_n to 1	1	0111	1111	010
shl EAQ	0	1111	1110	
add $\bar{B}+1$, carry to E		0101		
E=1, set Q_n to 1	1	0100	1111	001
shl EAQ	0	1001	1110	
add $\bar{B}+1$, carry to E		0101		
E=0, leave Q_n=0	0	1110	1110	
add B		1011		
restore remainder	1	1001	1110	000
		remainder	quotient	

10-10(b) $\frac{1111}{0011} = 0101$ $B = 0011$ $\overline{B}+1 = 1101$

	E	A	Q	SC
Dividend in Q, A=0		0000	1111	100
shl EAQ	0	0001	1110	
add $\overline{B}+1$		1101		
E=0, leave $Q_n=0$	0	1110	1110	
add B		0011		
restore partial remainder	1	0001		011
shl EAQ	0	0011	1100	
add $\overline{B}+1$		1101		
E=1, set Q_n to 1	1	0000	1101	010
shl EAQ	0	0001	1010	
add $\overline{B}+1$		1101		
E=0, leave $Q_n=0$	0	1110	1010	
add B		0011		
restore partial remainder	1	0001		001
shl EAQ	0	0011	0100	
add $\overline{B}+1$		1101		
E=1, set Q_n to 1	1	0000	0101	000
		remainder	quotient	

10-11

$A+\overline{B}+1$ performs: $A+2^n-B = 2^n+A-B$
adding B: $(2^n+A-B)+B = 2^n+A$
remove end-carry 2^n to obtain A.

10-12

To correspond with correct result. In general:

$$\frac{A}{B} = Q + \frac{R}{B}$$

Where A is dividend, Q the quotient and R the remainder.
Four possible signs for A and B:

$\frac{+52}{+5} = +10 + \frac{+2}{+5} = +10.4$ $\frac{-52}{+5} = -10 + \frac{-2}{+5} = -10.4$

$\frac{+52}{-5} = -10 + \frac{+2}{-5} = -10.4$ $\frac{-52}{-5} = +10 + \frac{-2}{-5} = +10.4$

The sign of the remainder (2) must be same as sign of dividend (52).

10-13

Add one more stage to Fig. 10-10 with 4 AND gates and a 4-bit adder.

10-14 (a) $(+15) \times (+13) = +195 = (0\ 0110000011)_2$

BR = 0 1111 (+15); $\overline{BR}+1 = 10001$ (-15); QR = 0 1101 (+13)

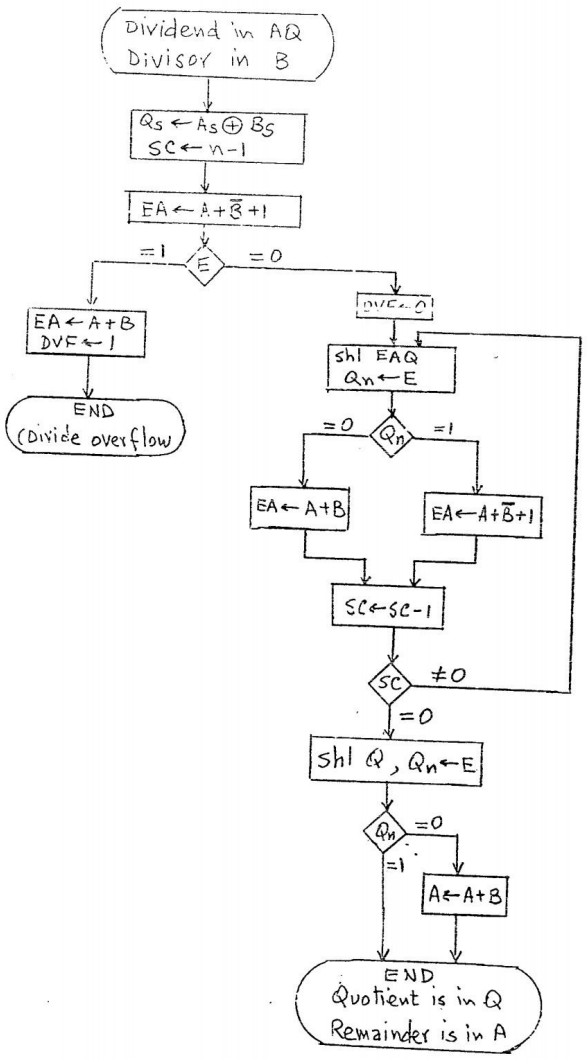
Q_n Q_{n+1}		AC	QR	Q_{n+1}	SC
	Initial	00000	01101	0	101
1 0	Subtract BR	10001			
		10001			
	ashr	11000	10110	1	100
0 1	Add BR	01111			
		00111			
	ashr	00011	11011	0	011
1 0	Subtract BR	10001			
		11100			
	ashr	11010	01101	1	010
1 1	ashr	11101	00110	1	001
0 1	Add BR	01111			
		01100			
	ashr	00110	00011	0	000
		+ 195			

(b) $(+15) \times (-13) = -195 = (1100\ 111101)_{2's\ comp.}$

BR = 0 1111 (+15); $\overline{BR}+1 = 10001$ (-15); QR = 10011 (-13)

Q_n Q_{n+1}		AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	10001			
		10001			
	ashr	11000	11001	1	100
1 1	ashr	11100	01100	1	011
0 1	Add BR	01111			
		01011			
	ashr	00101	10110	0	010
0 0	ashr	00010	11011	0	001
1 0	Subtract BR	10001			
		10011			
	ashr	11001	11101	1	000
		− 195			

10-15



10-16 The algorithm for square-root is similar to division with the radicand being equivalent to the dividend and a "test value" being equivalent to the divisor.

Let A be the radicand, Q the square-root, and R the remainder such that $Q^2 + R = A$ or:

$\sqrt{A} = Q$ and a remainder

General coments:

1. For k bits in A (k even), Q will have $k/2$ bits:
$$Q = q_1 q_2 q_3 \cdots q_{k/2}$$
2. The first test valve is 01
The second test value is $0q_1 01$
The third test value is $00q_1q_201$
The fourth test value is $000q_1q_2q_301$ etc.
3. Mark the bits of A in groups of two starting from left.
4. The procedure is similar to the division restoring method as shown in the following example:

```
   q1  q2  q3  q4
    1   1   0   1   = Q = 13
√ 10  10  10  01   = A = 169
   01                subtract first test value 01
  ---
   01                Answer positive; let q1 = 1
   01 10             bring down next pair
   01 01             subtract second test value 0q1 01
  ----
   00 01             answer positive; let q2 = 1
   00 01 10          bring down next pair
   00 11 01          subtract third test value 00q1q201
  ----------
   negative          answer negative; let q3 = 0
   00 01 10          restore partial remainder
   00 01 10 01       bring down next pair
   00 01 10 01       subtract fourth test value 000q1q2q301
  ------------
Remainder = 00000    answer positive (zero); let q4 = 1
```

10-17 (a) e = exponent $e+64$ = biased exponent

e	$e+64$	biased exponent
−64	−64+64 = 0	0 000 000
−63	−63+64 = 1	0 000 001
−62	−62+64 = 2	0 000 010
−1	−1+64 = 63	0 111 111
0	0+64 = 64	1 000 000
+1	1+64 = 65	1 000 001
+62	62+64 = 126	1 111 110
+63	63+64 = 127	1 111 111

(b) The biased exponent follows the same algorithm as a magnitude comparator. See Sec. 9-2

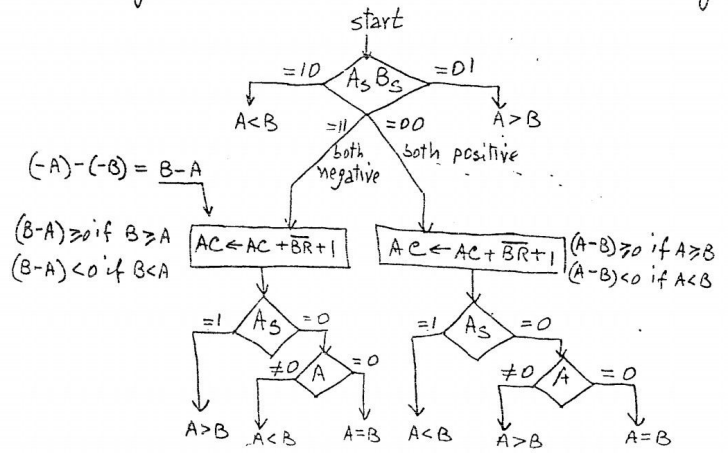
(c) $(e_1+64)+(e_2+64) = (e_1+e_2+64)+64$
subtract 64 to obtain biased exponent sum

(d) $(e_1+64)-(e_2-64) = e_1+e_2$
add 64 to obtain biased exponend difference.

10-18

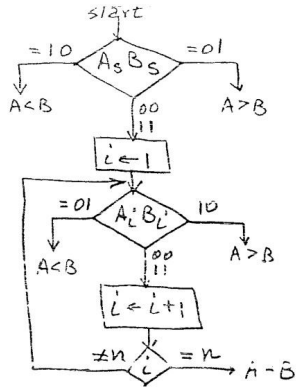
(a) $AC = A_S\ A_1\ A_2\ A_3 \ldots A_n$
$BS = B_S\ B_1\ B_2\ B_3 \ldots B_n$

If signs are unlike – the one with a 0 (plus) is larger.
If signs are alike – both numbers are either positive or negative

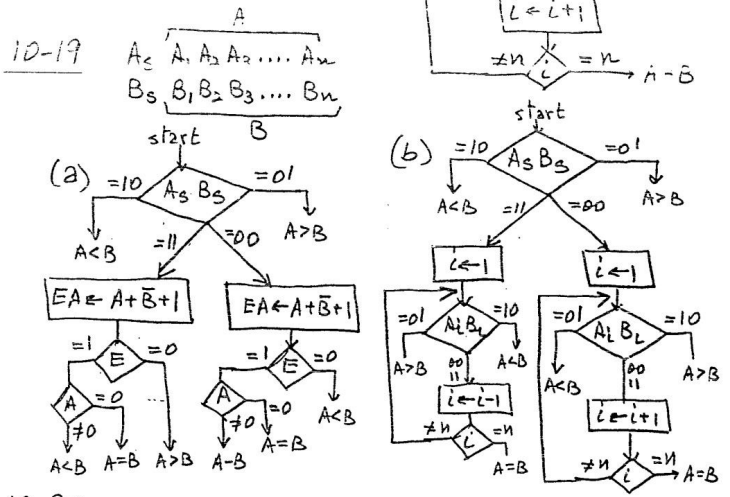


10-18 (b)

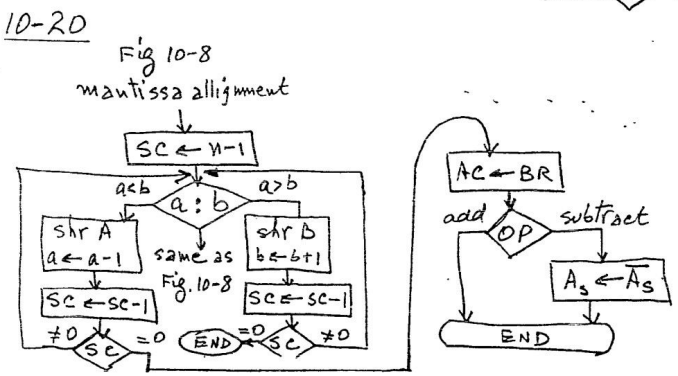
	$A_S\ A_1\ A_2 \ldots A_n$
+2	0 000010
+1	0 000001
0	0 000000
−1	1 111111
−2	1 111110
−3	1 111101



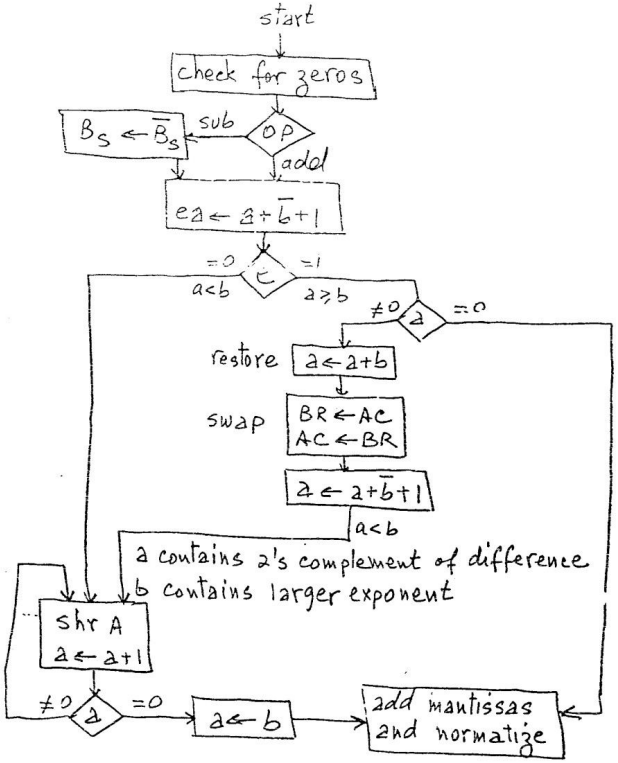
10-19



10-20



10-21 Let "e" be a flip-flop that holds end-carry after exponent addition.



10-22

When 2 numbers of n bits each are multiplied, the product is no more than 2n bits long — see Prob. 9-7.

10-23 $\frac{\text{dividend}}{\text{divisor}}$ $A = 0.1xxxx$ $B = 0.1xxxx$ where $x = 0,1$

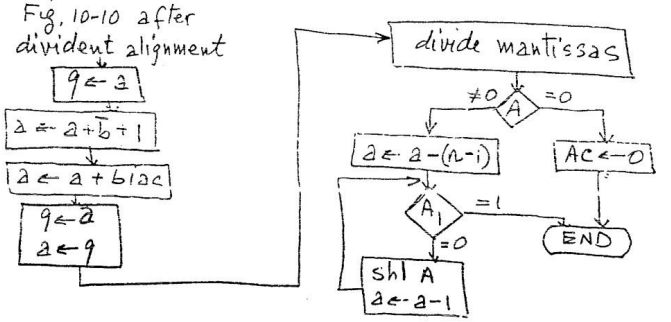
(a) If A<B then after shift we have $A = 1.xxxx$ and 1st quotient bit is a 1.

(b) if A≥B, dividend alignment results in $A = 0.01xxxx$ then after the left shift A≥B and first quotient bit = 1.

10-24

$$\frac{\text{dividend}}{\text{divisor}} = \frac{\overbrace{.1xxxx}^{n-1\text{ bits}} * 2^{e_1}}{.1yyyy * 2^{e_2}} = .1aaaa * 2^{e_1-e_2} + \frac{\overbrace{.0000rrrrr}^{\text{remainder}} * 2^{e_1}}{.1yyyy * 2^{e_2}}$$

Remainder bits rrrrr have a binary-point (n-1) bits to the left.



10-25

(a) When the exponents are added or incremented

(b) When the exponents are subtracted or decremented

(c) Check end-carry after addition and carry after increment or decrement.

10-26

Assume integer mantissa of n-1=5 bits (excluding sign)

(a) Product : A Q

xxxxx xxxxx. $* 2^{a}$ ← binary-point for integer

Product in AC : $xxxxx. * 2^{a+5}$

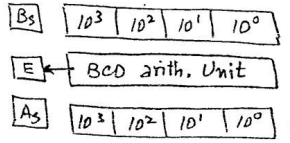
(b) Single precision normalized dividend : $xxxxx. * 2^{a}$

Dividend in AQ : A Q

xxxxx 00000. $* 2^{a-5}$

10-27

Neglect Be and Ae from Fig. 10-14. Apply carry directly to E.

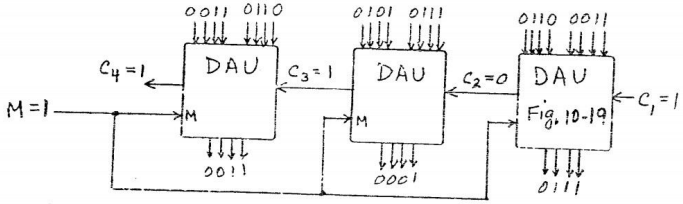


10-28

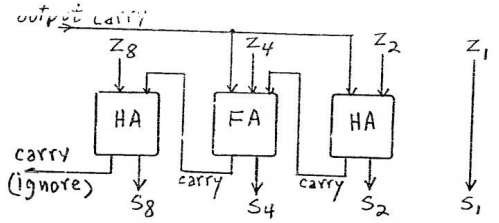
$$\begin{array}{r} 673 \\ -356 \\ \hline 317 \end{array}$$

10's comp. of 356 = 644

$$\begin{array}{r} 673 \\ +644 \\ \hline \text{carry} \rightarrow 1)\ 317 \end{array}$$

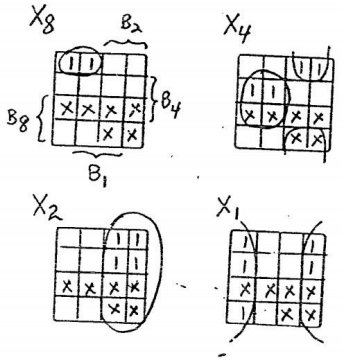


10-29



10-30

inputs $B_8B_4B_2B_1$	outputs $X_8X_4X_2X_1$	
0 0 0 0	1 0 0 1	9
0 0 0 1	1 0 0 0	8
0 0 1 0	0 1 1 1	7
0 0 1 1	0 1 1 0	6
0 1 0 0	0 1 0 1	5
0 1 0 1	0 1 0 0	4
0 1 1 0	0 0 1 1	3
0 1 1 1	0 0 1 0	2
1 0 0 0	0 0 0 1	1
1 0 0 1	0 0 0 0	0



$d(B_8B_4B_2B_1) = \Sigma(10,11,12,13,14,15)$
are don't-care conditions

$X_8 = B_8'B_4'B_2'$
$X_4 = B_4B_2' + B_4'B_2$
$X_2 = B_2$
$X_1 = B_1'$

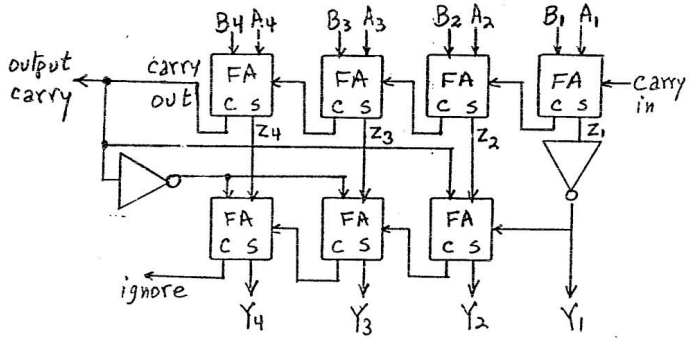
10-31

dec	Z uncorrected	Y corrected
0	0110	0011
1	0111	0100
2	1000	0101
3	1001	0110
4	1010	0111
5	1011	1000
6	1100	1001
7	1101	1010
8	1110	1011
9	1111	1100

No output carry
$Y = Z - 3 = Z + 13 - 16$
ignore carry

dec	Z uncorrected	Y corrected
10	1 0000	1 0011
11	1 0001	1 0100
12	1 0010	1 0101
13	1 0011	1 0110
14	1 0100	1 0111
15	1 0101	1 1000
16	1 0110	1 1001
17	1 0111	1 1010
18	1 1000	1 1011
19	1 1001	1 1100

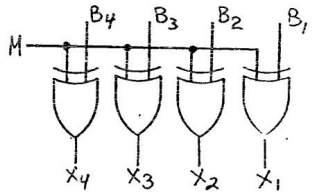
Uncorrected carry = output carry
$Y = Z + 3$



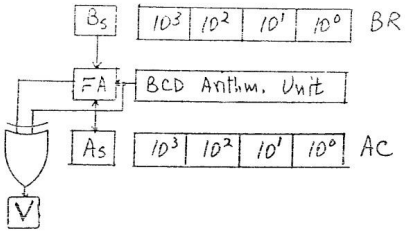
10-32 The excess-3 code is self-complementing code. Therefore, to get 9's complement we need to complement each bit.

M=0 for x=B
M=1 for x=9's comp. of B

M	B_i	$x_i = B_i \oplus M$	
0	0	0	$x_i = B_i$
0	1	1	
1	0	1	$x_i = B_i'$
1	1	0	



10-33



Algorithm is similar to flow chart of Fig. 10-2

10-34 (a) B = 410

	Ae	A	Q ↓ Q_L	SC
initial	0	000	152	3
$Q_L \neq 0$	0	470	151	
$Q_L \neq 0$	0	940	150	
$Q_L = 0$, dshr	0	094	015	2
$Q_L \neq 0$	0	564	014	
	1	034	013	
	1	504	012	
	1	974	011	
	2	444	010	
$Q_L = 0$, dshr	0	244	401	1
$Q_L \neq 0$	0	714	400	
$Q_L = 0$, dshr	0	071	440	0
		Product		

(b)

```
  999
 ×199
------
 8991   - first partial product     Ae = 8
+89910
------
 98901  - second partial product    Ae = 9
+99900
------
198801  - final product              Ae = 1
```


10-35 $\frac{1680}{32} = 52 + \frac{16}{32}$

B = 032
$\bar{B}+1 = 968$ (10's comp.)

	E	Ae	A	Q	SC
initial	0	0	16	80	2
dshl		1	68	00	
add $\bar{B}+1$		9	68		
E = 1	1	1	36	01	
add $\bar{B}+1$		9	68		
	1	1	04	02	
		9	68		
	1	0	72	03	
		9	68		
	1	0	40	04	
		9	68		
	1	0	08	05	
		9	68		
E = 0	0	9	76		
add B		0	32		
restore remainder	1	0	08	05	1

(continued here)

	E	Ae	A	Q	SC
dshl		0	80	50	1
add $\bar{B}+1$		9	68		
E = 1	1	0	48	51	
		9	68		
	1	0	16	52	
		9	68		
E = 0	0	9	84		
add B		0	32		
	1	0	16	52	0
		remainder		quotient	

10-36

(a) At the termination of multiplication we shift right the content of A to get zero in Ae.

(b) At the termination of division, B is added to the negative difference. The negative difference is in 10's complement so Ae = 9. Adding Be = 0 to Ae = 9 produces a carry and makes Ae = 0.

10-37

Change the symbols as defined in Table 10-1 and use same algorithms as in Sec. 10-4 but with multiplication and division of mantissas as in Sec. 10-5.