$R \leftarrow R_1$ + 2's	Subtract
$R \leftarrow R_1$	Transfer
$R \leftarrow R_1 + 1$	Increment
$R \leftarrow R_1$ + (All is)	Decrement

6.5 LOGIC MICROOPERATIONS

Logic operations are basically the binary operations which are performed on the string of bits stored in the registers. For a logic microoperation each bit of a register is treated as a variable. For example, if R_1 and R_2 are 8 bits registers and

R_1 contains 10010011

R_2 contains 01010101

R_1 AND R_2 00010001

Some of the common logic microoperations are AND, OR, NOT or complements, exclusive OR, NOR, NAND.

We can have four possible combination of input of two variables. These are 00, 01, 10 and 11. Now, for all these 4 input combination we can have $2^4 = 16$ output combinations of a function. This implies that for two variables we can have 16 logical operations.

6.6 SHIFT MICROOPERATIONS

Shift is a useful operation which can be used for serial transfer of data. Shift operations can also be used along with other operations. For example, for implementing a multiply operation arithmetic microoperation can be used along with shift operation. The shift operation may result in shifting the contents of a register to the left or right.

$R \leftarrow$ SHL (R_1) shift left

$R \leftarrow$ SHR (R_1) shift right

In a shift left operation a bit of data is input at the rightmost flip-flop while in shift right a bit of data is input at the leftmost flip-flop. Depending on what bit enters the register and where does the shift out bit goes, the shifts are classified in three types. These are:

- Logical,
- Arithmetic and
- Circular.

In logical shift the data entering by serial input to leftmost of right flip-flop is a 0.

If we connect the serial output of a shift register in its serial input then we encounter a circular shift. In circular shift left or circular shift right information is not lost, but is circulated. In arithmetic shift a signed binary number is shifted to the left or to the right. Thus, an arithmetic shift left causes a number to be multiplied by 2, on the other hand a shift right causes a division by 2. But as in division, on multiplication by 2 the sign of a number should not be changed, therefore, arithmetic shift must leave the sign bit unchanged.

The only place where these operations can be considered as microoperations is in the digital system, where they are implemented by means of combinational circuits. To implement the add microoperation with hardware that performs the arithmetic sum of two bits and a previous carry is called full adder. Digital circuit that generates the arithmetic sum of two binary number of any length is called binary adder constructed with full adder circuit connected in cascade. The subtraction of binary number can be done by including an exclusive-OR gate with each full adder. Increment microoperation can be easily implemented with the binary counter.

Check Your Progress

State whether the following statements are true or false:

1. Wilkes' idea was that each machine instruction was divided into a number of sub-instructions, or microinstructions.
2. When one sequence of microinstructions has finished, the next is not determined by the contents of the machine instruction register.
3. A microoperation is an elementary operation performed with the data stored in registers.
4. There are two memory transfer operations: Read and Write.
5. Shift is a useful operation which can be used for serial transfer of data.

6.7 LET US SUM UP

If a digital system has implemented division and multiplication by means of combinational circuits then we can call these as the microoperations for that system. The control parts of computers prior to the mid 60s were constructed, essentially, of electronic components structured into logic gates. It was quickly discovered that building computers, especially the control logic, was complex and error-prone — hence techniques were developed to further structure systems and reduce errors. The idea is that each microinstruction will be divided up into two parts — the control part, which controls the operation of the data path (or data unit), and the address part, which is the address of the next microinstruction to be executed under certain conditions.

The processor registers are 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 earlier. The control unit has a control address register CAR and a subroutine register SBR. Logic operations are basically the binary operations which are performed on the string of bits stored in the registers. Shift is a useful operation which can be used for serial transfer of data. Shift operations can also be used along with other operations.

6.8 KEYWORDS

PC: Program Counter

AR: Address Register

DR: Data Register

AC: Accumulator Register

CAR: Control Address Register

SBR: Subroutine Register

6.9 QUESTIONS FOR DISCUSSION

1. Explain register transfer.
2. Define Microoperation. Describe its categories.
3. What is the difference between arithmetic microoperations and logic microoperations?
4. Describe arithmetic microoperations.

Check Your Progress: Model Answers

1. True
2. False
3. True
4. True
5. True

6.10 SUGGESTED READINGS

Sajjan G. Shiva; *Computer Design and Architecture*; Marcel Dekker

Silvia Melitta Mueller, Wolfgang J. Paul; *Computer Architecture*; Springer

Joseph D. Dumas II; *Computer Architecture*; CRC Press.

Nicholas P. Carter; *Schaum's Outline of Computer Architecture*; Mc. Graw-Hill Professional

LESSON 7

CONTROL FUNCTIONS

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define control memory
- Explain the concept of address sequencing
- Explain the concept of microprogram sequencer
- Discuss micro instruction formats
- Identify the advantages and applications of micro programming

7.1 INTRODUCTION

The central processing unit is the brain of the computer system. The input and output devices may vary for different applications, but there is only one CPU for a particular computer. The specifications of a computer are basically characterized by its central processing unit.

The central processing unit can be further divided into:

1. Arithmetic Logic Unit (ALU)
2. Control Unit
3. Main Memory

The control unit controls the entire operation of the computer and the CPU. The control unit upon receiving an instruction decides what is to be done with it. The control unit has an electronic clock that transmits electronic pulses at equal intervals of time. In this lesson, we will describe the control memory of the control units, Microprogramming, Hardwired control unit, micro instructions formats and advantages of micro programming.

7.2 CONTROL MEMORY

An advance 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 use 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 called a control memory.

A computer that uses a microprogrammed control unit usually has two separate memories - a main memory and a control memory. The main memory is available to the user for storing their programs. The contents of main memory may change when the data are manipulated and every time the program is changed. The user's program in main memory consists of machine instructions and data, whereas, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of micro-instructions 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.

A microprogrammed control unit is shown in the block diagram of Figure 7.1. The control memory is usually a ROM, which stores all control information permanently. The control MAR 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 micro-operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction is generally the one next in sequence, otherwise, 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 microintstruction. Hence a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

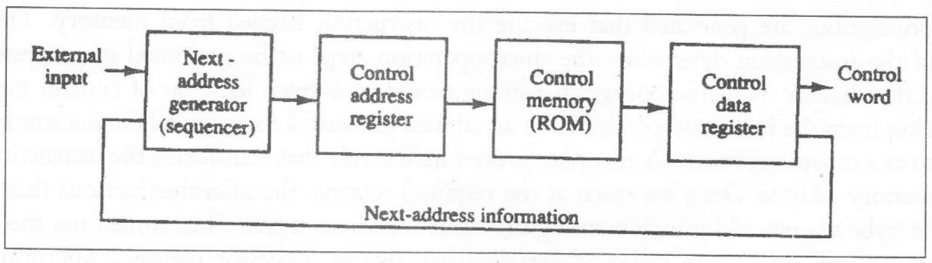


Figure 7.1: Microprogrammed Control Organization

A microprogram sequencer is the next address generator, 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. The 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.

The control DR(data register) stores the present microinstruction while the next address is computed and read from memory. The data register is also 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 main advantage of the microprogrammed control is that once the hardware configuration is built, there should be no need for further hardware or wiring changes. If we want to make a different control sequence for the system, all we need to do is to specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations. We have to change only the microprogram residing in control memory.

7.3 ADDRESS SEQUENCING

Control memory stores the microinstructions in groups, with each group specifying a routine. Different computer instructions have different microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware 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.

When power is turned on in the computer, an initial address is loaded into the control address register. This address is 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 subroutine that decides the effective address of the operand. A machine instruction may have bits that specify various addressing modes, for example, indirect address and index registers. The effective address calculation 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.

Now microoperations are generated that execute the instruction fetched from memory. The operation code part of the instruction determines the microoperation steps to be generated in processor registers. 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 we reach at the required routine, the microinstructions that execute the instruction maybe sequenced by incrementing the control address register, but sometimes the sequence of micro-operations 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.

After completion of instruction execution, control must return to the fetch routine. This is done 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.

7.3.1 Conditional Branching

The branch logic of Figure 7.2 gives decision-making qualities in the control unit. The status conditions are special bits that give parameter information, for example, 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.

We can implement the branch logic hardware in a variety of ways. The easiest way is to test the specified condition and branch to the indicated address if the condition is met. Otherwise, the address register is incremented.

A multiplexer can be used to implement it. Assume 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 give 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 of the multiplexer causes the address register to be incremented.

We can implement an unconditional branch microinstruction by loading the branch address from control memory into the control address register. This can be done by fixing the value of one status bit at the input of the multiplexer, so that 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.

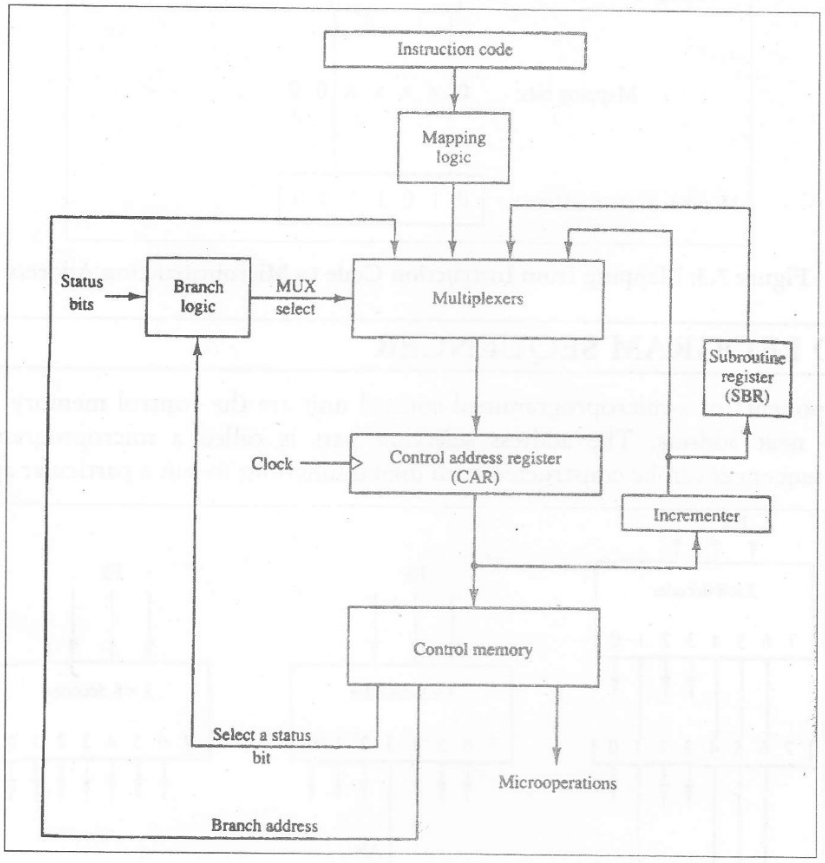


Figure 7.2: Selection of Address for Control Memory

7.3.2 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 Figure 7.3 an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring 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 Figure 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.

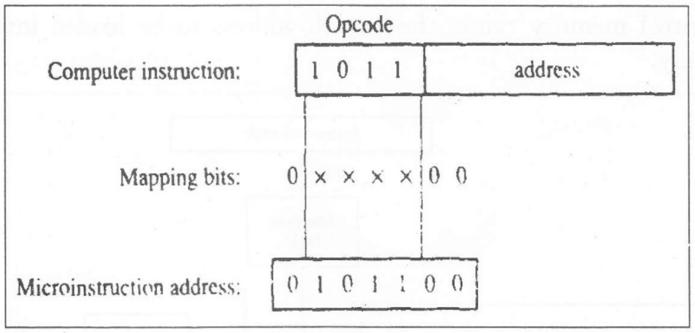


Figure 7.3: Mapping from Instruction Code to Microinstruction Address

7.4 MICRO PROGRAM 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.

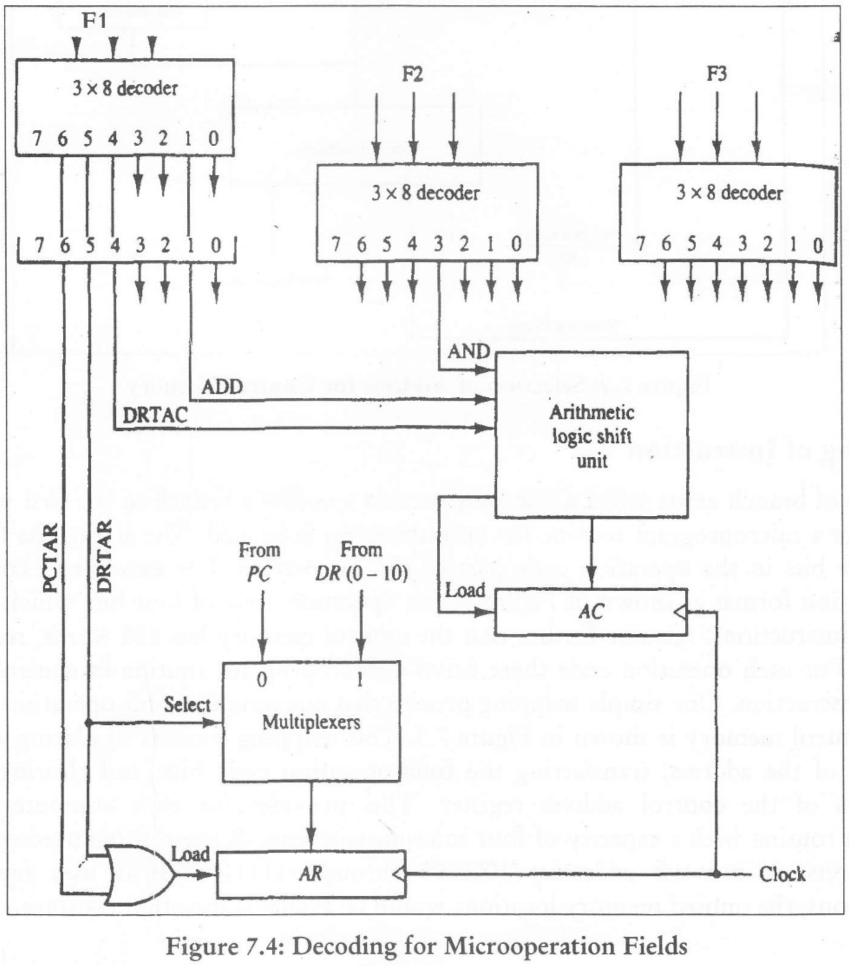


Figure 7.4: Decoding for Microoperation Fields

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.

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 Figure 7.4. 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 selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field, go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are - increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

The input logic circuit in Figure 7.5 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 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 7.1. 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:

$$S_1 = I_1$$

$$S_0 = I_1I_0 + I'_1T$$

$$L = I'_1I_0T$$

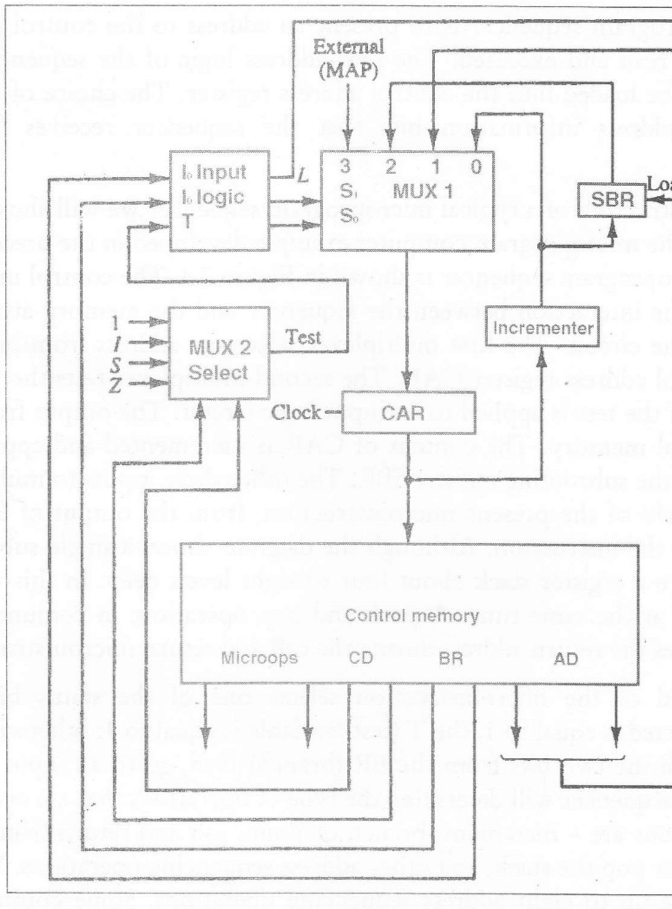


Figure 7.5: Microprogram Sequencer for a Control Memory

Table 7.1: Input Logic Truth Table for Microprogram Sequencer

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

7.4.1 Hardwired Control Unit

Another alternative to microprogrmmed control is hardwired control unit. A hardwired control unit typically has a large array of diode circuit (decoder) that produces the necessary control words to execute the microoperations. Hardwired control units are cheap, easy to design and easy to implement. However, they are not as flexible as micro-programmed control units are.

The microinstructions are executed by dedicated hardware circuits unlike a macro-programmed unit where certain algorithms are implemented to carry out an operation.

7.5 MICROINSTRUCTION FORMAT

The microinstruction format for the control memory is shown in Figure 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 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.

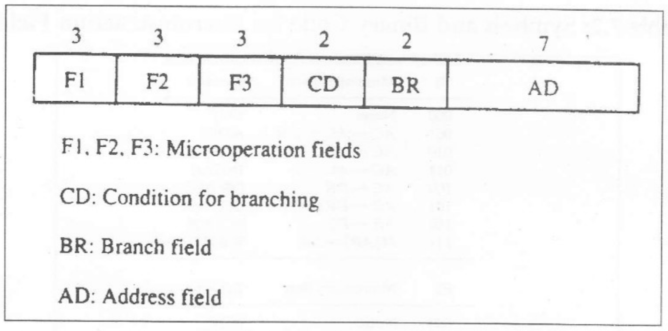

3
3
3
2
2
7
F1
F2
F3
CD
BR
AD
F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 7.6: Microinstruction Code Format (20 Bits).

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 7.2. 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

$DR \leftarrow M[AR]$ with F2 = 100

and $PC \leftarrow PC + 1$ 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 7.2 is defined with a register transfer statement and is assigned a symbol to use in a symbolic microprogram. All transfer-type microoperation 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.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions assisted in Table 7.2. 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 I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit. The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero. We will use the symbols U, I, S and Z for the four status bits when we write microprograms in symbolic form.

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in Table 7.2, 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.

Table 7.2: 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	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

7.5.1 Symbolic Microinstructions

The symbols defined in Table 7.2 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 earlier. 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 7.2. 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 7.2.
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.

7.5.2 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:

$AR \leftarrow PC$

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

$AR \leftarrow DR(0\text{-}10),\ CAR(2\text{-}5) \leftarrow DR(11\text{-}14),\ CAR(0,1.6) \leftarrow 0$

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 7.2.

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.

7.5.3 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. The table also shows the symbolic microprogram for the fetch routine and the microinstruction routines that execute four computer instructions.

7.6 ADVANTAGES AND APPLICATIONS OF MICROPROGRAMMING

- The Organization of Control
- Development in Performance
  - ❖ A high degree of parallelism in data paths e.g., multiple bit microinstructions are performed in one cycle
  - ❖ A high degree of decision logic (in table search and sorting routines)
- Computer-series Well-suited
- Compatibility of Instruction Set between smaller and larger machines of series e.g. intel 286, 386, Pentium, Motorola 68000 series

- Emulation
    - Emulation is the mutual software/hardware interpretation of the machine instruction of one machine by another. Target's machine architecture is mapped onto the host machine.
    - Emulator – A set of microprograms that interpret a exacting instruction set or language L1, Computer C1 emulates computer C2 if it can interpret machine language L2.
- Microdiagonostics

  Microprogram diagnostics custom have allowed refinements and increased the speed of detecting and localizing faults, including error detection and correction of micro storage itself.
    - Software Diagonostics
    - Hardware Diagnostics
    - Microdiagnostics
- Software Support
    - Effortlessness Programming
- Special-purpose Device

  E.g. special processor for data communication, data acquisition, device controllers
- Dynamic Microprogramming

  This allows routines to be easily microprogrmmed. Computer to limited to represent any instruction vocabulary by use of Writable Control Memory(WCM). It allows the instruction set of the machine to be changed and be tailored to specific applications.

Check Your Progress

Fill in the blanks:

1. The ....memory is usually a ROM, which stores all control information permanently.
2. The .... part of the instruction determines the microoperation steps to be generated in processor registers.
3. The basic components of a microprogrammed control unit are the control memory and the .... that select the next address.
4. To microprogram the control memory, it is necessary to determine the ....values of each of the 128 words.
5. Alternative to .... control is hardwired control unit.

7.7 LET US SUM UP

Dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. The control memory holds a fixed microprograms that specify various interval control signals for execution of register micro-operations. The main advantage of the microprogrammed control is that once the hardware configurations is built, there should be no need for further hardware or writing changes. We have to change only the microprogram residing in control memory.

Different computer instructions have different micropogram routine in control memory to generate the micro-operations that execute the instructions.

We can implement that branch logic to test the specified condition and branch to the indicated address if the condition is met. Otherwise the address register is incremented. A multiplexer can be used to implement it. We can implement an unconditional branch microinstruction by loading the branch address from control memory into the control address register.

A special type of branch exists a microinstruction specified a branch to the first word in control memory where a microprogram routine far an instruction is located.

The microcode for the control memory is called microprogramming and is a process similar to conventional machine language programming. The transfer of information among the registers rather than a common bus.

The control memory has 128 words, and each word contain 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The bit of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.

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 sequence. A microprogram sequence can be constructed with digital functions to suit a particular application.

7.8 KEYWORDS

Control Memory: A writable memory used by control units that use dynamic microprogramming.

Branch Logic: Logic which gives decision-making qualities in the control unit.

Multiplexed: Device used to implement the branch logic.

Microprogramming: Generation of the microcode for the control memory.

Symbolic Microinstructions: Microinstructions defined with the help of symbols that can be translated into its binary equivalent by means of an assembler.

Micro Program Sequence: component (circuit) of a microprogrammed control unit that select the next address.

7.9 QUESTIONS FOR DISCUSSION

1. Describe various fields of microoperations.
2. Describe the function Fetch.
3. Describe the microprogram sequence for a control memory.
4. What is the difference between a direct and an indirect address instruction? How many references to memory are needed for such type of instruction to bring an operand into a processor resister?

5. Explain why each of the following microoperations cannot be executed during a single clock pulse.

   IR ← M[PC]

   AC ← AC + TR

   DR ← DR + AC (AC does not change)

6. Can the letter I be used as a symbolic address in the assembly language program defined for the basic computers? Justify the answer.

7. Is it possible to have a hardwired command associated with a control memory?

8. Define the following:

   (a) Micro operation

   (b) Micro instruction

   (c) Micro program

   (d) Micro code

Check Your Progress: Model Answers

1. Control
2. Operation code
3. Circuits
4. Bit
5. Microprogrmmed

7.10 SUGGESTED READINGS

Albert Y. Teng, William A. Malmgren, *Experiments in Logic and computer Design*, Prentice Hall

Sajjan G. Shiva; *Computer Design and Architecture*; Marcel Dekker

Silvia Melitta Mueller, Wolfgang J. Paul; *Computer Architecture*; Springer

Joseph D. Dumas II; *Computer Architecture*; CRC Press

Nicholas P. Carter; *Schaum's Outline of Computer Architecture*; Mc. Graw-Hill Professional

UNIT IV

LESSON

8

ARITHMETIC OPERATIONS

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the design of Arithmetic processor
- Describe the comparison and subtraction of unsigned binary numbers
- Discuss the addition and subtraction algorithm
- Discuss the multiplication and division algorithm

- Explain floating point arithmetic operations.
- Identify decimal arithmetic operations

8.1 INTRODUCTION

The Central Processing Unit (CPU) is called the brain of the computer that performs data-processing operations. Figure 8.1 shows the three major parts of CPU.

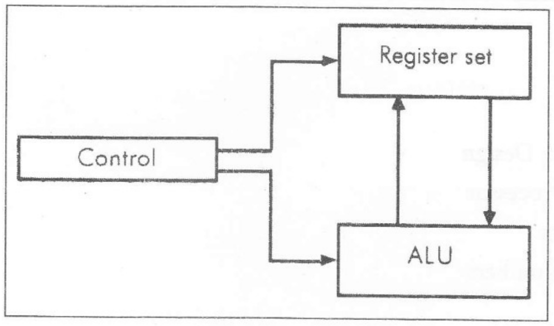


Figure 8.1: Major Components of CPU

Intermediate data is stored in the register set during the execution of the instructions. The microoperations required for executing the instructions are performed by the arithmetic logic unit whereas the control unit takes care of transfer of information among the registers and guides the ALU. The control unit services the transfer of information among the registers and instructs the ALU about which operation is to be performed. The computer instruction set is meant for providing the specifications for the design of the CPU. The design of the CPU largely, involves choosing the hardware for implementing the machine instructions.

8.2 ARITHMETIC PROCESSORS DESIGN

The very first question in this regard is: "What is an arithmetic processor?" and, "What is the need for arithmetic processors". A typical CPU needs most of the control and data processing hardware for implementing non-arithmetic functions. As the hardware costs are directly related to chip area, a floating point circuit being complex in nature are costly to implement. They are normally not included in the instruction set of a CPU. In such systems, floating-point operations are implemented by using software routines. This implementation of floating point arithmetic is definitely slower than the hardware implementation. Now, the question is whether a processor can be constructed only for arithmetic operations? A processor if devoted exclusively to arithmetic functions can be used to implement a full range of arithmetic functions in the hardware at a relatively low cost. This can be done in a single IC. Thus, a special purpose arithmetic processor, for performing only the arithmetic operations, can be constructed. Although this processor physically is separate, yet will be utilized by the CPU to execute a class of arithmetic instructions. Please note in the absence of arithmetic processors, these instructions may be executed using the slower software routines by the CPU itself. Thus, this auxiliary processor enhances the speed of execution of programs having lot of complex arithmetic computations. In addition it also helps in reducing program complexity, as it provides more instructions to a machine. Some of the instructions, which can be assigned, to arithmetic processors can be: add, subtract, multiply and divide fixed and floating point numbers of various lengths; exponentiation; logarithms and trigonometric functions.

How can this arithmetic processor be connected to the CPU? Two mechanisms are used for connecting arithmetic processor to the CPU.

If an arithmetic processor is treated as one of the I/O or peripheral unit then it is termed as *peripheral processor*. The CPU sends data and instructions to the peripheral processor, which performs the required operations on the data and communicates the results back to the CPU. A peripheral processor has several registers to communicate with the CPU. These registers may be addressed by the CPU as Input/Output register addresses. The CPU and peripheral processors are normally quite independent and communicate with each other by exchange of information using data transfer instructions. This data transfer instructions must be specific instructions in the CPU. This type of connection is called loosely coupled.

On the other hand, if the arithmetic processor has register and instruction set which can be considered extension of the CPU registers and instruction set then it is called tightly coupled processor. Here the CPU reserves a special subset of code for arithmetic processor. In such a system the instructions meant for arithmetic processor are fetched by CPU and decoded jointly by CPU and the arithmetic processor, and finally executed by arithmetic processor. Thus, these processors can be considered logical extension of the CPU. Such attached arithmetic processors are termed as *Co-processors.* Let us discuss them in more details.

8.2.1 Peripheral Processor

An example of one such arithmetic processor is the AMD 9511/12 one chip floating point processor. The advantage of this processor is that they can be utilized with any CPU, while the disadvantages are that they need explicitly programmed and slow communication links with the CPU. These processors can be utilized as given in figure:

Table 8.1: Communication between the CPU and Peripheral Processor

Performer	Instructions executed	Processing details
1) CPU	Data-transfer	These instruction help in sending a set input operands and commands, e.g. arithmetic operations, to the peripheral processor.
2) Peripheral Processor	Decode & Execute the command received	Results are generated and placed in registers directly accessible to the CPU From CPU using the operands.
3) CPU	Checks status by polling a status register or by Receiving interrupt from The peripheral processor.	It determines whether the peripheral processor has completed the task.
4) CPU	Data transfer instruction is executed	CPU obtains the results from the peripheral processor by executing this data transfer instruction.

In certain implementations CPU has to wait for peripheral processor to finish, therefore, remains idle for that time.

8.2.2 Coprocessors

Coprocessors, unlike peripheral processors, are tailor made for a particular family of CPUs. Normally, each CPU is designed to have a processor interface the control signal circuits of the CPU is designed for the interface beforehand. Special instructions are earmarked for execution by coprocessors. These coprocessor instructions can appear in any assembly or machine language program similar to any other instruction. The CPU hardware takes care of the instruction execution by the coprocessors. The coprocessor instructions can be executed even if a coprocessor is not present, by already stored software routines at pre-determined memory locations. If a coprocessor is not attached, then the CPU issues a software (coprocessor) trap which executes a desired software location routine for the instruction. Thus, without changing the source of object code we can execute the coprocessor instructions by the CPU even if the coprocessor is not present. Figure 8.2 shows a general structure, along with some of the control lines between the CPU and the coprocessor.

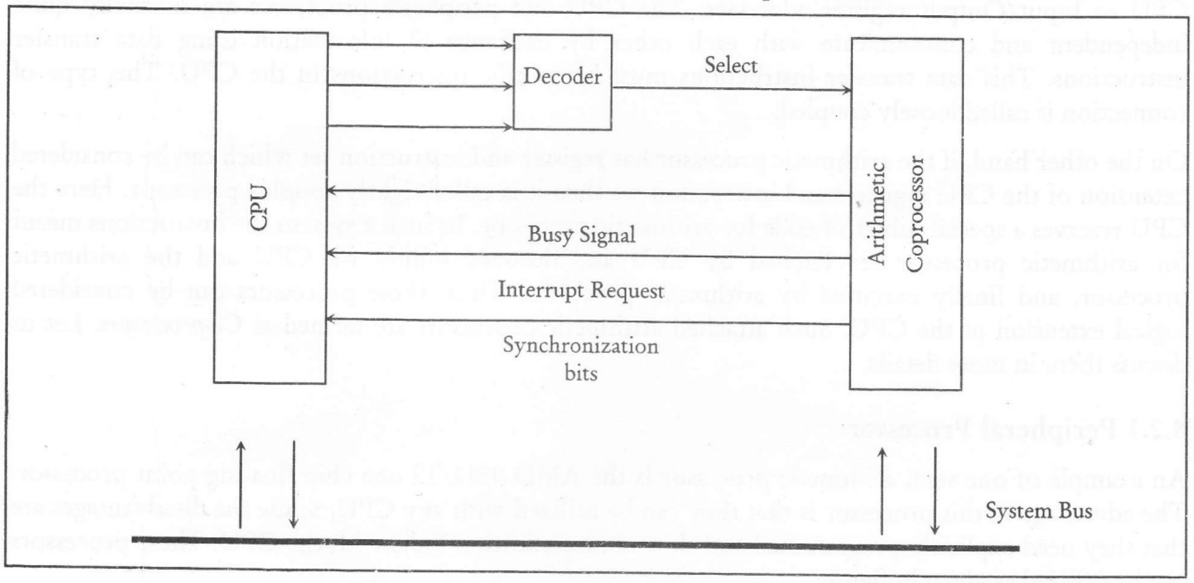


Figure 8.2: General Structure of CPU-Coprocessor

As both the processors are directly linked, therefore, they can be synchronized easily. The control lines between them are few. The data transfer between the processors can take place through the system bus. The CPU may act as the master of coprocessor. The registers of coprocessor can be written into or read by the CPU directly as it do for the main memory. Sometimes it useful to allow the coprocessor to control the bus as in such cases it can control data transfer from memory or can initiate data transfer to the CPU.

In case coprocessor can control the system bus then it is allowed to decode and identify the instructions at the same time CPU is doing. The coprocessor then can execute the instructions meant for it directly. This type of approach is followed in 8087 arithmetic coprocessor of 8086. While in some CPUs, only the CPU can decode the coprocessor instruction. This is the case for the 68881 floating-point coprocessor of Motorola 68000 series. A CPU can employ more than one different coprocessor if desired.

8.3 ADDITION OF BINARY NUMBERS

Digital computers perform a variety of information- processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 =1, and 1+ 1 =10. The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1. The binary sum consists of two digits. The higher significant bit of this result is called a carry. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a half-adder. One that performs the addition of three bits (two significant bits and a previous carry) is full-adder. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder. The two adder circuits are the first of the combinational circuits we shall design.

8.3.1 Half-adder

From the verbal explanation of a half-adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. It is necessary to specify two output variables because the result may consist of two binary digits. We arbitrarily assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.

Now that we have established the number and names of the input and output variables, we are ready to formulate a truth table to identify exactly the function of the half-adder. This truth table is shown below:

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

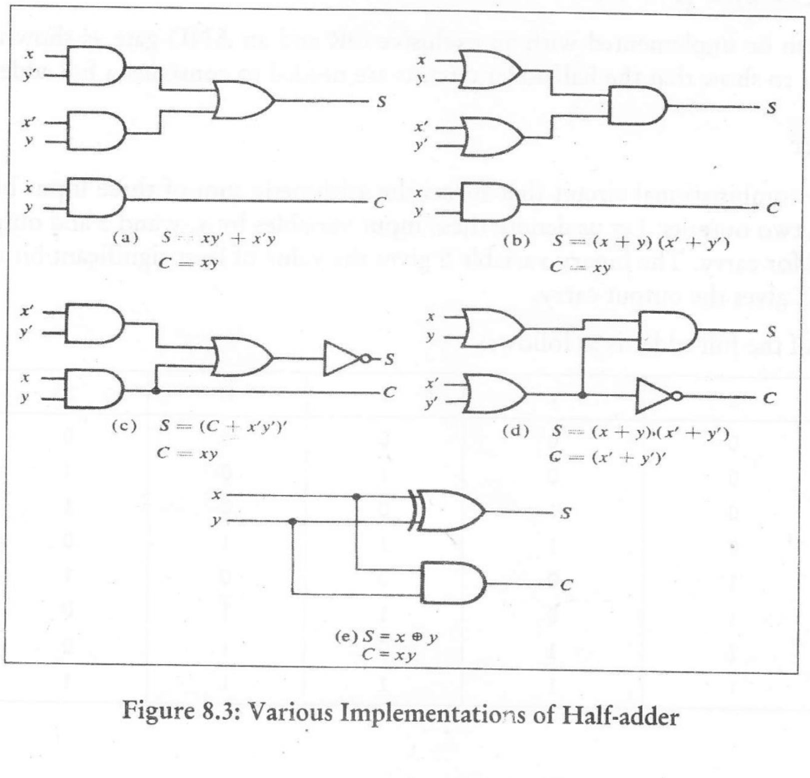


Figure 8.3: Various Implementations of Half-adder

The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are:

$$S = x'y + xy'$$
$$C = xy$$

The logic diagram for this implementation is shown in figure, as are four other implementations for a half-adder. They all achieve the same result as far as the input-output behavior is concerned. They illustrate the flexibility available to the designer when implementing even a simple combinational logic function such as this.

Figure as mentioned above, is the implementation of the half-adder in sum of products. Figure shows the implementation in product of sums:

$$S = (x + y)(x' + y)$$
$$C = xy$$

To obtain the implementation of figure, we note that S is the exclusive-OR of x and y. The complement of S is the equivalence of x and y

$$S' = xy + x'y'$$

But C = xy, and therefore we have:

$$S = (C + x'y')'$$

In figure, we use the product of sums implementation, with C derived as follows:

$$C = xy = (x' + y')'$$

The half-adder can be implemented with an exclusive-OR and an AND gate as shown in figure. This form is used later to show that the half-adder circuits are needed to construct a full adder circuit.

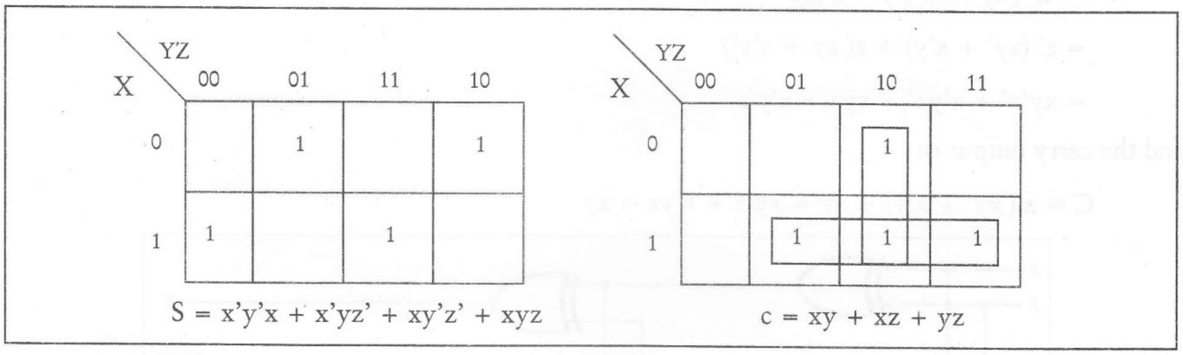
8.3.2 Full-adder

A full adder is a combinational circuit that forms the arithmetic sum of three input bits it consists of three inputs and two outputs. Let us denote these input variables by x, y and z and output variables by S for sum and C for carry. The binary variable S gives the value of least significant bit of the sum. The binary variable C gives the output carry.

The truth table of the full-adder is as follows:

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The eight rows under the input variable designate all possible combinations of 1's and 0's that these variables may have. The 1's and 0's for the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0's, the output is zero. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equals to 1.



S = x'y'x + x'yz' + xy'z' + xyz

c = xy + xz + yz

Figure 8.4: Maps for Full-adder

The input output logical relationship of the full adder circuit may be expressed in two Boolean functions, one for each output variables. Each output Boolean function requires a unique map for simplification. Each map must have eight squares, since each output is a function of three input variables. The maps of Figure 8.4 are used for simplifying the two output functions. The 1's in the squares for the maps of S and C are determined directly from the truth table. The squares with 1's for the S output do not combine in adjacent squares to give a simplified expression in sum of products. The C output can be simplified to a six literal expression. The logic diagram for the full adder implemented in sum of products is shown in Figure 8.5. This implementation uses the following Boolean expressions:

S = x'y'z + x'yz' + x y'z' + xyz

C = xy + xz + yz

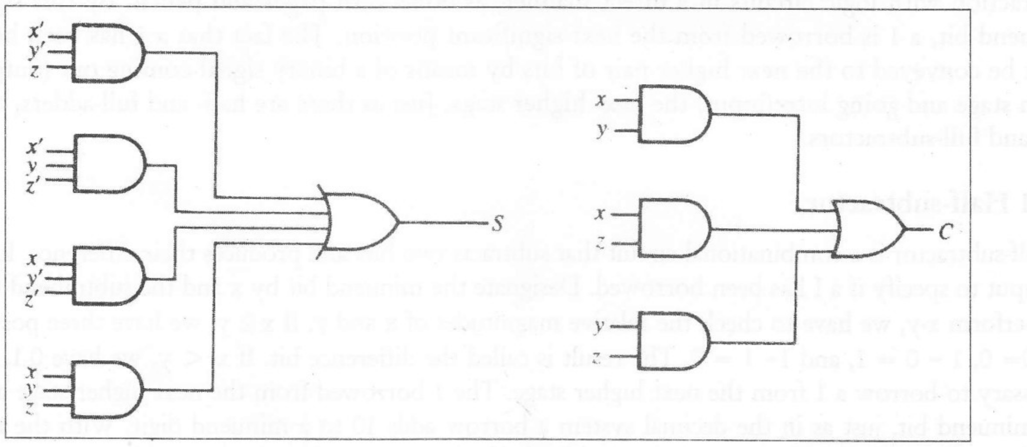


Figure 8.5: Implementation of Full-adder in Sum of Products

Other configurations for a full-adder may be developed. The product-of-sums implementation requires the same numbers of gates as in Figure 8.6, with the number of AND and OR gates interchanged.

A full adder can also be implemented with two half-adders and OR gate. The S output from the second half-adder is the exclusive-OR of z and the output of the first half-adder, giving:

$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

and the carry output is:

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

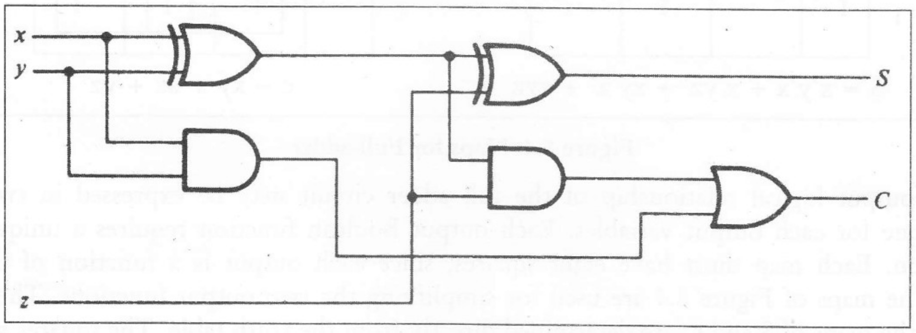


Figure 8.6: Implementation of A Full-adder with Two Half-adders and an OR Gate

8.4 SUBTRACTION OF BINARY NUMBERS

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes an addition operation requiring full adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this significant minuend bit, a 1 is borrowed from the next significant position. The fact that a 1 has been borrowed must be conveyed to the next higher pair of bits by means of a binary signal coming out (output) of a given stage and going into (input) the next higher stage. Just as there are half- and full-adders, there are half-and full-subtractors.

8.4.1 Half-subtractor

A half-subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an input to specify if a I has been borrowed. Designate the minuend bit by x and the subtrahend bit by y. To perform x-y, we have to check the relative magnitudes of x and y. If $x \geq y$, we have three possibilities: $0 - 0 = 0$, $1 - 0 = 1$, and $1 - 1 = 0$. The result is called the difference bit. If $x < y$, we have 0-1, and it is necessary to borrow a 1 from the next higher stage. The 1 borrowed from the next higher stage adds 2 to the minuend bit, just as in the decimal system a borrow adds 10 to a minuend digit. With the minuend equal to 2, the difference becomes $2 - 1 = 1$,. The half-subtractor needs two outputs. One output generates the difference and will be designated by the symbol D. The second output, designated B for borrow, generates the binary signal that informs the next stage that a 1 has been borrowed. The truth table for the input –output relations of a half-subtractor can now be derived as follows:

X	Y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow B is a 0 as long as x > y. It is a 1 for x = 0 and y = 1. The D output is the result of the arithmetic operation 2B + x – y.

The Boolean function for the two output of the half-subtractors are derived directly from the truth table:

$$D = x'y + xy'$$

$$B = xy'$$

It is interesting to note that the logic for D is exactly the same as the logic for output S in the half-adder.

8.4.2 Full-subtractor

A full–subtractor is a combinational circuit that performs a subtration between two bits, taking into account that a 1 may have been borrowed by a lower significant stage. This circuit has three inputs and two outputs. The three inputs, x, y, and z, denote the minuend subtrahend, and pervious borrow, respectively. The two output, D and b represent the difference and output borrow, respectively. The truth table for the circuit is as follows:

X	Y	Z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations of 1's and 0's that the binary variables may take. The 1's and 0's for the output variable are determined from the subtraction of x – y – z. The combinations having input borrow z=0 reduce to the same four conditions of the half-adder. For x=0, y=0 and z=1, we have to borrow a 1 from the next stage, which makes b-1 and adds 2 to x. since2-0-1=1, D=1. For x = 0 and y z = 1 1, we need to borrow again, making B = 1 and x=2. Since 2-1-1=0, D = 0. For x = 1 And xy=01, we have x –y=0, which makers B =1 and x =3, and 3-1-1=1, making D=1

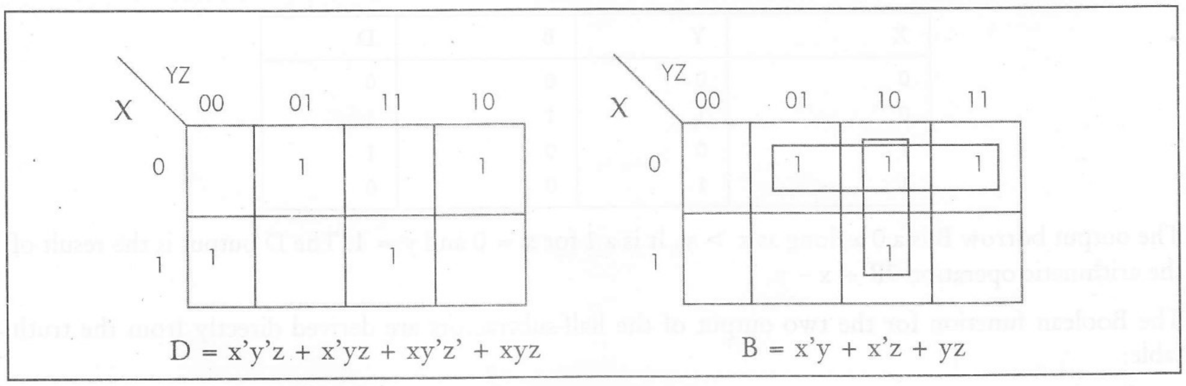


Figure 8.7: Map for Full-subtractor

The simplified Boolean functions for the two output of the full-subtractor are derived in the maps of Figure 8.7, The simplified sum of products output functions are:

$$D = x'y'z + x'y'z + x'y'z + xyz$$

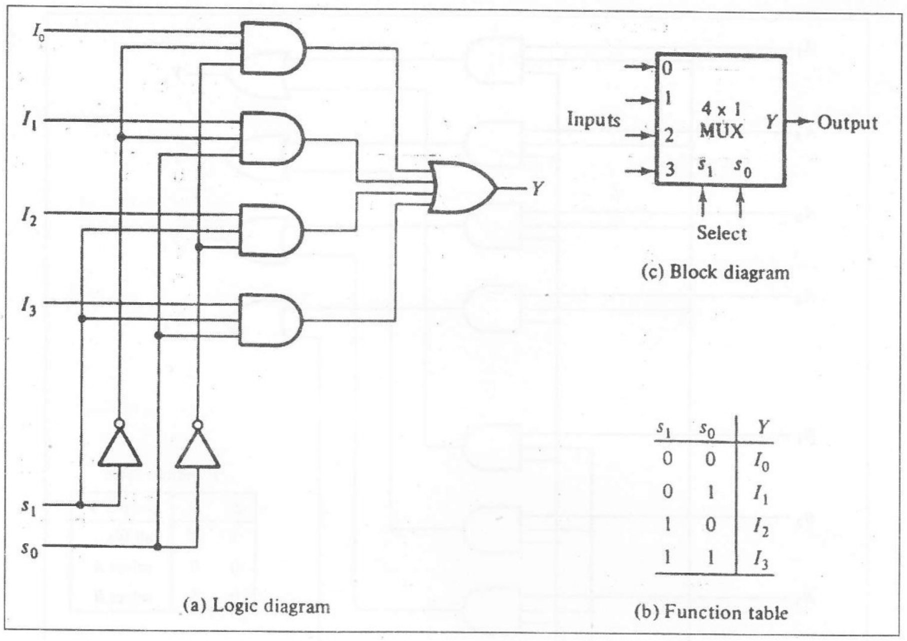
$$B = x'y + x'z + yz$$

Again we note that the logic function for output D in the full-subtractor is exactly the same as output S in the full-adder. Moreover, the output B resembles the function for C in the full adder, expect that the input variable x is complemented. Because of these similarities, it is possible to convert a full-adder into a full-subtractor by merely complementing input x prior to its application to the gates that from the carry output.

8.4.3 Multiplexers

Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. A digital multiplexer is a combinational circuit that selects binary information for one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

A 4-line to 1-line multiplexer is shown in Figure 8.8. Each of the four input lines, I_0 to I_3, is applied to one input of an AND gate. Selection lines s_1 and s_0 are decoded to select a particular AND gate. The function table in the figure lists the input-to-output path for each possible bit combination of the selection lines. When this MSI function is used in the design of a digital system, it is represented in block diagram form as shown in figure. To demonstrate the circuit operation, consider the case when s_1s_0 = 10. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third equal to 0., which makes providing a path from the OR-gate output is now equal to the value of I_2, thus providing a path from the selected input to the output. A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output line.



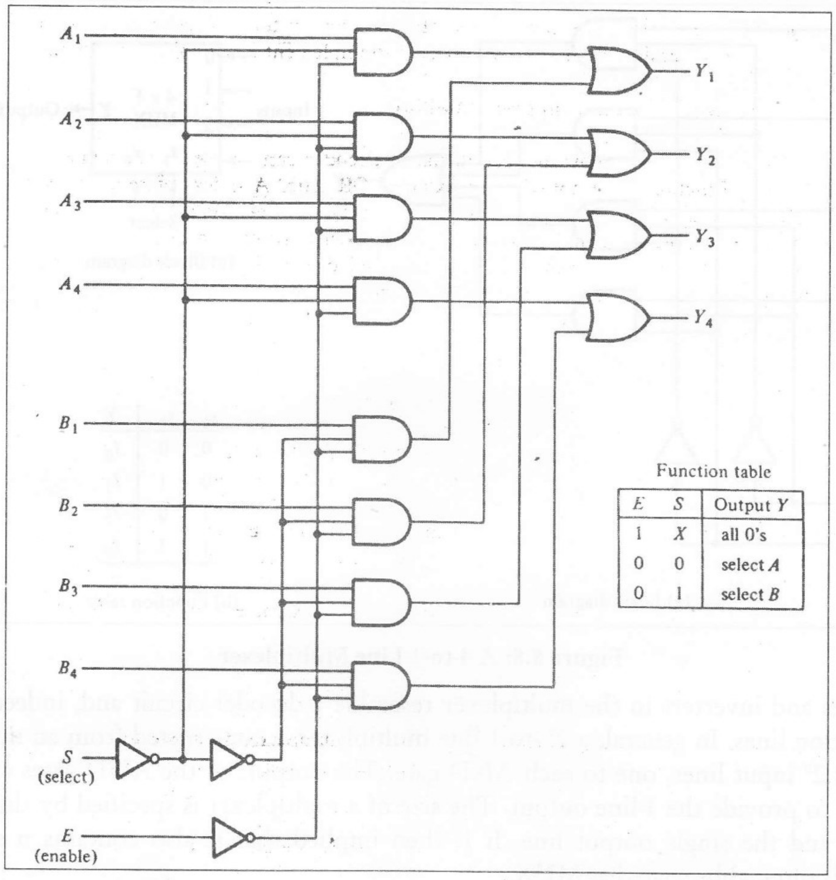
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure 8.8: A 4-to-1 Line Multiplexer

The AND gates and inverters in the multiplexer resemble a decoder circuit and, indeed, they decode the input selection lines. In general, a 2^n -to-1 line multiplexer is constructed from an n-ton 2^n decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1-line output. The size of a multiplexer is specified by the number 2^n of its input lines and the single output line. It is then implied that it also contains n selection lines. A multiplexer is often abbreviated as MUX.

As in decoder, multiplexer ICs may have an enable input to control the operation of the unit. When the enable input is in a given binary state, the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as a normal multiplexer. The enable input (sometimes called strobe) can be used to expand two or move multiplexer ICs to a digital multiplexer with a larger number of inputs.

In some cases two or more multiplexers are enclosed within one IC package. The selection and enable inputs in multiple-unit ICs may be common total multiplexers. As an illustration, a quadruple 2-line to1-line multiplexer IC is shown in figure. It has four multiplexers, each capable of selection one of two input lines. Output Y1 can be selected to be equal to either A_1 or B_1. Similarly, output Y_2 may have the value of A_2 or B_2, and so on. One input selection line, S, suffices to select one of two lines in all four multiplexers. The control input E enables the multiplexers in the O state and disables them in the 1 state. Although the circuit contains four multiplexers, we may think of it as a circuit that selects one in a pair of 4-input lines. As shown in the function table, the unit is selected when E = 0. Then, if S = 0, the four A inputs have a path to the outputs. On the other hand, if S = 1, the four B inputs are selected. The outputs have all 0's when E = 1, regardless of the value of S.



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

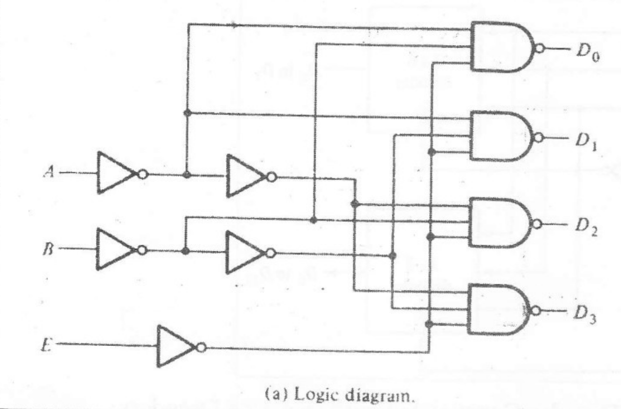
Figure 8.9: Quadruple 2-to-1 Line Multiplexers

The multiplexer is a very useful MSI function and has a multitude of applications. It is used for connecting two or more sources to a single destination among computer units, and it is useful for constructing common bus system.

Some IC decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Most, if not all, IC decoders include one or more enable inputs to control the circuit operation. A 2-to-4 line decoder with an enable input to constructed with NAND gates is shown in figure. All outputs are equal to1 if enable input E is 1, regardless of the values of inputs A and B. When the enable input is 0, the circuit operates as a decoder with complemented outputs. The truth table lists these conditions. The X's under A and B are don't-care conditions. Normal decoder operation occurs only with E = 0, and the outputs are selected when they are in the 0 state.

The block diagram of the selected output is shown in figure. The small circle at input E indicates that the decoder is enabled when E = 0. The small circles at the output indicate that all outputs are complemented.

A decoder with an enable input can function as a demultiplexer. A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output.



(a) Logic diagram.

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

Figure 8.10: A 2-to-4 line Decoder with Enable (E) Input

Line is controlled by the values of n selection lines. The decoder of figure can function as a demultiplexer if the E line is taken as a data input line and lines A and B are taken as the selection lines. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary value of the two selection lines A and B. This can be verified from the truth table of this circuit. For example, if the selection lines AB = 10, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder/demultiplexer. It is the enable input that makes the circuit a demultiplexer; the decoder itself can use AND, NAND, or NOR gates.

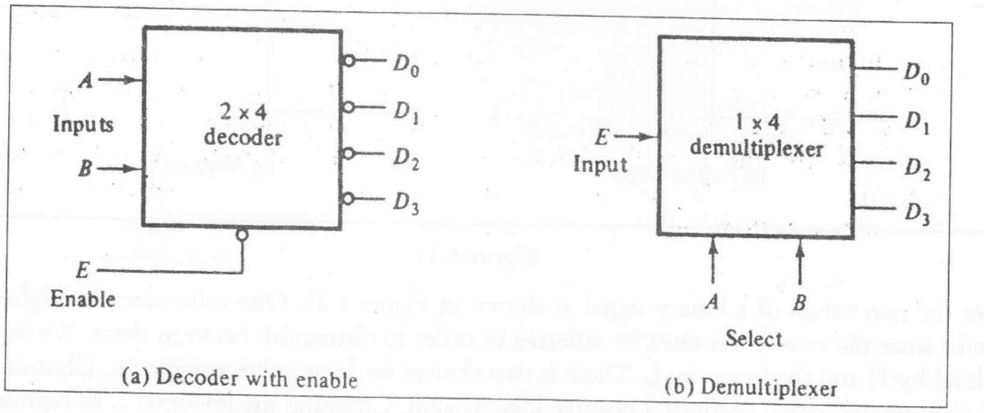


Figure 8.11: Block Diagrams for the Circuit

Decoder/demultiplexer circuits can be connected together to form a larger decoder circuit. Figure 8.12 shows two 3 × 8 decoders with enable inputs connected to form a 4*16 decoder. When w = 0, the top decoder is enabled and the Other is disabled. The bottom decoder outputs are all 0's and the top eight outputs generate minterms 0000 to0111. When w = 1, the enable conditions are reversed; the bottom decoder are all 0's. This example demonstrates the usefulness of enable inputs in ICs. In general, enable lines are a convenient feature for connecting two or more IC packages for the purpose of expanding the digital function into a similar function with more inputs and outputs.

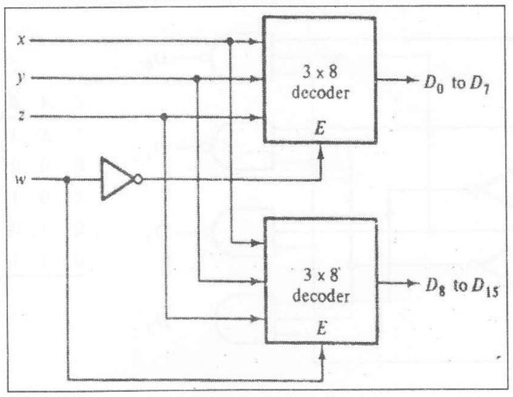


Figure 8.12: A 4×16 Decoder Constructed with two 3×8 Decoders

8.4.4 Positive and Negative Logic

The binary signals at the inputs or outputs of any gate can have only two values except during transition. One signal value represents logic 1 and the other, logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signals to logic. Because of the duality of Boolean algebra, and interchanged of signal-value assignment results in a dual-function implementations.

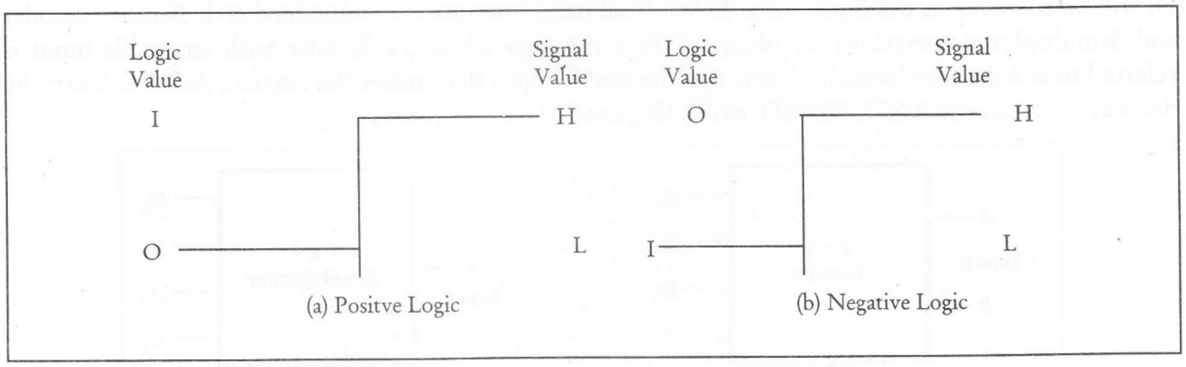


Figure 8.13

Consider the two values of a binary signal as shown in Figure 8.13. One value must be higher than the other must since the two values must be different in order to distinguish between them. We designate the higher level by H and the lower by L. There is two choices for logic value assignment. Choosing the high level H to represent logic-1, defined a positive logic system. Choosing the low-level L to represent logic 1 represent a negative logic system. The term positive and negative are somewhat misleading since both signal values may be positive or both may be negative. It is not signal polarity that determines the type of logic, but rather the assignment of logic values according to the amplitudes of the signals.

8.4.5 Addition, Subtraction, Multiplication and Division Algorithms

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most of the computers carry instructions for all four operations. For computers which have only addition and possibly subtraction instructions, the other two operations i.e. multiplication and division must be generated by means of software subroutines. These four basic arithmetic operations are adequate for providing solutions to scientific problems when expressed in terms of numerical analysis methods.