

UNIT 1:

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.

Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

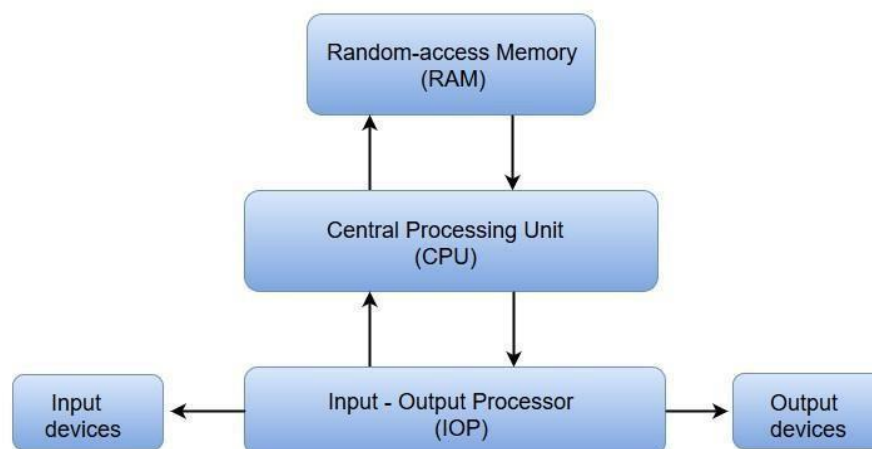
DIGITAL COMPUTERS

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations. By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Block diagram of a digital computer:



- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions

- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

BASIC COMPUTER ORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

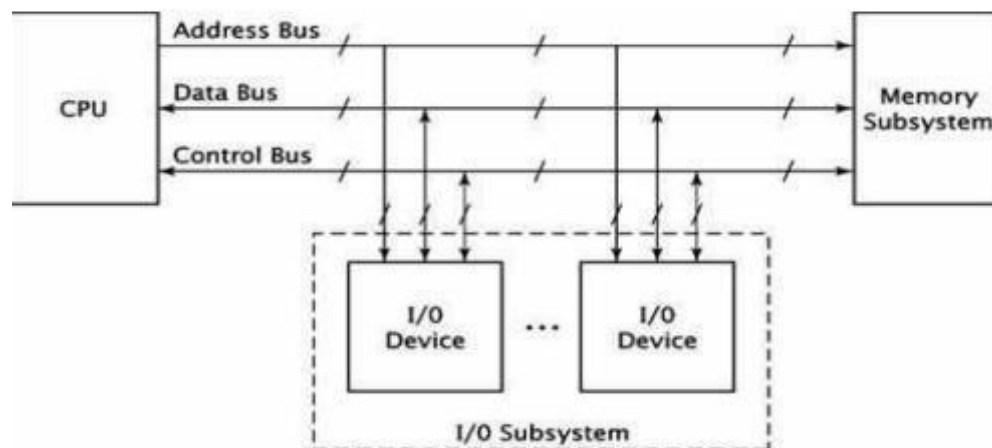


Fig 1.1 Generic computer Organization

Computer organization: Computer organization is the knowing, what the functional components of a computer are, how they work and how their performance is measured and optimized. Computer

Organization refers to the level of abstraction above the digital logic level, but below the operating system level.

Computer design and architecture:

Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation. Computer architecture is concerned with the structure and behavior of the computer as seen by the user.

Register Transfer Language and Micro Operations:

Register Transfer language:

- ☐ Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- ☐ The modules are interconnected with common data and control paths to form a digital computer system
- ☐ The operations executed on data stored in registers are called micro operations
- ☐ A micro operation is an elementary operation performed on the information stored in one or more registers
- ☐ Examples are shift, count, clear, and load
- ☐ Some of the digital components from before are registers that implement micro operations
- ☐ The internal hardware organization of a digital computer is best defined by specifying
 - o The set of registers it contains and their functions
 - o The sequence of micro operations performed on the binary information stored
 - o The control that initiates the sequence of micro operations
- ☐ Use symbols, rather than words, to specify the sequence of micro operations
- ☐ The symbolic notation used is called a register transfer language
- ☐ A programming language is a procedure for writing symbols to specify a given computational process

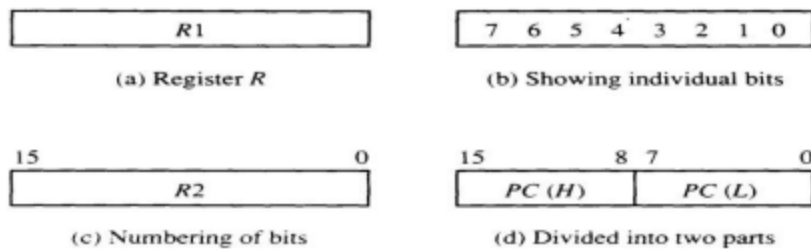
□ Define symbols for various types of micro operations and describe associated hardware that can implement the micro operations

Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC.

- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1
- Refer to Figure 4.1 for the different representations of a register

Figure 4-1 Block diagram of register.

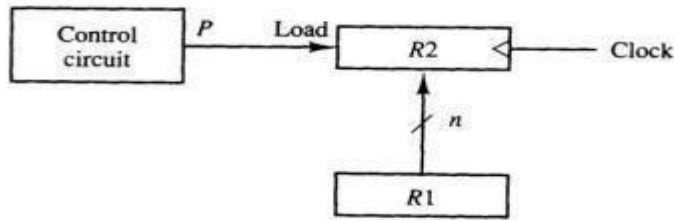


- Designate information transfer from one register to another by $R2 \leftarrow R1$
- This statement implies that the hardware is available
 - The outputs of the source must have a path to the inputs of the destination
 - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by
 - If $(P = 1)$ then $(R2 \leftarrow R1)$
 or,
 - $P: R2 \leftarrow R1,$

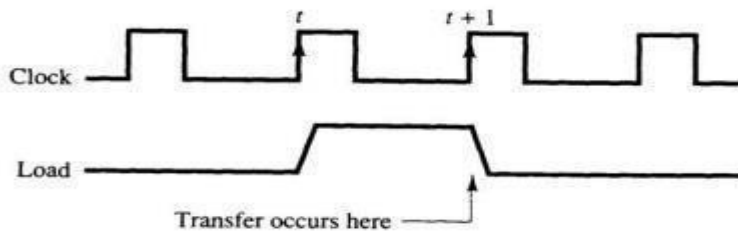
where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



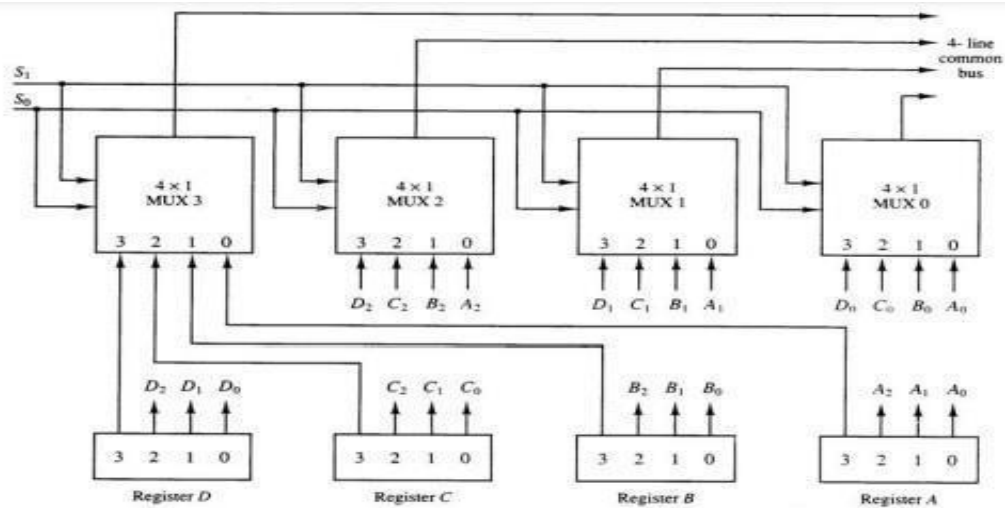
- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time T : $R2 \leftarrow R1, R1 \leftarrow R2$

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$MAR, R2$
Parentheses ()	Denotes a part of a register	$R2(0-7), R2(L)$
Arrow $\leftarrow$	Denotes transfer of information	$R2 \leftarrow R1$
Comma ,	Separates two microoperations	$R2 \leftarrow R1, R1 \leftarrow R2$

Bus and Memory Transfers

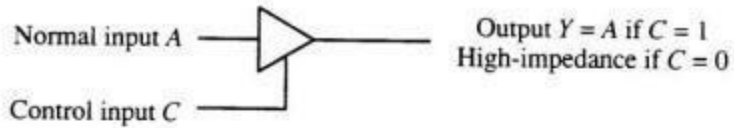
- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer



- In general, a bus system will multiplex k registers of n bits each to produce an n -line common bus
- This requires n multiplexers – one for each bit
- The size of each multiplexer must be $k \times 1$
- The number of select lines required is $\log k$
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as
 BUS $\square$ C, R1 $\square$ BUS,
 use R1 $\square$ C, since the bus is implied



Figure 4-4 Graphic symbols for three-state buffer.



- The three-state buffer gate has a normal input and a control input which determines the output state
- With control 1, the output equals the normal input
- With control 0, the gate goes to a high-impedance state
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects

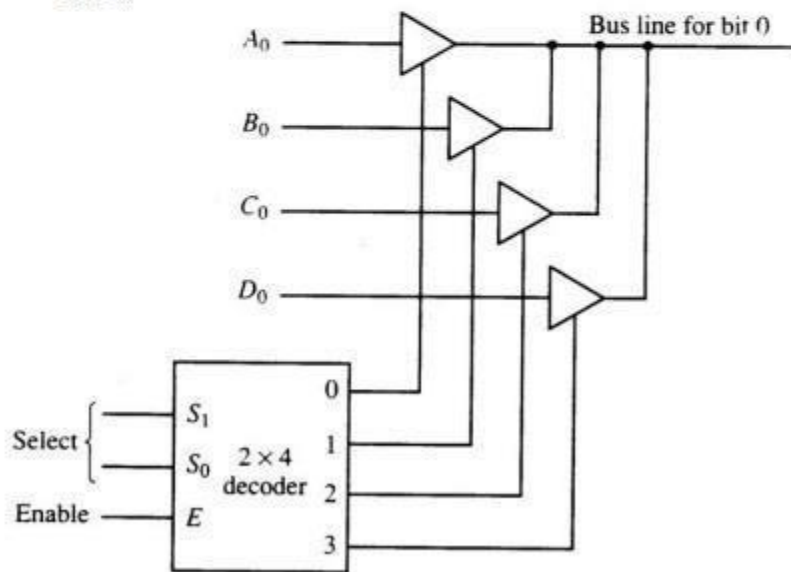


Figure 4-5 Bus line with three state-buffers.

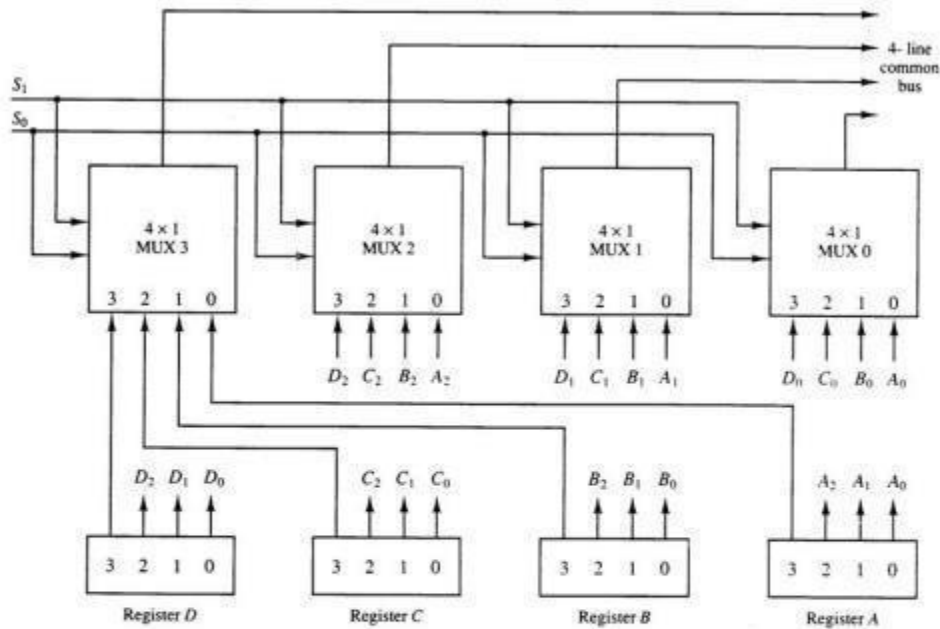
- Designate the address register by AR and the data register by DR
- The read operation can be stated as: Read: DR $\leftarrow$ M[AR]
- The write operation can be stated as:
Write: M[AR] $\leftarrow$ R1

Arithmetic Microoperations

- There are four categories of the most common microoperations:
 - Register transfer: transfer binary information from one register to another
 - Arithmetic: perform arithmetic operations on numeric data stored in registers
 - Logic: perform bit manipulation operations on non-numeric data stored in registers
 - Shift: perform shift operations on data stored in registers
- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift
- Example of addition: R3 $\leftarrow$ R1 + R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 $\leftarrow$ R1 + ~~R2~~ + 1 (strikethrough denotes bar on top – 1's complement of R2)
- Adding 1 to the 1's complement produces the 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting



Figure 4-3 Bus system for four registers.



- Multiply and divide are not included as microoperations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift)
- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry
 - An n-bit binary adder requires n full-adders

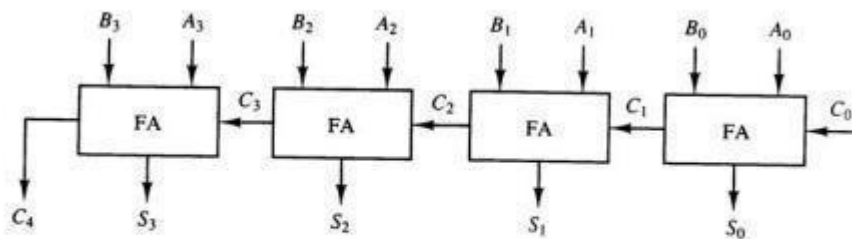


Figure 4-6 4-bit binary adder.

- The subtraction $A-B$ can be carried out by the following steps
 - Take the 1's complement of B (invert each bit)



- Get the 2's complement by adding 1
- Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

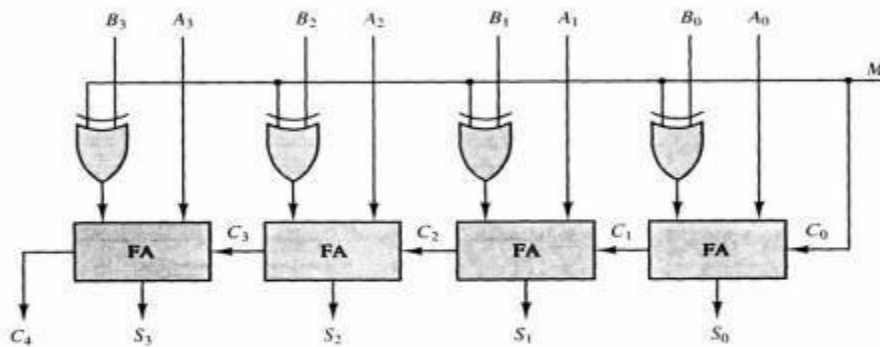


Figure 4-7 4-bit adder-subtractor.

- The increment microoperation adds one to a number in a register
- This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
- If the increment is to be performed independent of a particular register, then use half-adders connected in cascade
- An n-bit binary incrementer requires n half-adders

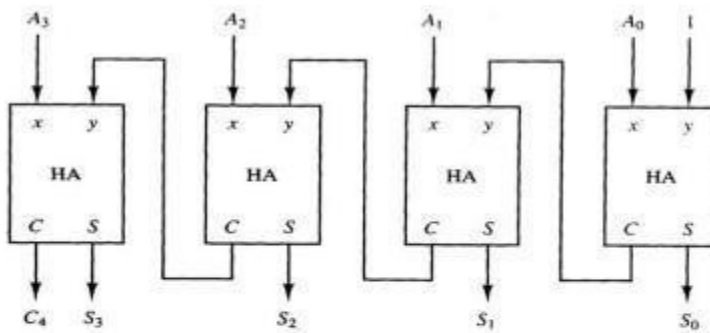


Figure 4-8 4-bit binary incrementer.

- Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

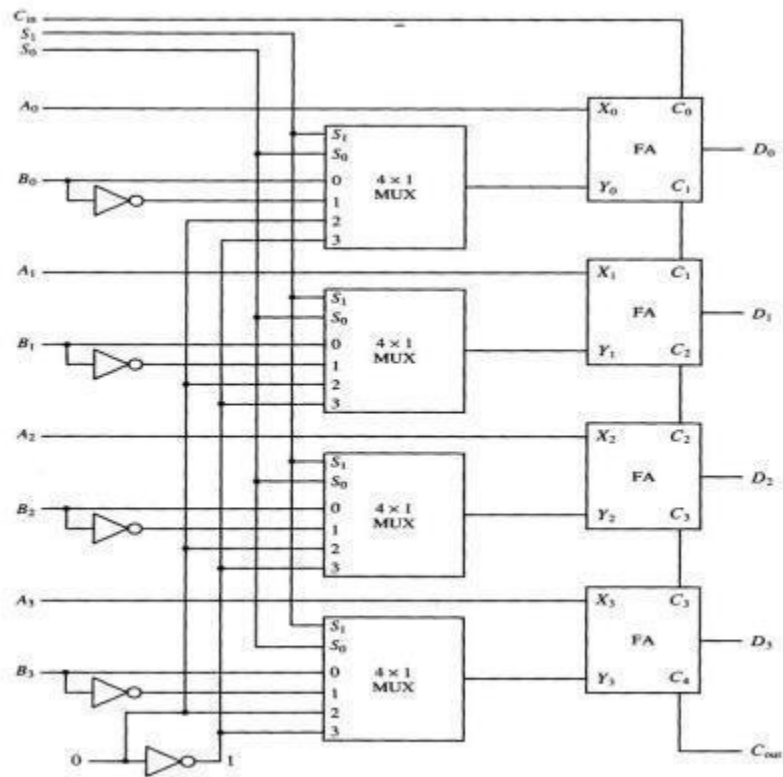


Figure 4-9 4-bit arithmetic circuit.



TABLE 4-4 Arithmetic Circuit Function Table

Select			Input Y	Output $D = A + Y + C_{in}$	Microoperation
S_1	S_0	C_{in}			
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\bar{B}$	$D = A + \bar{B}$	Subtract with borrow
0	1	1	$\bar{B}$	$D = A + \bar{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

$$P: R1 \square R1 \oplus R2$$

- Example: $R1 = 1010$ and $R2 = 1100$
 $\begin{array}{ll} 1010 & \text{Content of R1} \\ \underline{1100} & \text{Content of R2} \end{array}$

Logic Micro operations

□ Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately

□ Example: the XOR of R1 and R2 is symbolized by

$$P: R1 \square R1 \oplus R2$$

□ Example: $R1 = 1010$ and $R2 = 1100$

1010 Content of R1

1100 Content of R2

0110 Content of R1 after $P = 1$

- Symbols used for logical microoperations:
 - OR: $\vee$
 - AND: $\wedge$
 - XOR: $\oplus$
- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:
 - $P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$
- There are 16 different logic operations that can be performed with two binary variables

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

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

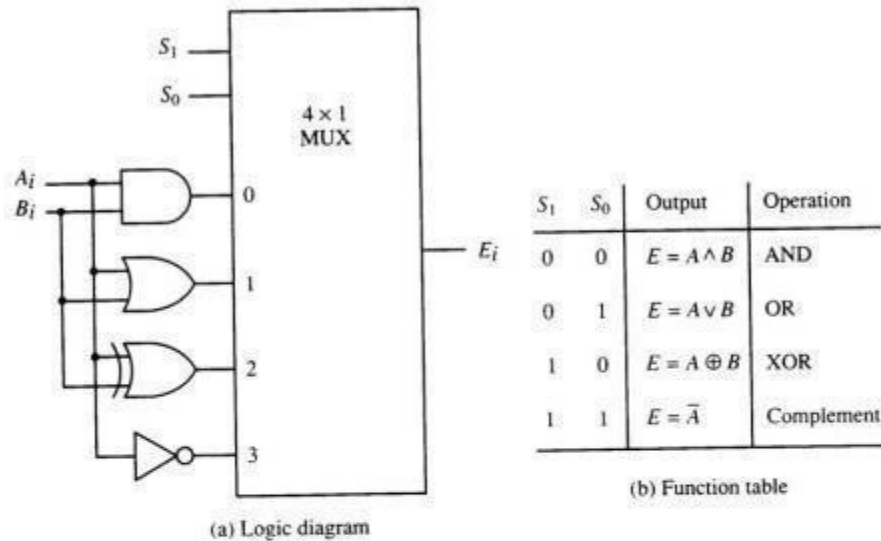
TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers

- All 16 microoperations can be derived from using four logic gates.

Figure 4-10 One stage of logic circuit.



- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The selective-set operation sets to 1 the bits in A where there are corresponding 1's in B

1010 A before
 1100 B
 (logic
 operand)
 1110 A
 after

$$A \leftarrow A \vee B$$

- The selective-complement operation complements bits in A where there are corresponding 1's in B

1010 A before
 1100 B
 (logic
 operand)
 0110 A
 after

$$A \leftarrow A \oplus B$$

- The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B

1010 A before
 1100 B
 (logic

operand)
 0010 A
 after
 A □ A □ B

- The mask operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

1010 A before
1100 B
 (logic
 operand)
 1000 A
 after
 A □ A □ B

- The insert operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010 A before
0000 1111 B (mask)
 0000 1010 A after masking



0000	1010	A before
<u>1001</u>	<u>0000</u>	B (insert)
1001	1010	A after insertion

- The clear operation compares the bits in A and B and produces an all 0's result if the two number are equal

1010	A
<u>1010</u>	B
0000	$A \square A \oplus B$

Shift Microoperations

- Shift microoperations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data- processing operations
- There are three types of shifts: logical, circular, and arithmetic
- A logical shift is one that transfers 0 through the serial input
- The symbols shl and shr are for logical shift-left and shift-right by one position $R1 \square \text{shl}R1$
- The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols cil and cir are for circular shift left and right



TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

- The arithmetic shift shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
- An overflow flip-flop V_s can be used to detect

$$\text{the overflow } V_s = R_{n-1} \oplus R_{n-2}$$

**Figure 4-11** Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift



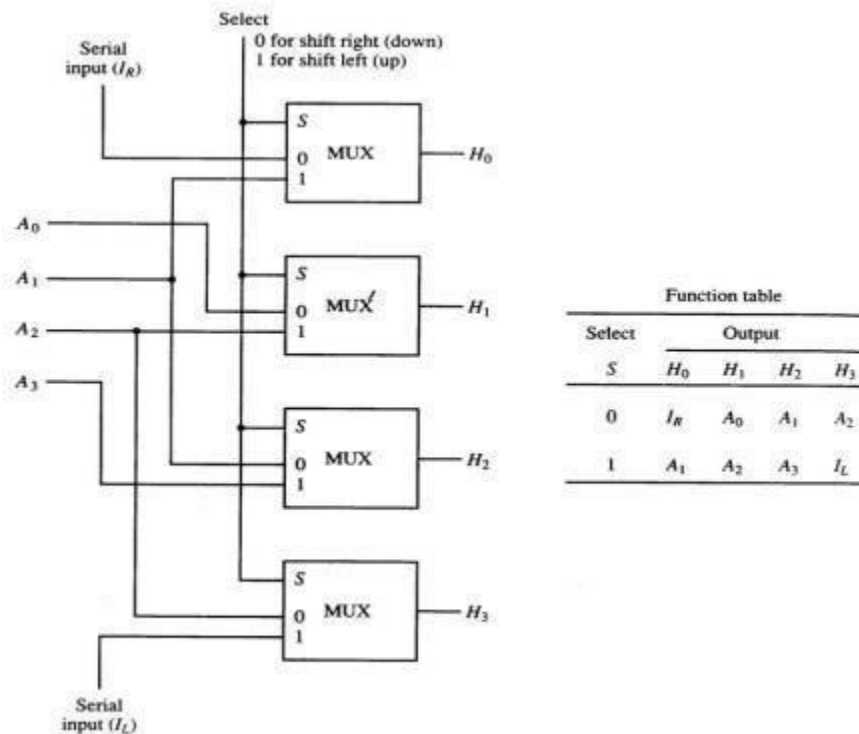


Figure 4-12 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

□ The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers

□ To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU

The ALU performs an operation and the result is then transferred to a destination register

□ The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

Figure 4-13 One stage of arithmetic logic shift unit.

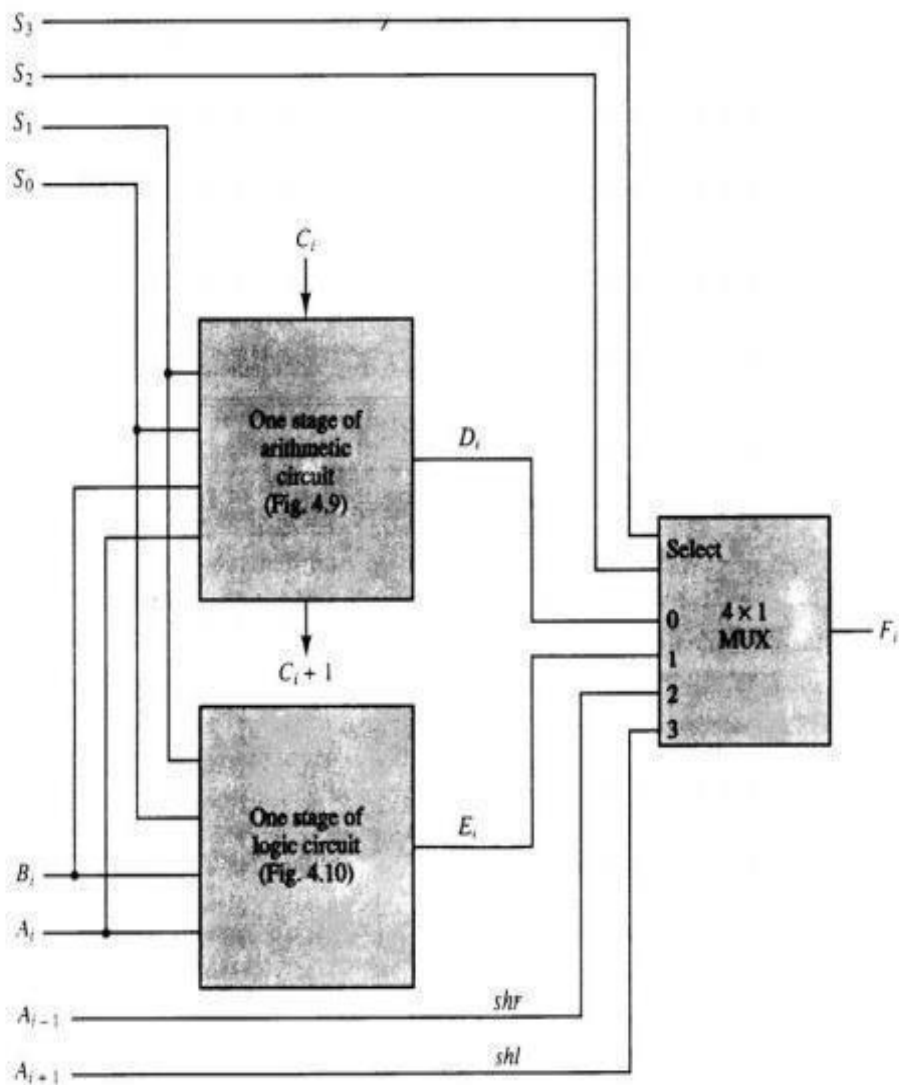


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	$\times$	$F = A \wedge B$	AND
0	1	0	1	$\times$	$F = A \vee B$	OR
0	1	1	0	$\times$	$F = A \oplus B$	XOR
0	1	1	1	$\times$	$F = \overline{A}$	Complement A
1	0	$\times$	$\times$	$\times$	$F = \text{shr } A$	Shift right A into F
1	1	$\times$	$\times$	$\times$	$F = \text{shl } A$	Shift left A into F

Instruction Formats:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

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

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

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

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5. Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations: 1 Single accumulator organization. 2 General register organization. 3 Stack organization. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD, Where X is the address of the operand. The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X. An example of a general register type of organization was presented in Fig. 7.1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 To denote the operation $R1 \leftarrow R2 + R3$. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction ADD R1, R2 Would denote the operation $R1 \leftarrow R1 + R2$. Only register addresses for R1 and R2 need be specified in this instruction. Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 Denotes the transfer $R1 \leftarrow R2$ (or $R2 \leftarrow R1$, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation $R1 \leftarrow R + M[X]$. It has two address fields, one for register R1 and the other for the memory address X. The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction PUSH X Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack. To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement $X = (A + B) * (C + D)$. Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB,

MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X. Three-Address Instructions Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates $X = (A + B) * (C + D)$ is shown below, together with comments that explain the register transfer.

operation of each instruction.

ADD R1, A, B $R1 \leftarrow$

$M[A] + M[B]$

ADD R2, C, D $R2 \leftarrow$

$M[C] + M[D]$

MUL X, R1, R2 $M[X]$

$\leftarrow R1 * R2$

It is assumed that the computer has two processor registers, R1 and R2. The symbol $M[A]$ denotes the operand at memory address symbolized by A.

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

Two-Address Instructions

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

follows:

MOV R1, A $R1 \leftarrow M[A]$

ADD R1, B $R1 \leftarrow R1 + M[B]$

MOV R2, C $R2 \leftarrow M[C]$

ADD R2, D $R2 \leftarrow R2 + M[D]$

MUL R1, R2 $R1 \leftarrow R1 * R2$

MOV X, R1 $M[X] \leftarrow R1$

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

One-Address Instructions

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

$(A + B) * (C + D)$ is

LOAD A $AC \leftarrow M[A]$

ADD B $AC \leftarrow AC + M[B]$

STORE T $M[T] \leftarrow AC$

LOAD C $AC \leftarrow M[C]$

ADD D $AC \leftarrow AC + M[D]$

MUL T $AC \leftarrow AC * M[T]$

STORE X $M[X] \leftarrow AC$

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

Zero-Address Instructions

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

written for a stack organized computer. (TOS stands for top of stack)

PUSH A $TOS \leftarrow A$

PUSH B TOS $\leftarrow$ B

ADD

PUSH C

TOS $\leftarrow$ (A + B)

TOS $\leftarrow$ C

PUSH D TOS $\leftarrow$ D

ADD TOS $\leftarrow$ (C + D)

MUL

POP X

TOS $\leftarrow$ (C + D) * (A + B)

M[X] $\leftarrow$ TOS

To evaluate arithmetic expressions in a stack computer, it is necessary

to convert the expression into reverse Polish notation. The name “zeroaddress” is given to this type of computer because of the absence of an address field in the computational instructions.

Instruction Codes

A set of instructions that specify the operations, operands, and the sequence by which processing has to occur. An instruction code is a group of bits that tells the computer to perform a specific operation part.

Format of Instruction

The format of an instruction is depicted in a rectangular box symbolizing the bits of an instruction. Basic fields of an instruction format are given below:

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

Computers may have instructions of different lengths containing varying number of addresses. The number of address field in the instruction format depends upon the internal organization of its registers.

Addressing Modes

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

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

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

There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence. In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

1. The operation code specified the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

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

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

Op code Mode Address

Figure 1: Instruction format with mode field

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

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

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

3 Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A kbit field can specify any one of 2^k registers.

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

5. Auto increment or Auto decrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction

and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in previous chapter.

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

7 Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

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

9 Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation. Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any

one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

10 Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

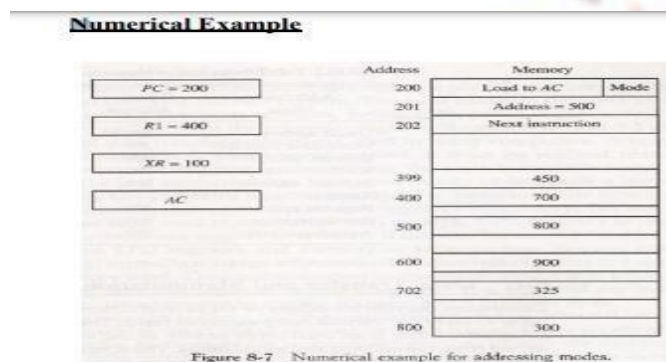


TABLE 8-4 Tabular List of Numerical Example

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

Computer Registers

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

❑ Data Register(DR) : hold the operand(Data) read from memory

❑ Accumulator Register(AC) : general purpose processing register

❑ Instruction Register(IR) : hold the instruction read from memory

❑ Temporary Register(TR) : hold a temporary data during processing

❑ Address Register(AR) : hold a memory address, 12 bit width

❑ Program Counter(PC) :

»hold the address of the next instruction to be read from memory after the current instruction is executed

»Instruction words are read and executed in sequence unless a branch instruction is encountered

»A branch instruction calls for a transfer to a nonconsecutive instruction in the program

»The address part of a branch instruction is transferred to PC to become the address of the next instruction

Input Register(INPR) : receive an 8-bit character from an input device

❑ Output Register(OUTR) : hold an 8-bit character for an output device

The following registers are used in Mano's example computer.

Register Number Register Register

symbol of bits name Function-----

DR 16 Data register Holds memory operands

AR 12 Address register Holds address for memory

AC 16 Accumulator Processor register

IR 16 Instruction register Holds instruction code

PC 12 Program counter Holds address of instruction

TR 16 Temporary register Holds temporary data

INPR 8 Input register Holds input character

OUTR 8 Output register Holds output character

Computer Instructions:

The basic computer has 16 bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

1. Memory Reference – These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12 bit address, 3 bit opcode (other than 111) and 1 bit addressing mode for direct and indirect addressing.

Example –

IR register contains = 0001XXXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

Hence, $DR \leftarrow M[AR]$

$AC \leftarrow AC + DR$, $SC \leftarrow 0$

2. Register Reference – These instructions perform operations on registers rather than memory addresses. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.

Example –

IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

Hence, $AC \leftarrow \sim AC$

3. Input/Output – These instructions are for communication between computer and outside environment. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.

Example –

IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputting character. Hence, INPUT character from peripheral device.

Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register. At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

- ☐ A hardwired control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).
- ☐ A microprogrammed control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the microoperations needed by a particular instruction.

Instruction Cycle

The CPU performs a sequence of microoperations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution

a. Instruction 1

i. Fetch instruction

ii. Decode ,Fetch operand

iii. Execute

b. Instruction 2

i. Fetch instruction , ii Decode

ii. Fetch operand

iii. Execute

c. Instruction 3 ...

Program execution begins with:

$PC \leftarrow \text{address of first instruction}, SC \leftarrow 0$

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

Instruction Fetch and Decode

The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: $AR \leftarrow PC$

T1: $IR \leftarrow M[AR], PC \leftarrow PC + 1$,T2: $D0-7 \leftarrow \text{decoded } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

For every timing cycle, we assume $SC \leftarrow SC + 1$ unless it is stated that $SC \leftarrow 0$.