UNIT I

BASIC STRUCTURE OF COMPUTER AND DATA REPRESENTATION

Syllabus:
Basic Structure Of Computers : Computer Types, Functional unit, Basic Operational concepts, Bus structures
Data Representation: Data types, Complements, Fixed Point Representation. Floating – Point Representation. Other Binary Codes, Error Detection codes.

Definition of Computer Organization: It describes the hardware components, their inter connection and their working to make up a computer system.

BASIC STRUCTURE OF COMPUTERS:

COMPUTER TYPES:

A digital computer or simply a computer is a fast electronic calculating machine that accepts digitized information as input, processed according to the stored instructions, produces result as the output. The list of instructions is called a computer program and the internal storage is called the computer memory.

Computers can be classified into different times based on their size, cost and computational power. Following are various types of computers.

1. Personal computers(desktops)
2. Notebook computers
3. Workstations
4. Mainframes(Enterprise systems)
5. Super computers

The most common type of computer is the personal computer. This is commonly found in homes, schools and offices. This type of computers are also called as desktop computers. These are having a keyboard, mouse as input device. Monitor, printer, audio as output units. It also contains storage and processing units. The desktops use storage medium like CD ROMS, DVDs, hard disks, diskettes.

Portable notepad computers are a compact version of personal computer with all these components are packed into a single unit. It has size of thin briefcase.

Workstations have high resolution input/output capability. These computers have more computational power than the PCs. Workstations are used in engineering applications, especially for interactive design work.

Mainframes or Enterprise systems are used for business data processing in medium or large scale organizations. Mainframes require much more storage capacity servers and require more computing power than the workstations. These main frame severs are accessed by the education, business, personal user communities.

Super computers are used for large scale numerical calculations. These are used in applications such as weather forecasting, aircraft design and simulation, scientific calculations. The super computers have multiple functional units and multiple processors.

FUNCTIONAL UNITS:

A computer consists of 5 functionally independent main parts:

1. Input Unit
2. Output Unit
3. Memory Unit
4. Arithmetic Logic Unit(ALU)
5. Control Unit(CU).

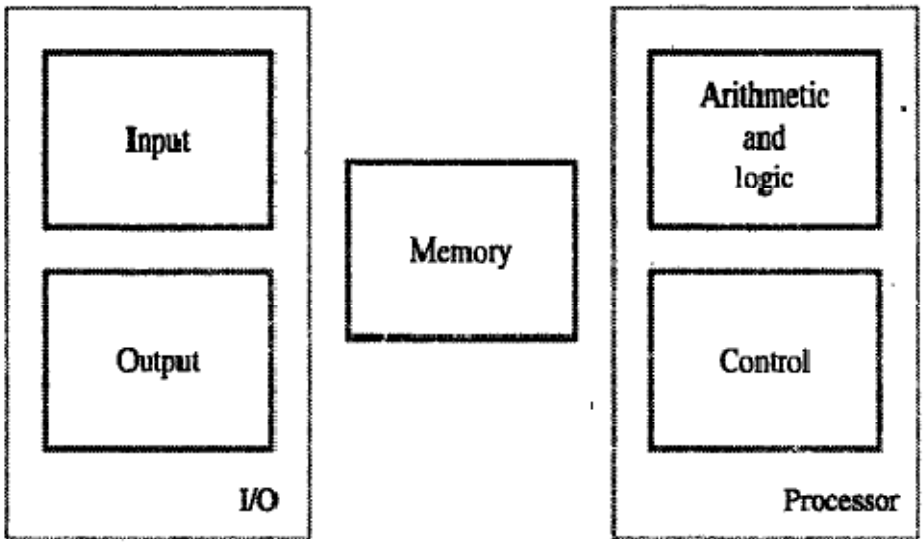


Figure 1.1 Basic functional units of a computer.

The input unit accepts the coded information from the electromechanical devices such as keyboards. The information received either stored in computer memory or immediately used by the ALU to perform the desired operations. The processing steps are determined by the program stored in the memory. Finally the results are sent to the outside world by the output unit.

Fig 1.1 shows the connections among the functional units. We refer the ALU and Control circuits as the processor. Input output equipment is collectively called input-ouput(I/O) unit.

Input unit:

Computers accept coded information through the input units. The common input device is the keyboard. Whenever a key is pressed the corresponding letter or digit is automatically translated into corresponding binary code and transmitted over a cable to either the memory or processor.

Numbers are represented in computer in Binary-Coded Decimal(BCD) format. Alpha numeric characters are expressed by using any one of the two coding schemes.

ASCII(American Standard Code for Information Interchange) which is a 7-bit code or EBCDIC(Extended Binary-Coded Decimal Interchange Code) which is 8-bit code.

Many kinds of input devices are available such as trackball, joystick and mouse's. These are called graphic input devices. Microphones can be used to capture the audio input and which is then converted into digital code for storage and processing.

Memory Unit:

The function of memory unit is to store program and data. There are two types of storage:

- Primary storage
- Secondary Storage

The primary storage is a fast memory that operates at electronics speeds. The memory is made up semiconductor cell and each cell is capable of storing one bit of information. These cells are processed in groups of fixed size called words. A distinct address is associated to each word location for easy access. A given word can be accessed by specifying its address and issuing a control command. The word lengths range from 16 to 64 bits.

Programs must reside in the primary memory during execution. The processor reads program and data from the memory. The RAM (Random Access memory) locates and the data in short and fixed amount of time (access time), independent of the location. The smallest and fast RAM units are called caches. The larges and slowest unit is referred to as the main memory.

Although the primary memory is essential it is very expensive. Additional cheaper secondary storage (magnetic tapes, disks, CD-ROMS) is used for the data large amount of data the is accessed infrequently.

Arithmetic and Logic Unit(ALU):

Most computer operations are executed in ALU of the processor. For example two numbers that are in the memory are to be added. The two numbers are brought into the processor and the actual addition is carried out by the ALU.

Any arithmetic or logic operation such as multiplication, division or comparison of numbers, is done by bringing the required operands into the processor and stored in the registers and the operation is performed by the ALU.

The control and Arithmetic Logic units are many time fasters than the other devices connected to the computer system. This enables a single processor to control the number of devices such as keyboards, displays, disks, sensors and mechanical devices.

Output Unit:

Its function is to send the processed results to the outside world. The most familiar examples of such devices are monitors and printers. Monitors are graphical display devices. Printers are mechanical devices. Examples of printers include laser, inkjet and line printers.

Control Unit:

The memory, ALU, and I/O devices store and process information and perform the input and output operations. These devices should be coordinated in some way. The Control Unit(CU) is responsible for controlling and coordinating different devices activities. The CU sends the control signals to various devices and senses their states. The CU controls the I/O transfers and data transfers between processors and memory through the timing signals. The control unit interacts with the other parts of the machine in well-define manner. A large set of control lines (buses) carries the signals to all the units.

The operation of a computer can be summarized as follows:

- **The computer accepts information in the form of programs and data through an input unit and stores it in the memory.**
- **Information stored in the memory is fetched, under program control, into an arithmetic and logic unit, where it is processed.**
- **Processed information leaves the computer through an output unit.**
- **All activities inside the machine are directed by the control unit.**

BASIC OPERATIONAL CONCEPTS:

Instructions and data are stored in the memory. To perform a given task, the instructions and data are brought from the memory to the processor. Consider the following instruction:

ADD LOCA, R0

This instruction adds the operand at the memory location LOCA to the operand in a register of a processor R0, and places the sum into register R0.

The above ADD instruction combines a memory access operation with an ALU operation. The above operation can be done with the following two-instruction sequence.

LOAD LOCA,R1

ADD R1,R0

Fig 1.2 shows how the memory and processor can be connected.

In addition to the ALU and the control circuitary, the processor contains number of registers used for several different purposes.

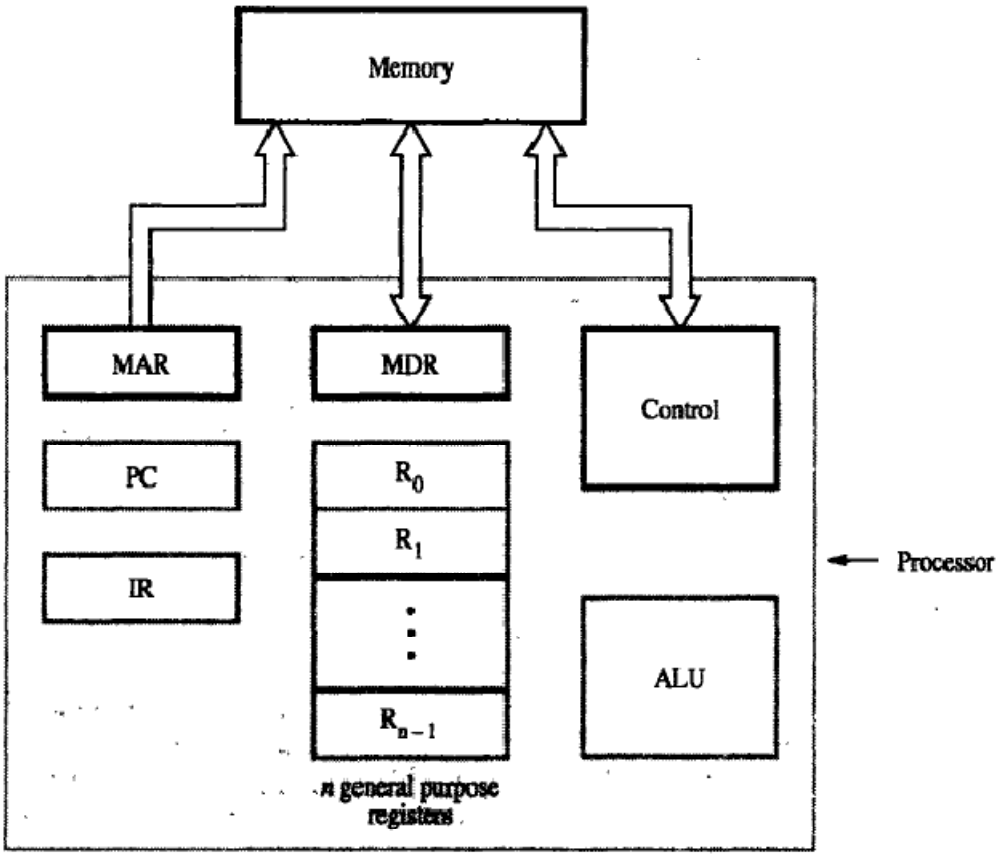


Figure 1.2 Connections between the processor and the memory.

The Instruction Register(IR) holds the instruction that is currently being executed. The IR output generates the timing signals the controls the various processing elements involved in the executing the instruction.

The Program Counter(PC) is the specialized register, that contains the address of the next instruction to be executed. The PC points to the next instruction to be fetched from the memory.

Beside PC and IR, there are n general purpose registers R0...Rn-1.

The Memory Address Register (MAR) and Memory Data Register (MDR) facilitate communication with the memory. The MAR holds the address of the location to be accessed. The MDR contains the data to be written into or read out of the addressed location.

The processor unit shown in figure 1.2 is usually implemented on a single Very Large Scale Integrated(VLSI) chip.

BUS STRUCTURES:

To form an operational system, the functional units of the computer are connected by using bus. A bus is contains a group of lines that serves as a connecting path for several devices. There are different types of buses:

- Address bus
- Data bus
- Control bus

A data bus carries the data words between the devices. The Address bus and control bus carries address and control information.

The simplest way to interconnect functional units is to use a single bus, as shown in the following fig:

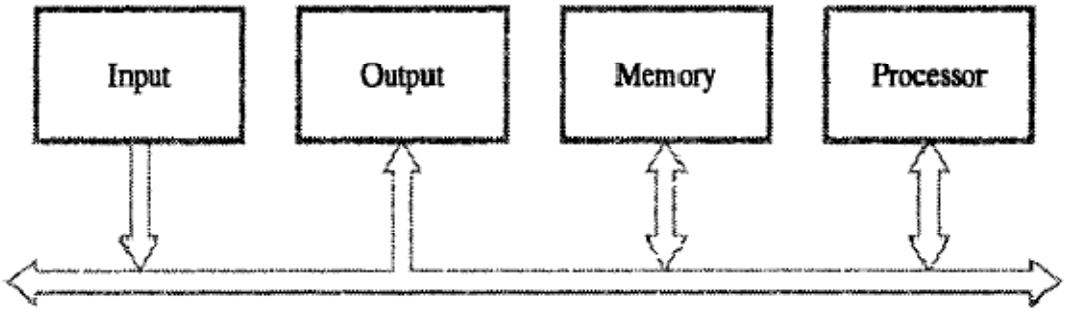


Figure 1.3 Single-bus structure.

All units are connected to one bus. In single bus structure only two units can actively use the bus at a given time. The advantage of the single bus structure is low cost and the flexibility for attaching the peripheral devices.

The system that contains multiple buses allows the two or more transfers to be carried out at the same time. It leads to better performance but at an increased cost.

The devices connected to the bus vary widely in the speed of their operation. The keyboards, Printers are relatively slow where as the disks are considerably faster. Memory and processor units are fastest parts of computer that operate at electronic speeds.

A common approach is used to eliminate the timing differences between various devices and facilitate the smooth communication is using the buffer registers with the devices. This approach prevents the high speed processor from being locked to a slow I/O device during a sequence of data transfers.

DATA REPRESENTATION:

DATA TYPES:

Binary information in computers is stored in memory or process registers. The data types may be classified into the following categories:

1. Numbers used in arithmetic computation.
2. Letters of the alphabet used in data processing.
3. Other discrete symbols used for specific purposes.

All types of data are represented in computer in binary coded form. The binary system is the most natural system to use in a digital computer. There are the other types of number systems as follows:

1) Decimal Number system.
2) Octal number system.
3) Hexa-decimal number system.

Number systems:

Numbers consisting of string of digits. Base or radix(r) can be used to identify the number system of given number. The binary system uses the radix 2. For the decimal number the radix is 10, for the octal number the radix is 8 and for the hexa-decimal system the radix is 16.

Decimal number system:

The decimal number system has the base or radix 10. It uses the digits 0 to 9. The string of digits 724.5 is interpreted as follows:

$$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

Ie. 7 hundreds, plus 2 10's, plus 4 1's, plus 5 tenths.

Conversion of decimal integer into base r representation is done by successive divisions of r and accumulation of the remainders. The conversion of decimal fraction to radix r representation is accomplished

Examples:

1) Conversion of decimal number into binary

Integer = 41		Fraction = 0.6875
41		0.6875
20	1	2
10	0	1.3750
5	0	x 2
2	1	0.7500
1	0	x 2
0	1	1.5000
		x 2
		1.0000

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

Binary Number system:

The binary number system uses the radix 2. The two symbols used are 0 and 1. Consider the following example for converting the given binary number to decimal number.

Examples:

$(101101)_2 = (?)_{10}$

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

$$(101101)_2 = (45)_{10}.$$

Octal Number System:

The octal number system or radix 8 number system consisting of digits 0 to 7. A number with radix r is converted to the decimal system by forming the sum of the weighted digits. Forexample. Octal 736.4 is converted to decimal as follows:

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$$
$$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$$

Hexa-Decimal Number system:

The Hexa-Decimal Number system or radix 16 number system consisting of digits 0 to 15.

Example:

$(F3)_{16} = (?)_{10}$

$$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3 = (243)_{10}$$

Conversion between binary, octal and hexa-decimal number systems:

The conversion from and to binary,octal, and hexa-decimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4=16$ each octal digit corresponds to 3-binary digits and each hexa-decimal number corresponds to 4-binary digits.

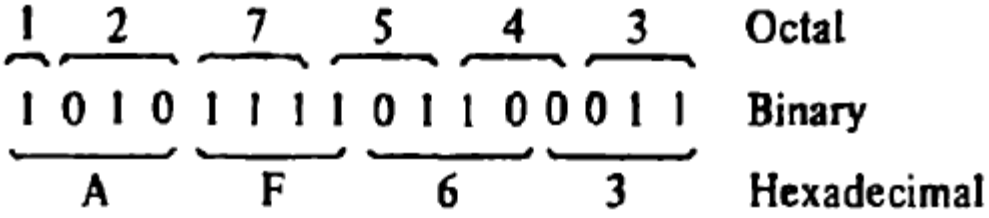


Figure 3-2 Binary, octal, and hexadecimal conversion.

Binary-Coded Decimal numbers(BCD):

In BCD each decimal number can be expressed in 4-digit binary form. The following table represents how the decimal numbers can be represented in BCD-form.

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Alphanumeric Representation:

A alpha numeric character is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters such as $,+,= . The ASCII(American Standard Code for Information Interchange) code is used to represent alpha number characters. It is a 7-bit representation to represent 2^7 = 128 characters which include numbers, upper and lower characters, and special symbols. Decimal digits in ASCII can be converted into BCD by removing the three high-order bits.

TABLE 3-4 American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

COMPLEMENTS:

Complements are used to simplify the subtraction operation and for logical manipulation in digital computers. There are two types of complements for each base r system.

1. r's complement.

2. (r-1)'s complement.

For the binary systems(r=2), the two types of complements are referred to as 2's and 1's complements. For the decimal system(r=10), the two types of complements are referred to as 10's and 9's complement.

(r-1)'s complement:

Given a number N in base r having n digits.

The (r-1)'s complement of N is defined as $(r^n-1) - N$.

For decimal numbers (r-1)'s complement, i.e, the 9's complement of N is: $(10^n - 1) - N$. The 9's complement for decimal number can be obtained by subtraction each digit from 9.

Example: (r-1)'s complement or 9 s complement of 546700 is

999999 – 546700 = 453299

For binary numbers (r-1)'s complement, i.e 1's complement of N is: $(2^n - 1) - N$. Thus, the 1'complement can be obtained by subtracting each digit from 1. Similarly, 1's complement can be obtained by changing 0's into 1's and 1's into 0's.

Example:

1's complement of 1011001 is 0100110

r's complement::

Given a number N in base r having n digits.

The r's complement of N is defined as $r^n - N$. i.e r's complement can be obtained by finding (r-1)'s complement and adding 1.

$$r^n - N = [(r^n - 1) - N] + 1$$

For decimal numbers r's complement, i.e, the 10's complement of N is: $[(10^n - 1) - N] + 1$. For binary numbers r's complement, i.e 2's complement of N is: $[(2^n - 1) - N] + 1$.

Example:

10's complement of the decimal number 2389 is

(9999 – 2389) + 1 = 7610 + 1 = 7611

Example:

2' complement of the binary number 101100 is

(111111 – 101100) + 1 = 010011 + 1 = 010100

Note: 10's complement of N can be obtained by leaving all least significant 0's unchanged, subtracting the first non-zero least significant bit from 10, and subtracting all other higher bits from 9.

Similarly 2's complement of N can be obtained by leaving all least significant 0's unchanged, subtracting and first 1 unchanged and replace all other higher significant bits 1's into 0's and 0's into 1's.

The complement of complement restores the number to is original value.

The r's complement of N is r^n – N. The complement of complement is

$$r^n - (r^n - N) = N$$

N is the original number.

For example:

N = 2389

r's complement of 2389 is : 7611

Again r's complement of 7611 is: 2389 (Original number).

Subtraction of Unsigned numbers:

Positive integers including 0 can be represented as unsigned numbers. In digital computers the subtraction is carried out by the complements. The subtraction of two n-digit unsigned numbers M – N in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

For example:

Consider, for example, the subtraction 72532 − 13250 = 59282. The 10's complement of 13250 is 86750. Therefore:

$$\begin{aligned} M &= \quad 72532 \\ \text{10's complement of } N &= +86750 \\ \text{Sum} &= \quad 159282 \\ \text{Discard end carry } 10^5 &= -100000 \\ \text{Answer} &= \quad 59282 \end{aligned}$$

Now consider an example with $M < N$. The subtraction 13250 − 72532 produces negative 59282. Using the procedure with complements, we have

$$\begin{aligned} M &= \quad 13250 \\ \text{10's complement of } N &= +27468 \\ \text{Sum} &= \quad 40718 \end{aligned}$$

There is no end carry

Answer is negative 59282 = 10's complement of 40718

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction $X - Y$ and $Y - X$ using 2's complements:

$$\begin{aligned} X &= \quad 1010100 \\ \text{2's complement of } Y &= +0111101 \\ \text{Sum} &= \quad 10010001 \\ \text{Discard end carry } 2^7 &= -10000000 \\ \text{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

$$\begin{aligned} Y &= \quad 1000011 \\ \text{2's complement of } X &= +0101100 \\ \text{Sum} &= \quad 1101111 \end{aligned}$$

There is no end carry

Answer is negative 0010001 = 2's complement of 1101111

SIGNED NUMBERS:

To represent negative values we need a notation. In ordinary arithmetic, a negative number is indicated by the minus sign and positive number by plus sign. Due to hardware limitations, computers must represent everything with 1's and 0's, including the sign of the number.

Computers use leftmost bit as a sign bit of the number. If the leftmost bit is 0 it is positive, if leftmost bit is 1 the number is negative.

Signed integer representation:

When integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1, the rest of the number may be represented by one of the three possible ways:

1. Signed-magnitude representation
2. Signed-1's complement representation
3. Signed 2's complement representation

In sign magnitude representation of a negative number consists of magnitude and negative sign. In other two representations, the negative number is represented in either 1's or 2's complements of the positive value.

For example +14 can be represented in way as follows:

00001110

However, there are 3 ways to present the -14:

In signed-magnitude representation	1 0001110
In signed-1's complement representation	1 1110001
In signed-2's complement representation	1 1110010

The sign magnitude representation of -14 can be obtained by the +14 by complementing only the sign bit.

The signed 1's complement representation of -14 can be obtained by complementing all the bits of +14 including the sign bit.

The signed 2's complement representation of -14 can be obtained by complementing all the bits of +14 including the sign bit.

Computer makes use of sign 2's complement representation for negative numbers.

Arithmetic addition for signed numbers(using 2's) complement:

Adding two numbers in signed 2's complement system require addition and complementation. Procedure for 2's complement addition is as follows:

1) Obtain 2's complement for negative numbers.
2) Add the two numbers
3) Discard the carry.

Example:

+6	00000110	−6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111
+6	00000110	−6	11111010
−13	11110011	−13	11110011
−7	11111001	−19	11101101

In the above examples, the result is negative. i.e It is in 2's complement form. To obtain the positive number we have to 2' complement the result.

Arithmetic subtraction for signed numbers(using 2's) complement:

Subtraction of two signed numbers when negative numbers are in 2's complement form is very simple and can be stated as follows:

Procedure:

1) Find the 2's complement of subtrahend including the sign bit
2) Add the result to minuend
3) Discard the carry

This is demonstrated by the following relationship.

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

Example:

11111010 – 11110011

The subtraction can be changed to the addition by obtaining the 2'scomplement for the subtrahend(i.e the second number)

i.e 11111010 + 00001101 = 100000111

There is carry(left most bit) can be discarded

Finally, The result is 00000111 which is +7.

Overflow:

The overflow is problem in digital computers because the width of the registers is finite. i.e The result contains n+1 bits cannot be accommodated in a register with a standard length of n-bits. In many computers when overflow occurs overflow flag in flag register will be set.

An overflow cannot occur when one number is positive and another number negative.An overflow can occur when two numbers are either positive or negative.

Example:

Consider the following additions

carries:	0	1	carries:	1	0
+70	0	1000110	−70	1	0111010
+80	0	1010000	−80	1	0110000
+150	1	0010110	−150	0	1101010

From the above example 9-bit answer are produced which cannot be accommodated in 8-bit register. Hence, we say an overflow occurred.

An overflow condition can be detected by observing the carry into the sign bit position and carry out of the sign bit position. If the two carriers are not equal, an overflow condition is produced.

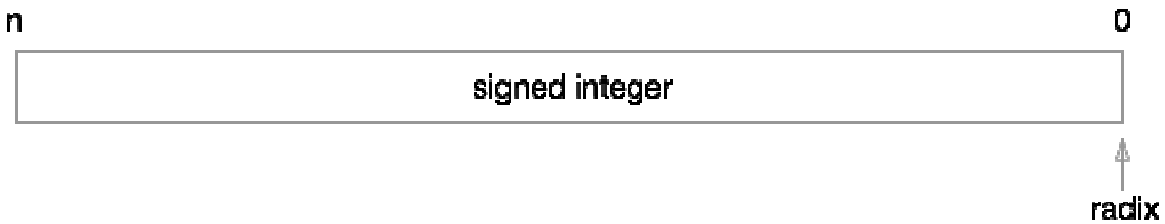
REPRESENTATION FOR DECIMAL OR BINARY POINTS IN NUMBERS:

In addition to sign, a number may have binary or decimal point. There are two ways to represent the binary point in a register:

1) Fixed point representation.
2) Floating point representation.

FIXED POINT REPRESENTATION:

The fractional numbers in the memory are represented by the fixed point representation. In Fixed Point Notation, the number is stored as a signed integer in two's complement format.



The fixed point representation assumes that the binary point always fixed in one position. Fixed point representation uses two positions widely:

1. Binary point in the extreme left of the register for the fractional numbers.
2. Binary point in the extreme right of the register for the integer numbers.

Binary point is not actually present, but its presence is assumed.

We reposition the radix point by multiplying the stored integer by a fixed scaling factor. As the scaling factor is a power of 2 it relocates the radix point some number of places to the left or right of its starting position.

During this conversion there are three directions that the radix point can be moved:

- **The radix point is moved to the right:** This is represented by a1 scaling factor whose exponent is 1 or more. In this case additional zeros are appended to the right of the least-significant bit and means that the actual number being represented is larger than the binary integer that was stored.

- **The radix point remains where it is:** This is represented by a scaling factor whose exponent is 0 and means that the integer value stored is exactly the same as the integer value being represented.

- **The radix point is moved to the left:** This is represented by a scaling factor whose exponent is negative. This means that the number being represented is smaller than the integer number that was stored and means that the number being represented has a fractional component.

Let's take a look at a couple of examples.

Examples of Fixed Point Numbers

Lets assume we have an 8-bit signed binary number 00011011_2 that is stored in memory using 8-bits of storage (hence the leading zeros).

In our first scenario, lets also assume this number was stored as a signed fixed-point representation with a scale factor of 2^2.

As our scale factor is greater than 1, when we translated the bits stored in memory into the number we are actually representing, we move the radix point two places to the right. This gives us the number: 1101100_2 (Note the additional zeros that are appended to the right of the least significant bit).

In our second scenario, let us assume that we start off with the same binary number in memory but this time we'll assume that it is stored as a signed fixed-point representation with a scale factor of 2^{-3}. As the exponent is negative we move the radix point three places to the left. This gives us the number 00011.011_2

Advantages and Disadvantages of Fixed Point Representation

Advantage: The major advantage of using a fixed-point representation is performance. As the value stored in memory is an integer the CPU no additional hardware or software logic is required.

Disadvantage: Fixed Point Representations have a relatively limited range of values that they can represent.

FLOATING POINT REPRESENTATION:

The floating point representation of a number has two parts.

The first part represents a signed fixed point is call mantissa. The second part designates the position of the decimal(or binary) point is called the exponent. The fixed point mantissa may be a fraction or an integer.

For example, the decimal number +6132.789 is represented in floating point with a fraction or exponent as follows:

Fraction	*Exponent*
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point. In our example, the position of the decimal point is four positions right of the indicated decimal point in the fraction.

The above representation is equivalent to the scientific notation:

$$+0.6132789 \times 10^{+4}.$$

Floating point always represent a number in the following form:

$$m \times r^e$$

Only mantissa m and exponent e are physically represented in the computer. The radix r and radix position of the mantissa are always assumed.

A floating point for the binary number is represented in similar manner except that it uses base 2 for the exponent.

For example, the binary number +1001.11 is represented with 8-bt fraction and 6-bit exponent as follows:

Fraction	*Exponent*
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The exponent has the equivalent binary number +4. The floating point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

We can normalize the floating point numbers. Normalized numbers provide the maximum possible precision for the floating point number.

For example, the 8-bit binary number 00011010 is normalized because of three leading 0's. The number can be normalized by shifting it three positions left and discarding leading 0's to obtain 11010000. The three shifts multiply the number by 23 = 8. To keep the same value for the floating point-number the exponent must be subtracted by 3.

Arithmetic operations with floating point numbers are more complicated than the arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware.

However, floating-point representation is must for scientific calculations because scaling problem involved with fixed-point computations. Many electronic devices and computer have the built in capability of performing floating-point arithmetic operations.

OTHER CODES:

OTHER BINARY CODES:

Digital computers have other binary code for special applications. The additional binary code is as follows:

- Gray codes

Gray code:

The analog-to-digital conversion is done by the gray code which is also called reflected binary code. The advantage of gray code is, it changes only one digit (0 to 1 or 1 to 0) to generate the next number in sequence.
Gray code counters are sometimes used to provide the timing sequences.

TABLE 3-5 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

Other decimal codes:
The BCD code is the straight binary representation for the given decimal digit. The disadvantage of BCD representation is by using the BCD representation it is difficult to compute the 9's complement. The 9's complement can be easily obtained by the following codes:

- 2421 code
- Excess-3 code

These two codes have self-complementing property which means that, the 9's complement can be simply obtained by changing 1's to 0's and 0's to 1's. This property is useful when arithmetic operations are done by using signed-complement representation

The 2421 is called the weighted code because, the bits are multiplied by the weights and the sum of weighted bits gives the decimal digit. The BCD-code is also referred ad 8421 code.

The Excess-3 is an unweighted code and it can be obtained from BCD equivalent binary number by addition of binary 3(0011).

Following table shows four different representations for the decimal digit.

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

Other alphanumeric codes:

In ASCII, each character is represented by 7-bit code. This code represents 128 characters. The IBM computers make use of another code which is called EBCDIC(Extended Binary Coded Decimal Interchange Code). EBCDIC has same character set as ASCII but it uses different bit assignment to the characters.

ERROR DETECTION CODES:

When the binary information is transmitted over some communication medium, there is chance bit changes due external noise in transmitted medium.

An error detection code is a binary code that detects digital errors during transmission. The most common error detection is the ***parity bit***. The parity bit is the extra bit included with the binary message make the total number 1's either odd or even.

We have two types of schemes for error detection that uses parity bit.

1) Odd parity scheme
2) Even parity scheme

Consider the following table to add parity bit to the message xyz:

TABLE 3-7 Parity Bit Generation

Message *xyz*	*P*(odd)	*P*(even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

The P(odd) is chosen in such a way that the sum 1's including parity bit is odd. The P(Even) is chosen in such a way that the sum of 1's including parity bit is even.

During the transfer the parity bit is handled as follows:

1)At the sending end, the message is applied to a parity generator to generate the required parity bit.

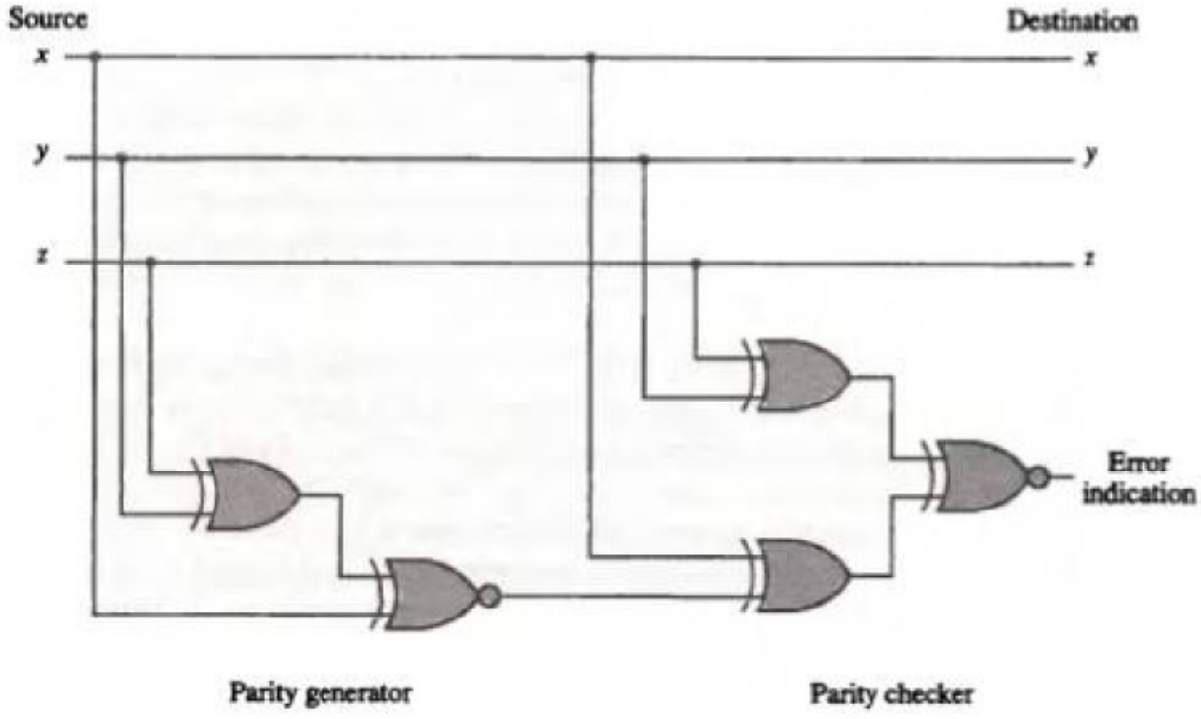
2) The message including the parity bit is transferred to the destination.

3) At the receiving end all the incoming bits are applied to the parity checker which checks the proper parity adopted. An error is detected if the checked parity does not match the adopted parity.

The parity method detects the presence of one, three or any odd number of errors. An even number of errors are not detected.

Parity generator and checker circuits are constructed with the Exclusive-OR gates.

Figure 3-3 Error detection with odd parity bit.



UNIT-II

REGISTER TRANSFER LANGUAGE AND MICROOPERATIONS

AND

BASIC COMPUTER ORGANIZATION AND DESIGN

Syllabus :

REGISTER TRANSFER LANGUAGE AND MICROOPERATIONS: 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, Design of basic computer, Design of Accumulator Logic.

REGISTER TRANSFER LANGUAGE AND MICROOPERATIONS:

REGISTER TRANSFER LANGUAGE:

A digital computer is the interconnection of hardware modules. The modules are constructed from digital components such as registers, decoders, arithmetic elements and control logic.

Modules that contains registers. Registers contains the data. The operations performed on the data of the register are called microoperations.

A microoperation is the elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another. Examples of microoperations include shift, count, clear and load.

The internal hardware organization of a digital computer is defined as follows:

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

The symbolic notation used to describe the microoperation on register is called Register Transfer Language.

The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation on registers. A programming language is a procedure for writing symbols that specify a given computational process.

TYPES OF MICROOPERATIONS:

A microoperation is an elementary operation performed with the data stored in register. The microoperations are classified into 4 categories:

1. ***Register transfer microoperations***: transfer binary information from one register to another.
2. ***Arithmetic microoperatoins:*** perform arithmetic operation on numeric data stored in registers.
3. ***Logic microoperations:*** Perform bit manipulation operations on nun-numeric data stored in registers.
4. ***Shift microoperations:*** perform shift operations on data stored in registers.

The register transfer microperation does not change the content when the binary information moves from the source register to the destination register. The other three microoperations change the information content during the transfer.

REGISTER TRANSFER:

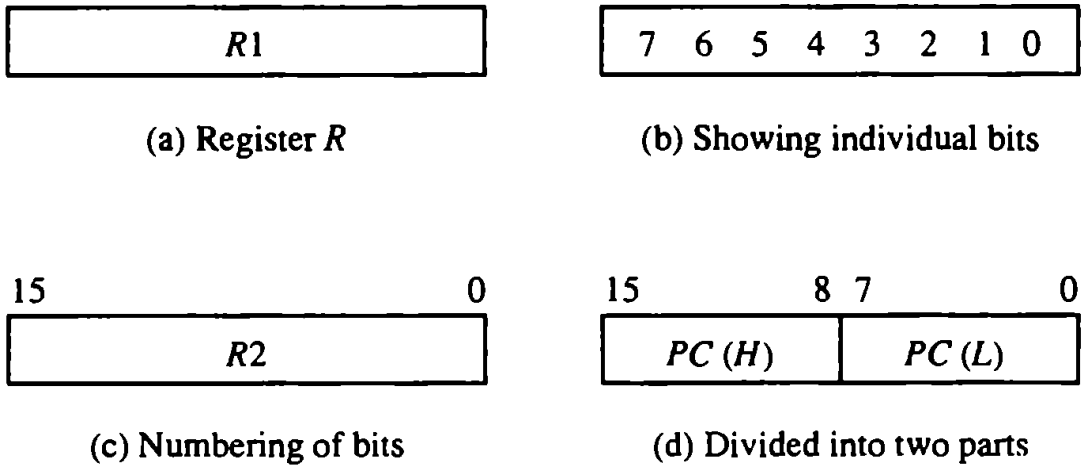
Computer registers are represented by the capital letters as follows:

MAR,MDR,IR,PC,$R_{0.....}$Rn

Following diagram shows different ways to represent a register:

1. The most common way to represent a register is a rectangular box with the name inside.(fig 4-1(a)).
2. The individual bits can be distinguished.(fig 4.1(b)).
3. The number of bits in16-bit register can be marked on the top of the box(fig 4.1(c)).
4. Partitioning of 16-bit register in two parts(fig 4.1(d)).. Bits 0 to 7 are assigned to the lower order bytes and represented by L. Bits 8 to 15 are assigned to the higher order byte and represented by H. For example, PC is the 16-bit register. PC(L) represents the lower order byte. PC(H) represents the higher order byte.

Figure 4-1 Block diagram of register.



Information transfer from one register to another register is represented in symbolic form as follows:

$$R2 \leftarrow R1$$

It denotes the transfer of the content of register R1 into register R2. It designates a replacement of content of R2 by R1. The content of the register R1 does not change after the transfer.

This means that the circuits are available the outputs of the source register to the inputs of the destination register and the destination register has the parallel load capability.

Suppose, the transfer is performed under a predetermined control condition as follows:`

$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

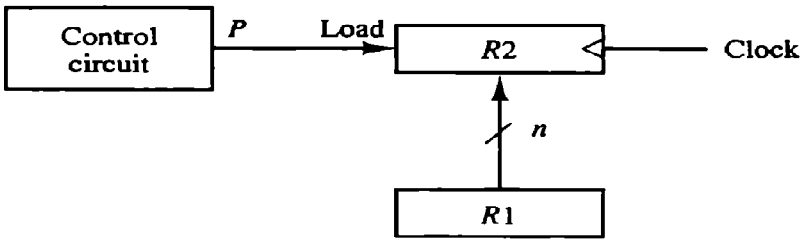
Where P is the control signal generated in the control section.

The above statement can be written as follows:

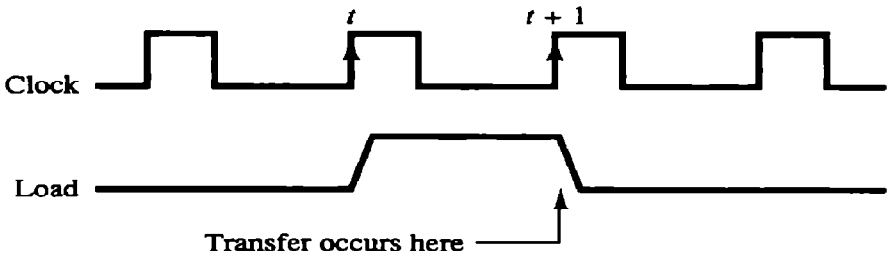
$$P:\quad R2 \leftarrow R1$$

The following diagram shows the block diagram for transfer R1 to R2. The n outputs of register R1 are connected to the register R2. The letter n represents bits of the register. Register R2 has a load input that is activated by the control variable P. P is activated in the control section by the rising edge of a clock pulse at time t. The next positive clock transition of the clock at time t+1 finds the load inputs active and data inputs of R2 then loaded into the register.

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



(a) Block diagram



(b) Timing diagram

The basic symbols of the register transfer notation are shown in the following table.

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 ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

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

$$T: \quad R2 \leftarrow R1, \quad R1 \leftarrow R2$$

BUS AND MEMORY TRANSFERS:

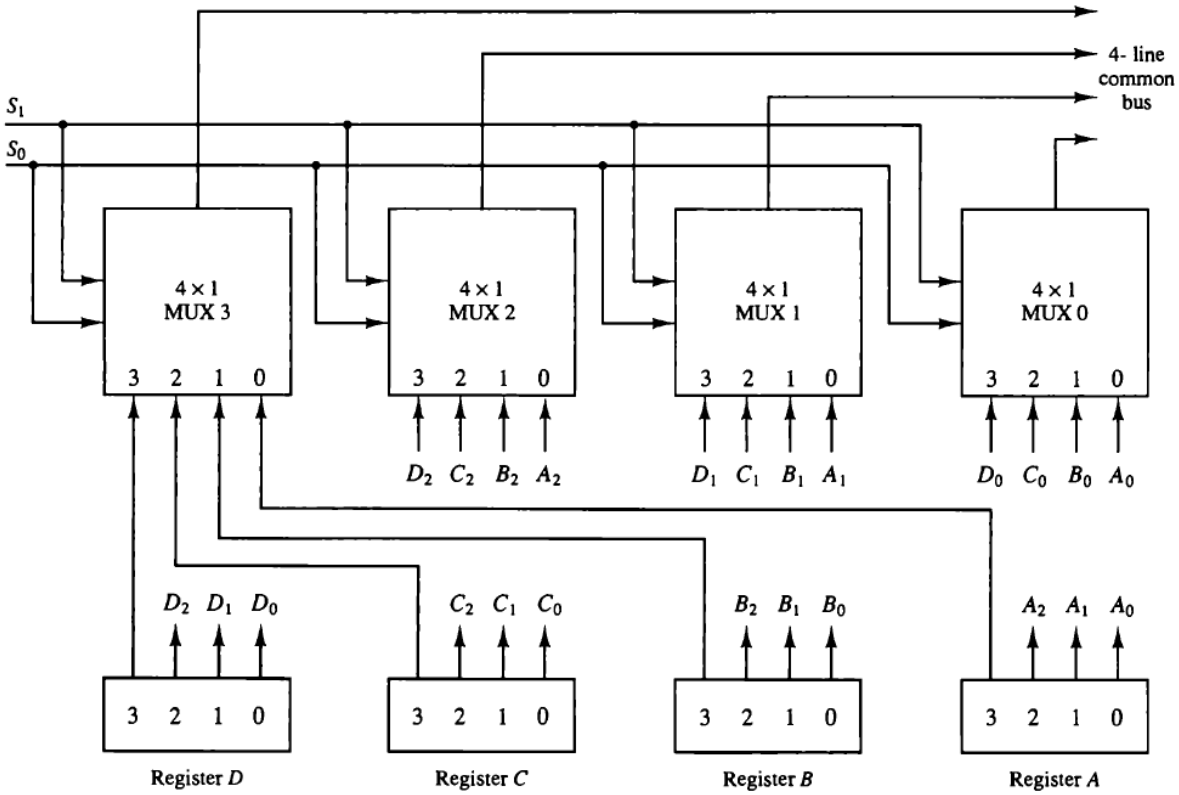
A digital computer has many registers. Paths must be provided to transfer information from one register to another. A more efficient scheme to transfer information between multiple registers is a common bus system.

A bus structure consists of set of common lines one for each bit of the register, through which binary information is transferred. Which register is to be selected for transfer is determined by the control signals.

One way of constructing the common bus is by using multiplexers. The multiplexers select the source register whose binary information is then palace in the bus.

Following diagram show the construction of a bus system for four registers:

Figure 4-3 Bus system for four registers.



The two selection lines s1 s0 are connected to the selection inputs of four multiplexers. The selection lines choose the four bits of one register and transfer them into the four line common bus. When s1s0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs. This causes the bus lines to receive the contents of the register A. Similarly, register B is selected if s1so=01 and so on.

Following table shows the register that is selected by the bus for each combination of the selection lines.

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

S_1	S_0	Register selected
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

The number of multiplexers needed to construct the bus is equal to n . here, n is the number of bits in the register. The size of the multiplexers must be k x 1 since it multiplexes k data lines.

For examples, a common bus for eight registers of 16 bit each requires 16 multiplexers, one for each line in the bus. Each multiplexers must have eight data input lines and three selection lines to multiplex on significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of particular destination register selected.

The symbolic statement for this transfer is as follows:

$$BUS \leftarrow C, \qquad R1 \leftarrow BUS$$

The content of the register C is place on the bus, and the content of the bus is loaded into the register R1 by activating its load control input.

The above transfer can be shown as the direct transfer

$$R1 \leftarrow C$$

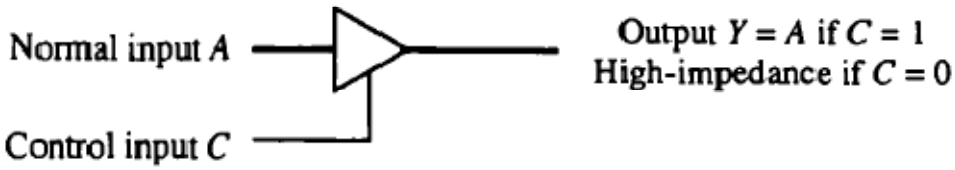
Three-State bus buffers:

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0. The third state is the high-impedance state that behaves as open circuit which

means the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND.

The graphic symbol for three-state buffer gate is shown in fig 4.4. It is distinguished from the normal buffer by having both normal input and the control input. The control input determines the output state.

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



When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output is equal to normal input. When the control input is 0. The output is disabled and gate goes to the high impedance state, regardless of the value in the normal input.

The high-impedance state of the three-state buffer provides three state gate provides a special feature not available in the other gates. Because of this feature, a large number of three-state gate output can be connected with wire to form a common bus line without endangering loading effects.

The contruction of a bus systems with three-state buffers is demonstrated in the fig 4.5. the outputs of 4 buffers are connected together to form a single bus line.

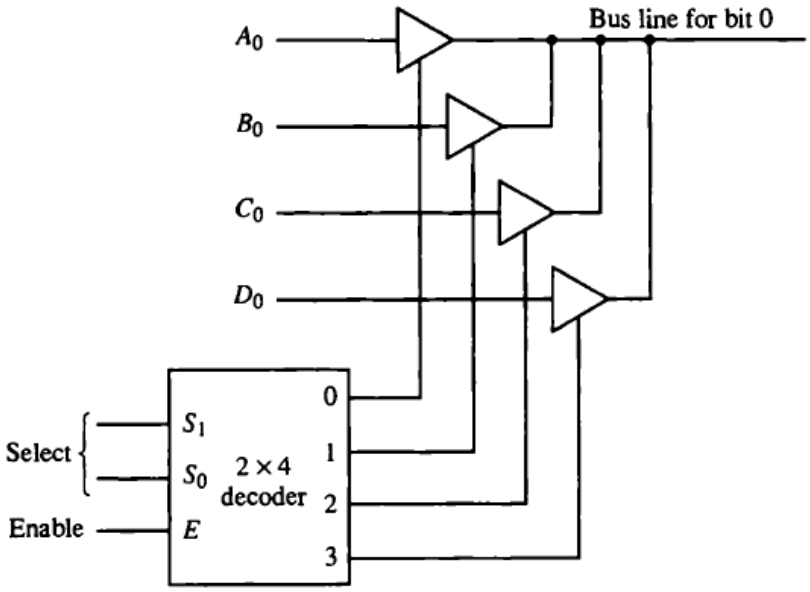


Figure 4-5 Bus line with three state-buffers.

The control inputs to the buffer determine which of the which of four normal inputs will communicate the bus line. No more than one buffer may be in the active state at the given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in high-impedance state.

The decoder is used to make sure that only one control input is active at any given time. When enable input is 0, all of its four outputs are 0 and the bus line is in high-impedance state. When the enable input is active, one of the three-state buffers will active, depending on the binary value of the selects input of the decoder.

Memory Transfer:

The transfer information from the memory to the outside environment is called read operation. The transfer of new information to be stored into the memory is called wirte operations

A memory word is symbolized by the letter M. The memory address is used to select a particular word in the memory. It is necessary to specify the address of M during the memory transfer operations.

Consider the following read operation:

$$\text{Read:} \quad DR \leftarrow M[AR]$$

The above read operation transfers the information into DR from the memory word M selected by the address AR.

The write operation transfers the content of the data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR. The write operation can be stated as follows:

Write: M[AR] ← R1

This causes the transfer of information from R1 into the memory word M selected by the address in AR.

ARITHMETIC MICROOPERATIONS:

The basic arithmetic microoperations are addition, subtraction, increment and decrement. The arithmetic microoperation is defined by the statement:

R3 ← R1 + R2

The above specified an add operation. It states that the contents of register R1 are added to the contents of register R2 and the sum transfer to register R3.

To implement the above operation we need three register and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in table 4.3.

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Subtraction is most often implemented through complementation and addition as follows:

$$R3 \leftarrow R1 + \overline{R2} + 1$$

Adding the contents of R1 to the 2's complement of R2 is equivalent to R1-R2.

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

The multiplication operation can be done in computer as sequence of add and shift operations. Division operation is implemented with a sequence of subtract and shift operations.

Binary Adder:

To implement the add microoperation, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms arithmetic sum of two bits and a previous carry is called a full adder.

The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full adder circuits connected in cascade. Ie. The output carry of one full-adder connected to the input carry of the next full-adder. Fig 4.6 show the interconnection of four full-adders(FA) to provide a 4-bit binary adder.

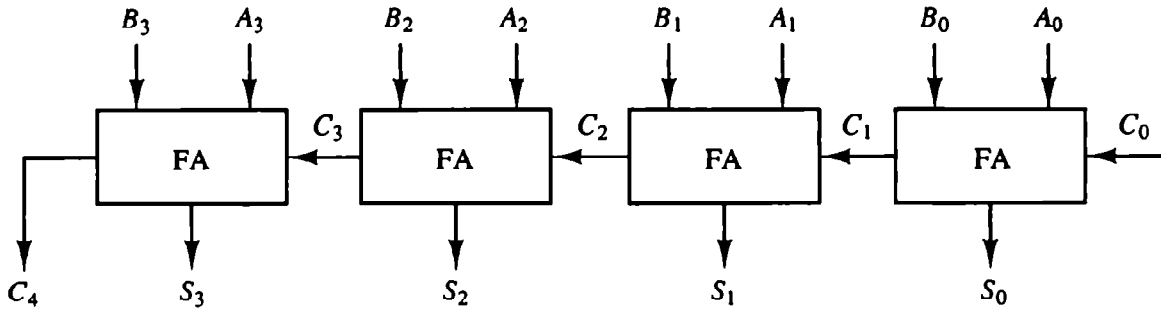


Figure 4-6 4-bit binary adder.

The augend bits of A and the addend bits of b are designated by subscript number from left to right. The carries are connected in chain trough full-adders. The input carry for the binary adder is C0 and the output carry is C4. The S outputs of the full-adders generate the required sum bits.

The n-bit binary adder requires n full-adders. The n data bits for the A inputs come from one register(R1), and the n data bits for the B inputs come from another register(R2). The sum can be transferred to a third register or to one of the source registers(R1 or R2), replacing the previous contents.

Binary Adder-Subtractor.

The subtraction of binary number can be done by means complements. The subtraction A – B can be done by taking the 2' complement of B and adding it to A.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-circuit is shown in fig 4.2. The mode input M controls the operation.

When, M=0 the circuit is an adder and when M = 1 the circuits become subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M=0, we have $B \oplus 0 = B$. The full adders receive the value of B, the input carry is 0, and the circuit perform A plus B.

When M =1 , We have, $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2' complement of B.

For unsigned numbers, this gives A – B if A>=B or the 2's complement of (B-A) if Aa<b. For signed numbers, the result is A – B provided that there is no overflow.

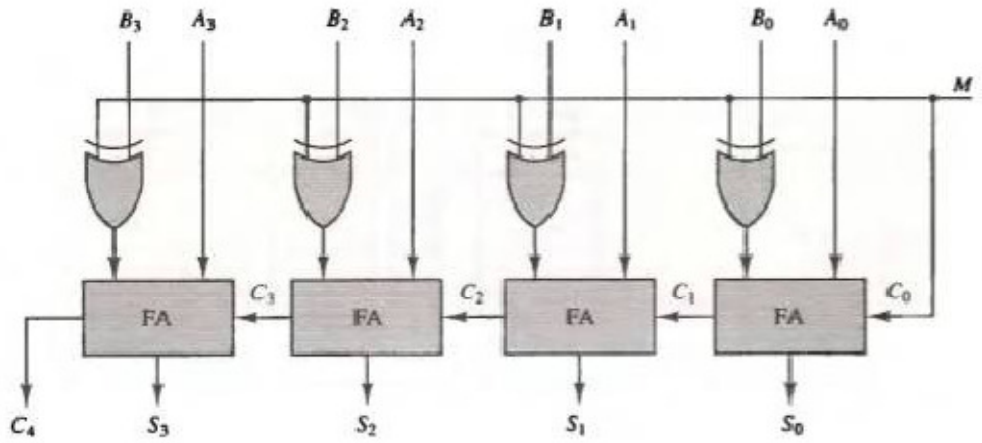


Figure 4-7 4-bit adder-subtractor.

Binary Incrementer:

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. The increment microoperation can be accomplished by means of half-adders connected in cascade.

The diagram for 4-bit combinational circuit incrementer is shown in fig 4.8.

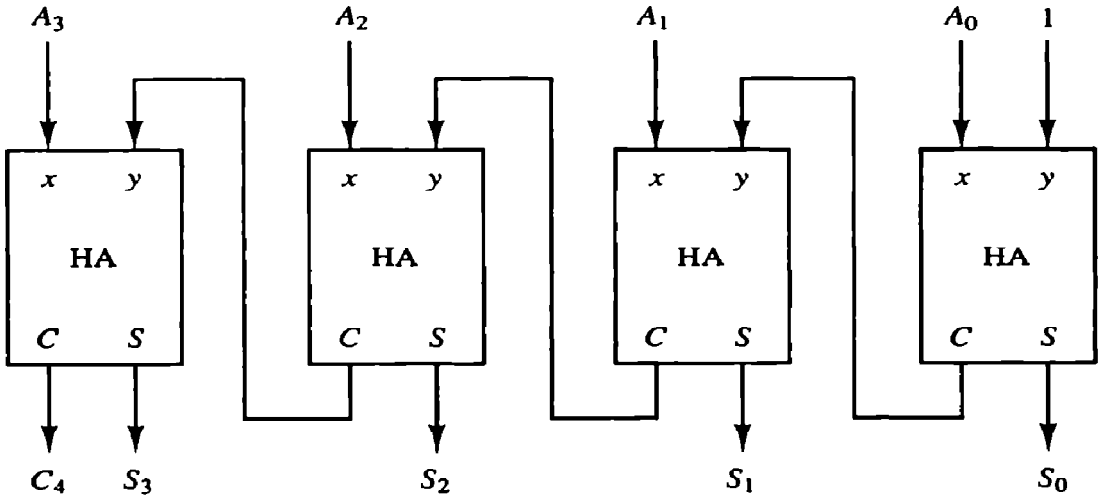


Figure 4-8 4-bit binary incrementer.

As shown in the diagram, one of the inputs to the least significant half-adder(HA) is connected to logic 1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher order half-adder. The circuit receives the four bits from A_0 though A_3, add one to it, and generates the incremented output in s_0 through s_3. The output carry C4 will be 1 only after incrementing binary 1111. This also causes outputs s_0 through s_3 to go to 0.

The circuit of fig 4.8 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders.

Arithmetic Circuit:

The arithmetic microoperations listed in table 4.3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By, controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in fig 4.9.

It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations.

There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1.

The four multiplexers are controlled by two selection inputs, S1 and S0. The input carry Cin goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

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

$$D = A + Y + C_{in}$$

Where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. Cin is the input carry, which can be equal to 0 or 1.

By controlling the value of Y with the two selection inputs S1 and S0 and making cin equal to 0 or 1, it is possible to generated the eight arithmetic microoperations listed in table 4.4.

Addition:

When $S_1S_0 = 00$, the value of B is applied to the Y inputs of the adder. If $C_{in} = 0$, the output $D = A + B$. If $C_{in} = 1$, output $D = A + B + 1$. Both cases perform the add microoperation with or without adding the input carry.

Subtraction:

When $S_1S_0 = 01$, the complement of B is applied to the Y inputs of the adder. If $C_{in} = 1$, then $D = A + \overline{B} + 1$. This produces A plus the 2's complement of B, which is equivalent to a subtraction of $A - B$. When $C_{in} = 0$, then $D = A + \overline{B}$. This is equivalent to a subtract with borrow, that is, $A - B - 1$.

Increment:

When $S_1S_0 = 10$, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes $D = A + 0 + C_{in}$. This gives $D = A$ when $C_{in} = 0$ and $D = A + 1$ when $C_{in} = 1$. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

Decrement:

When $S_1S_0 = 11$, all 1's are inserted into the Y inputs of the adder to produce the decrement operation $D = A - 1$ when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces $F = A$ + 2's complement of $1 = A - 1$. When $C_{in} = 1$, then $D = A - 1 + 1 = A$, which causes a direct transfer from input A to output D. Note that the microoperation $D = A$ is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

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

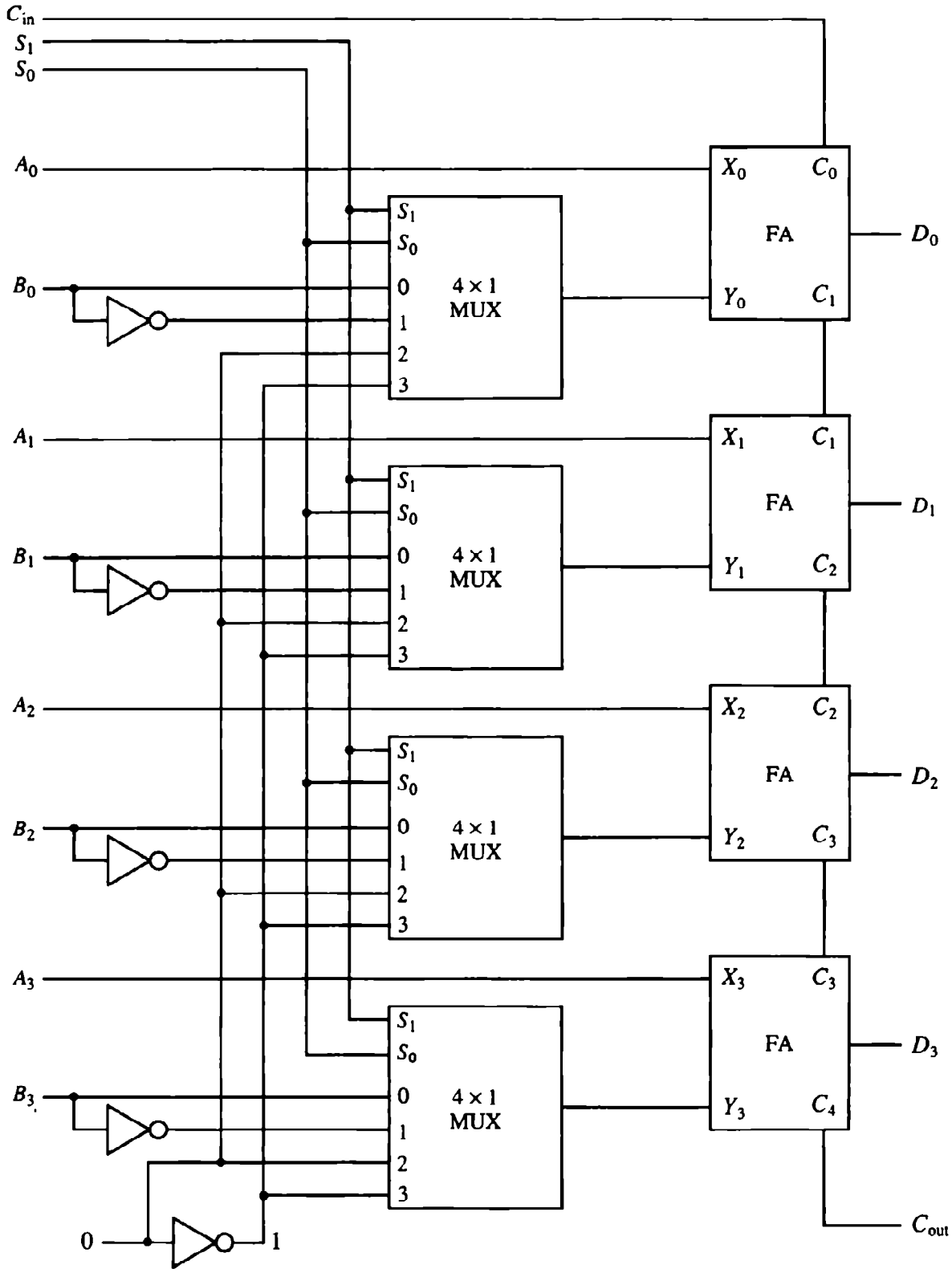


Figure 4-9 4-bit arithmetic circuit.

LOGIC MICROOPERATIONS:

Logic microoperations specify binary operations on strings of bits stored in the registers. These microoperations consider each bit of the register separately and treat them as binary variables.

For example, the exclusive-OR microoperation with the contents of two register R1 and R2 is symbolized by the statement:

$$P: \quad R1 \leftarrow R1 \oplus R2$$

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as in the table 4.5

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

The following table shows 16 logic microoperations that are derived from the 16 boolean functions of two variables x and y. The binary content of variable x is stored in register A and the binary content of variable y is stored in register B. The logic microoperations are performed on the string of bits stored in the register.

Hardware Implementation:

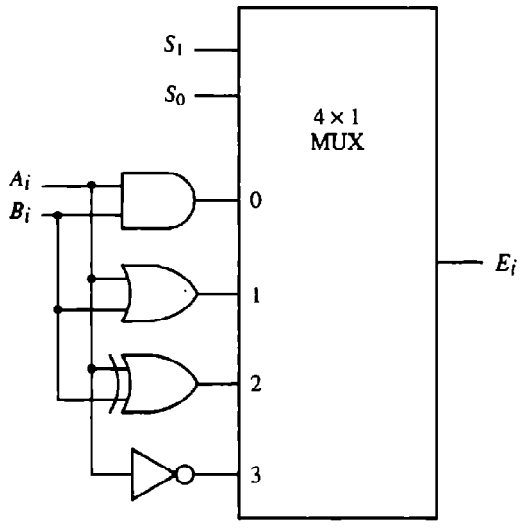
The hardware implementation of logic microoperations requires logic gates be inserted for each bit or pair of bits in the registers to perform the require logic function. Although there are 16 logic microoperations, most computers perform only four-AND,OR,XOR, and complement from which all others can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

Fig 4-10 shows a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated though a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs s1 and s0 choose one of the data inputs of the multiplexer and direct its value to the output.

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
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Figure 4-10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Applications:

Logic operations are very useful for manipulation individual bits or a portion of word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register.

Examples:

Selective set:

The selective-set operation set to 1 the bits in register A where there are corresponding 1's in register B.

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

Selective Complement:

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
```

Selective clear:

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

```
1010    A before
1100    B (logic operand)
0010    A after
```

Mask/AND operation:

The mask operation in which the bits of A are cleared only where there are correspondin 0's in B. The mask operation performs the AND operation :

```
1010    A before
1100    B (logic operand)
1000    A after masking
```

Insert/OR operation:

This operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, Suppose that an A register

contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits as follows:

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

and then insert the new value:

```
0000 1010      A before
1001 0000      B (insert)
1001 1010      A after insertion
```

The mask operation is an AND operation and the Insert operation is an OR operation.

Clear/Exclusive-OR operation:

The clear operation compared the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive OR microoperation as shown below:

```
1010      A
1010      B
0000      A ← A ⊕ B
```

SHIFT MICROOPERATIONS:

Shift microoperations are used for serial transfer of data. They are used in conjunction with arithmetic, logic and other data-processing operations.

The contents of a register can be shifted to the left or right. At the time of bits shift the first-flip flop receives its binary information from the serial input. During a sift-left operation the serial input transfers a bit into the rightmost position. During the shift-right operation the serial input transfers a bit into the leftmost position.

A logical shift is one that transfers 0 though the serial input. shl and shr are the symbols used for logical shift-left and shift-right microoperations.

Example:

$R1 \leftarrow \text{shl } R1$

$R2 \leftarrow \text{shr } R2$

The above are two microoperations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2.

The circular shift(or rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. The symbols cil and cir are used circular shift left and right respectively. The following table shows different shift microoperations.

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

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sigh bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Fig 4.11 shows a typical register of n bits.

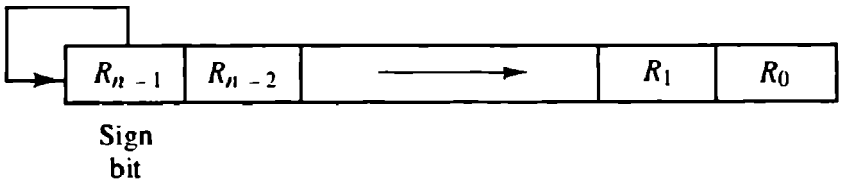


Figure 4-11 Arithmetic shift right.

Bit R_{n-1} holds the sight bit. R_{n-2} is the most significant bit of the number and R0 is the least significant bit.

The arithmetic shift-left transfers a 0 to R_0, and shifts all other bits to the left. The initial R_{n-1} is lost and replace by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow.

An overflow occurs after the arithmetic shift left R_{n-1} is not equal to R_{n-2} before shift. An overflow flip-flop Vs can be used to detect an arithmetic shift overflow.

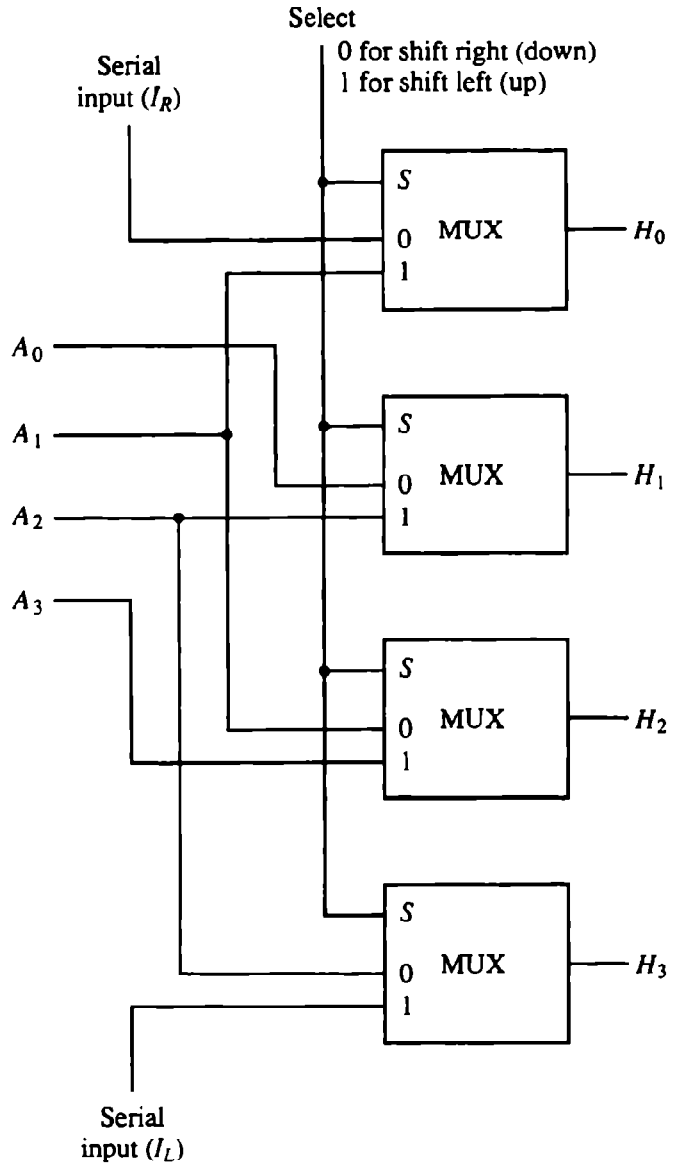
$$V_s = R_{n-1} \oplus R_{n-2}$$

If Vs = 0, there is no overflow, but if Vs=1, there is overflow and a sign reversal after the shift.

Hardware Implementation:

The choice to construct a shift unit would be a bidirectional shift register with parallel load. In a processor with may registers it is more efficient to implement the shift operations with a combinational circuit.

A combinational circuit shifter can be constructed with multiplexers as shown in fig 4.12.



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

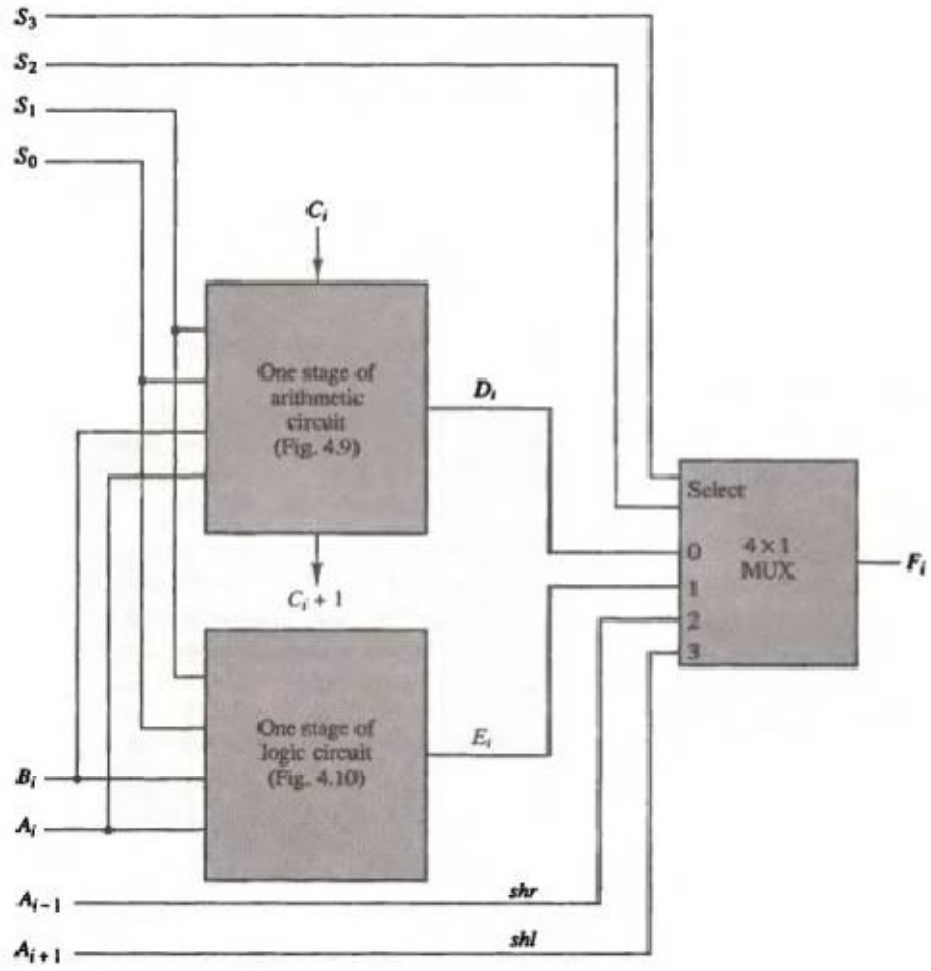
A combination circuit shifter can be constructed with multiplexers as shown in fig. 4.12. The 4-bit shifter has four data inputs, A0 through A3, and four data outputs, H0 through H3. There are two serial inputs, one for shift left(I_L) and other for shift right(I_R). When the selection input S-0, the input data are shifted right(down), When S-1, the input data are

shifted left(up). The function table in fig: 4.12 shows which input goes to each output after the shift. A shifter with n data inputs and output requires n multiplexers.

ARITHMETIC LOGIC SHIFT UNIT(ALU):

The arithmetic,logic and shift circuits are combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in the fig 4.13. The subscript i designates a typical stage.

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



Inputs Ai and Bi are applied to both the arithmetic and logic units. A particular microoperation is selected with input S1 and S0. A 4x1 multiplexer chooses between an arithmetic output in E_i and a logic output in H_i. The data in the multiplexer are selected with S3 and S2. The other data inputs to the multiplexer receive inputs Ai-1 for the shift-right operation and Ai+1 for the shift left operation. For the n-bit ALU the circuit must be repeated for n-times. The output carry Ci+1 of a given arithmetic stage must be connected to the input carry Ci of the next stage in sequence.

The above circuit provides eight arithmetic operations, four logic operations, and two shift operations. Each operation is selected with the five variables S3, S2,S1,S0 and Cin. The input carry cin is used for selecting an arithmetic operation only.

Table 4.8 lists the 14 operations of the ALU. The first eight are arithmetic and are selected with S_3S_2=00. The next 4 are logic operations and are selected with S_3S_2=01. The input carry has no effect during the logic operations and is marked with don't care Xs. The last two operations are shift operations and are selected with S_3S_2=10 and 11. The other three selection inputs have no effect on the shift.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
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	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	F = shr A	Shift right A into F
1	1	×	×	×	F = shl A	Shift left A into F

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
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	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	F = shr A	Shift right A into F
1	1	×	×	×	F = shl A	Shift left A into F

BASIC COMPUTER ORGANIZATION AND DESIGN:

INSTRUCTION CODES:

The design of the computer is defined by its internal registers, the timing and control structure, and set of instructions that it uses.

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction code together with data are stored in memory. The computer reads each instruction from memory and places it control register. The control then interprets the binary code of each instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set.

A instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts. The first part is the operation code. Operation code is a group of bits that specifies the operation to be performed such as add, subtract, multiply, shift and complement. The number of operation available in computer depends number of bits in the operation code. The number of bits in operation code is n, if the computer has 2 n distinct operations. For example, a computer with 64 distinct operations, the operation code consists of six bits.

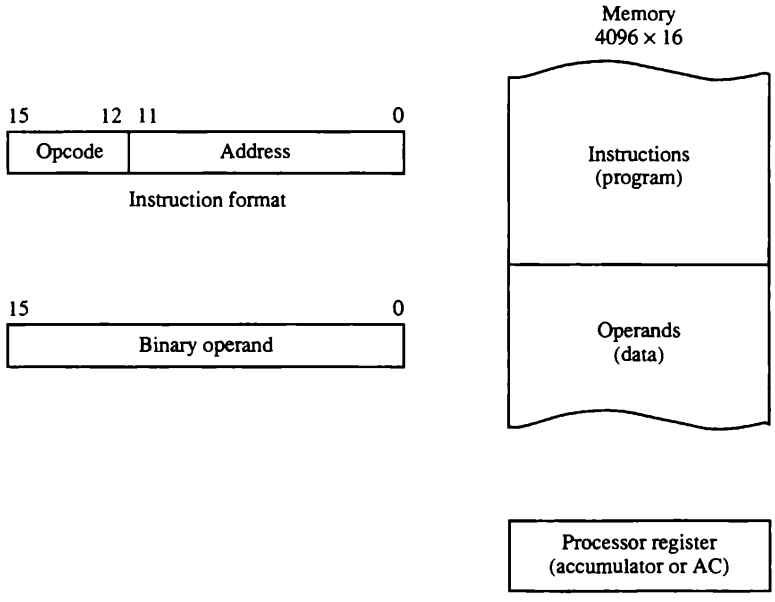
The second part of the instruction code consists of address, where operands are stored. Each computer has its own instruction code format.

Stored program organization:

The simplest way to organize a computer is to have a processor register and an instruction code format. The memory address in the instruction code format tells the control where to find an operand in the memory. This operand is read from memory and operated together with the data stored in the processor register.

Fig 5.2 shows this type of organization.

Figure 5-1 Stored program organization.



The available operations are 16 because there are 4-bits in the operation code. There are 4096 memory words because address field consists of 12 bits. The computer reads 16-bit instruction from the program portion of the memory. It uses 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

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

If the operation does not need an operand, the rest of the bits in the instruction can be used for other purposes. The operations that do not require operands are Clear AC, Complement AC. The operations operate on data stored in the AC register.

Indirect address:

When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have direct address. In the indirect address, the second part of the instruction represents the address of the memory word in which the address of the operand is found.

One bit of the instruction code can be used to distinguish between direct and an indirect address.

Consider the instruction code format shown in fig 5.2(a). It consist of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address.

A direct address instruction is shown in fig 5.2(b) placed in address 22 in memory. The I bit is 0,so the instruction is recognized as a direct address instruction. The control finds the operand in memory at address 57 and add it to the content of AC.

The instruction in address 35 show in fig 5.2(c), has mode bit I=1. Therefore, it is recognized as an indirect address instruction. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two reference to memory to fetch the operand.

The **effective address** is the address of the operand in computation-type instruction or the target address in a branch type instruction.

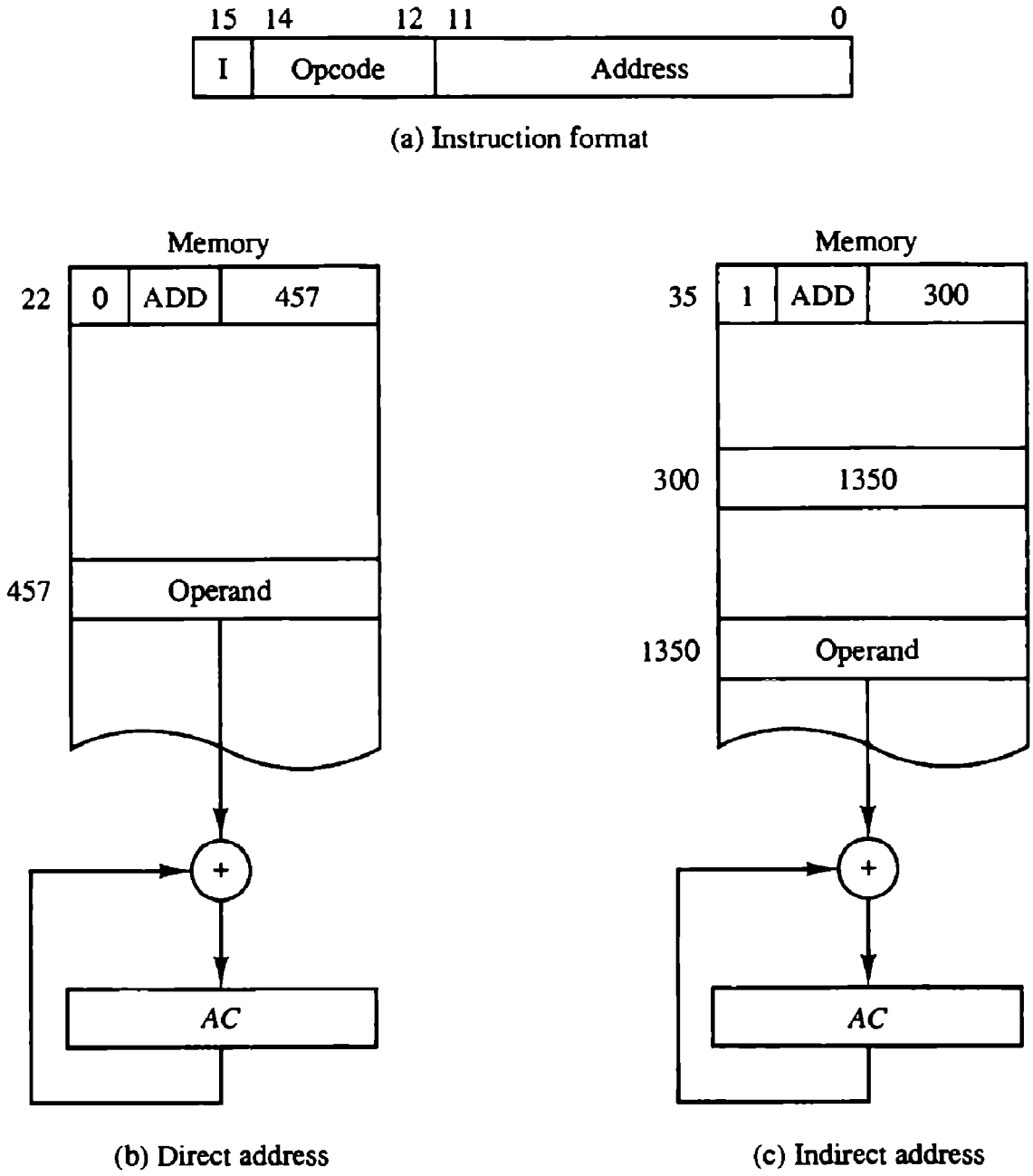


Figure 5-2 Demonstration of direct and indirect address.

COMPUTER REGISTERS:

The computer requires different kinds of registers for different purposes:

1) A counter to calculate the address of the next instruction.
2) A register in the control unit for storing the instruction code after it is read from memory.
3) Process register for data manipulation.

4) A register for holding memory address.

The fig 5.3 shows the basic computer registers and memory.

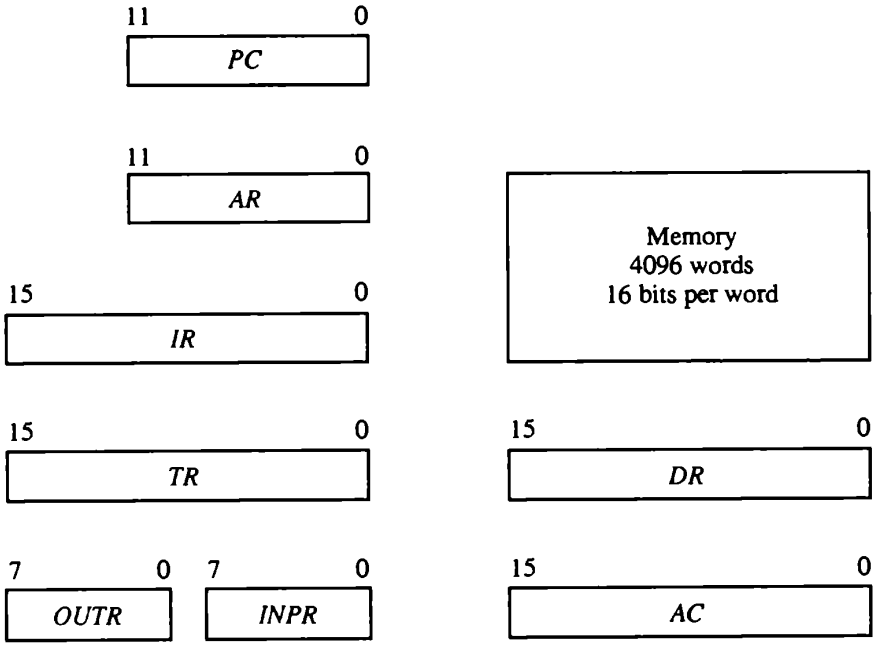


Figure 5-3 Basic computer registers and memory.

The table 5.1 list the registers and the brief description of their function.

TABLE 5-1 List of Registers for the Basic Computer

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

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand, three bits for the operation and a bit to specify a direct or indirect address.

The data register(DR) holds the operand read from memory. The accumulator(AC) register is a general purpose processing register. The instruction read from memory is placed int the instruction register(IR). The temporary register(TR) is used for holding temporary data during the processing.

The address register(AR) has 12 bits because this is the width of a memory address. The program counter(PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed.

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

Common bus system:

The basic computer has eight registers, a memory unit and a control unit. Paths must be provides to transfer information from on register to another and between memory and registers. Transferring of information can be done by a common bus. The connection of the register and memory of the basic computer to a common bus system is shown in fig 5.4.

The outputs of seven registers and memory are connected to the common bus. The specific output to be selected for the bus at any given time is determined from the binary value of the selection variables $S_2S_1S_0$. There is a decimal number along each output, that show the binary equivalent of $S_2S_1S_0$ required to select its output for the bus. For example 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_{0=}$ 011.

The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD(load) input is enable receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory place its 16-bit output into the bus when the read input is activated and $S_2S_1S_{0=}$ 111.

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

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

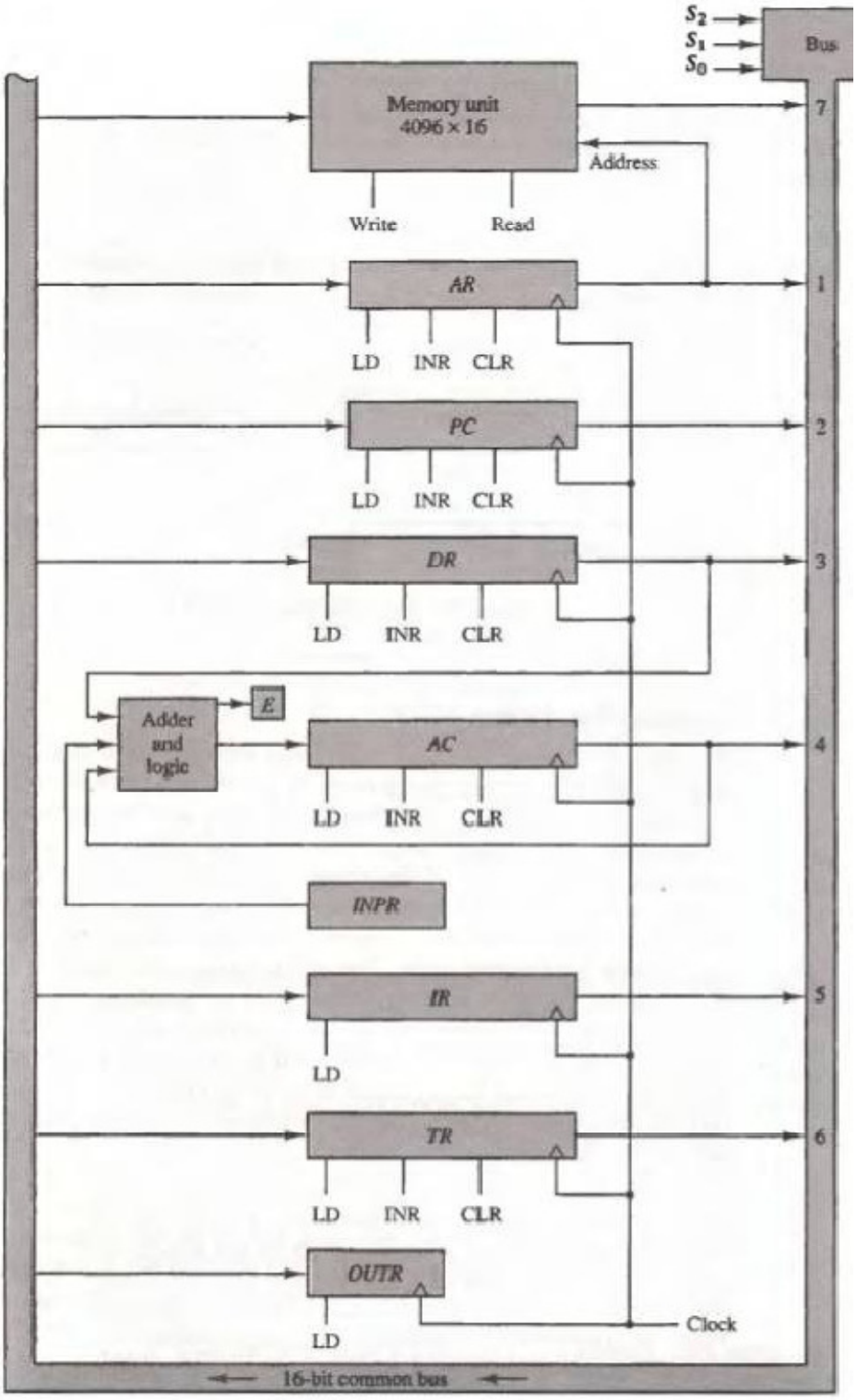


Figure 5-4 Basic computer registers connected to a common bus.

The 16 lines of the common bus receive information from six register and memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD(load), INR(increment), and CLR(clear).

The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address.

The inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One input come from the outputs of AC. They are used to implement register microoperations such as complement AC and shift AC. Another input come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microperations, such as add DR to AC or AND DR to AC. The result of addition is transferred to AC and the end carry-out of the addition is transferred to the flip-flop E. A third input come from the input register INPR.

For example, consider the two microoperations:

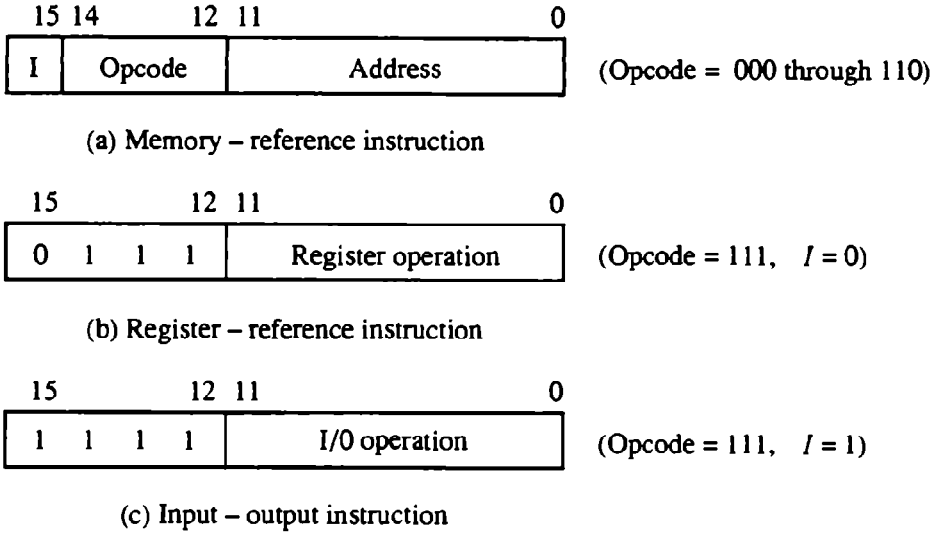
$$DR \leftarrow AC \quad \text{and} \quad AC \leftarrow DR$$

The above can be executed at the same time. This can be done by placing the content of AC on the bus(with S2S10=100), enabling the LD(load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD(load) input of AC, all during the same clock cycle.

COMPUTER INSTRUCTIONS:

The basic computer has three instruction code formats, as shown in fig 5.5

Figure 5-5 Basic computer instruction formats.



A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.

The register reference instructions are recognized by the operation code 111 with 0 leftmost bit of the instruction. These instructions do not require operand from the memory.

Similary, an input-output instruction is recognized by operation code 111 with a 1 in the left most bit position and does not need a reference to the memory.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction.

The instructions for the computer are listed in table 5.2

TABLE 5-2 Basic Computer Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

By using the hexadecimal equivalent the 16-bits of an instruction code to four digits. A memory reference instruction has an address part of 12 bits. The address part is denoted by three x's. Register reference instructions use 16 bits to specify an operation. The leftmost 3-bits are always 0111, which is equivalent to hexadecimal 7. The input-output instructions

allow use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Instruction set completeness:

The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

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

TIMING AND CONTROL:

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

There are two types of control organization:

1) Hardwired control
2) Microprogrammed control

In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.

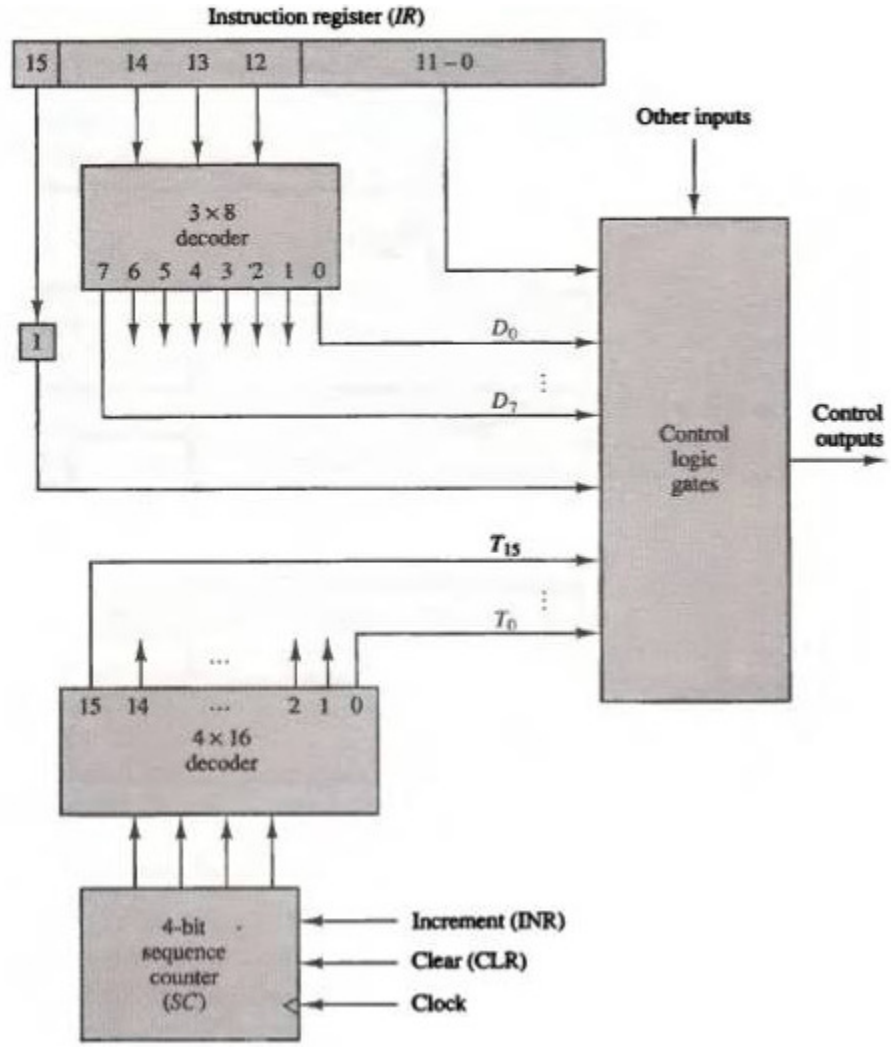
Advantage: This organization produces the fast mode of the operation.

Disadvantage: Difficult to change the design.

In the microprogrammed organization, the control information is stored in the memory. The control memory is programmed to initiate the required sequence of microoperations.

Advantage: Easy to change the design by modifying updating the microprogram in control memory.

The block diagram of the hardwire control unit is shown in fig 5.6. It consists of two decoders, a sequence counter and a number of control logic gates. An instruction is read from memory is place in the instruction register (IR). The instruction register is divided into three parts: The I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3x8 decoder. D0 thorough D7 are the outputs of the decoder. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count 0 through 15. The outputs of the counter are decoded into 16 timing signals T0 through T15.



Once in a while, the counter is cleared to 0, causing the next active timing signal to be T0. For example, consider the case where SC is incremented to provide timing signals T0,T1,T2,T3 and T4 in sequence. At the time T4, SC is cleared to 0 if the decoder output D3 is active. This is expressed symbolically by the statement:

$$D_3T_4: \quad SC \leftarrow 0$$

The timing diagram fig 5.7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active.

The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T0 out of the decoder. T0 is active during one clock cycle. The positive clock transition labelled T0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T0. SC is incremented with every positive clock transition, unless its CLR input is active, and so on. If SC is not cleared, the timing signals will continue with T1,T2 up to T15 and back to T0.

A memory read or write cycle will be initiated with the rising edge of a timing signal. A memory read or write cycle initiated by a timing signal will be completed by the time next clock goes through its positive transition.

For example, the register transfer statement:

$$T_0: \quad AR \leftarrow PC$$

The above statement specified a transfer of the content of PC into AR is timing signal T0 is active. T0 is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus(with S2S1S0=010) and the LD(load) input of AR is enabled. The actual transfer does not occur until next positive clock transition(T1). The same transition(T1) increments the sequence counter SC from 0000 to 0001.

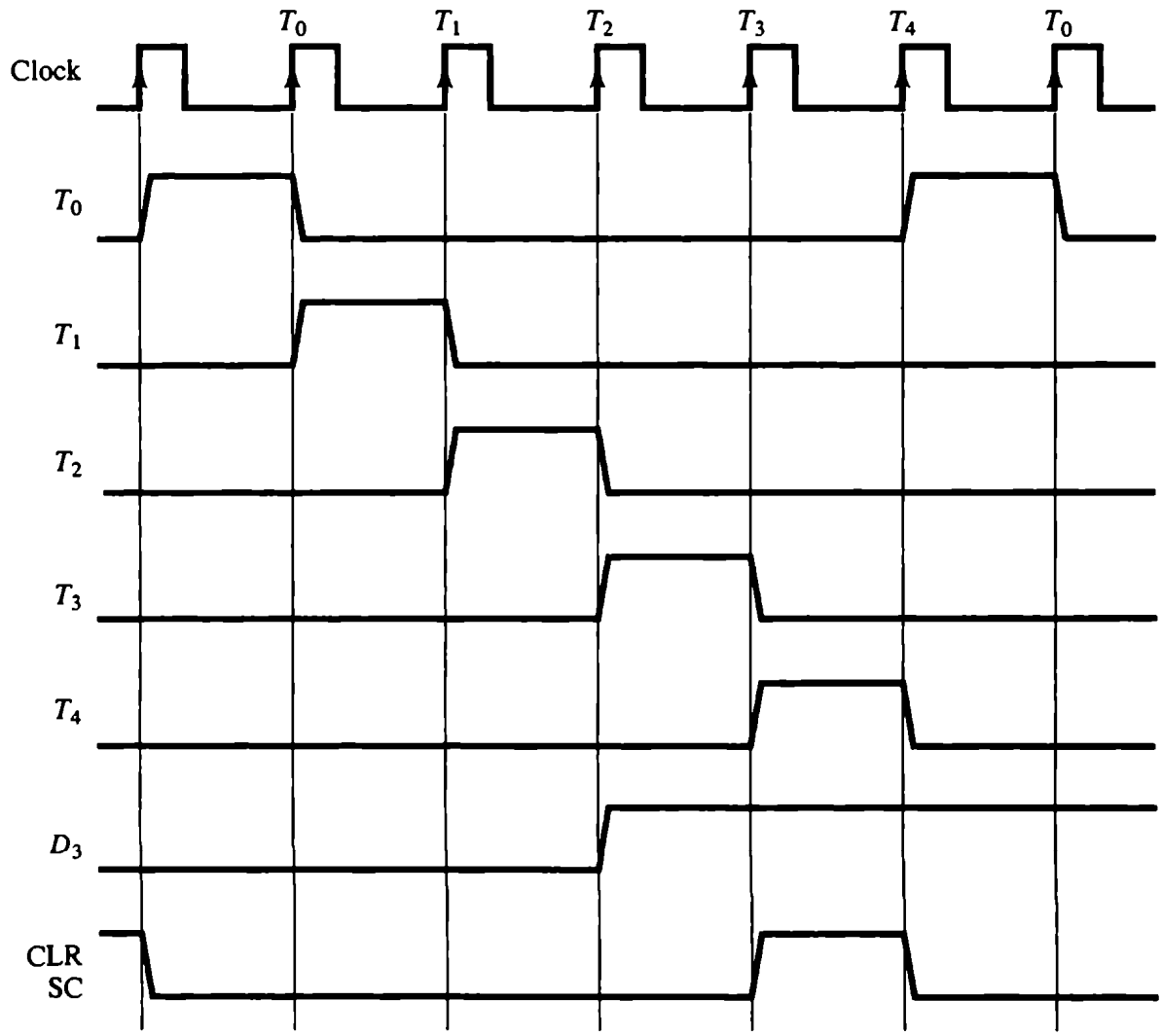


Figure 5-7 Example of control timing signals.

INSTRUCTION CYCLE:

There are different phases to execute an instruction is called instruction cycle. The instruction cycle consisting of the following phases:

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

After completion of step 4, the control goes back to step1 to fetch, decode and execute the next instruction. This process continues until a HALT instruction is encountered.

Fetch and decode:

Initially, the PC is loaded with the address of the first instruction in the program. The microoperations for the fetch and decode phases can be specified by the following statements.

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR], \quad PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14), \quad AR \leftarrow IR(0\text{–}11), \quad I \leftarrow IR(15)$

Since only AR is connected to the address inputs of memory, it is necessary to transfer address from PC to AR during T0. The instruction read from memory is then place in the instruction register IR during T1. At the same time PC is incremented by one to point to the next instruction in the program. During T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

Fig 5.8 shows how the first two register transfer statements are implemented in the bus system.

Determine the type of instruction:

After decoding the instruction the timing signal T3 activated, during this T3, the control unit determines the type of the instruction. The fig 5-9 shows a flowchart for instruction cycle. There are 3 possible instruction types available in the basic computer.

1. Register reference instructions.
2. Memory reference instructions
3. Input-Output instructions.

Decoder output D7 is equal to 1 if the operation code is 111. If D7=1, the instruction must be register-reference or input-output instruction. If D7=0, the operation coder must be 000 through 110, specifying the memory reference instruction.

If D7=0 and I=1, we have a memory reference instruction with indirect address. It is necessary to read effective address(EA) from memory. The microoperation for the indirect address condition can be as follows:

$$AR \leftarrow M[AR]$$

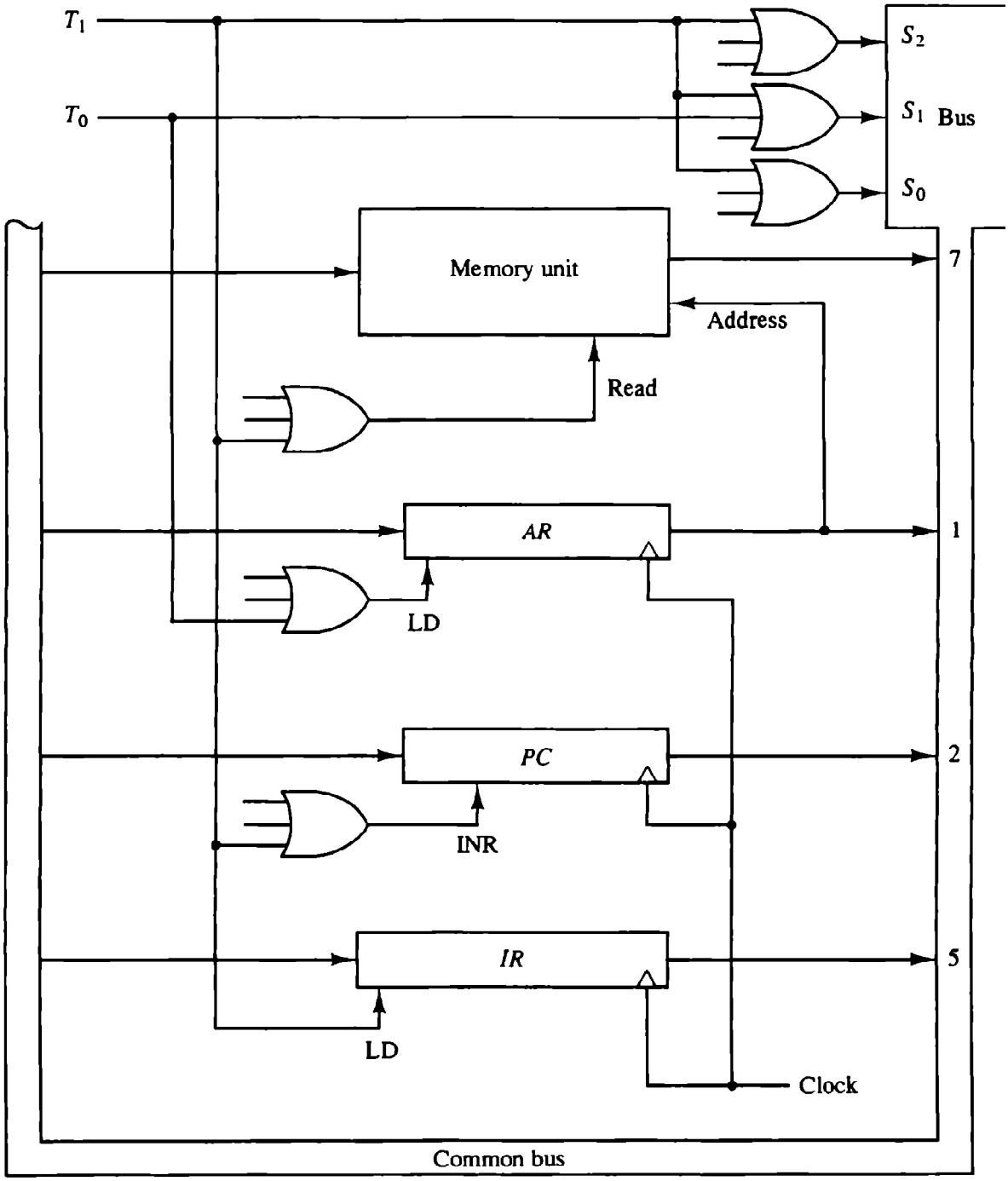


Figure 5-8 Register transfers for the fetch phase.

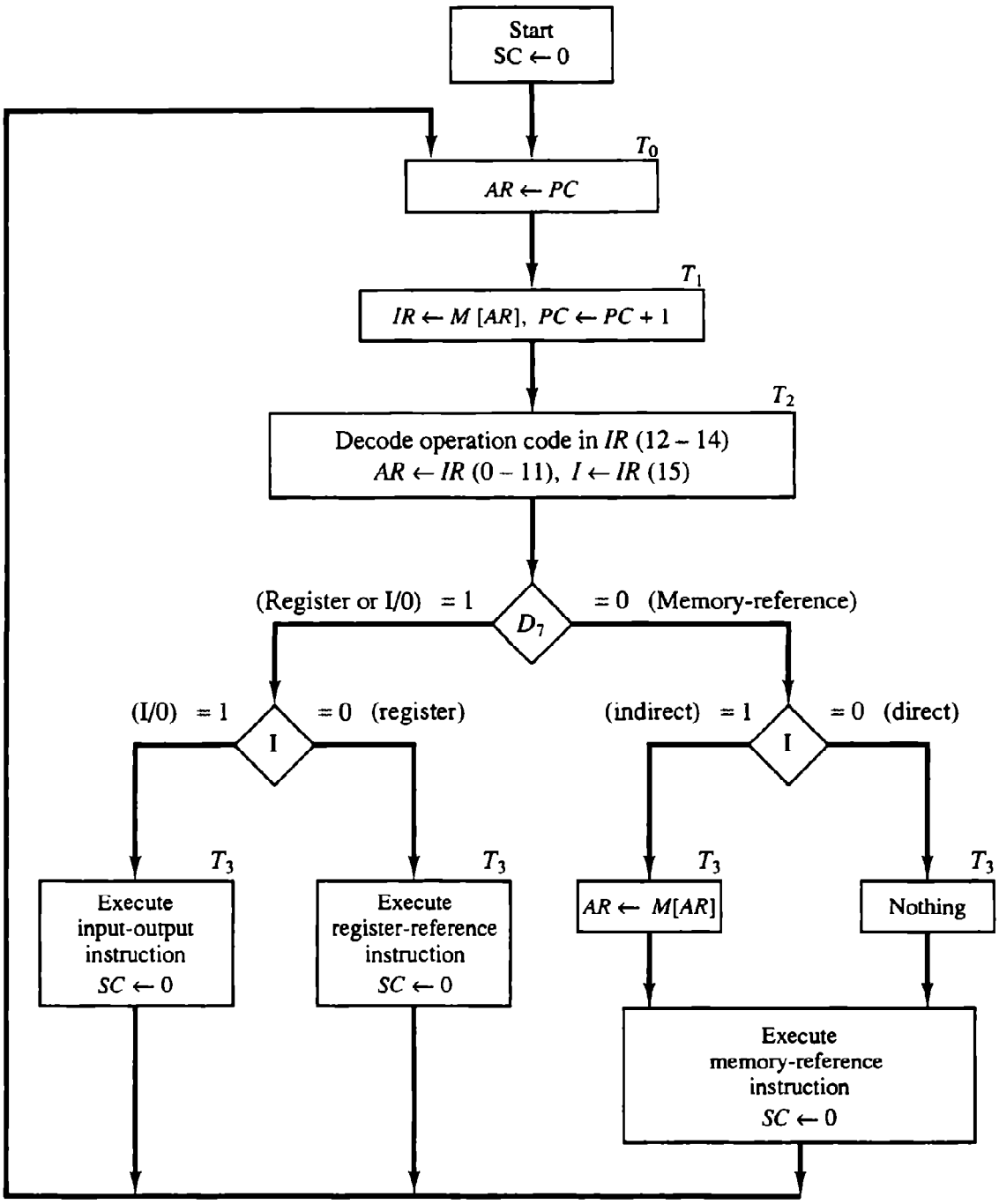


Figure 5-9 Flowchart for instruction cycle (initial configuration).

The 3 instructions are subdivided into four separate paths as follows:

$D_7' I T_3$: $AR \leftarrow M[AR]$
$D_7' I' T_3$: Nothing
$D_7 I' T_3$: Execute a register-reference instruction
$D_7 I T_3$: Execute an input–output instruction

REGISTER-REFERENCE INSTRUCTIONS:

The register-reference instructions are recognized by the control when D7=1 and I=0.These instructions use bits 0 through 11 of the instruction code to specify one of the 12 instructions. The register reference instructions are listed in table 5-3.

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip to next instruction, when condition is satisfied. The skipping of instruction is achieved by incrementing PC once again. The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting.

MEMORY-REFERENCE INSTRUCTIONS:

Table 5.4 lists the seven memory reference instructions. The decoded output Di for i = 0,1,2,3,4,5 and 6 from the operation decoder that belongs to each instruction is included in the table.

The effective address(EA) of the instruction is in the address register AR and was placed there during timing signal T2 when I=0, or during timing signal T3 when I=1. The execution to memory-reference instructions starts with timing signal T4. The memory reference operations are:

1) AND to AC
2) ADD to AC
3) LOAD to AC
4) Store AC

5) Branch Unconditionally
6) Branch and Save Return Adress
7) Increment and Skip if Zero

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

A control flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in fig 5-11.

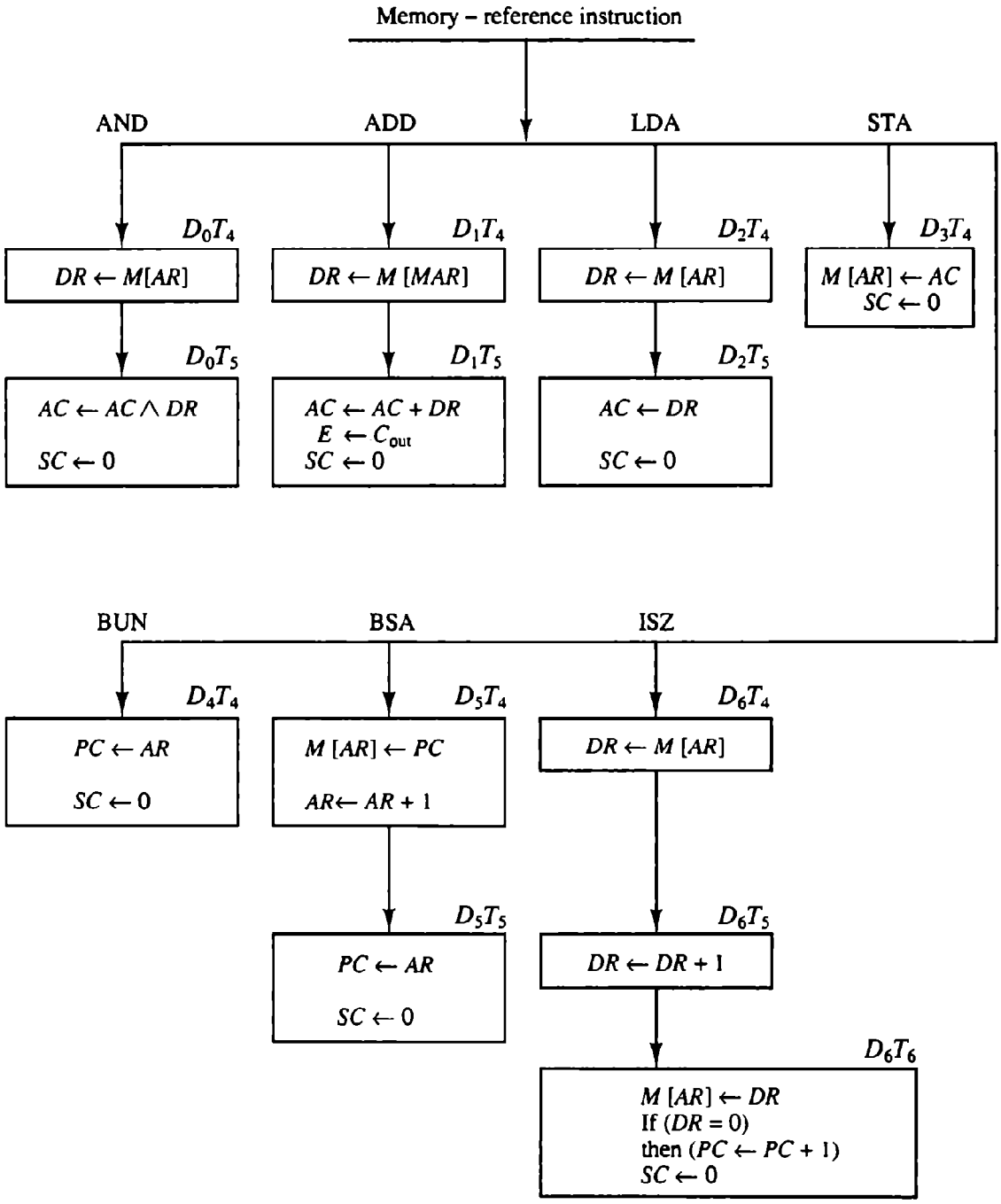


Figure 5-11 Flowchart for memory-reference instructions.

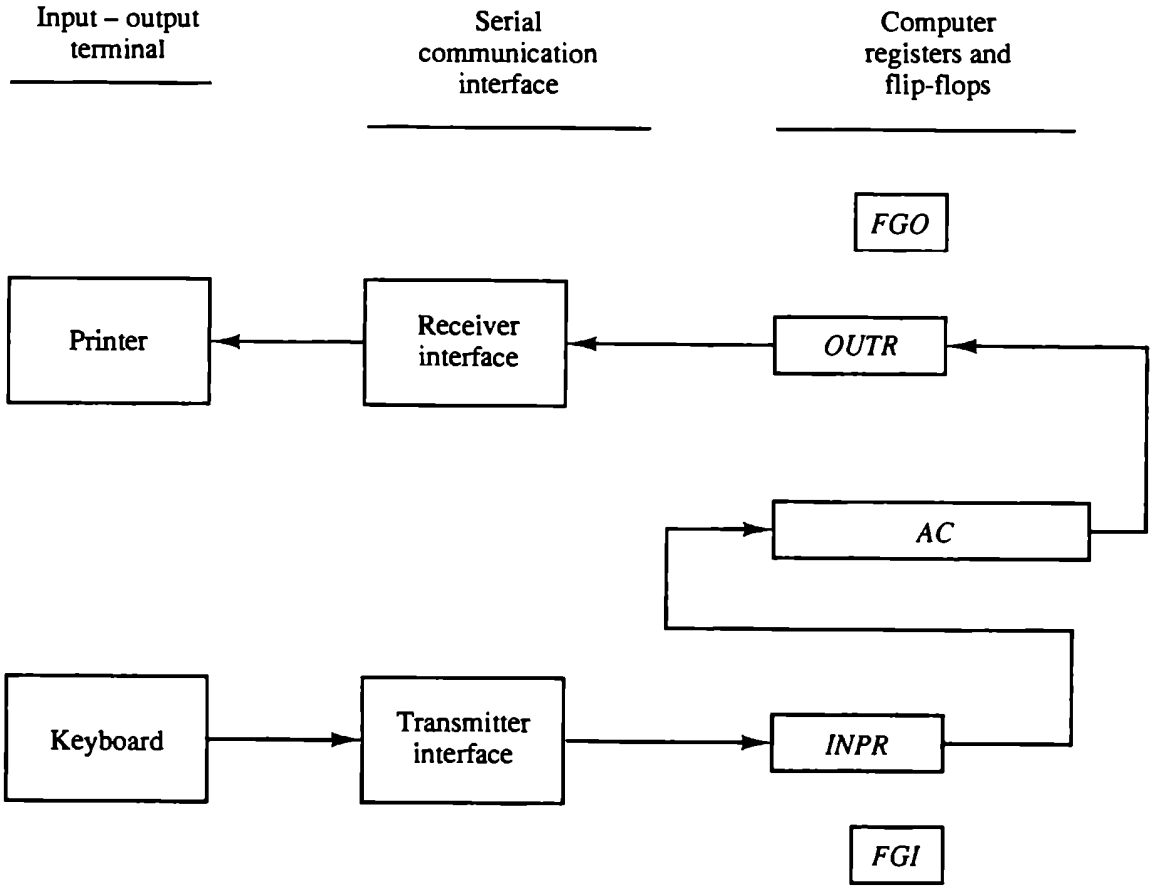
INPUT-OUTPUT AND INTERRUPT:

Instructions and data stored in memory must come from input device. Results must be transmitted to the user through some output device.

Input-Output configuration:

The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two register communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in fig 5-12.

Figure 5-12 Input-output configuration.



The input register INPR consists of eight bits and holds an alphanumeric character. The 1-bt flag FGI is a control flip-flop. The flag bit is set to 1when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.

The output register OUTR works similarly but het direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the information, and prints the corresponding character. When operation is completed, it sets FGO to 1.

Input-output instructions:

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.

Input-output instructions have an operation code 1111 and are recognized by the control when D7=1 and I=1. The remaining bits specify the particular operation. Table 5-5 show the input-output instructions.

TABLE 5-5 Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

These instructions are executed at timing signal T3.

The INP instruction transfers the input information from INPR into the eight lower-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag 0. The next two instructions, check the status of the flags and cause a skip of the next instruction if the flag is 1. The last two instructions set and clear an interrupt enable flip-flop IEN.

Program Interrupt:

In this mechanism the device that wants to transfer the information generates the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0(with IOF instruction), the flags can not interrupt the computer. When IEN is set to 1(With the ION instruction), the computer can be interrupted.

The fig 5-3 explains the way the interrupt is handled by the computer.

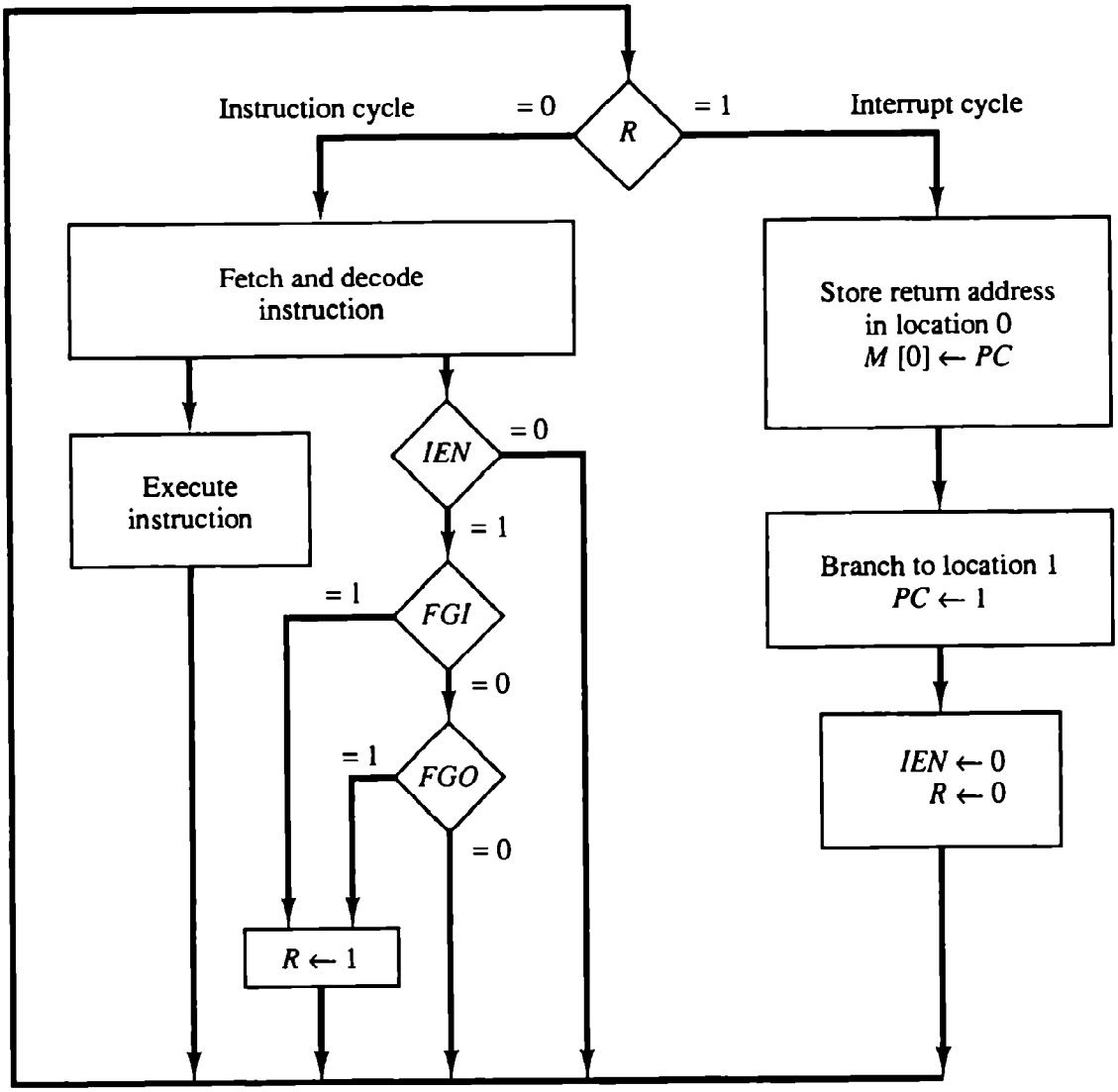


Figure 5-13 Flowchart for interrupt cycle.

When R=0, the computer goes through the instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.

If IEN is 1,control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output register are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to1 while IEN=1, flip-flop R is set 1. At the end of execute phase, control checks the value of R, and if it is equal to 1, it goes to the interrupt cycle instead of instruction cycle.

Interrupt Cycle:

DESIGN OF BASIC COMPUTER:

The basic computer consists of the following hardware components:

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

The functional block diagram of the hypothetic BASIC computer is as shown below:

Control Logic for gates:

The block diagram of the control logic gates is shown in fig 5-6. The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S_2, S_1, and S_0 to select a register for the bus
5. Signals to control the *AC* adder and logic circuit

The table 5.6 shows the specifications for various control signals

TABLE 5-6 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt:		
	$T_0'T_1'T_2'(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		
		$D_7I'T_3 = r$ (common to all register-reference instructions)
		$IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
		$D_7IT_3 = p$ (common to all input–output instructions)
		$IR(i) = B_i$ $(i = 6, 7, 8, 9, 10, 11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

Control logic for Registers and Memory:

The control inputs of the registers are LD(load), INR(increment), and CLR(clear). Suppose we want to design the gate structure associated with the control inputs of AR, we scan table 5.6 to find all the statement that change contents of AR.

$$\begin{aligned} R'T_0&: \quad AR \leftarrow PC \\ R'T_2&: \quad AR \leftarrow IR(0\text{–}11) \\ D'_7IT_3&: \quad AR \leftarrow M[AR] \\ RT_0&: \quad AR \leftarrow 0 \\ D_5T_4&: \quad AR \leftarrow AR + 1 \end{aligned}$$

The first 3 statements specify the transfer of information from a register or memory to AR. The 4th statement clears AR to 0. The last statement increases AR by 1.The control functions can be combined into Boolean expressions as follows:

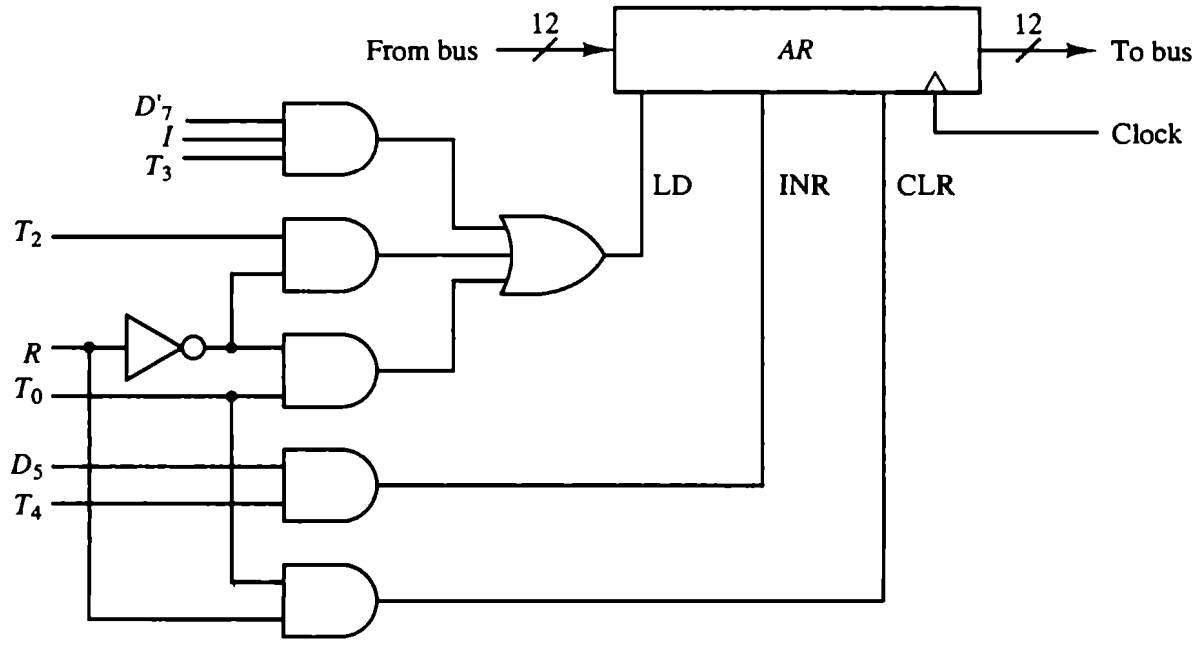
$$\begin{aligned} \text{LD}(AR) &= R'T_0 + R'T_2 + D'_7IT_3 \\ \text{CLR}(AR) &= RT_0 \\ \text{INR}(AR) &= D_5T_4 \end{aligned}$$

The control gate logic associated with AR is shown in the fig 5-16.

The logic gates associated with the read input of memory is derived by scanning table 5-6 and find the statements that specify the read operation. The read operation recognized from the symbol ←[MR]

$$\text{Read} = R'T_1 + D'_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

Figure 5-16 Control gates associated with *AR*.



Control of single flip-flops:

The control gates for seven flip-flops can be determined in similar manner. For example, from table 5-6 consider the IEN flip-flop.

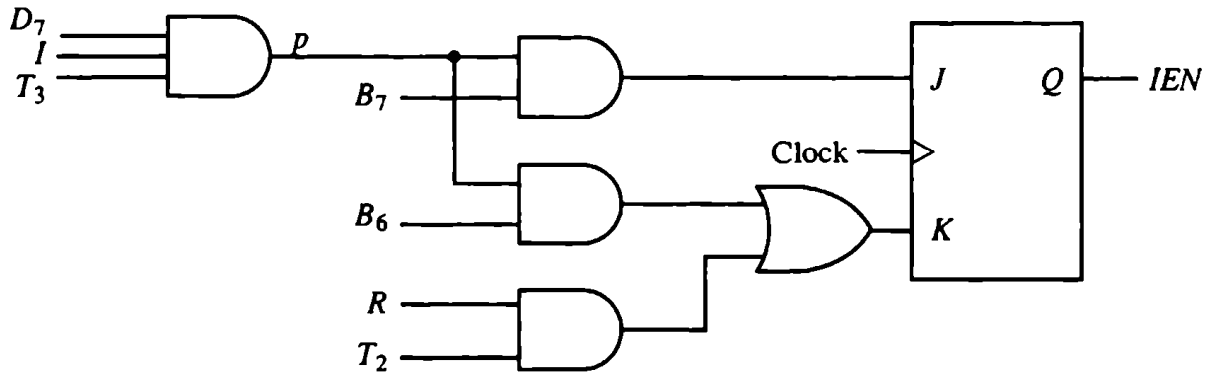
$$pB_7: \quad IEN \leftarrow 1$$
$$pB_6: \quad IEN \leftarrow 0$$

Where p=D7IT3 and b7 and b6 are bits 7 and 6 of IR, respectively. More over at the end of the interrupt cycle IEN is cleared to 0.

$$RT_2: \quad IEN \leftarrow 0$$

The control logic of IEN flip-flop by using JK flip is shown in the fig 5-17.

Figure 5-17 Control inputs for *IEN*.



Control logic for Common bus:

The 16-bit common bus shown in fig 5-4 is controlled by the selection inputs S2, S1 and S0. Table 5-7 specifies the binary number for S2S2S0 that select each register. Each binary number is associated with a Boolean variable x1 through x2, corresponding to the gate structure that must be active in order to select the register or memory for the bus.

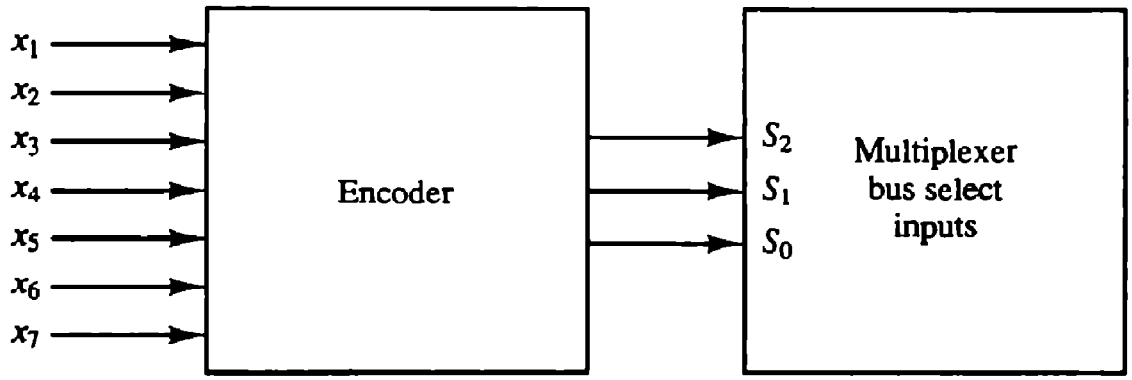
For example x1=1, the value of S2S1S0 must be 001 and the output of AR will be selected for the bus. Table 5-7 is recognized as the truth table of a binary encoder. The placement of encoder at the inputs of the bus selection logic is shown in fig 5-19. The Boolean functions for the encoder are:

$$S_0 = x_1 + x_3 + x_5 + x_7$$
$$S_1 = x_2 + x_3 + x_6 + x_7$$
$$S_2 = x_4 + x_5 + x_6 + x_7$$

TABLE 5-7 Encoder for Bus Selection Circuit

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	*AR*
0	1	0	0	0	0	0	0	1	0	*PC*
0	0	1	0	0	0	0	0	1	1	*DR*
0	0	0	1	0	0	0	1	0	0	*AC*
0	0	0	0	1	0	0	1	0	1	*IR*
0	0	0	0	0	1	0	1	1	0	*TR*
0	0	0	0	0	0	1	1	1	1	Memory

Figure 5-18 Encoder for bus selection inputs.



To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register on to the bus. For example, to find the logic that makes x1=1, we scan all register transfer statements in table 5-6 and extract those statements that have AR as a source.

$$x_1 = D_4T_4 + D_5T_5$$

The data output from memory are selected for the bus when X7=1 and S2S1S0=111. The gate logic that generates x7 must be applied to the read input of the memory. Therefore, the Boolean function for x7 is same as the read operation.

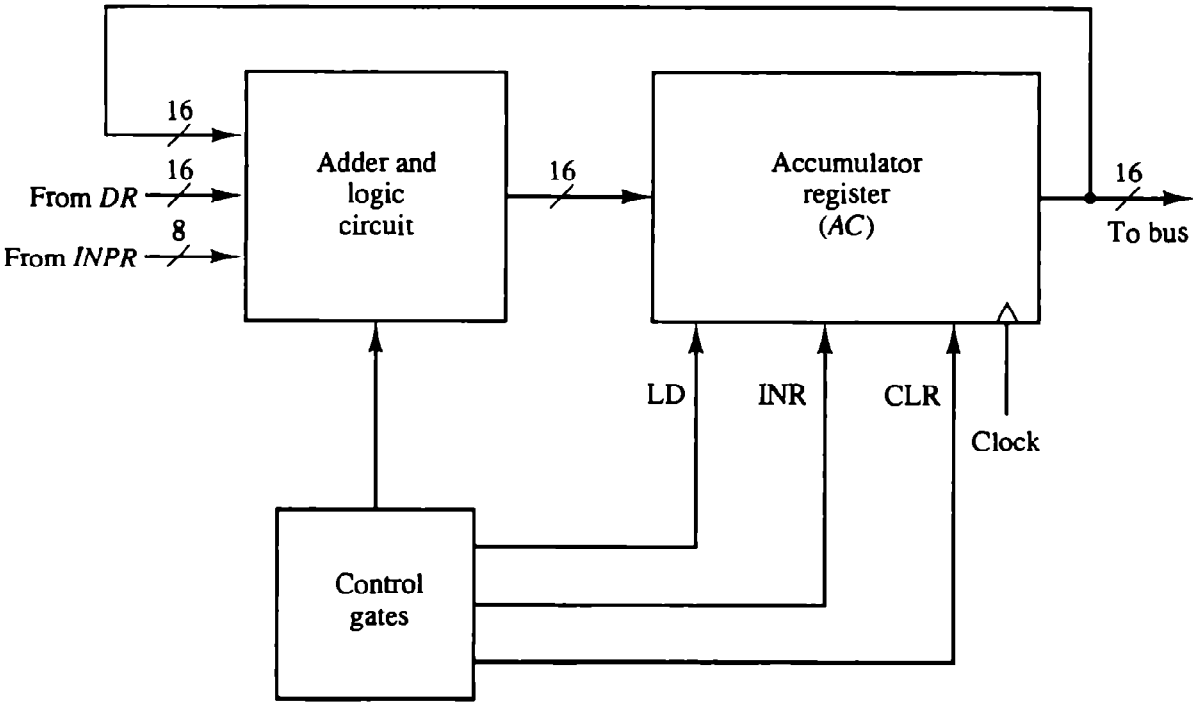
$$x_7 = R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

In similar manner we derive the logic for the other registers.

DESIGN OF ACCUMULATOR LOGIC:

The circuits associated with the AC registers are shown in fig 5-19.

Figure 5-19 Circuits associated with AC.



The adder and logic circuit has three inputs. The first input comes from the output AC, the second input come from data register DR and third input come from input register INPR. The output of the adder logic circuit provides the data input to the AC register. In addition it is necessary to include logic gates for controlling the LD,INR, and CLR in the register and controlling the operation of the adder and logic circuit.

In order to design the logic associated with AC, find all the statement from the table 5-6 that change the contents of AC.

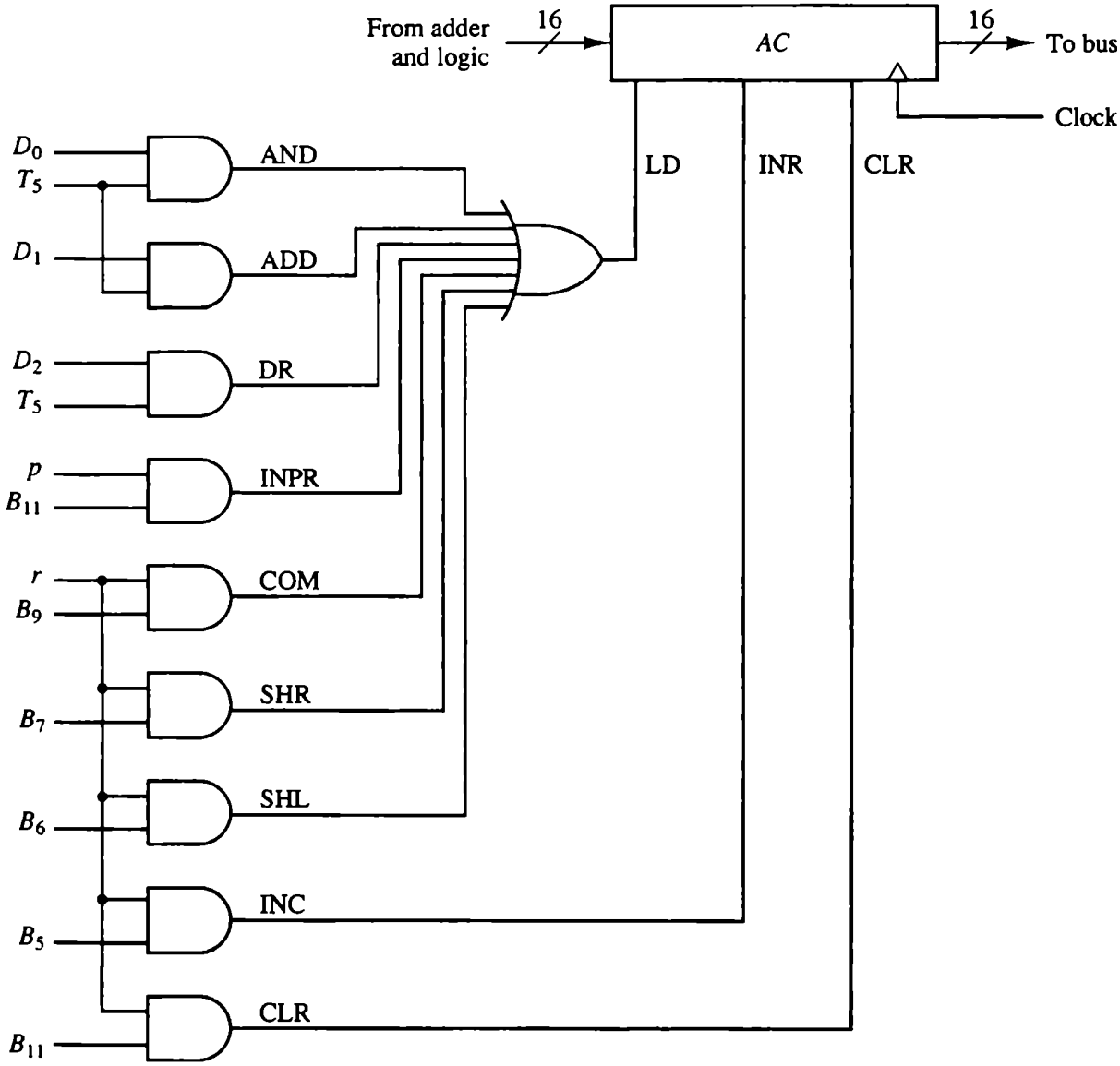
D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

From the above list we can derive the control logic gates and the adder and logic circuit.

Control of AC Register:

The gate structure that controls LD,INR, and CLR inputs of AC is show in fig 5-20.

Figure 5-20 Gate structure for controlling the LD, INR, and CLR of AC.



The gate configuration is derived from the control functions in the above list. The control function for the clear microoperation is rB_{11}, where

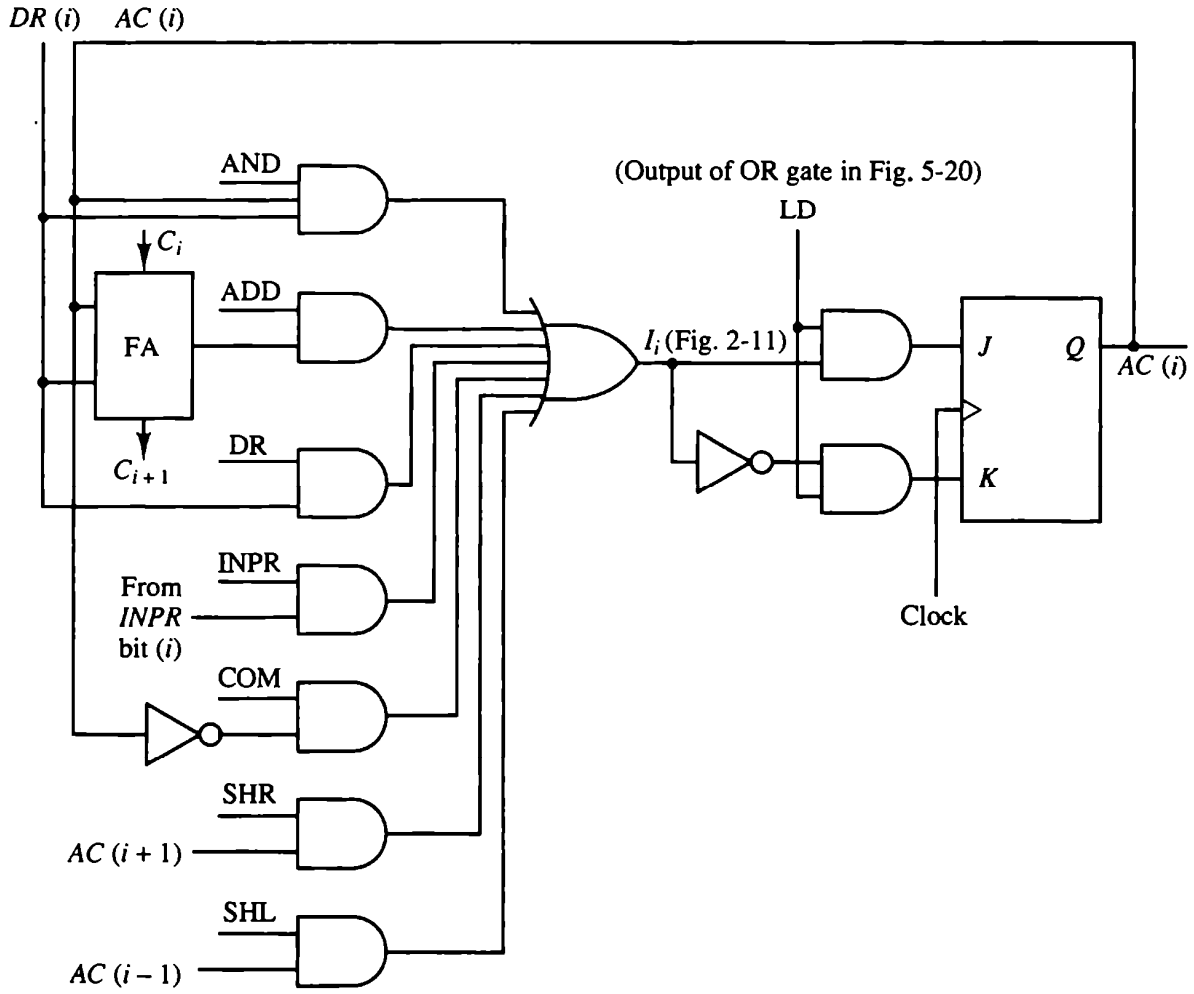
$$r = D_7 I' T_3 \text{ and } B_{11} = IR(11)$$

The output of the AND gate that generates the control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment operations is connected to the INR input of the register. The other seven microoperations are generated in the adder and logic circuit and are loaded into AC

Adder and Logic circuit:

The adder and logic circuit can be subdivided into 16 stages, with each stage correspoinding to one bit of AC. Fig 5.21 shows one AC register stage.

Figure 5-21 One stage of adder and logic circuit.



The circuit consists of seven AND gates, one OR gate and a full-adder(FA). The AND operation is achieved by ANDing AC(i) with the corresponding bit in the data register DR(i). The ADD operation is obtained using a binary adder. The transfer from INPR to AC is only for bits 0 through 7. The complement microoperation is obtained by inverting the bit value in AC. The shift-right operation transfers the bit from AC(i+1), and the shift-left operation transfers the bit from AC(i-1). The complete adder and logic circuit consists of 16 stages connected together.

UNIT-III

CENTRAL PROCESSING UNIT

AND

MICRO PROGRAMMED CONTROL

Syllabus:

CENTRAL PROCESSING UNIT : General Register Organization, STACK organization. Instruction formats. Addressing modes. DATA Transfer and manipulation. Program control. Reduced Instruction set computer.

MICRO PROGRAMMED CONTROL : Control memory, Address sequencing, micro program example, design of control unit

CENTRAL PROCESSING UNIT:

The part of the computer that performs bulk of data processing operations is called the central processing unit(CPU). The CPU consists of 3 major parts as shown in fig 8-1.

1. The register set for storing the temporary data during the execution of instructions.
2. The arithmetic logic unit(ALU) performs the required microoperations for executing the instructions.
3. The Control Unit(CU), that control transfer of information among the registers and instructs the ALU to which operation to perform.

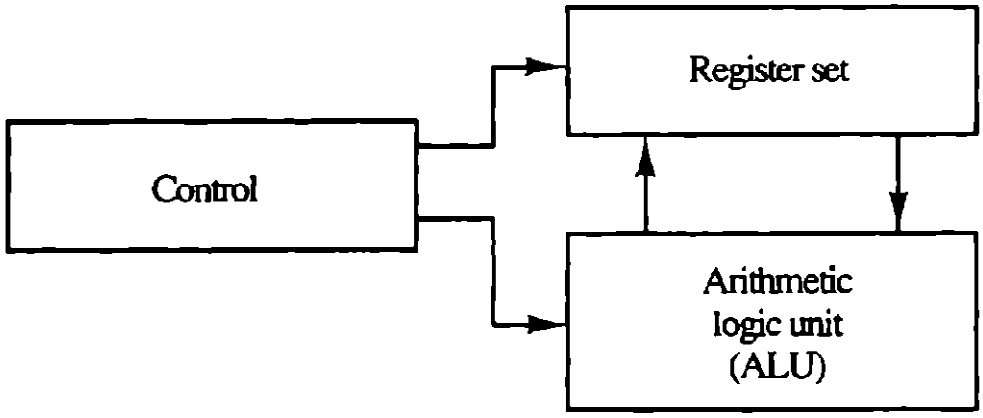


Figure 8-1 Major components of CPU.

From the designer's point of view, the computer instructions set provides the specifications for the design of the CPU.

GENERAL REGISTER ORGANIZATION:

It is more convenient and more efficient to store intermediate values in processor registers rather than in memory, because memory access is much slower. These registers communicate

by using the common bus. Hence it is necessary to provide a common unit that can perform all arithmetic, logic, and shift microoperations in the processor.

The bus organization for seven CPU registers is shown in fig 8-2. The output of each register is connected to two multiplexers(MUX) to from the two buses A and B. The selection lines in each multiplexer select one register data for the particular bus. The A and B buses form the inputs to ALU. The operation selected in the ALU determines the arithmetic or logic microoperation. The result of the microoperation(the ALU output) goes into the inputs of all the registers. The decoder select a particular register to store the result by activating the LD(load) input of the register.

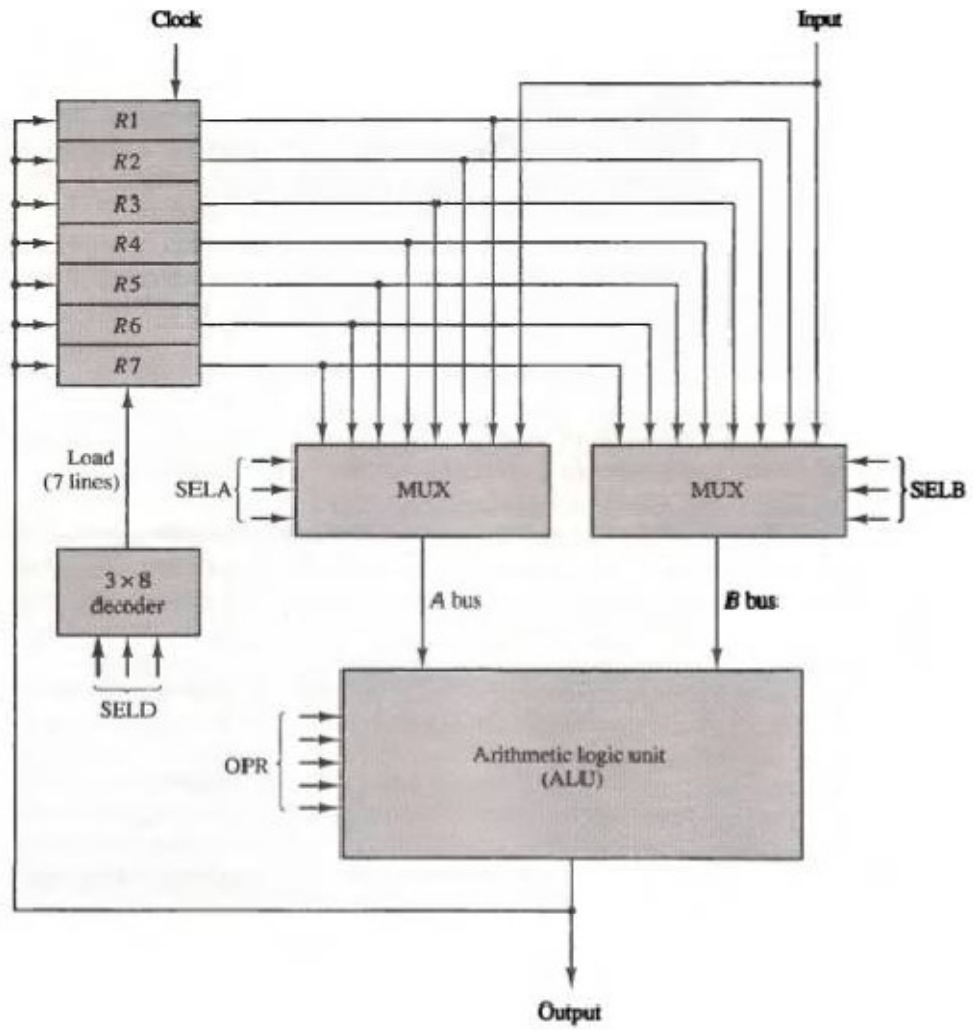
The control unit is responsible for controlling the bus system, registers and transfers between the registers.

For example, consider the following operation:

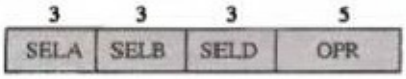
$R1 \leftarrow R2 + R3$

The control must provide 4 selection variables to the following selector inputs for performing the above operation:

1. MUX A selector (SELA): to place the content of $R2$ into bus A.
2. MUX B selector (SELB): to place the content of $R3$ into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.
4. Decoder destination selector (SELD): to transfer the content of the output bus into $R1$.



(a) Block diagram

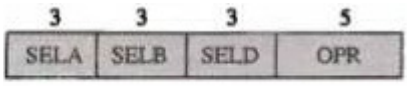


(b) Control word

Figure 8-2 Register set with common ALU.

Control word:

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in fig 8-2(b). It consists of four fields. Three fields contain three bits each, and one filed has five bits.



(b) Control word

The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The 5 bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

The encoding of the register selections is specified in table 8-1.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R1*	*R1*	*R1*
010	*R2*	*R2*	*R2*
011	*R3*	*R3*	*R3*
100	*R4*	*R4*	*R4*
101	*R5*	*R5*	*R5*
110	*R6*	*R6*	*R6*
111	*R7*	*R7*	*R7*

When SELA and SELB is 000, the corresponding multiplexer selects the external input data. When SELD=000, no destination register is selected but the contents of the output bus are available in the external output.

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide shift capability, of at the output of the ALU to provide the post shifting capability. The following table shows the encoding of the ALU operations. The OPR field has five bits and each operation is designated with a symbolic name.

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

Example of Microoperations:

A control word of 14-bits is needed to specify a microoperation in the CPU. The conrol word for a given microoperation. For example, consider the following subtraction microoperation:

$$R1 \leftarrow R2 - R3$$

The above operation specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and the ALU operation to subtract A – B. Thus, the control word is specified by the four fields and the corresponding binary value for each field is obtained from the table 8-1 and 8-2. The binary control word for the above subtraction operation is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

The increment and transfer microoperation do not use the B input of the ALU. For these cases, the B field is marked with dash.

The control word for different microoperations is listed in the following table:

TABLE 8-3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

STACK ORGANIZATION:

A Stack is a storage structure that stores information in such a way that the item stored last is the first item retrieved. For this reason stack is also called Last-in, First-out(LIFO) list.

The register that holds the address for the stack is called a stack pointer(SP). SP always points at the top item in the stack. The two operations of a stack are the insertion and deletion of items. The operation of insertion is called ***push*** and operation of deletion is called ***pop***. The operations push and pop can be achieved by incrementing and decrementing the SP.

The stack can be organized in two ways:

1. By using registers(Register stack)
2. By using Memory words(Memory stack)

Register stack:

Fig 8-3 shows the organization of a 64-bit word register stack. Three items A,B and C are placed in the stack. The address in the SP is the address of the word that is currently on the top of stack. Item C is on the top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next higher-location of the stack.

In a 64-word stack the SP contains 6 bits because 26 = 64. The maximum number SP can store is 111111. When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000. Similarly, 000000 is decremented by , the result is 111111. The 1-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty.

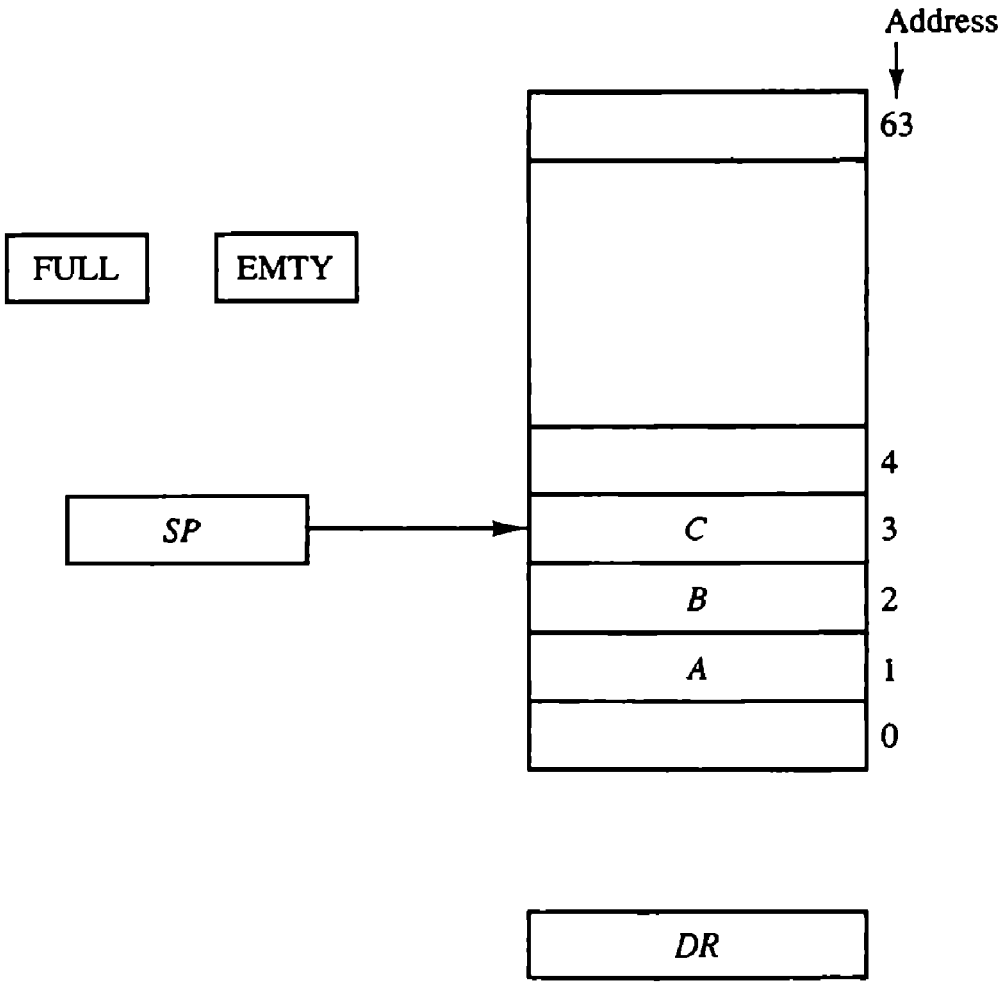


Figure 8-3 Block diagram of a 64-word stack.

The push operation is implemented with the following sequence of microoperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

The pop operation is implemented with the following sequence of microoperations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

Memory stack:

A stack can be implemented in a random-access memory attached to a CPU called memory stack. In this, the portion of memory is assigned to the stack operation and process register is used as a stack pointer. Fig 8-4 shows a portion of computer memory partitioned into three segments: ***Program, data and stack.***

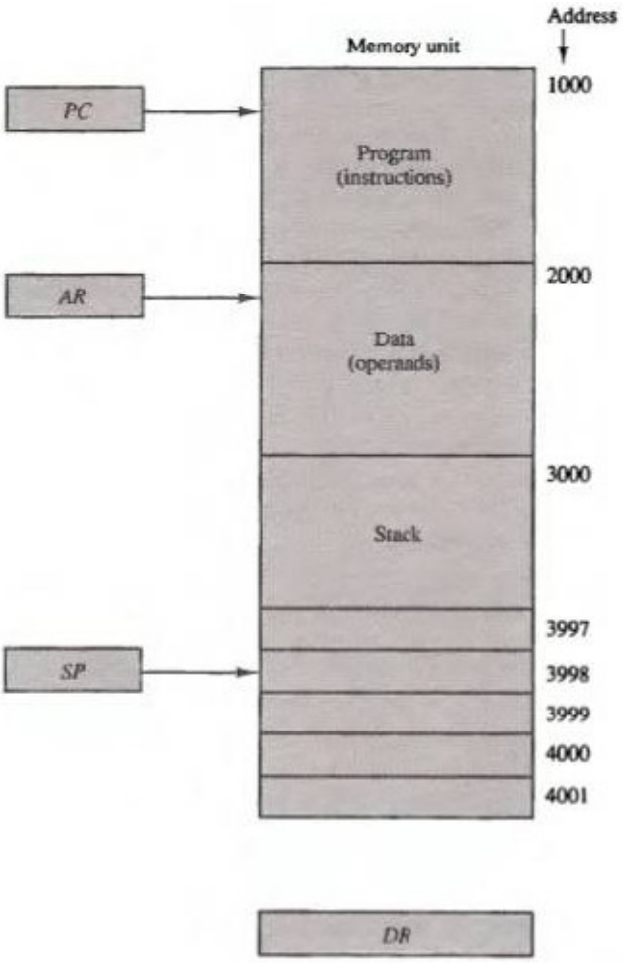


Figure 8-4 Computer memory with program, data, and stack segments.

The program counter PC points at the address of the next instruction. The address register AR point at an array of data. The stack pointer SP point at the top of the stack. The three registers are connected to a common address bus.

PC is used during the fetch phase to read an instruction. AR is used during the executed phase to read an operand. SP is used to push or pop items into or from the stack.

A new is item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

The two microoperations are need for either the push or pop

1) An access to memory through SP
2) Updating SP.

Reverse Polish Notation(RPN):

The stack organization is very effective for evaluation arithmetic expressions. The following are the three representations for the expressions:

$A + B$	Infix notation
$+AB$	Prefix or Polish notation
$AB+$	Postfix or reverse Polish notation

The reverse polish notation is suitable for stack manipulation. The expression

$$A * B + C * D$$

is written in reverse polish notation as:

$$AB * CD * +$$

The above expression is evaluated as follows:

Scan the expression from left to right. When an operator is reached, perform the operation with two operands found on the left side of the operator. Remove the two operands and operator and replace them by the result of the operation. Continue this process until there are no more operators.

Converting infix expression to postfix (reverse polish) notation:

This can be done by first performing arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Consider the expression:

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

To evaluate the above expression, we must perform the arithmetic inside the parentheses (A+B) and (D+E). Next we must calculate the expression inside the square brackets. The manipulation of C*(D+E) must be done prior to the addition of F since multiplication has

precedence over addition. The last operation is multiplication of the two terms between the parentheses and brackets.

$$AB + DE + C * F + *$$

Evaluation of arithmetic expressions:

Reverse polish notation combined with a stack is the most efficient way for evaluating arithmetic expressions. The procedure is as follows:

1. First convert the arithmetic expression into its equivalent reverse polish notation.
2. The operands are pushed into the stack in order in which they appear.
3. When we get the operator, the top most two operands are poped and perform the operation
4. The result of the operation replaces the lower operand.

By pushing the operands into the stack continuously and performing the operations as above, the expression is evaluated in the proper order and the final result remains on the top of the stack.

Example:

Consider the arithmetic expression:

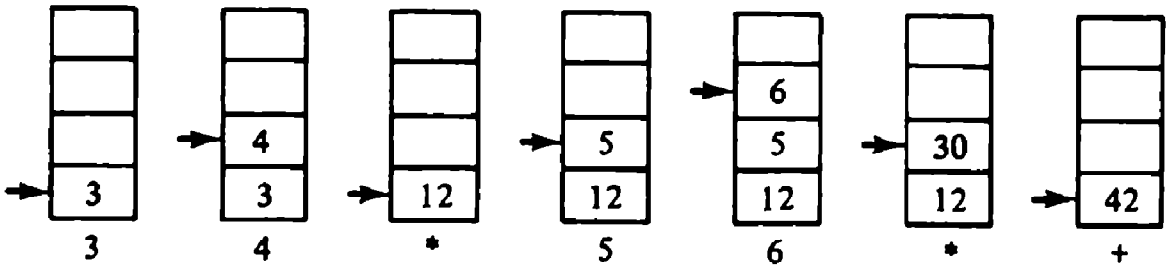
$$(3 * 4) + (5 * 6)$$

The reverse polish notation of the above expression is

$$34 * 56 * +$$

Fig 8-5 shows the stack operations for evaluating the above expression:

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



INSTRUCTION FORMATS:

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

Most computers fall into one of three types of CPU organizations.

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

Based on the type of CPU organization computers may have instructions of several lengths. The length of the instruction depends on the number of address fields in the instruction.

The most common fields found in instruction formats are:

1. An operation code(op code)
2. Address field
3. A mode field

The instruction format in accumulator-type organization has one address field. All operations are performed with accumulator. For example, the addition operation is represented as follows:

```
ADD    X
```

The above add operation is done as follows:

$$AC \leftarrow AC + M[X]$$

The instruction format in register type organization needs three register address fields. For example, the addition operation is written as follows:

```
ADD    R1, R2, R3
```

The above operation is denoted by:

$$R1 \leftarrow R1 + R2$$

The number of address fields in the above instruction can be reduced from three two if the destination register is the same as one of the source register. Thus the instruction:

```
ADD    R1, R2
```

The above operation is denoted by:

$$R1 \leftarrow R1 + R2$$

The register-types computers contain two or three fields in their instruction format. Each address field specify a processor register or a memory word. Consider the following instruction:

```
ADD    R1, X
```

The above instruction has two address fields, one for register R1 and the other for the memory address X.

The stack-organized CPU has PUSH and POP instructions which require one address field. Consider the following instruction:

```
PUSH    X
```

The above instruction pushes the word at address X, to the top of the stack.

The operation-type instruction does not need an address filed in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. For example:

```
ADD
```

The above instruction consists of an operation code only but no address field.

Based on the discussion, we have the following types of instruction formats:

1. Three-Address instructions
2. Two-Address instructions
3. One-Address instructions
4. Zero-Address instructions.
5. RISC instructions.

To understand the above instruction formats, consider how the following arithmetic statement is evaluated by using the above instruction formats.

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

Three-addressed instructions:

Computers with three-address instruction formats can use each address filed to specify either a process register or memory operand. Following the program in assembly language to evaluate the above expression:

```
ADD    R1, A, B      R1 ← M[A] + M[B]
ADD    R2, C, D      R2 ← M[C] + M[D]
MUL    X, R1, R2     M[X] ← R1 * R2
```

Here, The symbol M[A] denotes the operand at memory address A. The advantage of the three-address format is that it results in short programs, when evaluating arithmetic expressions. The disadvantage is that the instructions require too many bits to specify three addresses.

Two-address instructions:

There are most common in commercial computers. Here also the address filed specify either a processor register or a memory word. The program to evaluate the above expression by two-address instruction is as follows:

```
MOV    R1, A     R1←M[A]
ADD    R1, B     R1←R1 + M[B]
MOV    R2, C     R2←M[C]
ADD    R2, D     R2←R2 + M[D]
MUL    R1,R2     R1←R1*R2
MOV    X, R1     M[X]←R1
```

The MOV instruction moves the operands to and from memory and processor registers.

One-Address instructions:

One-address instructions use an implied accumulator(AC) register for all data manipulation. The program to evaluate the above expression is:

```
LOAD     A    AC←M[A]
ADD      B    AC←AC + M[B]
STORE    T    M[T]←AC
LOAD     C    AC←M[C]
ADD      D    AC←AC + M[D]
MUL      T    AC←AC*M[T]
STORE    X    M[X]←AC
```

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

Zero-Address instructions:

A stack-organized computer does not use an address field for the instructions ADD and MUL. However, PUSH and POP instructions need an address field. The following program evaluates the above arithmetic expression:

```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

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

RISC(Reduced Instruction Set Computer) instructions:

The instruction set of RISC processor is restricted to use the load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring the memory.

The LOAD and STORE instructions have one memory and one register address, and computational type instructions that have three addresses, all specifying the processor registers. The following is the program for evaluating the above expression:

```
LOAD    R1, A          R1 ← M[A]
LOAD    R2, B          R2 ← M[B]
LOAD    R3, C          R3 ← M[C]
LOAD    R4, D          R4 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
ADD     R3, R3, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1*R3
STORE   X, R1          M[X] ← R1
```

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

ADDRESSING MODES:

Program execution require, choosing the operands from memory. Address modes provides different ways to choose the operand from memory. Addressing modes specifies the rules for modifying the address field of the instruction for referring the operand.

1. Addressing modes provides flexibility in writing the programs.
2. To reduce the number of bits in the addressing field.

Fig 8-6 shows the instruction format with a distinct address mode field

Figure 8-6 Instruction format with mode field.

The mode field is used to located the operand need for operation in different ways .

Following are various addressing modes:

1. Implied mode
2. Immediate mode
3. Register mode
4. Register indirect mode
5. Autoincrement and Autodecrement mode.
6. Direct address mode
7. Indirect address mode
8. Relative address mode
9. Indexed addressing mode
10. Base register addressing mode.

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

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

Register mode: When the address field specifies a processor register, the instruction is said to be in the register mode. In this mode the operand are in the CPU registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In the other words, the selected register contains the address of the operand rather than the operand itself.

The advantage of a register indirect mode instruction is that, the address filed of the instruction uses fewer bits to select a register than specifying the memory address directly.

Autoincrement or Autodecrement 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.

For example, 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.

Direct Address Mode: In this mode the address part of the instruction contains the effective address of the operand. In the branch-type instruction the address filed specified the actual branch address.

Indirect Address Mode: In this mode the address filed 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

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

Effective address = address part of instruction + content of CPU register

The CPU register may be the PC, IR, OR BR(base register).

Relative Address Mode: in this mode the EA is obtained by adding the contents of PC to the address part of the instruction. The relative addressing is often used with branch-type instructions, when the branch address is in the area surrounding to the instruction. The advantage of this mode is, it requires a shorter address filed because relative address can be specified with a smaller number of bits.

Indexed Addressing Mode: In this mode the EA can be obtained by adding the content of an IR to the address part of the instruction. The index register is 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 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 values, stored in the index register. The index register can be incremented to facilitate access to consecutive operands.

Base Register Addressing Mode: In this mode the EA can be obtained by adding the content of base register to the address part of the instruction. This is similar to the indexed addressing mode except that the register is base register instead of an index register.

The difference between the two modes is, an index register holds an index number relative to the address part of the instruction. A base register holds a base address and the address filed of the instruction gives a displacement relative to this base address.

The base register addressing mode is used to facilitate the relocation of programs in memory. To achieve relocation or programs only the value of the base register is to be changed to reflect the beginning of a new memory segment.

Numerical Example:

To show the differences between the various modes, consider the two word instruction defined in fig 8-7 . The two-word instruction at address 200 and 201 is a “load to AC” instruction with an address filed equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. The mode filed of the instruction can specify any one of a number of modes. For each possible mode, we

calculate the EA and the operand that must be loaded in AC.

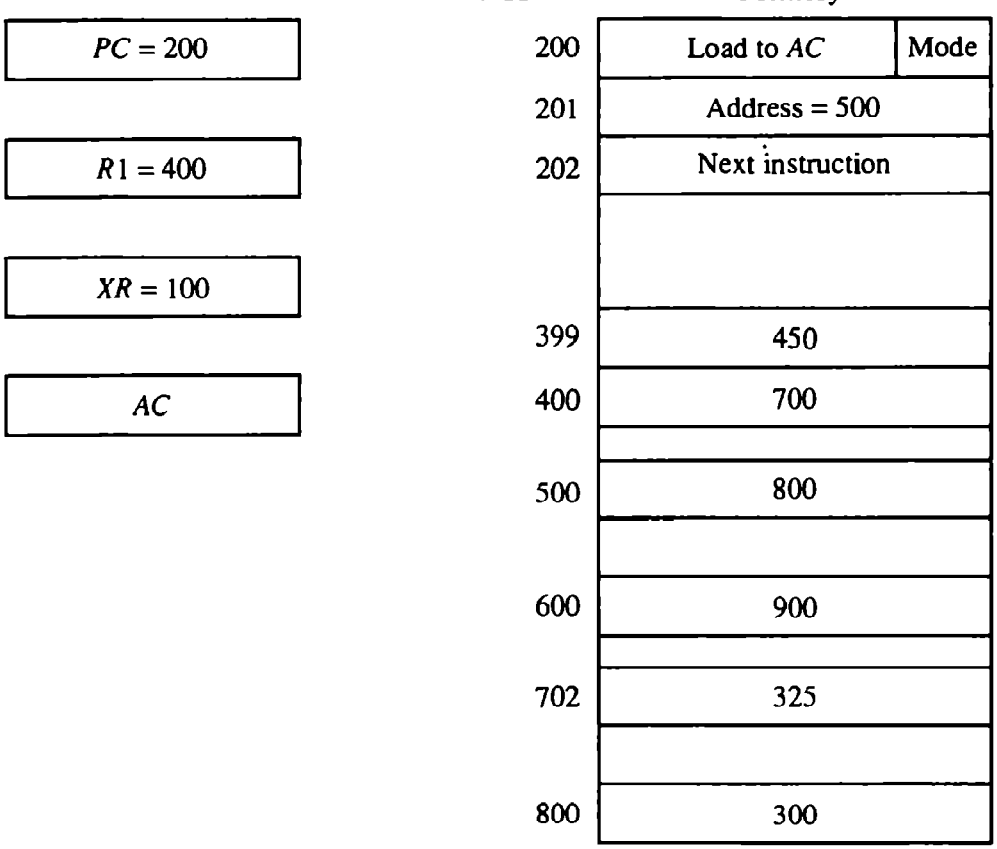


Figure 8-7 Numerical example for addressing modes.

Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

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

DATA TRANSFER AND MANIPULATION:

Computers provide and extensive set of instructions to give the uses the flexibility to carry out various computational tasks. The instructions set differs from computer to computer in the way the operands are determined from the address mode fields.

Most computer instructions can be classified into three categories:

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

Data Transfer Instructions:

Data transfer instructions move data from one place in the computer to another. These instructions do not change the data content. The common transfers are:

1) Transfers between memory and processor registers
2) Between processor registers and input or output.
3) Between the processor registers themselves.

Table 8-5 shows 8 data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

The **load** instruction transfers data from memory to process register, usually an accumulator. The **store** instruction transfers data from process register into memory. The **move** instruction transfers data from one processor register to another processor register. The move can also be used to transfer data between processor registers and memory or between two memory words. The **exchange** instruction swaps information between two registers or a registers and a memory word. The **input** and **output** instructions transfer data among processor registers and

input or output terminals. The **push** and **pop** instructions transfer data between processor registers and a memory stack.

The instructions are associated with a variety of addressing modes. The table 8-6 show load accumulator instruction with eight different addressing modes.

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

ADR stands for an address, *NBR* is a number or operand, *X* is an index register, *R1* is a processor register, and *AC* is the accumulator register. The @ symbolizes an indirect address. The $ before an address makes the address relative to PC. The # before the operand is an immediate mode instruction.

Data Manipulation Instructions:

These instructions perform operations on data and provide the computational capabilities for the computer. The data manipulations instructions are divided into three types:

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

Arithmetic Instructions: the four basic arithmetic operations are addition, subtraction, multiplication, and division. The table 8-7 gives a list of arithmetic instructions. The increment instruction adds 1 to the values stored in a register or memory word. The decrement operation subtracts 1 from a value stored in a register or memory word.

The add, subtract, multiply, and divide instructions may be available for different types of data such as fixed-point or floating-point , binary or decimal data, single-precision or double-precision data.

TABLE 8-7 Typical Arithmetic Instructions

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

=

The add instructions on different data types are shown below:

```
ADDI    Add two binary integer numbers
ADDF    Add two floating-point numbers
ADDD    Add two decimal numbers in BCD
```

Logical and Bit Manipulation Instructions:

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

Table 8-8, list the bit manipulation instructions.

The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND,OR and XOR instruction produce the corresponding logical operations on the individual bits of the operands.

A few bit manipulation instructions are listed in table 8-8.

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Shift instructions:

Shift instructions move bits of a word to the left or right. The bit shifted in, at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 8-9 lists four types of shift instructions.

TABLE 8-9 Typical Shift Instructions

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

The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The arithmetic shift-left operation inserts 0 to the end position and is identical to the logical shift-left instruction.

The rotate instructions produce a circular shift. Bits shifted out are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of register whose word being rotated. Thus rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

PROGRAM CONTROL:

The data transfer and manipulation instructions specify conditions for data-processing operations. The program control instructions control flow of program execution by altering the content of the PC. The program control instructions provide the capability for branching to different program segments.

Table 8-10 list some program control instructions:

TABLE 8-10 Typical Program Control Instructions

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

The branch and jump instruction may be conditional or unconditional. They do the same thing but use different addressing modes. These instructions transfer the value of ADR(address) into the PC.

The skip instruction does not need an address field therefore it is a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is satisfied. This is accomplished by incrementing the program counter during the execute phase in addition to being incremented during the fetch phase.

The call and return instructions are used in conjunction with subroutines.

The compare instruction performs a subtraction between two operands and set certain status bits conditions based on the result of the operation. Similarly, the test instruction performs logical AND of two operands and updates certain status bits.

The status bits are: carry bit, sign bit, zero bit, overflow bit.

Status bits(flag bits):

The ALU circuit is associated with the status register. The status register consist of status bit conditions for further analysis. The status bits are also called condition-code bits or flag bits. Fig 8-8 shows the block diagram of an 8-bit ALU with 4-bit status register. The four status bits are symbolized by *C,S,Z* and *V*. The bits are set or cleared as a result of an operation performed in the ALU.

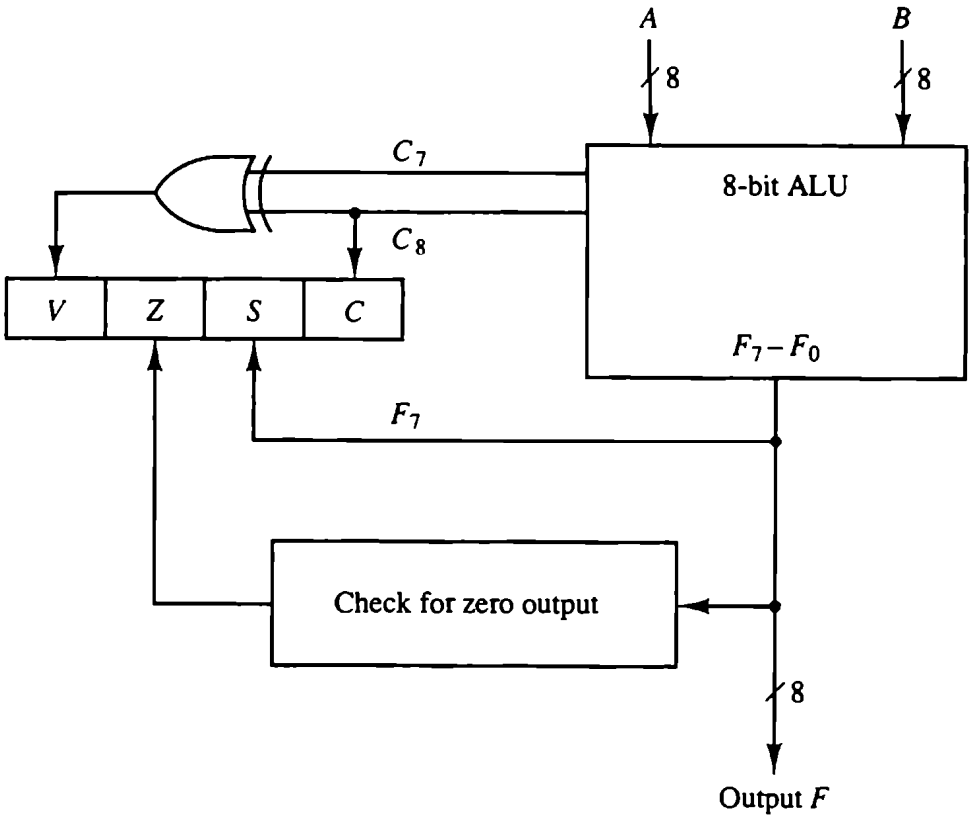


Figure 8-8 Status register bits.

1. Bit C(carry) is set to1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S(Sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit Z(zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V(overflow) is set to 1, if the X-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V=1 if the output is greater than +127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B.

Conditional Branch instructions:

Table 8-11 give a list of branch instructions. Each mnemonic starts with letter B(branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state.

TABLE 8-11 Conditional Branch Instructions

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

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

The branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1.

The compare instruction performs subtraction of two operands A and B and status bits are updated based on the result. Table 8-11 lists the conditional instructions that can be executed after the compare instruction for both unsigned and signed numbers.

Subroutine Call and Return:

A subroutine is a self-contained sequence of instructions that performs a given computational task. A subroutine is called many times during program execution to perform a function. Each time a subroutine is called, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to subroutine is known as call subroutine or jump to subroutine, branch to subroutine, or branch and save address.

The call to subroutine instruction is executed by performing two operations:

1) The address of the next instruction available in the PC(return address) is stored in a temporary location.
2) Control is transferred to the beginning of the subroutine.

The last instruction of every subroutine is the return from subroutine, transfers the return address from the temporary location into the PC.

A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

Program interrupt:

Program interrupt transfers the program control from currently running program to another service as a result of external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is as follows:

1) The interrupt is initiated by an internal or external signal
2) The address of the interrupt service program is determined by the hardware rather than the address field of an instruction.
3) An interrupt procedure usually stores all the information to define the state of the CPU rather than only storing the PC.

After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred.

The state of the CPU can be determined from:

1) The contents of the PC
2) The content of the Processor registers.
3) The content of status register.

The collection of all status bit conditions in the CPU is called a **program status word(PSW).**

Types of Interrupts:

There are three major types of interrupts:

1) External interrupts
2) Internal interrupts
3) Software interrupts.

External interrupts come from input-output (I/O) devices, from a timing device or from any other external source. Example include: I/O device requesting to transfer data, I/O device finished to transfer of data, or power failure.

Internal interrupts arise from illegal or erroneous use of an instruction. Internal interrupts are also called **traps**. Example include: register overflow, divide by zero, an invalid operation code, stack overflow, and protection violation.

A software interrupt is initiated by executing an instruction. Example of this instruction is switching from a CPU user mode to supervisory mode.

REDUCED INSTRUCTION SET COMPUTER (RISC):

The instruction set determines the way that the way machine language programs are constructed. Many computers have more than 100 or 200 instructions.

A computer with a large number of instructions is called as **Complex instruction set computer(CISC).**

A computer that uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is called as a **reduced instruction set computer (RISC).**

CISC characteristics:

1. A large number of instructions – typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.

3. A large variety of addressing modes-typically from 5 to 20 different modes.
4. Variable-length instruction formats.
5. Instructions that manipulate operands in memory.

RISC Characteristics:

The concept of RISC architecture is:

- Reducing the execution time
- Simplifying the instruction set of the computer.

RISC characteristic include:

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

Other characteristics of RISC architecture are:

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

Overlapped Register Windows:

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passing the results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time-consuming operations.

A characteristic of RIS processors is their use of ***overlapped register windows*** to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a new window containing a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing the pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have

overlapping registers that are shared to provide the passing of parameters and results. The concept of overlapped register window is illustrated in fig 8-9.

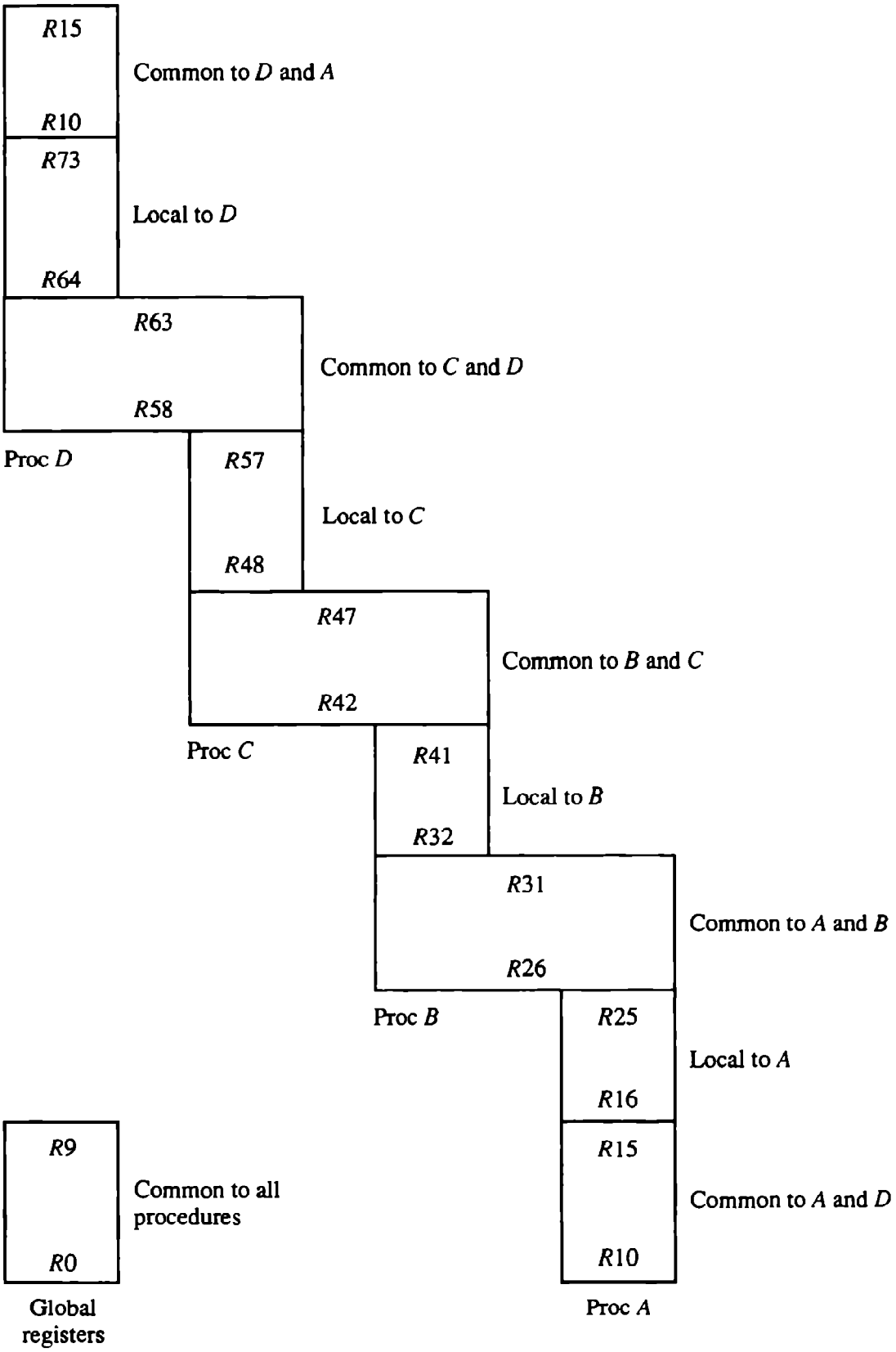


Figure 8-9 Overlapped register windows.

The organization of register windows will have the following relationships:

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

Berkeley RISC I:

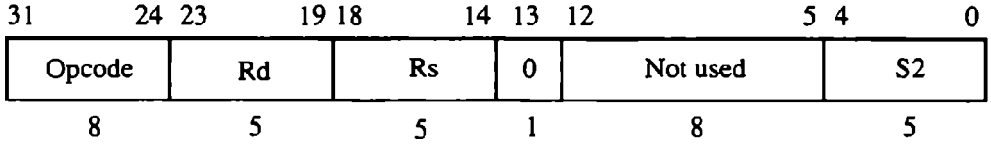
The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-,16-or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes:

- Register addressing
- Immediate addressing and
- Relative addressing

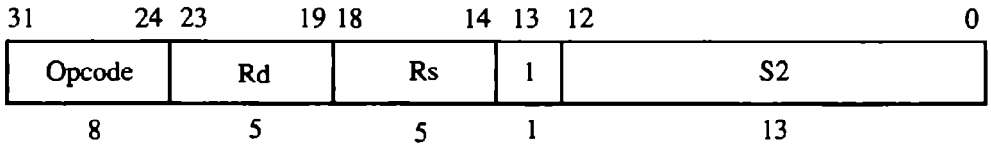
It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each. Since only one set of 32 registers in a window is accessed at any given time, the instruction format can specify a processor register with a register field of five bits.

Fig 8-10 shows the 32-bit instruction formats used for register-to-register instructions and memory access instructions.

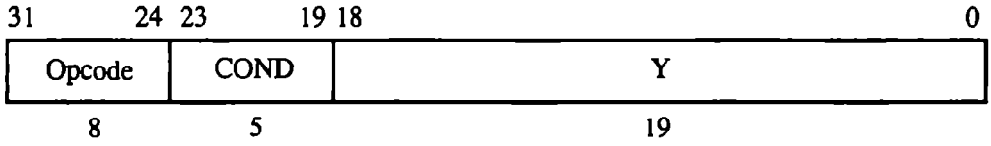
Figure 8-10 Berkeley RISC I instruction formats.



(a) Register mode: (S2 specifies a register)



(b) Register–immediate mode: (S2 specifies an operand)



(c) *PC* relative mode:

The 31 instructions of RISC I are listed in table 8-12. They have been grouped into 3 categories.

Data manipulation instructions perform arithmetic, logic and shift operations. All the instructions have three operands. The second source S2 can be either a register or an immediate operand. For example, consider the ADD instruction and how it can be used to perform variety operations:

```
ADD R22,R21,R23      R23←R22 + R21
ADD R22,#150,R23     R23←R22 + 150
ADD R0,R21,R22       R22←R21  (Move)
ADD R0,#150,R22      R22←150  (Load immediate)
ADD R22,#1,R22       R22←R22 + 1  (Increment)
```

The data transfer instructions consists of six load instructions, three store instructions, and two instructions that transfer the program status word PSW. The load and store instructions move data between a register and memory. Following are examples of load long instructions with different addressing modes:

```
LDL (R22)#150,R5     R5←M[R22] + 150
LDL (R22)#0,R5       R5←M[R22]
LDL (R0)#500,R5      R5←M[500]
```

The program control instructions operated with the PC to control the program sequence. There are two jump and two call instructions. One uses an index plus displacement addressing; the second uses a relative addressing. The call return instructions use a 3-bit CWP(Current Window Pointer) register which points to the currently active register window.

TABLE 8-12 Instruction Set of Berkeley RISC I

Opcode	Operands	Register Transfer	Description
Data manipulation instructions			
ADD	Rs,S2,Rd	$Rd \leftarrow Rs + S2$	Integer add
ADDC	Rs,S2,Rd	$Rd \leftarrow Rs + S2 +$ carry	Add with carry
SUB	Rs,S2,Rd	$Rd \leftarrow Rs - S2$	Integer subtract
SUBC	Rs,S2,Rd	$Rd \leftarrow Rs - S2 -$ carry	Subtract with carry
SUBR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs$	Subtract reverse
SUBCR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs -$ carry	Subtract with carry
AND	Rs,S2,Rd	$Rd \leftarrow Rs \wedge S2$	AND
OR	Rs,S2,Rd	$Rd \leftarrow Rs \vee S2$	OR
XOR	Rs,S2,Rd	$Rd \leftarrow Rs \oplus S2$	Exclusive-OR
SLL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-left
SRL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right logical
SRA	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right arithmetic
Data transfer instructions			
LDL	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load long
LDSU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short unsigned
LDSS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short signed
LDBU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte unsigned
LDBS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte signed
LDHI	Rd,Y	$Rd \leftarrow Y$	Load immediate high
STL	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store long
STS	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store short
STB	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store byte
GETPSW	Rd	$Rd \leftarrow PSW$	Load status word
PUTPSW	Rd	$PSW \leftarrow Rd$	Set status word
Program control instructions			
JMP	COND, S2(Rs)	$PC \leftarrow Rs + S2$	Conditional jump
JMPR	COND,Y	$PC \leftarrow PC + Y$	Jump relative
CALL	Rd,S2(Rs)	$Rd \leftarrow PC$ $PC \leftarrow Rs + S2$ $CWP \leftarrow CWP - 1$	Call subroutine and change window
CALLR	Rd,Y	$Rd \leftarrow PC$ $PC \leftarrow PC + Y$ $CWP \leftarrow$ CWP $- 1$	Call relative and change window
RET	Rd,S2	$PC \leftarrow Rd + S2$ $CWP - CWP + 1$	Return and change window
CALLINT	Rd	$Rd \leftarrow PC$ $CWP \leftarrow CWP - 1$	Disable interrupts
RETINT	Rd,S2	$PC \leftarrow Rd + S2$ $CWP \leftarrow CWP + 1$	Enable interrupts
GTLPC	Rd	$Rd \leftarrow PC$	Get last PC

MICROPROGRAMMED CONTROL:

CONTROL MEMORY:

The control unit(CU) is functional unit of CPU, that performs sequence of microoperations. The complexity of digital computer is derived from the number of sequences of microoperations that are performed.
The methods for implementing control unit are:

1) Hardwired control.
2) Microprogrammed control.

The design of hardwired control involves the use of fixed instructions, fixed logic blocks of arrays, encoders, decoders etc. In other words, when the control signals are generated by hardwire using conventional logic design techniques, the control unit is said to be hardwired.

The key characteristics of hardwired control logic are high speed operation, expensive, relatively complex and no flexibility of adding new instructions. Intel 8085, Motorola 6802, Zilog 80, and any RISC CPUs are hardwired.

The second alternative for designing the control unit of a digital computer is microprogramming. It is a systematic method for controlling the microoperation sequences in a digital computer. Intel 8080, Motorola 68000, and any CISC CPUs are microprogrammed.

The advantage of microporgrammed control is that , the hardwired configuration need not be changed once it is established. If we want to change control sequence for the system, it requires change of microprogram in the control memory.

A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microoperations constitutes a microprogram.
Since modification of microinstructions are not needed once the control unit is in operation, the control memory can be a read-only memory(ROM). Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing and reading. A memory that is part of control unit is referred to as a control memory.

A computer that employs a microprogrammed control unit will have two separate memories:

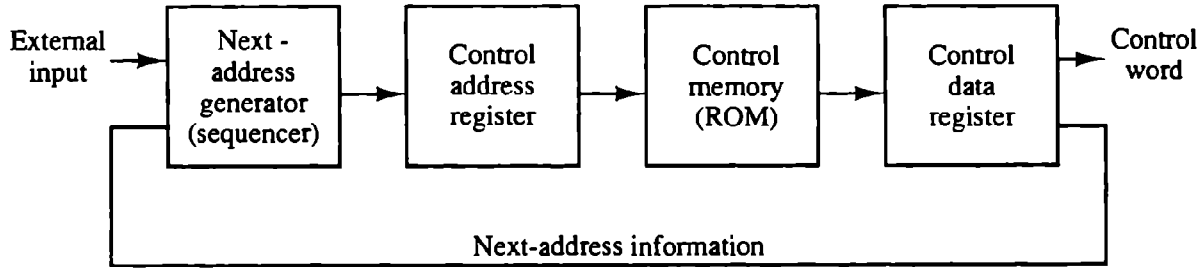
- A main memory
- Control memory

The main memory is available to uses for storing the program. The contents of main memory may be altered. In contrast, the control memory holds a fixed microprogram that cannot be altered by the normal user. The microprogram consists of microinstructions.

The microinstructions generate the microoperatons to fetch the instruction from main memory, evaluate effective address, and execute the operation specified by the instruction and return control to the fetch phase in order to repeat the cycle for the next instruction.

Fig 7-1 shows the block diagram for the general configuration of microprogrammed control Unit.

Figure 7-1 Microprogrammed control organization.



The control memory contains control information which is usually a ROM. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address.

The next address generator determines the address sequence that is read from the control memory, also called a microprogram sequencer. The control data register holds the present microinstruction. The data register also called pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next micro instruction.

This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

ADDRESS SEQUENCING:

Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware that controls the address sequencing of control memory must be capable of:

- Sequencing the microinstructions within the routine.
- Able to branch from one routine to another.

The address sequencing capabilities required in 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.

Fig 7-2 show a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

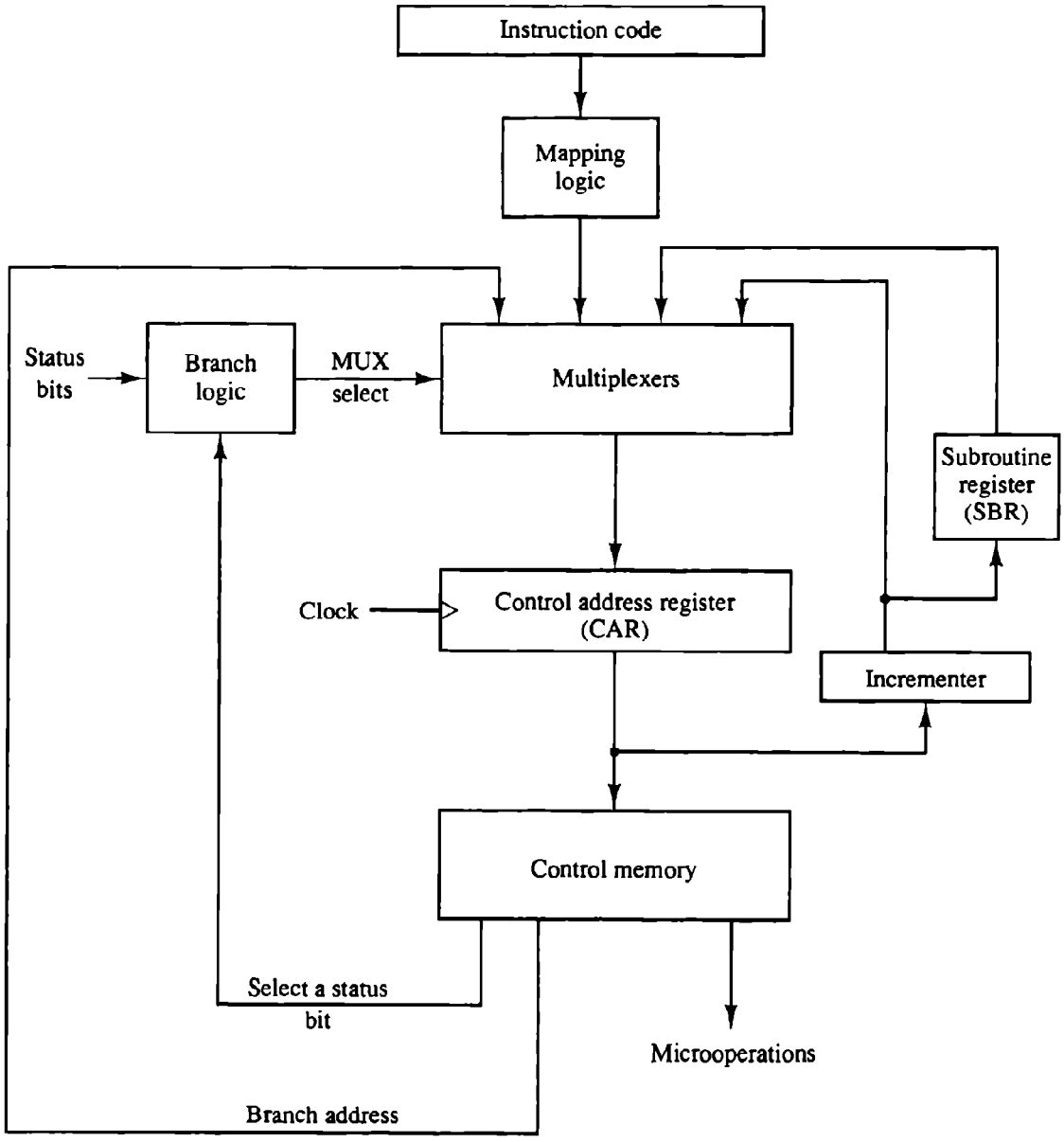


Figure 7-2 Selection of address for control memory.

The diagram shows four different paths from which the control address register(CAR) receives the address. The incrementer increments the content of the CAR by one, to select the next microinstruction in sequence.

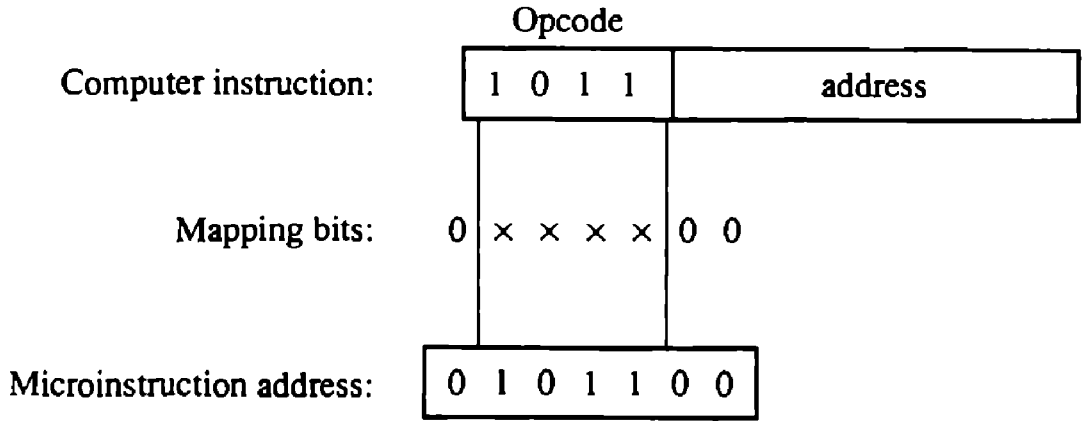
Conditional Branching: The branch logic of fig 7-2 provides decision-making capabilities in the control unit. The status bits provide information such as carry out of adder, the sign bit of a number, the mode bit of an instruction, and input or output status conditions. The status bits, together with the filed in the microinstruction specifies a branch address.

The branch logic hardware may be implemented as follows:
Test the specified condition and branch to the indicated address if the condition is met. This can be implemented by the multiplexer

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

Mapping of instruction: Assume the control memory has 128 words, requiring an address of 7 bits. The mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in the fig 7-3.

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



The mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two lest significant bits of the CAR. This provides for each computer instruction a microprogram routine with a capability of four microinstructions. If the routine needs more than four microinstructions, it can se addresses 1000000 through 1111111.

Another way of mapping instruction code to microinstruction address is by using a mapping function in the ROM. The mapping function can be implemented Programmable logic device (PLD).

Subroutines: Subroutines are programs that are called by main program to accomplish a particular task. For example, we can define a subroutine that consists of sequence of microoperations to compute the effective address (EA) of the operand for all memory reference instructions.
Microprograms that use subroutines must have a provision for storing the return address during the subroutine call and restoring the address during a subroutine return. The subroutine register can be used for this purpose. The subroutine register can be contains address for subroutines and is implemented as LIFO stack.

MICROPROGRAM EXAMPLE:

We present here a simple digital computer and show how it is programmed.

Computer Configuration:

The block diagram of the computer is shown in fig 7-4.

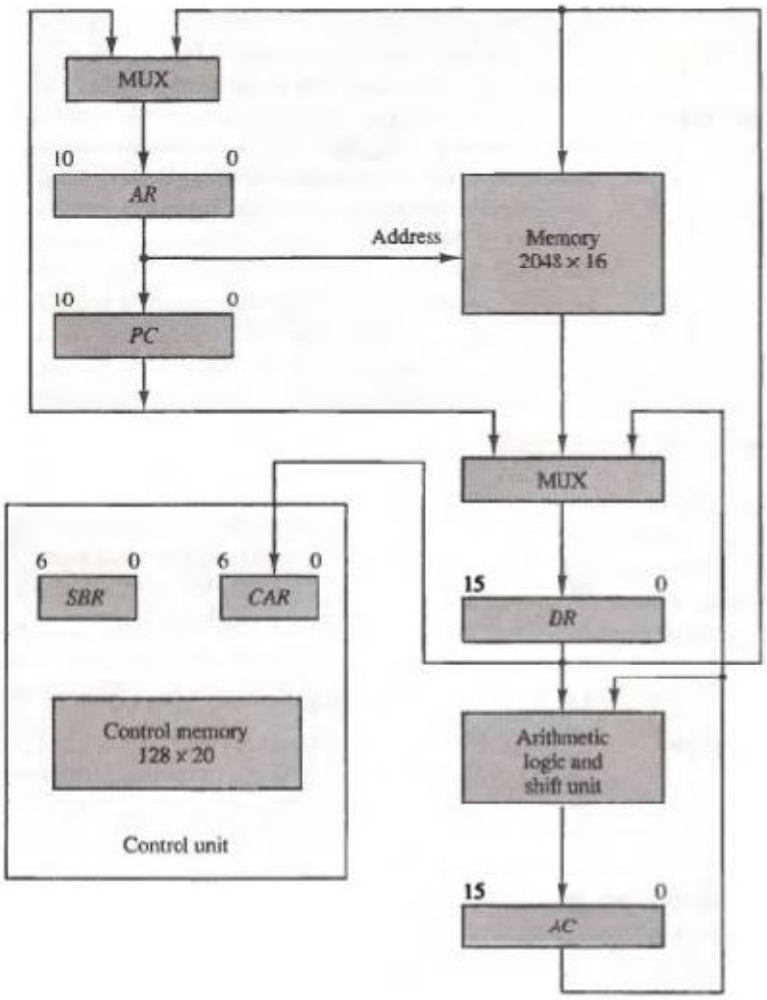


Figure 7-4 Computer hardware configuration.

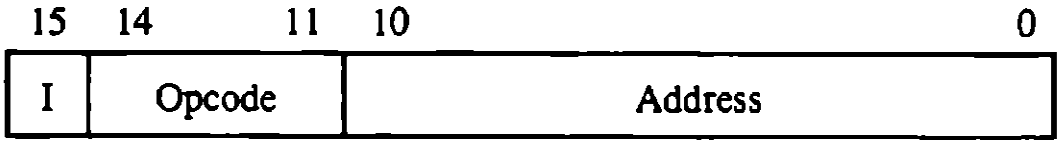
It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are PC,AR,DR and AC. The control unit has control address register(CAR) and a subroutine resister (SBR). The control memory and its registers are organized as a microprogrammed control unit, as shown fig 7-2.

The transfer of information among the registers in the processor is done through multiplexers. DR can receive information from AC,PC or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic,logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.

The computer instruction format is shown in fig 7-5(a). It consists of 3 fields: a 1- bit field for indirect addressing symbolized by I, a 4-bit operation code(opcode), and an 11-bit address field. Fig 7-5(b) lists four of the 16 possible memory-reference instructions. The ADD

instruction adds the content of the content of the operand found in the EA to the content of AC. The BRANCH instruction causes a branch to the EA if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit is a 1. The store instruction transfers the content of AC into memory word specified by the EA. The EXCHANGE instruction swaps the data between AC and the memory word specified by the EA.

Figure 7-5 Computer instructions.



(a) Instruction format

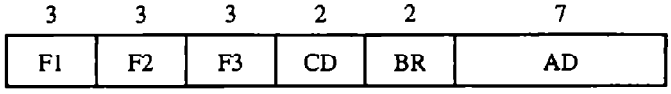
Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M\,[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M\,[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

Micro instruction format:

The micro instruction format is shown in fig 7-6



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

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

The 20 bits of the microinstrucition 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 specify the type or branch to be used. The AD field contains a branch address. The address filed is seven bits wide, since the control memory has $128 = 2^7$ words.

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations are listed in table 7-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction. One from each field. If fewer than three microoperations are used, one or more of the field will use the binary code 000 for no operation. For example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

$$DR \leftarrow M[AR] \qquad \text{with F2 = 100}$$

$$\text{and} \quad PC \leftarrow PC + 1 \qquad \text{with F3 = 101}$$

The nine bits of the microoperation fields will then be 000 100 101.

From the table 7-1, all transfer-type microoperations symbols use file letters. The first two letters designated the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC← DR(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 as listed in table 7-1. The first condition is always a 1, so that a reference to CD = 00(or symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR(branch) filed, it provides an unconditional branch operation. The indirect 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 value 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 four status bits when we write the microprograms in symbolic terms.

The BR(branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in table 7-1, when BR = 00, the control performs a jump(JMP) operation, 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 filed. It the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

The return from subroutine is accomplished with a BR filed equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. This mapping is as shown in fig 7-3. The bits of the operation code are in DR(11-14) after an instruction is read from memory.

Symbolic Microinstructions:

The symbols defined in table 7-1 can be used to specify symbolic microinstructions. A symbolic microinstruction can be translated into its binary equivalent by means of an assembler.

Each symbolic microinstruction is divided into five fields:

label, microoperations, CD, B R, and AD.

1. The label field specify a symbolic address. A label is terminated with a colon(:)
2. The microoperation field consists of one,two, or three microoperations.
3. The CD has one of the letters U ,I,S or Z.
4. The BR filed contains one of the four symbols defined in Table 7-1.
5. The AD field specifies a value for the address field of the microinstruction.

The Fetch Routine:

The micro instructions needed for the fetch routine are:

$$AR \leftarrow PC$$

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

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

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64,65, and 66. ***The symbolic microprogram*** for the above fetch routine as follows:

```
          ORG 64
FETCH:    PCTAR          U    JMP    NEXT
          READ, INCPC    U    JMP    NEXT
          DRTAR          U    MAP
```

Table 7-2 shows the symbolic microprogram for the fetch routine and the microinstruction routines that execute four computer instructions:

The translation of the symbolic microprogram to binary produces the ***binary microprogram***. Following table shows a binary microprogram. Following table shows a microprogram for above fetch symbolic program in 3 different ways. The bit values are obtained from table 7-1.

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

TABLE 7-2 Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

TABLE 7-1 Symbols and Binary Code for Microinstruction Fields

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

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

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

DESIGN OF CONTROL UNIT:

The microinstruction format consists of various fields, with each field performs a separate function. The fields contain control bits to initiate the microoperations, special bits to evaluate the next address and address field for branching.

Each filed requires a decoder to produce the corresponding control signals. Fog 7-7 shows the three decoders and connections.

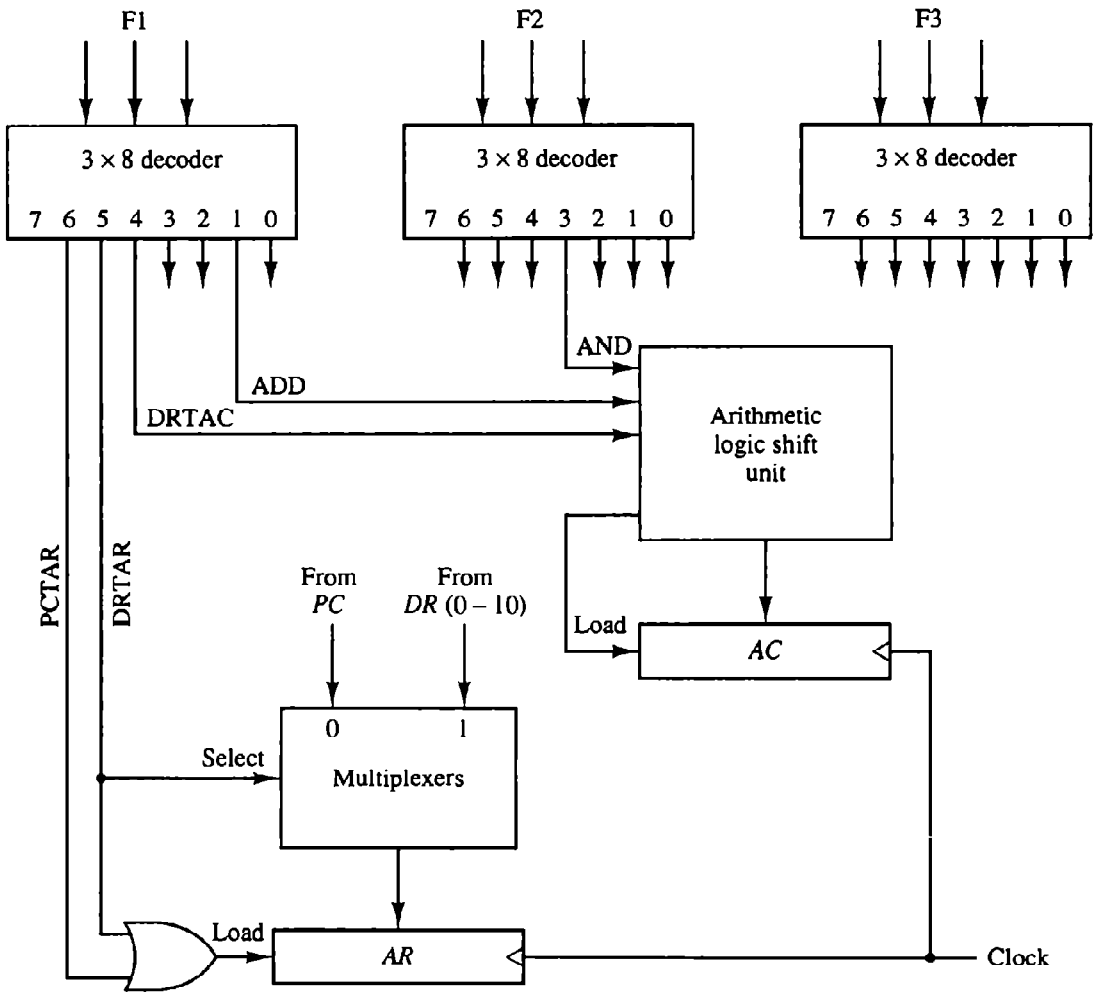


Figure 7-7 Decoding of microoperation fields.

In the above fig, each of the three fields of the microinstruction are decoded with a 3x8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuits to initiate the corresponding microoperation as specified in table 7-1.

For example, when F1=101(binary 5), the next clock pulse transition transfers the content of DR(0-10) to AR. Similarly, when F1=110 there is as transfer from PC to AR.

The control signals to ALU are from output of decoders and are marked as AND,ADD and DRTAC respectively. The other outputs of decoders are associated with an AC operation.

Microprogram Sequencer: The basic components of a microprogrammed control unit are the control memory and microprogram sequencer.

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 block diagram of the microprogram sequencer is shown in fig 7-8.

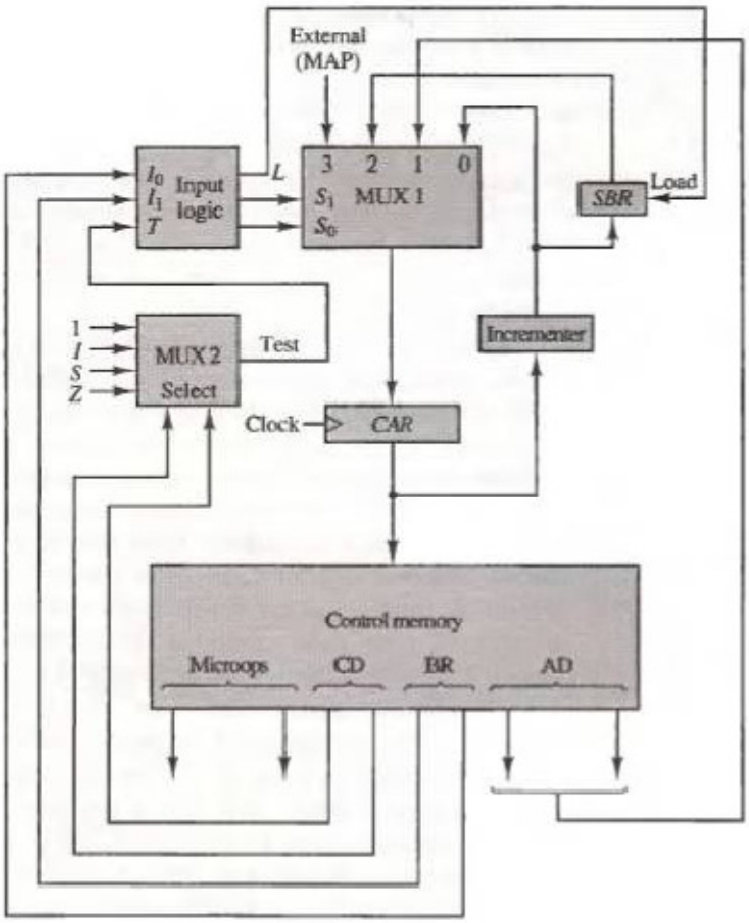


Figure 7-8 Microprogram sequencer for a control memory.

In the above diagram, there are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes into a control address register CAR. The second multiplexer tests the value of 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.

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 tow bits from the BR(branch) filed go to an input

logic circuit. The input logic in a particular sequencer will determince the type of the 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

With three inputs, the sequencer can provide upto eight address sequencing operations.

The input logic circuit has three inputs I_0,I_1, and T, and three outputs, S_0,S_1 and L. Varuables S0 and S1 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, S1S0=10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.

The truth table for the input logic circuit is show in table 7-4. Inputs I1 and I0 are identical to the bit values in the BR field. The function listed in each entry was defined in table 7-1. The bit values for S1 and S0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.

TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	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 subroutine register is loaded with the incremented values of CAR during a call microinstruction(BR=01) provider that the status bit condition is satisfied(T=1). The truth table can be use to obtain the simplified Boolean functions for the input logic circuit:

$$S_1 = I_1$$
$$S_0 = I_1 I_0 + I_1' T$$
$$L = I_1' I_0 T$$

UNIT-IV

COMPUTER ARITHMETIC

Syllabus: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

Arithmetic instructions manipulate data to produce necessary results. The four basic arithmetic operations are addition, subtraction, multiplication, and division.

ADDITION AND SUBTRACTION:

There are three way of representing negative fixed-point binary numbers:

- Signed-magnitude
- Signed 1's complement
- Signed 2's complement

Most computers use the signed 2's complement representation when performing arithmetic operations with integers. For floating-point representation, most computers use signed-magnitude representation for the mantissa.

Addition and Subtraction with Signed-Magnitude Data:

We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that, there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions listed in the first column of table 10-1.

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

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

The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction can be stated as follows:

Addition (subtraction) algorithm: When the signs of A and B are identical, add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of

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

Hardware Implementation:

To implement the two addition and subtraction with hardware, two number be stored in registers A and B. Ai and Bi are flip-flops that hold the signs of A and B. The result of the operation may be transferred to a third register called accumulator(AC).

The procedure requires a magnitude comparator for comparing A > B, A = B, OR A < B, a parallel-adder for microoperation A + B, and two parallel subtractors for the microoperations A- B and B - A.

Fig 10-1 shows a block diagram of the hardware for implementing the addition and subtraction operations.

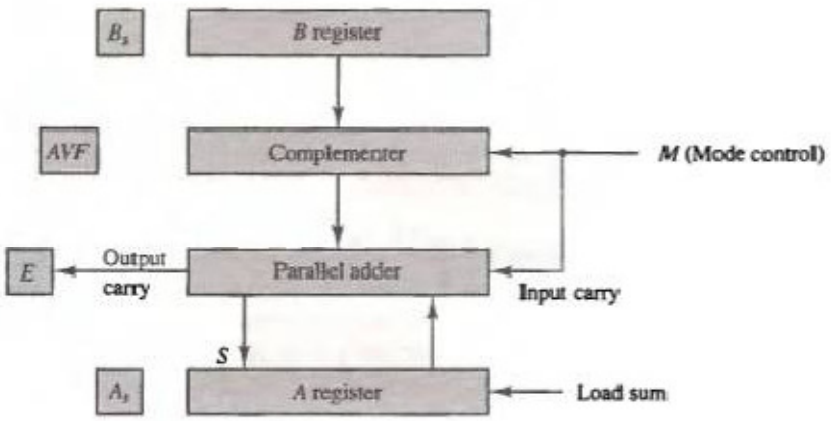


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

It consists of register A and B and sign flip-flops Ai, Bi. Subtraction is done by adding A to the 2's complement of B. The output carry is transdferred to flip-flop E. E can be used to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF hold the overflow bit when A and B are added.

The addition of A plus B done through the parallel adder. The S(sum) output of the adder is applied to the input of the A register. The complementer provides complement of B depending the state of the mode control M. The complementer consists of X-OR gates and the parallel adder consist of full-adder circuits.

When M = 0, S = A + B is performed.
When M = 0, S = A + B +1 is done. This is nothing but the subtraction A = B.

Hardware Algorithm:

The flowchart for the hardware algorithm is presented in fig 10-2.

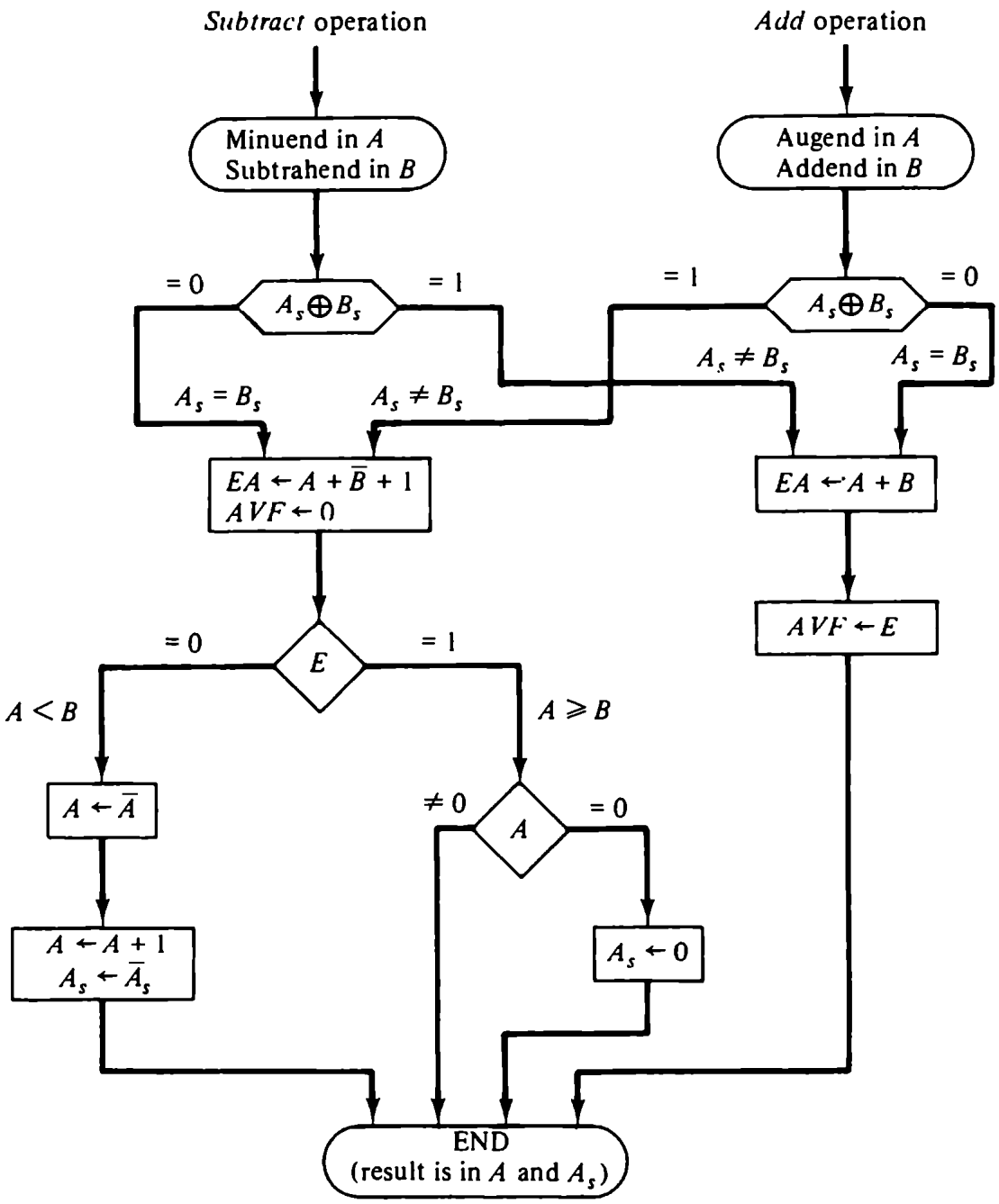


Figure 10-2 Flowchart for add and subtract operations.

The two signs Ai and Bi are compared by an X-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an add operation, identical signs dictate the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a microoperation

E A← A + B

.The 1 bit in E signifies the overflow. The value of E is transferred into the add-overflow flip-flop AVF.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted. So, AVF is cleared to 0. A 1 in the E indicates when A >=B and the number in A is the correct result. A 0 in E indicates A < B. For this case it is necessary to take the 2's complement of the value sin A. This operation can be done with one microoperation A ← A' + 1 .

In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in Ai is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement Ai to obtain the correct sign. The final result is found in register A and its sign in Ai.

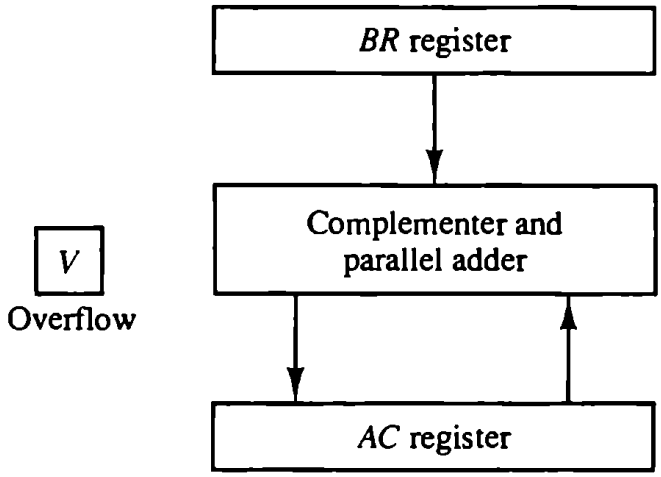
Addition and Subtraction with Signed-2's Complement Data:

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

When two numbers of n digits are added and the sum occupies n + 1digits, we say that an overflow occurred. An overflow can be detected in addition, by applying the last two carries to an X-OR gate. When the X-OR gate output is 1 then an overflow is detected.

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

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



The algorithm for adding and subtraction two binary number in signed 2's complement representation is show in the fig 10-4.

The sum is obtained by adding AC and BR(including their sign bits). The overflow bit V is set to 1 if the X-OR of the last two carries is 1. And it is cleared to 0 otherwise.

The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR. An overflow must be checked during this operation because the two numbers added could have the same sign.

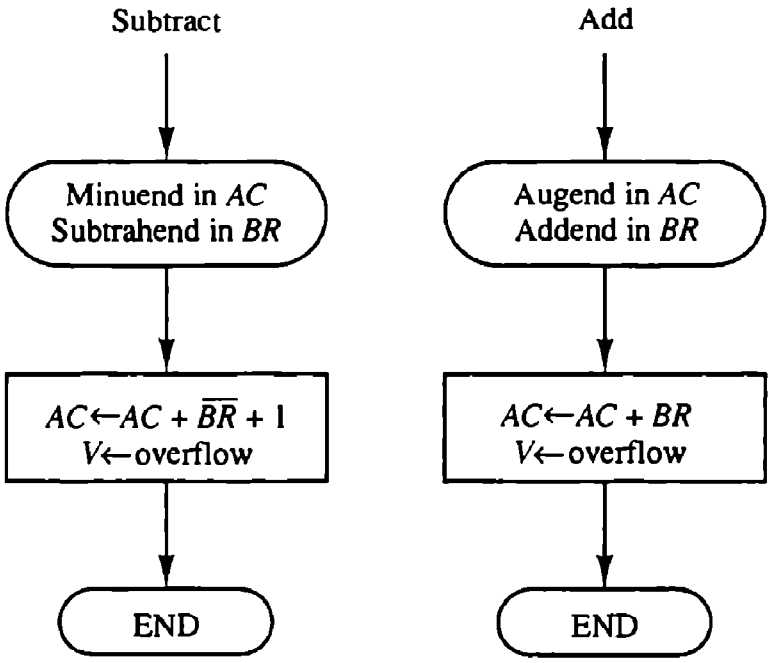


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

MULTIPLICATION ALGORITHMS:

Multiplication of two fixed-point binary numbers is signed-magnitude representation is done by a process of successive shift and add operations. This process can be explained by the flowing numerical example.

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

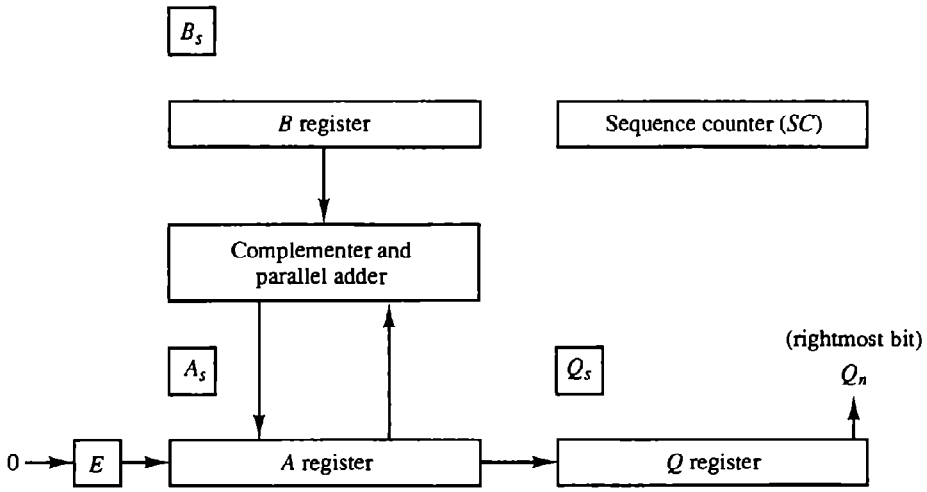
The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, then multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number.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

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

Hardware Implementation for Signed-Magnitude Data:

The hardware for multiplication consists of the equipment shown in fig 10-1 plus two more registers SC and Q register. There are shown in fig 10-5.

Figure 10-5 Hardware for multiply operation.



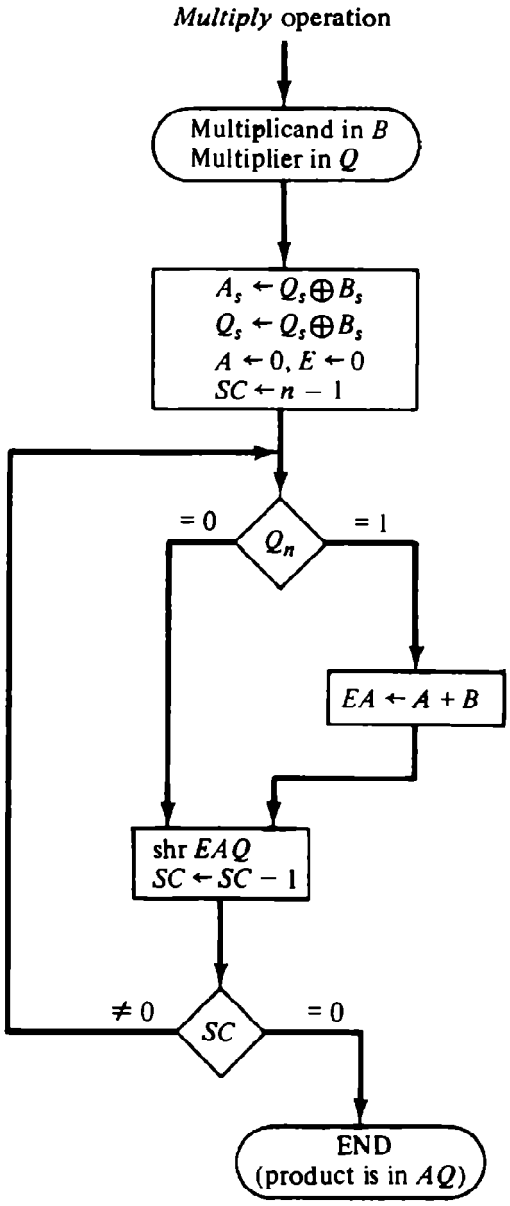
The multiplier is stored in the Q register and its sign Q_s. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to EA register. Both partial product and multiplier are shifter to the right. This shift will be denoted by the statement SHR EAQ. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the parallel product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Qs, will hold the bit of the multiplier.

Hardware Algorithm:

Figure 10-6 is a flowchart of the multiplication algorithm. Initially, the multiplicand is in B and the multiplier is in Q. Their corresponding signs are in Bs an Qs. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the SC is set to a number equal to the number of bits(only magnitude bits not the sign bit. Hence it is n-1) of the multiplier.

Figure 10-6 Flowchart for multiply operation.



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

The table 10-2 shows the hardware multiplication process that is explained in the above flow chart.

TABLE 10-2 Numerical Example for Binary Multiplier

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

Booth Multiplication Algorithm:

Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation. It operates on the fact that the string of 0's in the multiplier require no addition but just shifting, and the string of 1's in the multiplier from bit weight 2^k to weight 2m can be treated as 2^{k+1}- 2^m. For example, the binary number 001110(+14) has string of 1's from 2^3 to 2^1(k=3, m =1). The number can be represented as $^{2k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$.

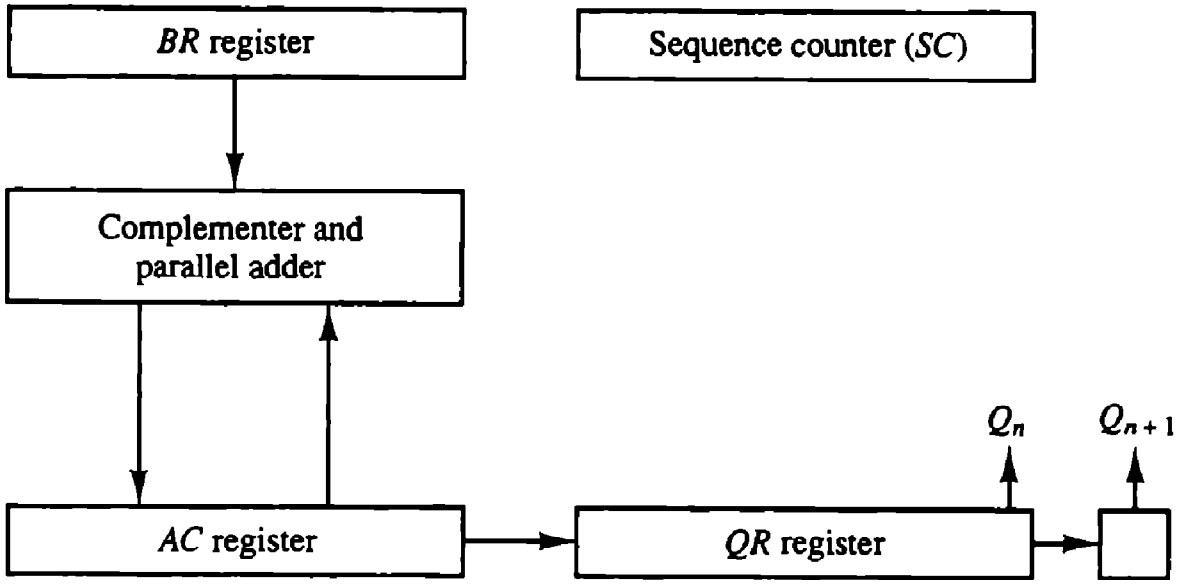
Therefore, the multiplication M x 14, where is M is the multiplicand and 14 the multiplier, can be done as M x 2^4 – M x 2^1. Thus the product can be obtained by shifting the binary multiplicand M four tines to the left and subtraction M shifted left once.

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

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

The hardware implementation of Booth algorithms require the register configuration shown in fig 10-7. This is similar to fig 10-5 except that the sign bits are not separated from the rest of the registers. Qn designates the least significant bit of the multiplier in register QR to facilitate a double bit inspection of the multiplier. The flowchart for both algorithm is shown in fig 10-8.

Figure 10-7 Hardware for Booth algorithm.



The AC and appended bit Qn+1 are initially cleared to 0 and the SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1 are inspected. If the two bits are equal to 10, it means that the first 1 in the string of 1's have been encountered. This requires a subtraction of the multiplicand from the partial product of AC. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right(ashr) operation which shifts AC and QR to the right and leave the sign bit in AC unchanged. The SC is decremented and the computational loop is replaced n times.

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

TABLE 10-3 Example of Multiplication with Booth Algorithm

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

DIVISION ALGORITHMS:

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift and subtract operations. The division process is illustrated by a numerical example in fig 10-11.

Figure 10-11 Example of binary division.

Divisor:
$B = 10001$

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

The divisor B consists of 5 bits and the dividend A of 10 bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we

try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The difference is called the partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data:

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

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

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

Division overflow:

The division operation may result in a quotient with an overflow. The condition for overflow can be stated as follows:

A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor. Another problem associated with division is that a division by zero. The divide-overflow condition takes care of this condition as well. Overflow condition is usually detected when a divide-overflow (DVF) flip-flop set. The best way to handle the divide overflow is by using the floating point representation.

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

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

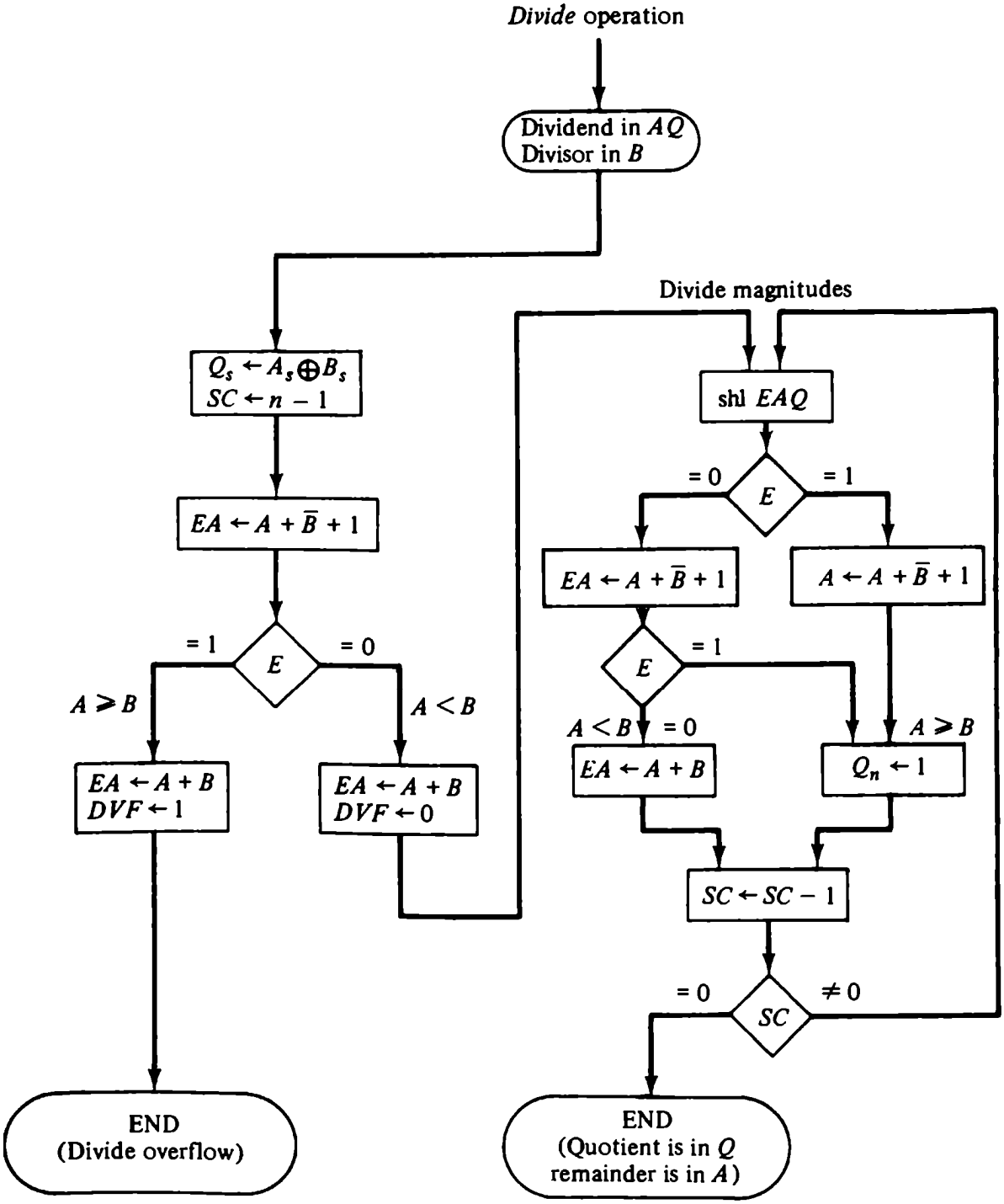
Figure 10-12 Example of binary division with digital hardware.

Hardware Algorithm:

The hardware divide algorithm is shown in the flowchart of fig 10-13. The dividend is in A and Q and the divisor is in B. The sign of the result is transferred into Qs to be part of the

quotient. A constant is set into the SC to specify the number of bits in the quotient.

Figure 10-13 Flowchart for divide operation.



One bit of the word will be occupied by the sign and the magnitude will consist of n – 1 bits.

A divide overflow condition is tested by subtraction the divisor in B from half of the bits of the dividend stored in A. If A >= B, the divide-overflow flip-flop DVF is set and the

operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

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

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

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

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

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

FLOATING-POINT ARITHMETIC OPERATIONS:

The computer must have provision for floating point arithmetic operations. These can be implemented by using hardware or by using compiler.

Basic considerations: Floating-point number in computer register consists of two parts: a mantissa m and an exponent e. The two pats represent a number obtained from multiplying m times a radix r raised to the value of e, thus:

$$m \times r^e$$

Arithmetic operations with floating-pint numbers are more complicated than with fixed-point numbers and their execution takes longer and requires more complex hardware.

Adding and subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

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

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissa can be added.

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

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the fight and incrementing the exponent.

When two numbers are subtracted, the result may contain the most significant zeros as shown in the following example:

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

A floating-point number that has a 0 in the MSB of the mantissa is said to have an underflow. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain .35000 X 10^3.

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

The mantissa can be represented same as with the fixed-point numbers. The exponent may be represented in any one of the following representations:

1. Signed-magnitude.
2. Signed-1'scomplement.
3. Signed-2's complement.
4. Biased exponent.

In biased representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive.

For example, consider an exponent that ranges from -50 to 49. Internally, it is represented by tw0 digits(without a sign) by adding to it a bias of 50. The exponent register contains the

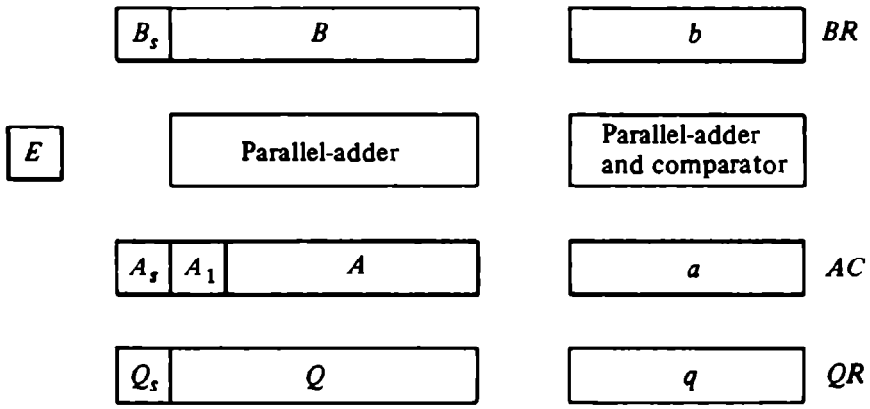
number e + 50. Where e is the actual exponent. The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compared these relative magnitude without being concerned with their signs.

Register configuration:

The register configuration for floating-point operations is quite similar to the fixed-point operations. The same registers and adder used for fixed-point arithmetic are used for processing the mantissa. The difference lies in the way the exponents are handled.

The register configuration for floating-point operations is shown in fig 10-14.

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



Addition and Subtraction:

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

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

The flowchart for adding or subtracting two floating-ping binary numbers is shown in fig 10-15.

If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissa.

The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitudes. If the two exponents are equal, we go to perform the arithmetic

operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted right and its exponent is incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed point addition and subtraction algorithm presented in fig 10-12. The magnitude part is added or subtracted depending on the operation and the sign of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitude were subtracted, one result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number is the AC is made zero. Otherwise, the mantissa must have at least one bit that is equal to1. The mantissa has an underflow if the most significant bit in position A1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until it is equal to 1. When A1 = 1, the mantissa is normalized and the operation is completed.

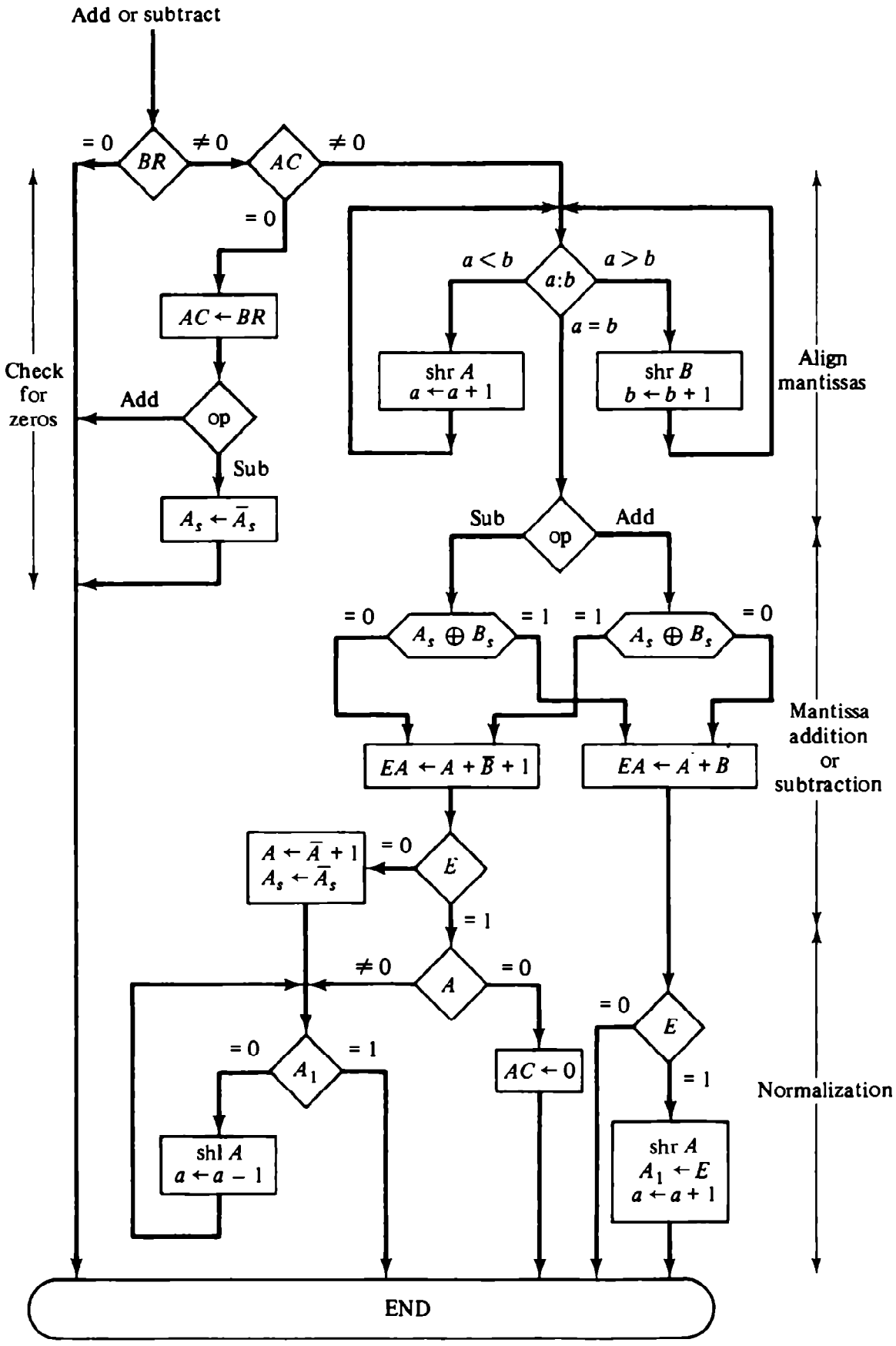


Figure 10-15 Addition and subtraction of floating-point numbers.

Multiplication:

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

The multiplication algorithm can be divided into four parts:

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

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

The flowchart for floating-point multiplication is shown in fig 10-16.

The two operands for floating-point multiplication is shown in Fig 10-16. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

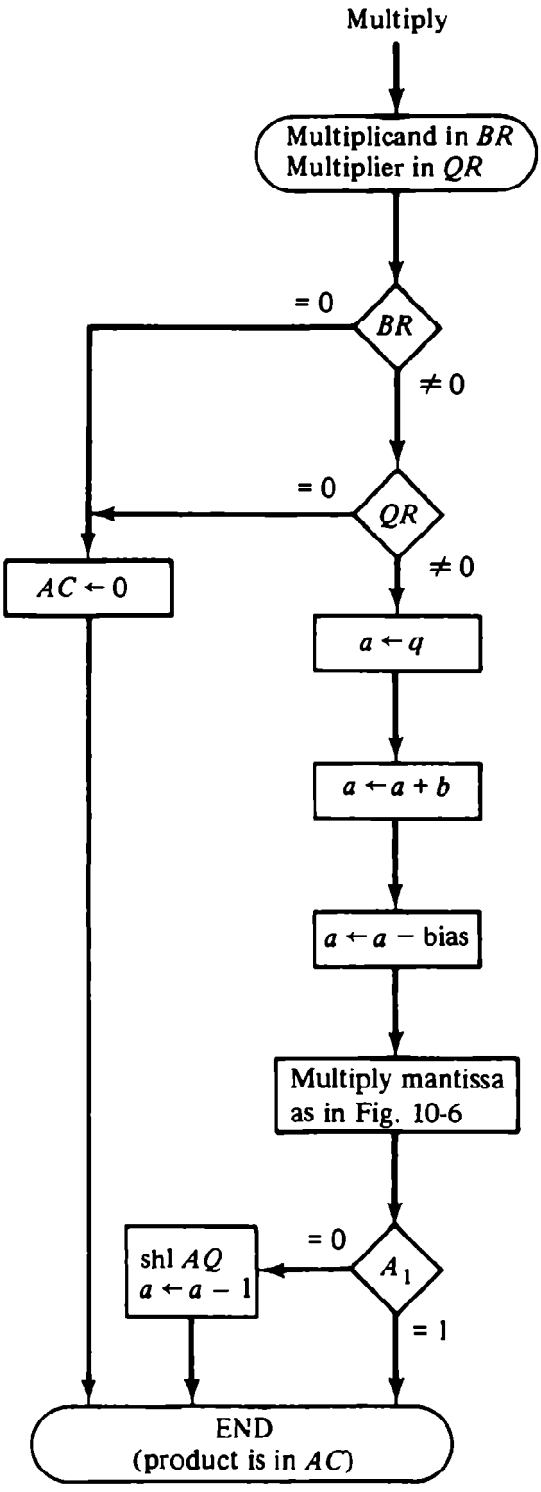
The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The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

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

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

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

Figure 10-16 Multiplication of floating-point numbers.



Division:

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

The check for divide-flow is the same as in fixed-point representation. However, with floating-pint numbers the divide-overflow imposes no problems. If the dividend is greater than equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. This operation is referred as a dividend alignment.

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

The division algorithm can be subdivided into five parts.

1. Check for zeros.
2. Initialize register and evaluated the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

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

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

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

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

The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lines (n-1) positions to the left of A1. The remainder can be converted to a normalized fraction by subtracting n – 1 from the dividend exponent and by shift and decrement until the bit in A1 is equal to 1. This is not shown in the flow chart.

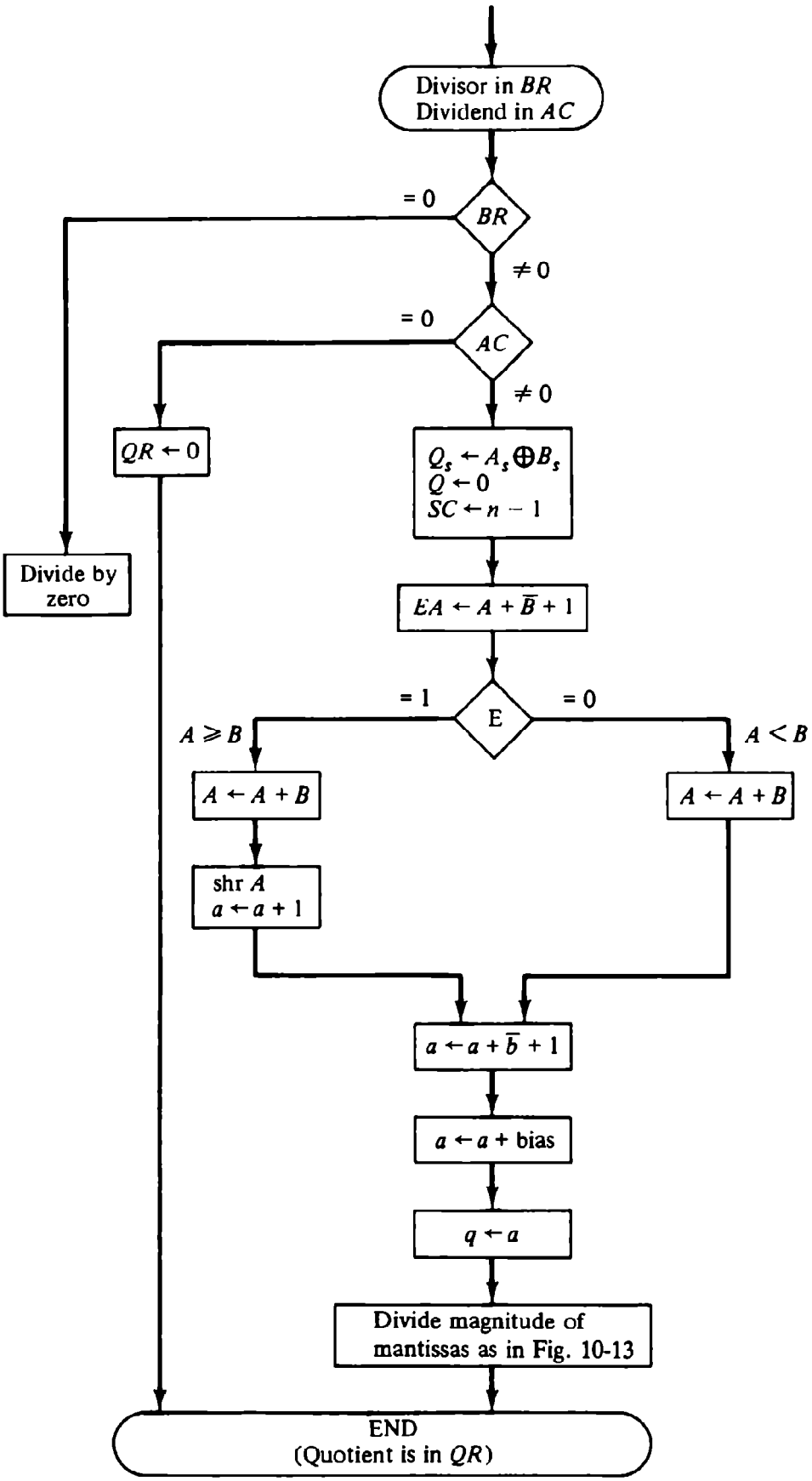


Figure 10-17 Division of floating-point numbers.

DECIMAL ARITHMETIC UNIT:

A CPU performs arithmetic operations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. When application require large amount of input and relatively small no. of arithmetic calculations, it is convenient to do the internal arithmetic directly with the decimal numbers.

Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations.

Electronic calculators use internal decimal arithmetic unit since inputs and outputs are frequent. Many computers have hardware for arithmetic calculations with both binary and decimal data.

A decimal arithmetic units is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's and 10's complement of the subtrahend.

BCD Adder:

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

In examining the contents of the table, it is clear that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6(0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

TABLE 10-4 Derivation of BCD Adder

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

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

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

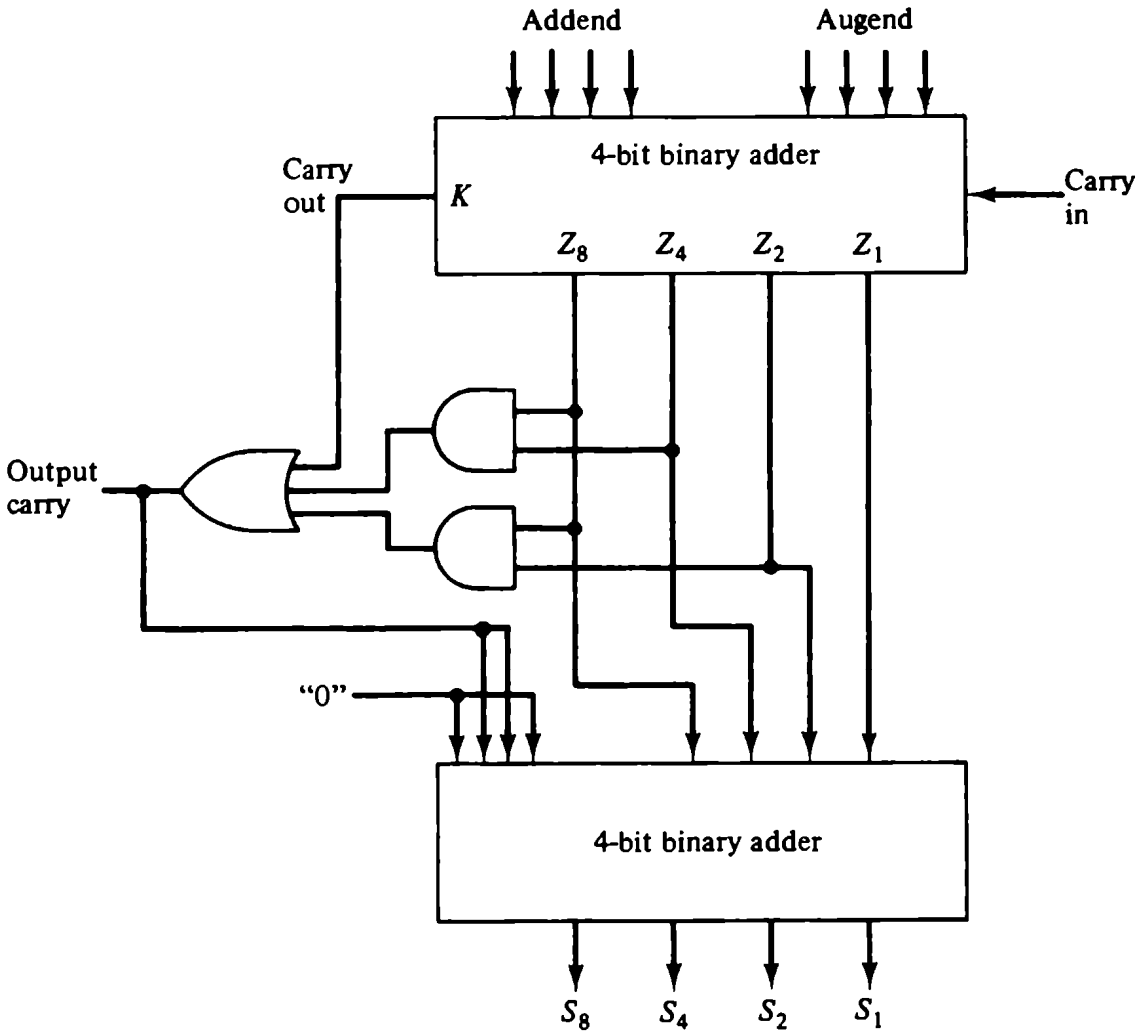
further that either Z4 or Z2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function:

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

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

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

Figure 10-18 Block diagram of BCD adder.



A decimal parallel-adder that adds n decimal digits needs an BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher order stage.

BCD Subtraction:

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It can be done by taking the 9's complement of the subtrahend and adding it to the minuend. To obtain the 9's complement a circuit is necessary to subtract each BCD digit from 9.

The above require a correction. There are two possible correction methods. In the first method, binary 1010(decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110(decimal 6) is added before the digit is complemented.

For example, the 9's complement of BCD 0111(decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010(decimal 2). By second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010.

The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend(or addend) digit be denoted by the four binary variables B_8,B_4,B_2, and B_1. Let M be a mode bit that controls the add/subtract operation. When M = 0, the two digits are added; when M=1, the digits are subtracted. Let the binary variables x_9,x_4,x_2, and x_1 be the outputs of the 9's complementer circuit. By an examination of the truth table for the circuit, it may be observed that B1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x4 is 1 when the X-OR of B_2 and B_4 is 1; and x8 is 1 when $B_8B_4B_2$=000. The Boolean functions for the 9's complement circuit are:

$$x_1 = B_1 M' + B_1' M$$

$$x_2 = B_2$$

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

$$x_8 = B_8 M' + B_8' B_4' B_2' M$$

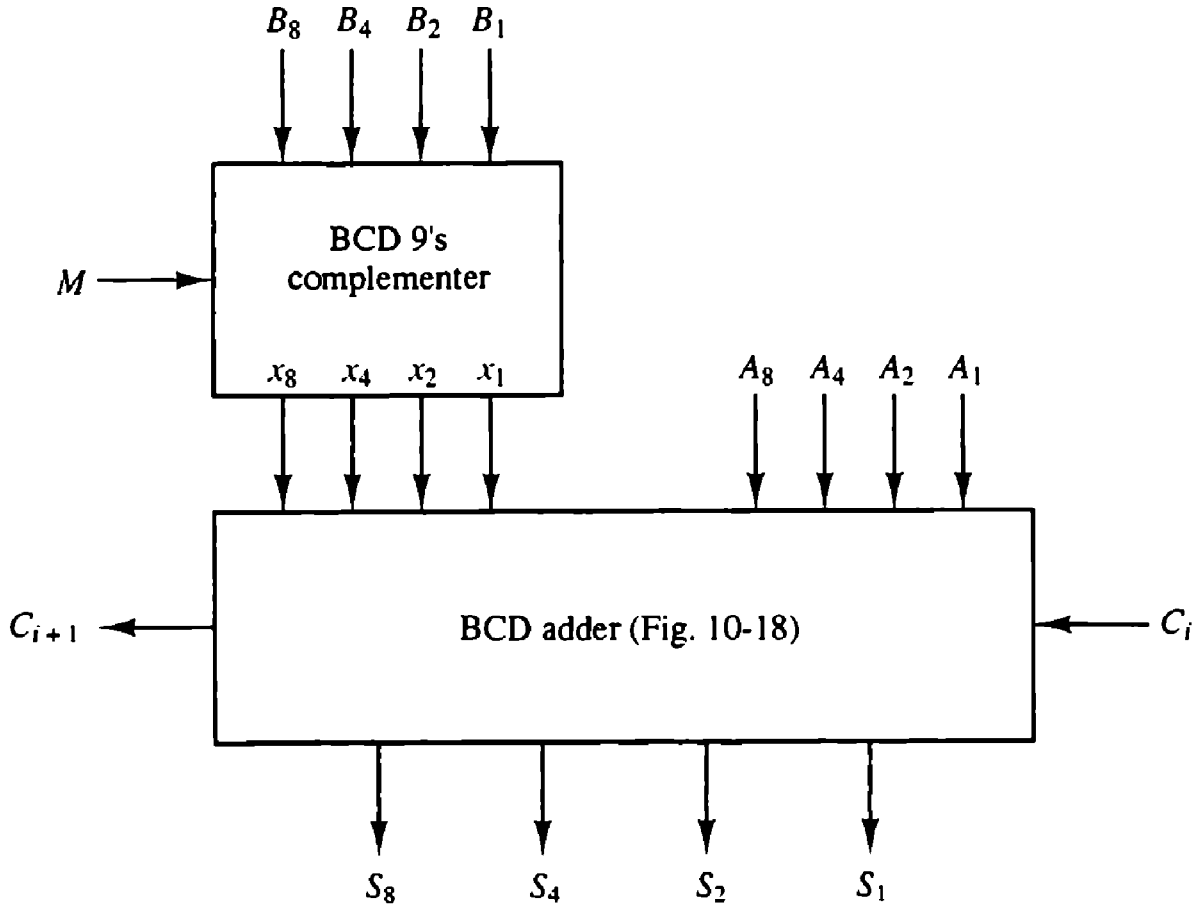
From these equations we see that x = B when M = 0. When M = 1, the x outputs produce the 9's complement of B.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in 10-19.

It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With M = 0, the S outputs form the sum of A and B . With M = 1, the S outputs from the

sum of A plus the 9's complement of B. For numbers with decimal digits we need n such stages. The output carrie C_{i+1} from one stage must be connected to the input carry of C_i of the next-higher-order stage. The best way to subtract the two decimal number sis to let M = 1 and apply a 1 to the input carry of Ci of the first stage. The outputs will form the sum of a plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out the last stage is discarded.

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



DECIMAL ARITHMETIC OPERATIONS:

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

Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit.

For convenience, we will use same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in table 10-5, a bar over

the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement.

TABLE 10-5 Decimal Arithmetic Microoperation Symbols

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

Thus for the decimal numbers, the symbol $A \leftarrow A + B + 1$ denotes transfer of the decimal sum formed by adding the original content A to the 10's complement of B. Incrementing or decrementing a register is same as binary and decimal except for the number of states that the register is allowed to have. A binary counter goes though 16 states, from 0000 to 1111, when incremented. A decimal counter goes though 0000 to 1001 and back to 0000. Similarly, when decremented the decimal counter goes from 1001 to 0000.

A decimal shift right or left is preceded by the letter d to indicate a decimal shift. For example, consider register A holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

0111 1000 0110 0000

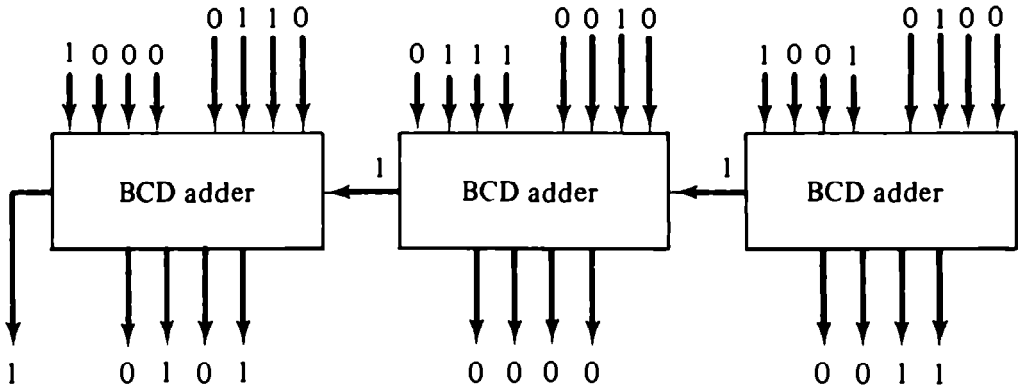
The microoperation *dshr A* shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

0000 0111 1000 0110

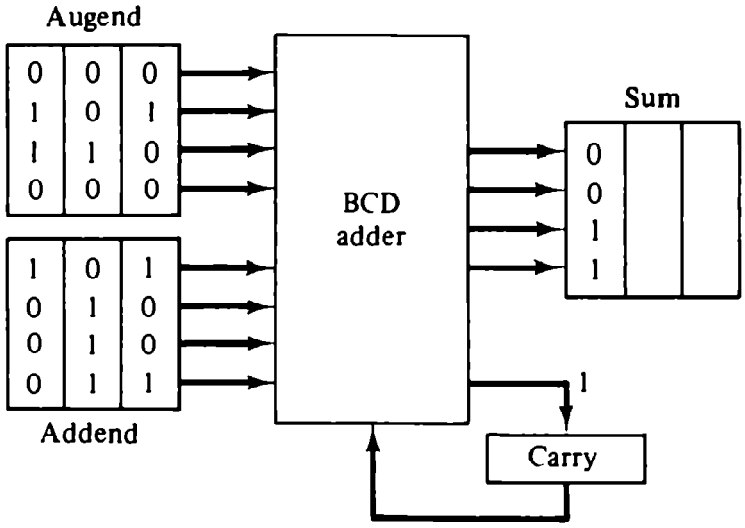
Addition and Subtraction:

The algorithm for addition and subtraction for decimal signed-magnitude numbers is similar as the algorithms for addition and subtraction of binary signed-magnitude numbers. Similarly, the algorithm for binary signed-2'sc complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder or a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9's complement of the subtrahend as shown in fig 10-19.

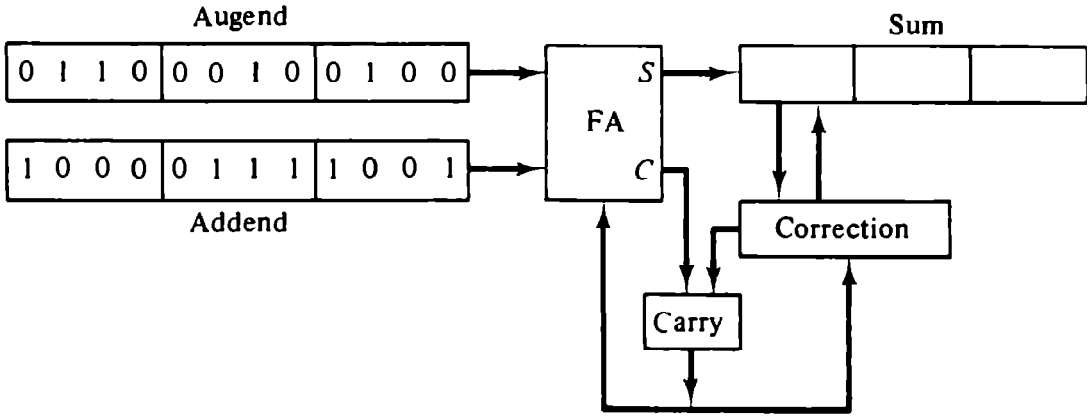
Decimal data can be added in three different ways as shown in fig 10-20.



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



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



(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.

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

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

Multiplication:

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

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

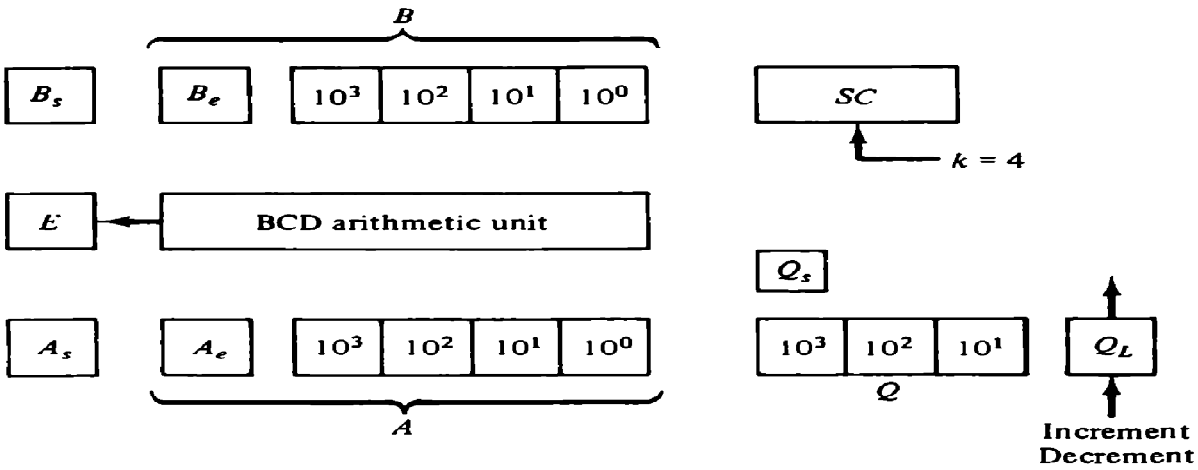


Figure 10-21 Registers for decimal arithmetic multiplication and division.

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

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

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

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

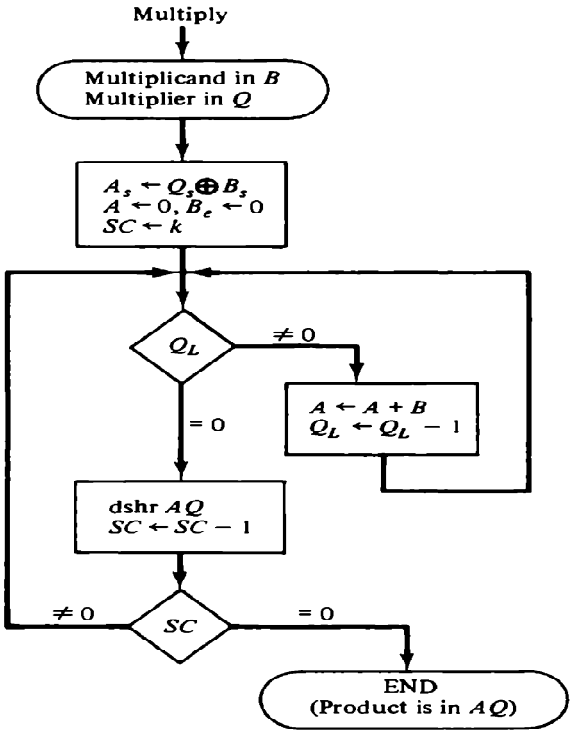


Figure 10-22 Flowchart for decimal multiplication.

Division:

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

The decimal division algorithm is shown in fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend(or partial remainder) is shifted to the left, with its most significant digit placed in Ae. The divisor is then subtracted by adding its 10's complement values. Since, Be is initially cleared, its complement value is 9 as required. The carry in E determines relative magnitude of A and B. If E = 0, it signifies that A < B. In this case the divisor is added to restore the partial remainder and QL stays at 0(inserted there during the shift). If E = 1, it signifies that A >= B. The quotient digit in Q_L is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0. When this occurs, the quotient digit is not incremented but the divisor is added to resotre the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder goes into the divisor.

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

Floating-Point operations:

Decimal floating-point arithmetic operations follow the same procedures as binary operations. The algorithms for floating-point binary data can be adopted for decimal data. The multiplication and division of the mantissas must be done by the methods described above.

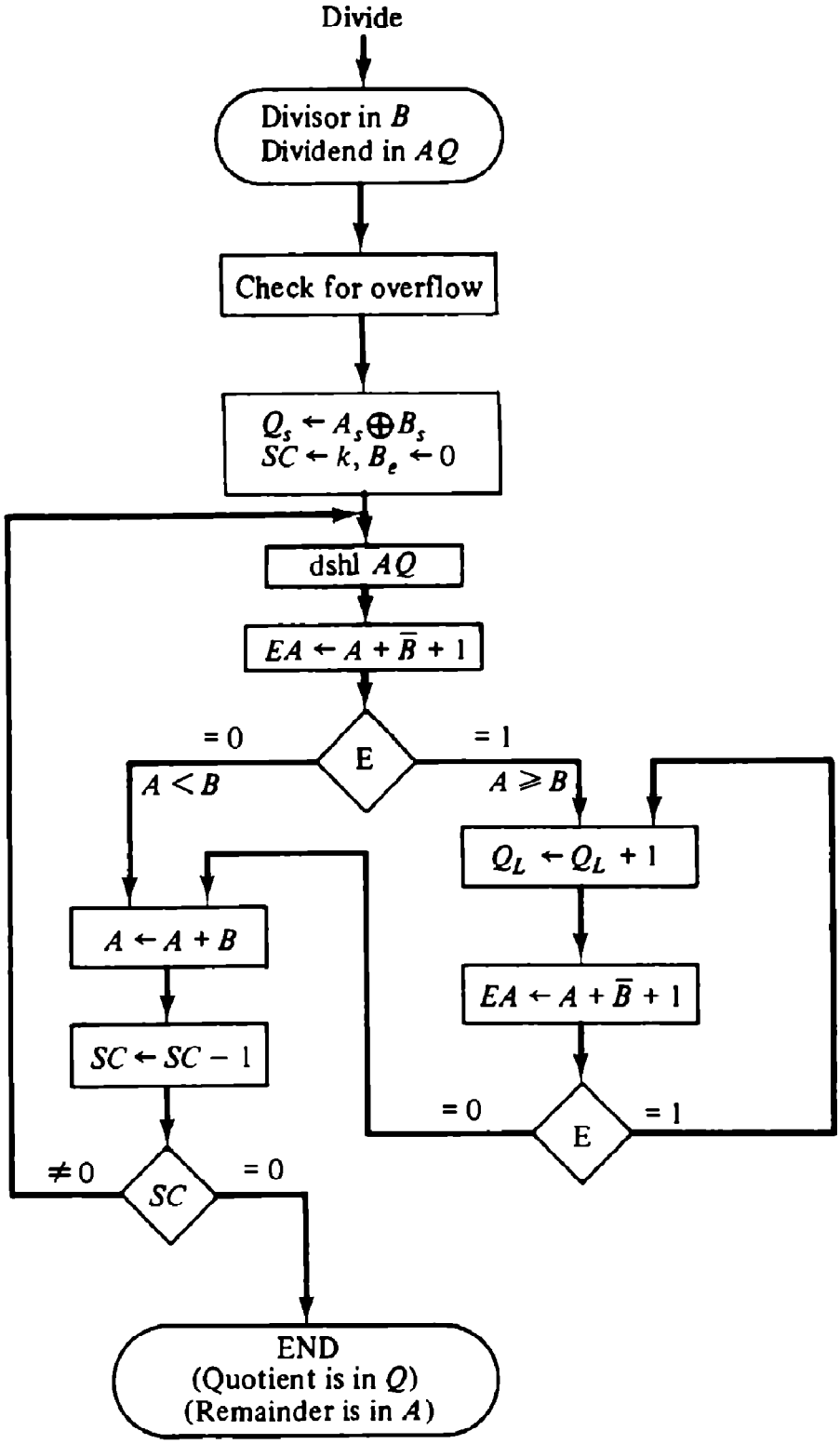


Figure 10-23 Flowchart for decimal division.

THE MEMORY SYSTEM

Syllabus:
Memory Hierarchy, Main memory, Auxiliary memory, Associative Memory, Cache Memory, Virtual Memory.

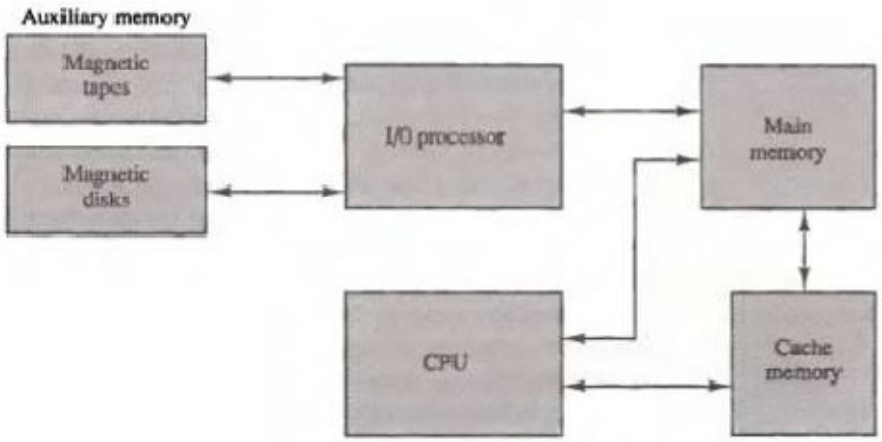
Memory Hierarchy:

The memory unit is an essential component in digital computer for storing programs and data. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary storages. The most common auxiliary devices are magnetic devices and tapes.

Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The hierarchy of memory gives us the total memory capacity of the computer. Fig 12-1 shows the components of typical memory hierarchy.

Figure 12-1 Memory hierarchy in a computer system.



At the bottom of the hierarchy there are relatively slow magnetic tapes used to store removable files. Next are the magnetic tapes used as backup storage. The main memory occupies the central position for communicating directly with the CPU, and with auxiliary memory devices through an I/O processor. Programs needed by the CPU are brought from auxiliary memory. Programs in main memory that are not currently needed are transferred into auxiliary memory.

A special very-high-speed expensive memory is called a cache, used to increase the speed of processing by making current programs and data available to the CPU at rapid rate. The cache memory is used to compensate the speed differences between memory and CPU. By making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU.

The reason for having two or three levels of memory hierarchy is economics. The cache memory is very small, expensive and has very high access speed. The auxiliary memory has a large storage capacity and is inexpensive but has low access speed. The goal of memory hierarchy is to obtain the highest possible average speed while minimizing the total cost of the entire memory system.

MAIN MEMORY:

The main memory is the central storage unit in a computer. It is a relatively large and fast memory used to store programs and data during the computer operation. The technology used for the main memory is semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible modes:

- Static
- Dynamic

The static RAM use flip-flops for storing binary information. They retain the information as long as power is applied to the unit. The dynamic RAM stores the information in the form of electric charges. The capacitors are provided inside the memory chip by MOS transistors. Dynamic RAM requires periodical refreshing to prevent the discharging of capacitors.

The static RAM is easier to use and has shorter read and write cycles. Static RAM are mainly used in implementing the cash memories. The dynamic RAMs are used for implementing the main memory. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.

The main memory of computer is constructed with RAM(Random Access Memory) integrated circuit chips, but portion of the memory may be constructed with ROM(Read-Only memory) chips.

ROM is used for strong an initial program called a boot strap loader. It is also used for storing programs and tables that do not change once the production of the computer is completed. Ram is used for storing the bulk of the programs and data that are subject to change. For this reason the ROM is called non-volatile memory and RAM is called volatile memory.

RAM and ROM chips are available in variety of sizes. For example a 1024x8 memory constructed with 128x8 RAM chips and 512 x 8 ROM chips.

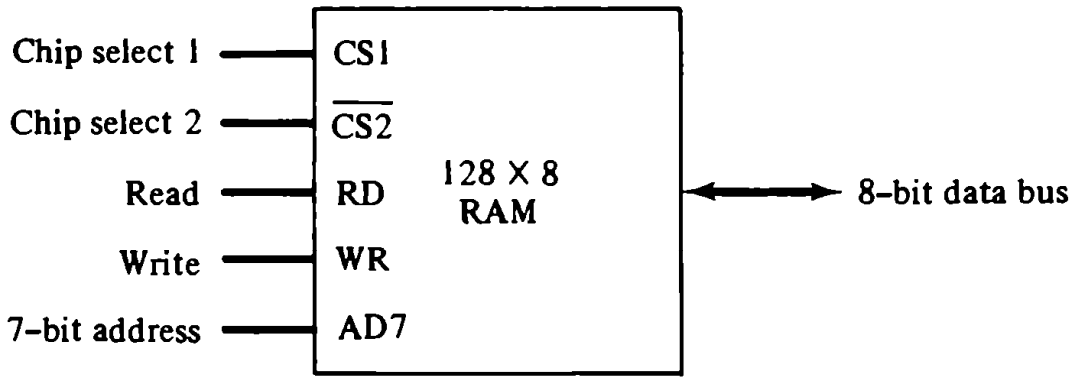
RAM and ROM chips:

A RAM chip is better suited for communication with the CPU. Another feature of RAM is a bidirectional data bus that allows the transfer of data either from memory to CPU during the read operation or from the CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffer.

The block diagram for the RAM chip is shown in fig 12-2. The capacity of the memory is 128 words and eight bits per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and two chip select(CS) control inputs are used to select a specific chip by the microprocessor when multiple chips are used.

The functional table listed in fig 12-2(b) specifies the operation of the RAM chip. The memory is place in Read or write mode when CS1=1 and $\overline{CS2} = 0$. Otherwise the memory is inhibited and data bus is in a high-impedance state.

Figure 12-2 Typical RAM chip.



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

The ROM chip is organized is organized in similar way. A ROM can only read, the data bus can only be in an output mode. Fig 12-3 shot the block diagram for ROM chip. For same size chip, ROM has more bits than RAM because the internal cells of ROM occupy less space than RAM.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. Two chip select inputs must be CS1=1 and $\overline{CS2} = 0$ for the unit to operate. There is no need for a read or write control because the unit can only read.

Memory Address Map:

The addressing of memory can be established by means of a table that specifies the memory addresses assigned to each chip. The table is called memory address map table. It is a pictorial representation of assigned address space for each chip in the system.

To demonstrate this, assume that a computer system has 512 bytes of RAM and 512 bytes of ROM. The memory address map for this configuration is shown in the table 12-1. In the table the hexadecimal address column assigns a range of hexadecimal addresses for each chip. The address bus lines are listed in third column. The small x's designate those lines that must be connected to the address inputs in each chip.

TABLE 12-1 Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines. The 4-RAM chips are distinguished with the bus lines 8 and 9. The distinction between RAM and Rom address is done with 10^{th} bus line. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The hexadecimal address for each chip is obtained from the information under the address bus assignment by subdividing into groups and each group contains four bits.

Memory connection to CPU:

The connection of memory chips to the CPU is shown in fig 12-4. RAM and ROM chips are connected to a CPU trough the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through chip select(CS) inputs.

This configuration gives the memory capacity of 512 bytes of RAM and 512 bytes of Rom. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2x4 decoder whose outputs go to the CS1 inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the fist RAM chip is selected. When 01, the second RAM chip is selected and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RAD control line for the ROM chip to be enabled only during the read operation.

Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, where as the data bus connected to the RAMs can transfer information in both directions.

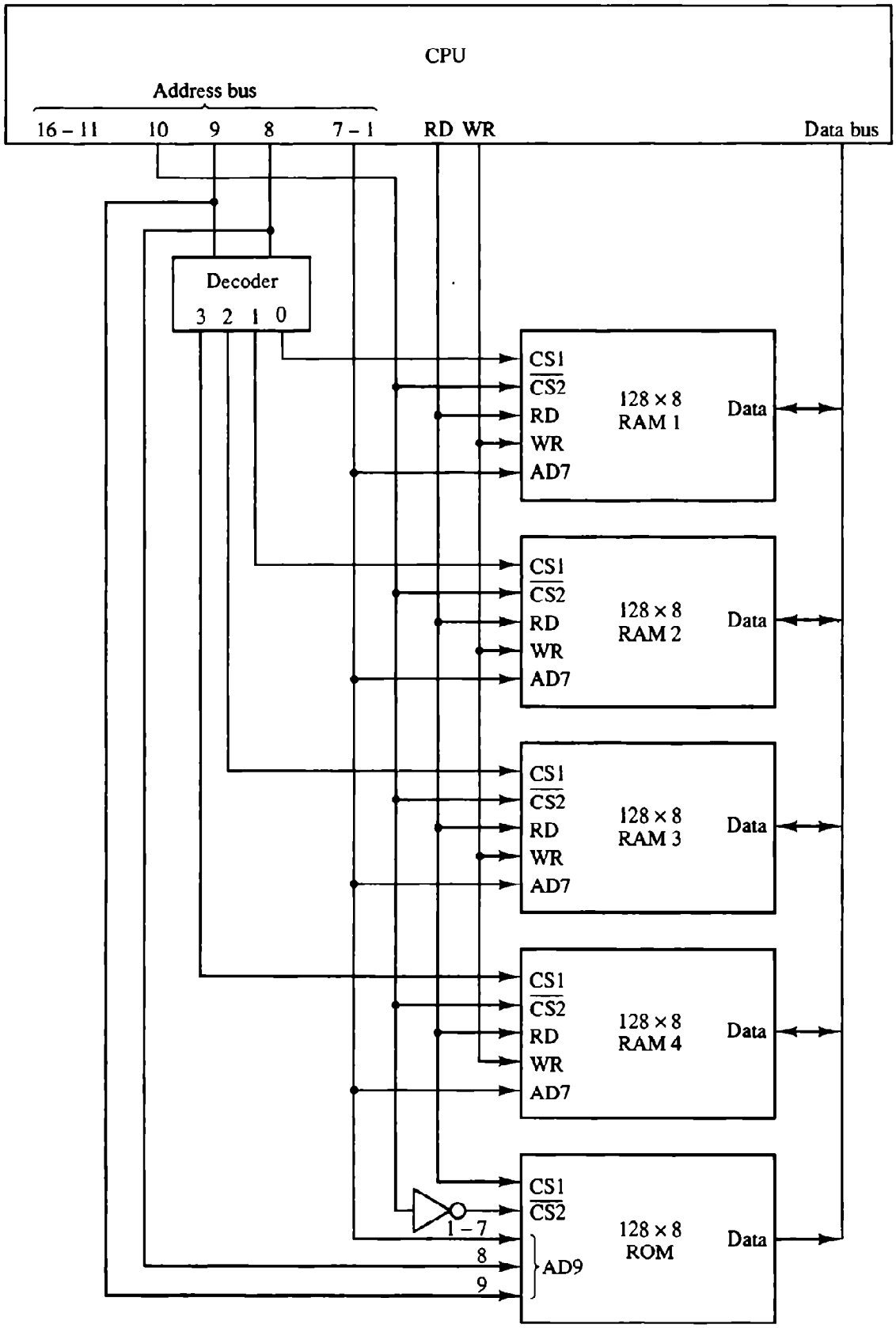


Figure 12-4 Memory connection to the CPU.

AUXILIARY MEMORY:

The most common auxiliary memory devices used in computer are magnetic disks and tapes. Other components are magnetic drums, magnetic bubble memory, and optical disks.

The disks and tapes are electromechanical device because they have moving parts. In these devices the access time (consists of seek time(to position the read-write head to a location) and transfer time(to transfer data to or from the device). The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

Magnetic Disks:

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be staked on one spindle with read/write heads available on each surface. All disks rotate together at high speed. Bits are stored in the magnetized surface along concentric circles called tacks. These tracks are divided into sections called sectors. The subdivision of one disk surface into tracks and sectors is shown in fig 12-5.

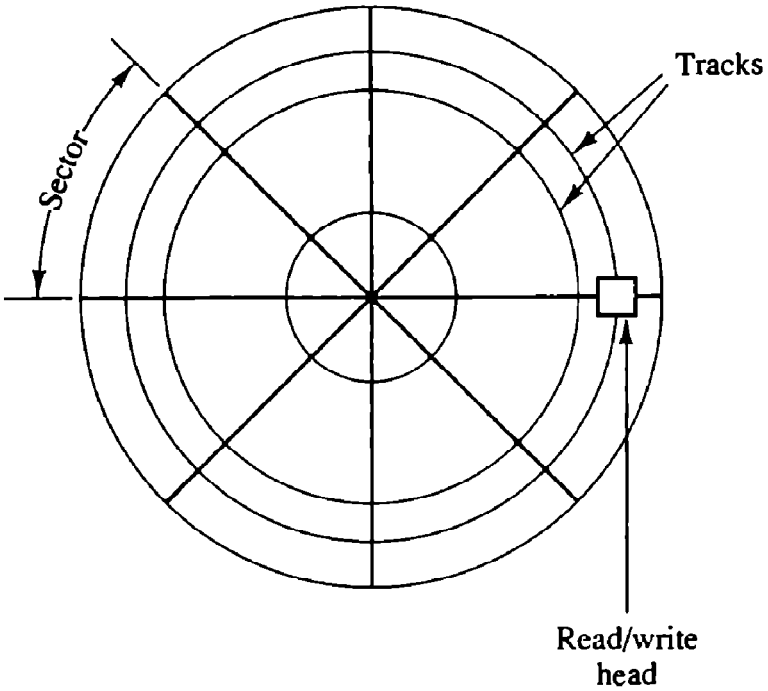


Figure 12-5 Magnetic disk.

Some units use a single read/write head for each disk surface. In other disk systems, separate read/write heads are provides for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit.

A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has

been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.

A track near the circumference is longer than a track near the center of the disk and some tracks will contain more recorded bits than others. Disks that are

A disk that cannot be removed from the system is called hard disk. A disk drive with removable disks is called floppy disk. There are two sizes commonly used floppy disks, with diameters of 5.25 and 3.5 inches.

Magnetic Tape:

electrical, mechanical and electronic components provide the parts and control mechanism for magnetic tape unit. The tape itself a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to from a character with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. Information is recorded in blocks called records. Gaps are inserted between the records where the tape can be stopped. Each record on a tape has an identification bit pattern at the beginning and end. By reading the bit at pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap.

A tape unit is addressed by specifying the record number and the number of characters in the record. Records may be fixed or variable length.

ASSOCIATIVE MEMORY:

Many data procession applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status.

The time required to fine an item stored in memory can be reduced considerably if store data can be identified by the content of the data itself rather than by an address. A memory unit accessed by content is called ***associative memory or content addressable memory(CAM).***

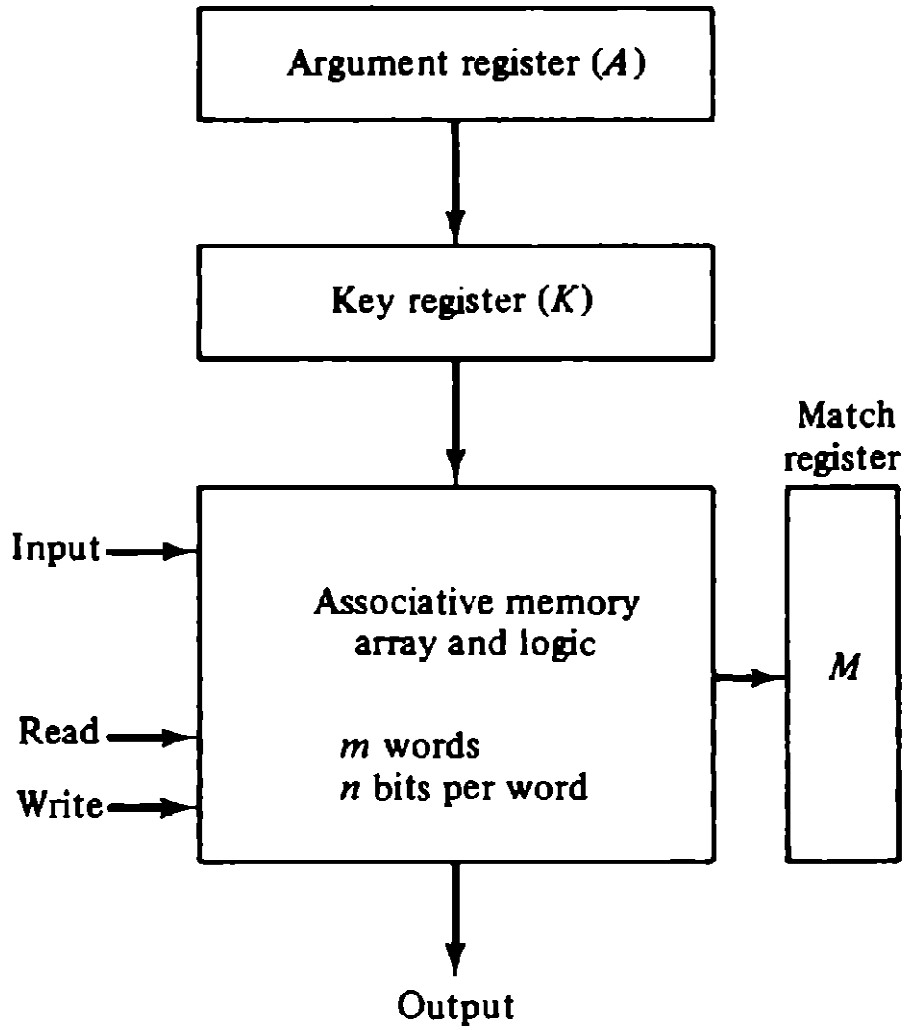
When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the world. When the world is to be read from an associative memory, part of the word is specified. The memory locates all words which match the specified content and marks them for reading.

An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external

argument. For this reason, associate memories are used in application where the search time is very critical and must be very short.

Hardware Organization:
The block diagram of an associative memory is shown in fig 12-6.

Figure 12-6 Block diagram of associative memory.



It consists of a memory array and logic for m words with n bits for word. The argument register A and key register K each have n bits, on for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set, indicate corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular filed or key in the argument word. The entire argument is compared with the memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

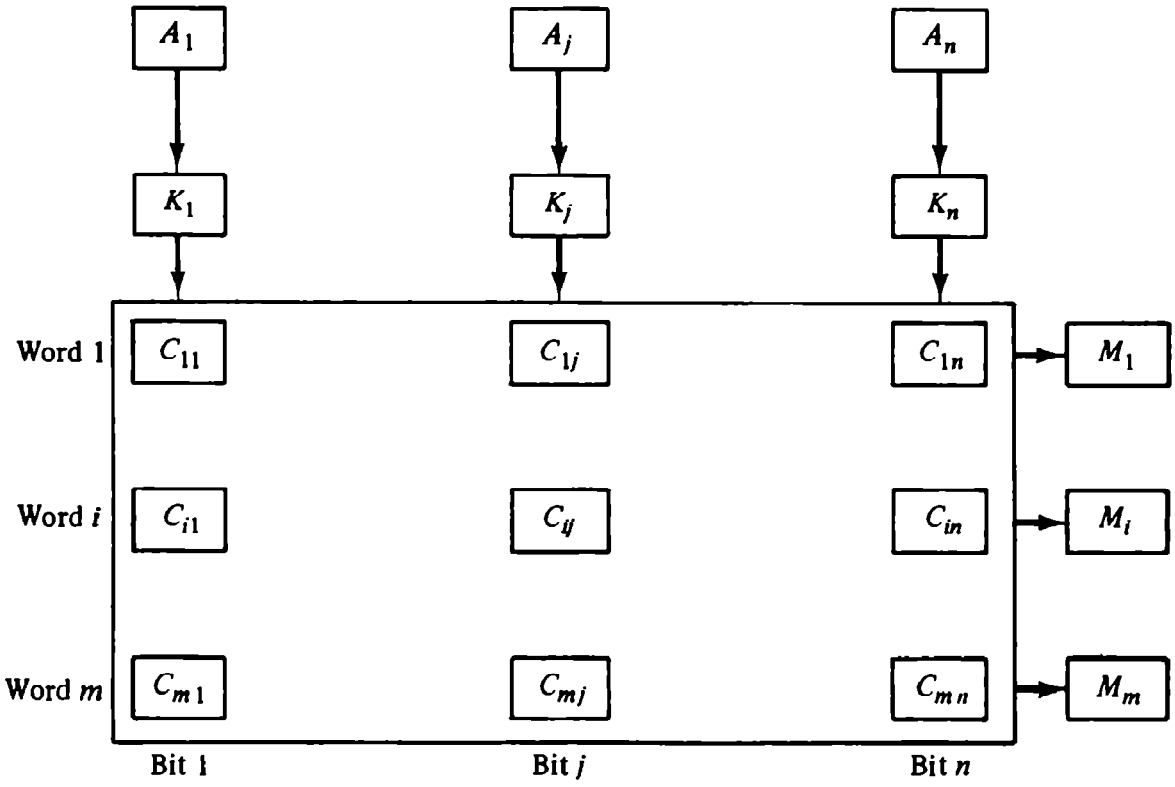
To illustrate this, consider the following example:

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word2 matches the argument field because the three leftmost bits of the argument and word are equal.

The relation between the memory array and external registers in associative memory is shown in fig 12-7.

Figure 12-7 Associative memory of *m* word, *n* cells per word.

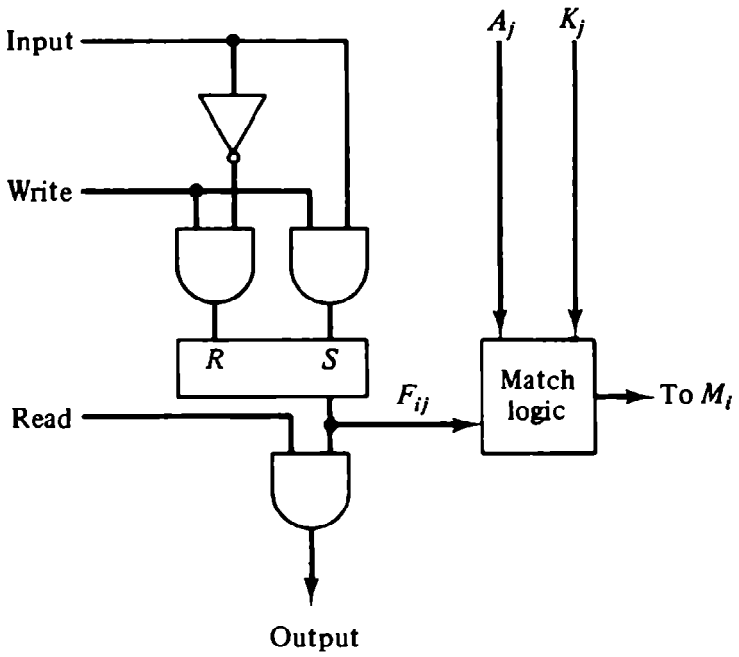


The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in the column j o the array provided that K_j=1. This is done for all columns j=1,23,.....,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bite M_i in the match register is set to 1, other M_i is cleared to 0.

The internal organization of cell C_{ij} is shown in fig 12-8.
It consists of a flip-flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input it is transferred into the storage cell during a write operation. The bit stored is read out during operation. The match logic compares the content of the storage cell

with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

Figure 12-8 One cell of associative memory.



Match Logic:

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if Aj = Fij for j = 1,2,....,n. Two bits are equal if they are both 1 for both 0. The equality of two bits can be expressed logically by the Boolean function

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

Where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For the word i to be equal to the argument A we must have all xj variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for the condition is:

$$M_i = x_1 x_2 x_3 \cdots x_n$$

The above is the AND operation of all pairs of matched bits in a word.

We now include the key bit K_j in the comparison logic. The requirement is that Kj = 0, the corresponding bits A_j an F_{ij} need no comparison. Only when Kj = 1 they must be compared. This requirement is achieved b ORing each term wih K_j', thus:

$$x_j + K_j' = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

When Kj = 1, we have Kj' = 0 and xj + 0 = xj,
when Kj=0, then Kj' = 1 and xj + 1 = 1.

A term (xj + Kj') will be in the 1 state if its pair of bits is not compared. This is necessary because each term in ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when Kj = 1.
The match logic for a word i in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

Each term in the expression will be equal to 1 if its corresponding Kj=0, if Kj =1, the term will be either 0 or 1 depending the value of Xj. A match will occur and Mi, will be equal to 1 if all terms are equal to 1.
If we substitute the original definition of xj, the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

Where PI is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word i = 1,2,3,....,m.

The circuit for matching one word is shown in fig 12-9.

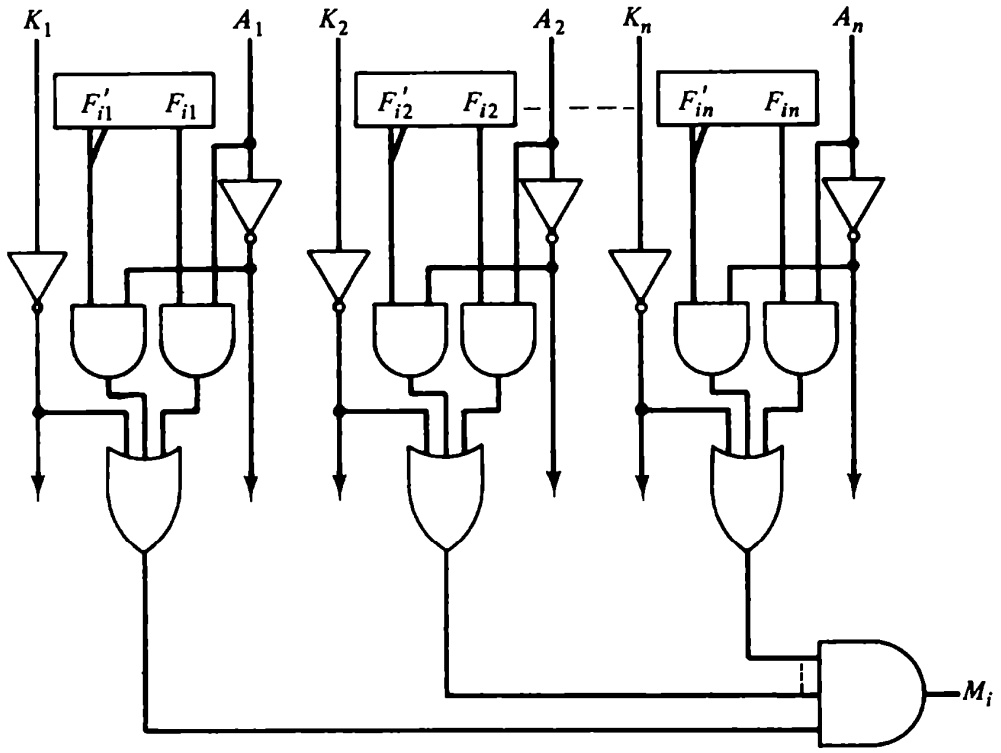


Figure 12-9 Match logic for one word of associative memory.

Each cell requires two AND gates and one OR gate. The inverters for Aj and Kj are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of common AND gate to generate the match signal to Mi. Mi will be logic 1 if a match occurs and 0 if no match occurs.

Read Operation:
If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the match register. It is then

necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a read signal to each word line whose corresponding Mi bit is a 1.

Write operation:
An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that, the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

CACHE MEMORY:

To reduce the memory access time and execution time in case of loops and subroutine calls, the references to memory must be restricted to few localized areas in memory. This is called ***property of locality of reference.***

We can reduce the average memory access time and program execution time by placing the active portions of the program and data in a fast small memory. Such a fast small memory is referred to as a ***cache memory.*** The cache memory is placed in between the CPU and main memory as shown in fig 12-1. The cache memory access time is less than the access time of main memory by a factor of 5 or 10. The cache is the fastest component in the memory hierarchy and has similar speed of CPU.

The fundamental idea cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory request will be found in the fast cache memory because of the locality of reference of property of the programs.

The basic operation of the cache is as follows:

When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the cache memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is measured in terms of ***hit ratio.*** When the CPU refers to memory and find the word in cache, it is said to be a hit. If the word is not found in cache, it is in main memory and it is called a miss. The ratio of the number of hits divided by the total CPU references to memory (hit plus misses) is the hit ratio.

The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are as follows:

1. Associative mapping.
2. Direct mapping.
3. Set associative mapping.

Consider the memory organization shown in fig 12-10 to explain the above three mapping procedures.

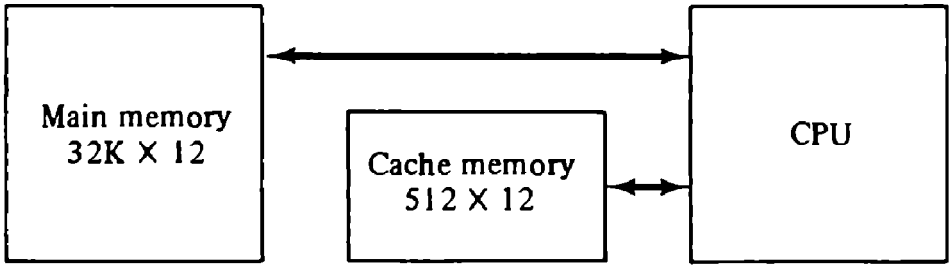


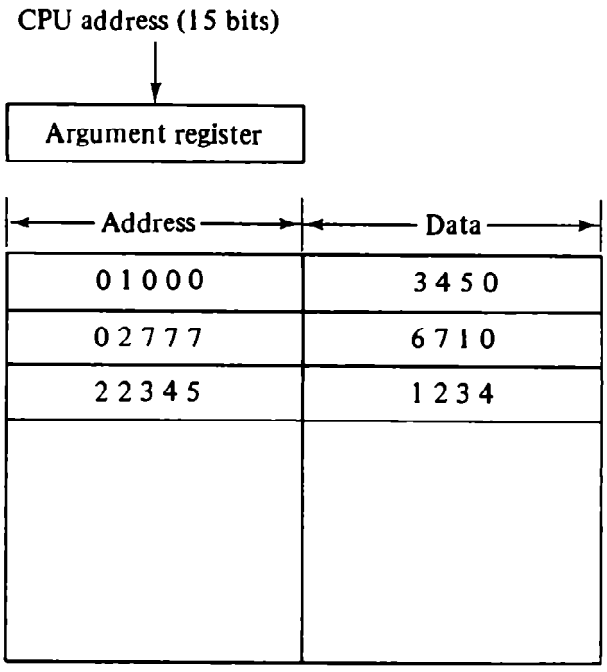
Figure 12-10 Example of cache memory.

from the fig, the main memory can store 32K words of 12 bits each. The cache is capable of storing 12 of these words at any given time. For every word stored in cache there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends 15-bit addresses to cache. If there is a hit, the CPU accepts the 12-bit data from the cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

Associative Mapping:

This is the fastest and most flexible cache organization used by an associative memory. This organization is illustrated in fig 12-11.

Figure 12-11 Associative mapping cache (all numbers in octal).



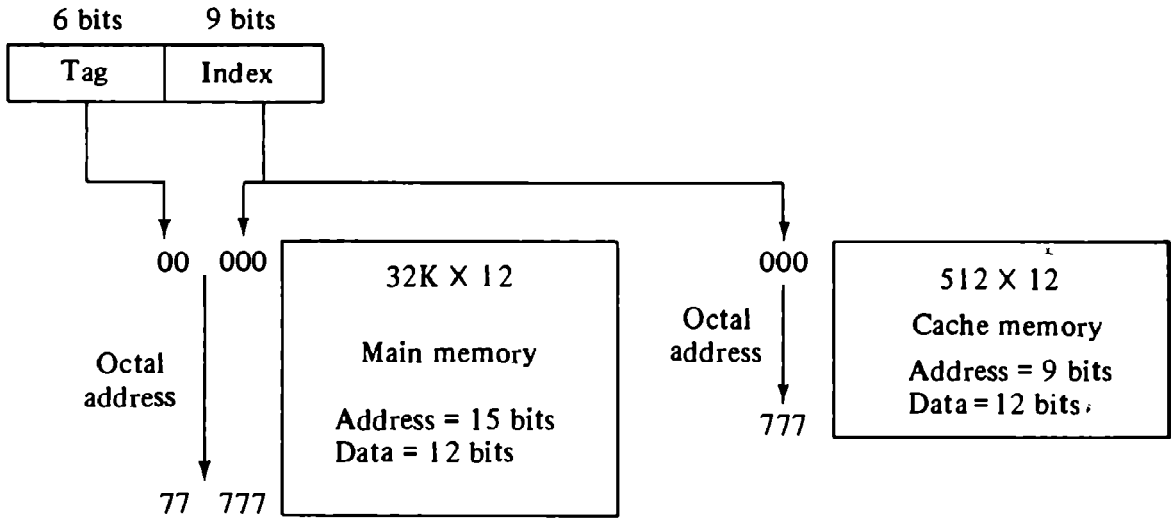
The associative memory stores both the address and data or the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal

number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address found, the corresponding 12-bit data is read and sent to the CPU. If not match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address-data pair must be replaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cell of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in-first-out(FIFO) replacement policy.

Direct Mapping:

Associative memories are expensive compared to Random-access memories, because of logic associated with each cell. The possibility of using a RAM for the cache is shown in fig 12-12.

Figure 12-12 Addressing relationships between main and cache memories.



The CPU address of 15 bits is divides into two fields. The nine least significant bits constitute the index filed and the remaining six bits from the tag filed. The fig shows the main memory needs an address that includes both the tag and the index bits. The number of bits in the index filed is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields. K bits for the index field and n – k bits for the tag field. The direct mapping cache organization uses n-bit addresses to access the main memory and the k-bit index to access the cache. The internal organization of the works in the cache memory is shown in fig 12-13(b). Each word in the cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits.

The CPU generates a memory request; the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache. If there is no match, there is a miss and the required word is read from main memory. It then stored in the cache together with the new tag, replacing the previous value.

The disadvantage of the direct mapping is that the hit ratio can drop considerably if two or more words whose address have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such works are relatively far apart in the address range.

To explain the direct mapping, consider the numeric example shown in fig 12-13.

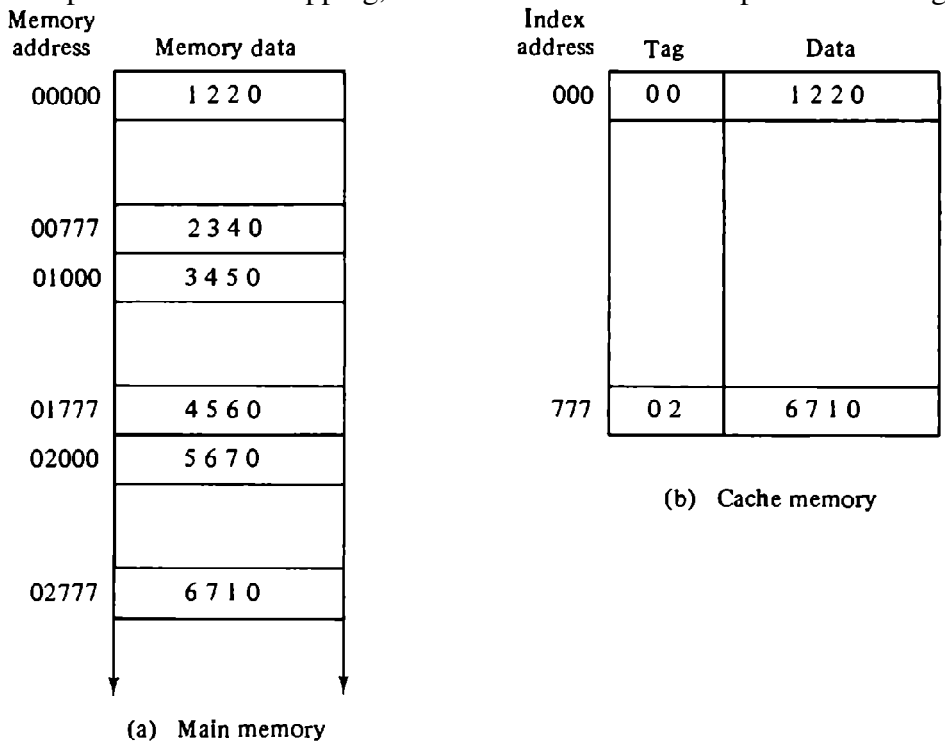


Figure 12-13 Direct mapping cache organization.

The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose the CPU now wants to access the word at address 02000. The index address is 000,so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

The above example is for a block size of one word. The same organization can be used for the block size of 8 words as shown in the fig 12-14. The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since 64x8 = 512. The block number is specified wit a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight word of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will likely improve with larger block size.

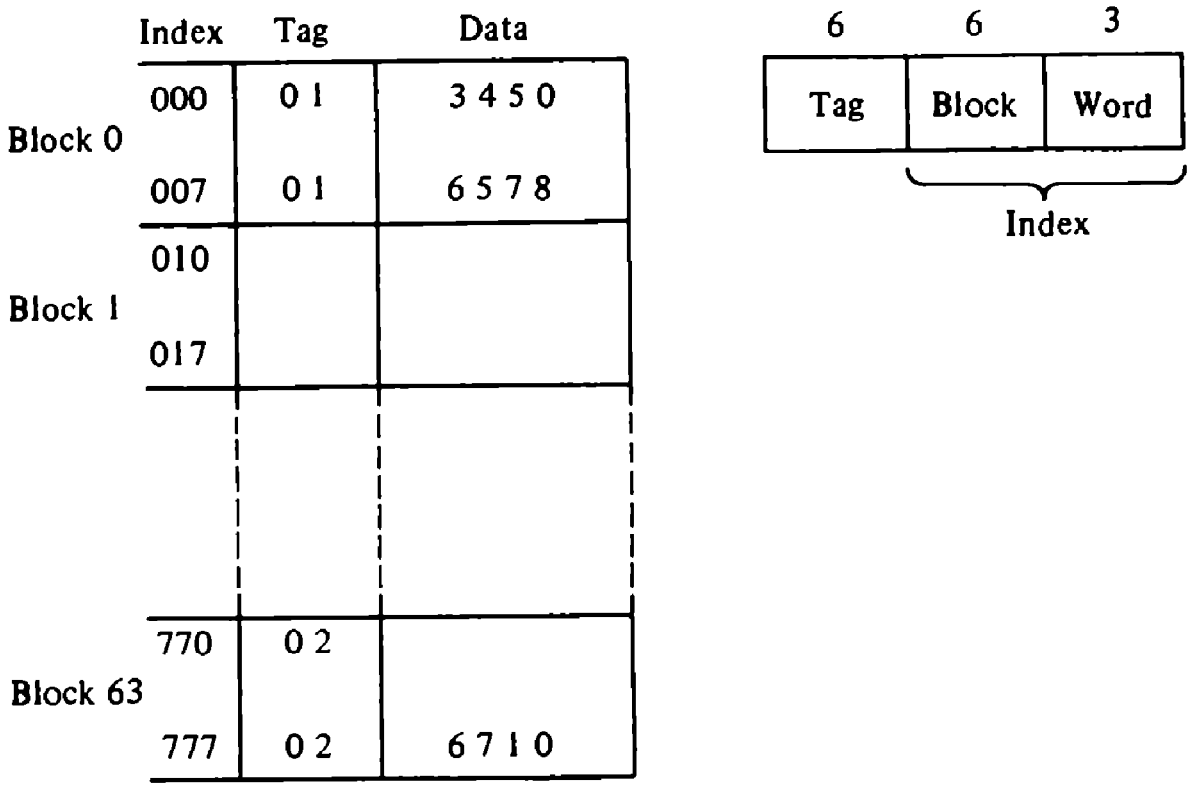


Figure 12-14 Direct mapping cache with block size of 8 words.

Set-Associative Mapping;

The disadvantage of direct mapping is that two words with the same index in their address but different tag values cannot reside in the cache memory at the same time. The set-associative mapping, each word of cache can store two or more words of memory under the same index address.

Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in fig 12-15.

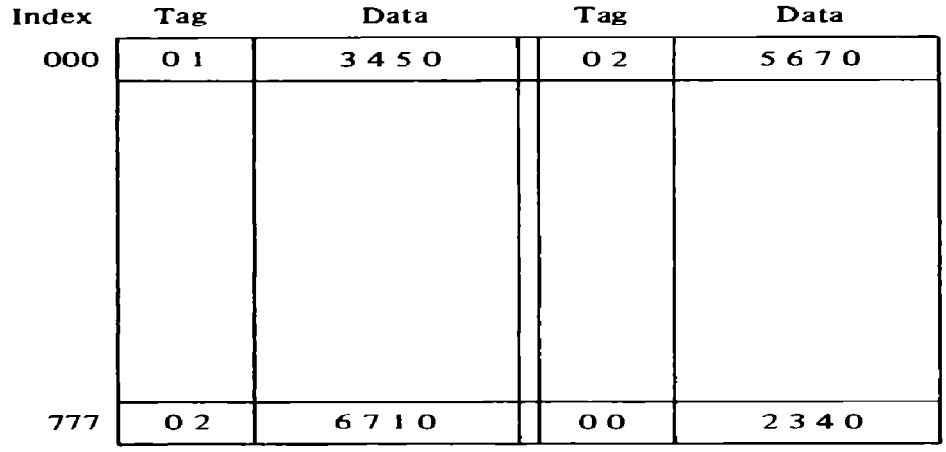


Figure 12-15 Two-way set-associative mapping cache.

Each index address refers two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 x 36. It can accommodate 1024 words of main memory, since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

From fig 12-15, when the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs. The comparison logic is done by an associative search of the tags in the set similar to an associate memory search; hence the name "set-associative. The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache. However, an increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.

When miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: Random replacement, FIFO, and LRU.

Writing into Cache:

When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways to do this

1) Write-through method.
2) Write-back method.

In write-through method the main memory is updated in parallel with cache memory write operation. This method has advantage that main memory always contains the same data as the cache.

In write-back method, only cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache is copied into main memory. It is useful when the words in cache will be updated several times.

VIRTUAL MEMORY:

Virtual memory is used to give the programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated address into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

Address Space and Memory Space:
The address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computer the address and memory spaces are identical.

The address space is allowed to be larger than the memory space in computers with virtual memory.

For example, consider a computer with a main memory capacity of 32K words fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 120K words. Thus auxiliary memory has a capacity which is equivalent to capacity of 32 main memories.

In multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory as shown in fig 12-16.

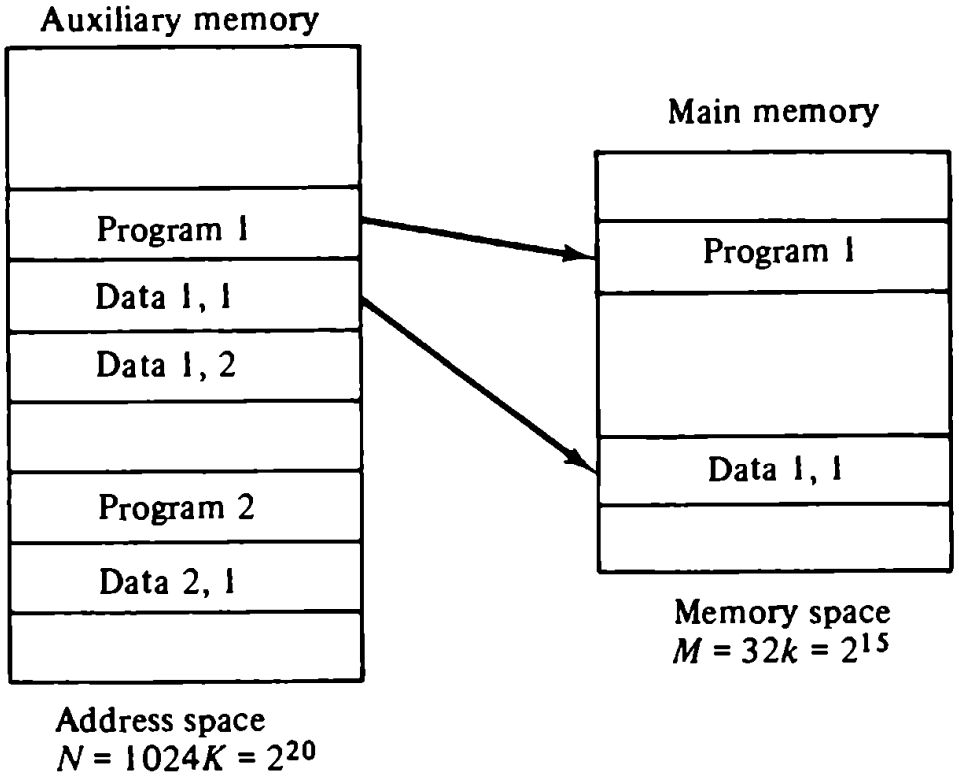
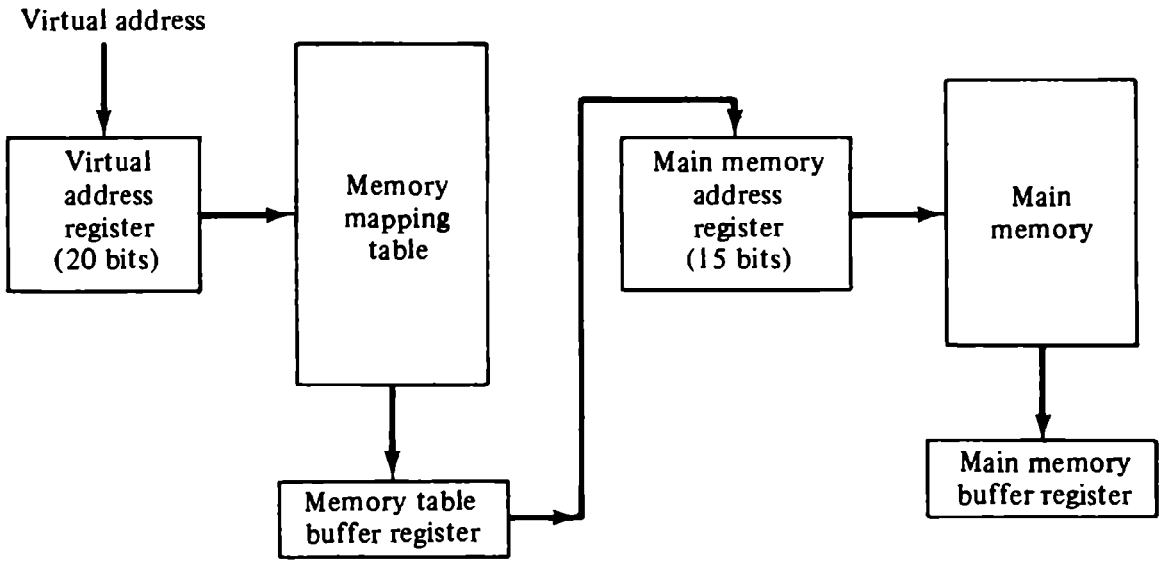


Figure 12-16 Relation between address and memory space in a virtual memory system.

In virtual memory system, programmers are told that they have the total address space called virtual addresses at their disposal. A memory table is needed to map a virtual address of 20 bits to a physical address of 15 bits as shown in fig 12-17.

The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU. The mapping table may be stored in a separate memory or in main memory. The third alternative is to use an associative memory as explained below.

Figure 12-17 Memory table for mapping a virtual address.



Address Mapping Using Pages:

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size.

For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term "page frame" is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in fig 12-18.

At any given time, up to four pages of address space may reside in main memory in any one of the four blocks. The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used o specify a line address and the remaining high-order bits of the virtual address specify the page number in the example of fig 12-18, a virtual address has 13 bits. Since each pages consists of 2^{10} = 1024 words, the high-order three bits of virtual address will specify one of the eight pages and the low-order 10 bits give the line address with the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

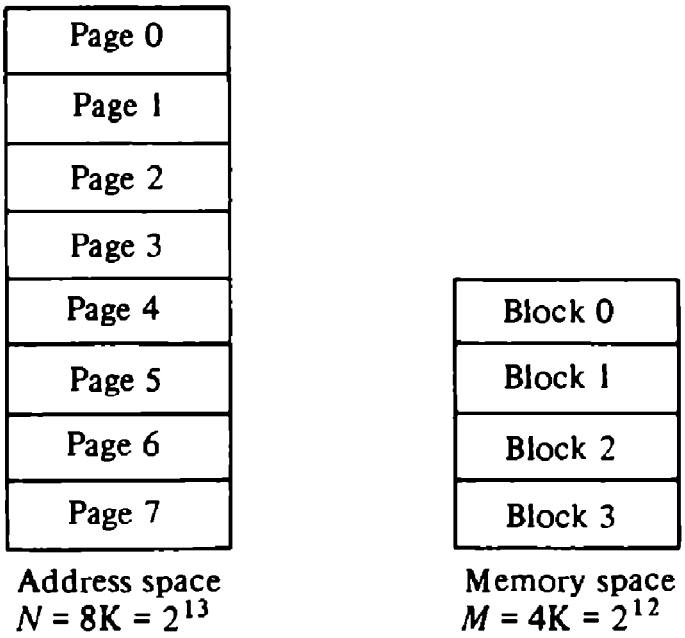
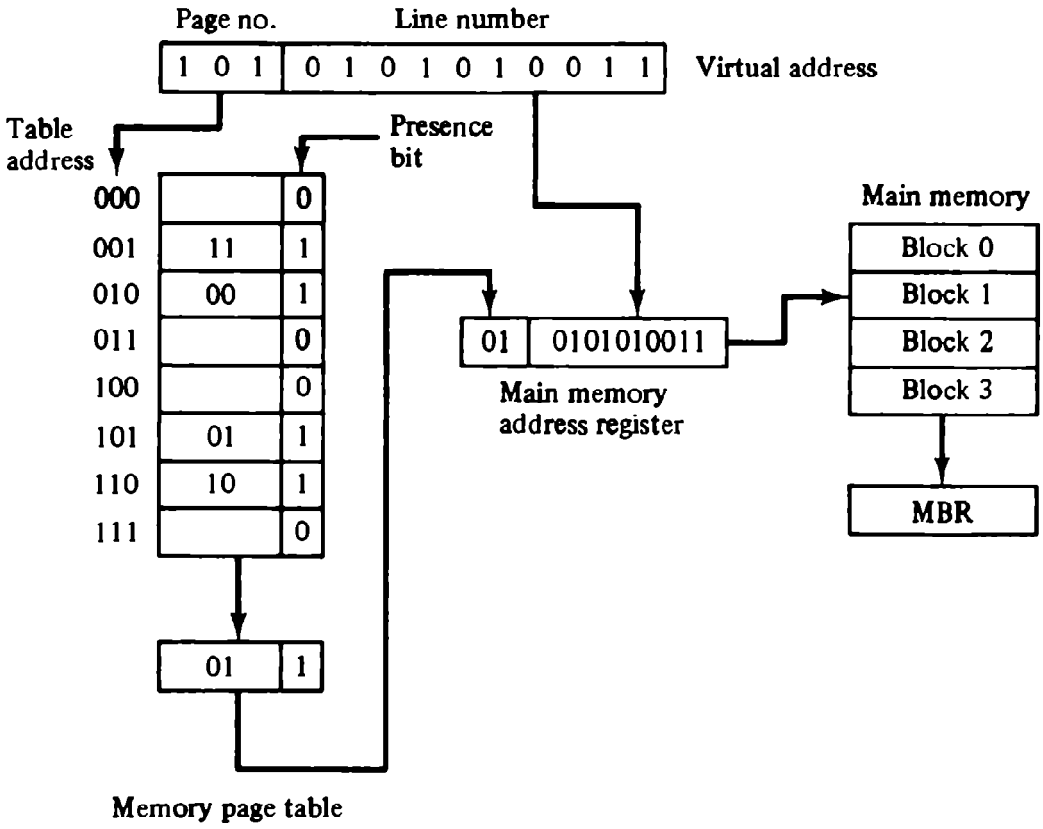


Figure 12-18 Address space and memory space split into groups of 1K words.

The organization of the memory mapping table in a paged system is shown in fig 12-19.

Figure 12-19 Memory table in a paged system.



The memory-page table consists of eight words, one for each page. The address in a page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows the pages 1,2,5, and 6 are now available in main memory in blocks 3,0,1, and 2 respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory page table. The content of the word in the memory page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

Associative Memory Page Table:

A Random-access memory page table is inefficient with respect to storage utilization. In the example of fig 12-19 we observer that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of associative memory, with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

We replace the random-access memory page table(fig 12-19) with an associative memory of four words as shown in fig 12-20. Each entry in the associative memory array consists of two fields. The first three bits specify a filed for storing the page number. The last two bits constitute a file for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument register are compared with all page numbers in the page filed of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

Page replacement:

A virtual memory system is combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space.

It must decide:
1) Which page in main memory need to be removed to make room for a new page
2) When a new page is to be transferred from auxiliary memory to main memory?
3) Where the page is to be placed in main memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program continues execution until it needs a page that is still in auxiliary memory. This condition is called ***page fault.*** When the page fault occurs, the execution of the program is suspended until the required page brought into main memory.

When page fault occurs, a required page need to transfer from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the page replacement algorithms. The goal of a replacement policy is, remove the page that is least referenced in immediate future. Two common replacement algorithms used are ***first-in first-out(FIFO)*** and the ***least recently used(LRU).***