

Number system:

- A system that is used for representing numbers is called the number system.
- In digital electronics, the numbers are used to represent the information.
 - Hence, it is important to learn and understand different types of number systems so we can easily represent and interpret the information in the form of numbers.
 - There are several types of number systems and the basis of this classification is the base or radix of the number of symbols used to denote the numbers in the number system.

Types of Number Systems:

Depending on the base or radix, number systems can be classified into 4 major types.

They are

1. Decimal Number System
2. Binary Number System
3. Octal Number System
4. Hexadecimal Number System.

- The number system which we human beings commonly use is the decimal number system consisting of digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.
- But in computers, instructions as well as data are stored in binary number system consisting of digits '0' & '1'.

It is so because binary digits 0 and 1 can be easily represented by an electrical switch. If the switch is closed it could be '1' and if it is open it could be '0'.

Decimal Number System:

The system of numbers which has base or radix 10, i.e. uses total 10 symbols to represent numbers of the system is called decimal number system.

- * The symbols used in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; where each of these symbols assigned a specific value.
- * It is a positional weighted system. i.e. the value attached to a symbol depends on its location with respect to the decimal point.
- * Each symbol in the number is called a digit.
- * The left most digit in any number representation, which has the greatest positional weight out of all the digits represent in that number, is called the most significant digit (MSD).
- * The right most digit, which has the least positional weight out of all the digits represent in that number is called the least significant digit (LSD).
- * The digits on the left side of the decimal point form the integer part of a decimal number and

those on the right side form the fractional part.

- * The digits to the right of the decimal point have weights which are negative powers of 10 and
- * The digits to the left of the decimal point have weights which are positive powers of 10.
- * The value of a decimal number is the sum of the products of the digits of that number with their respective column weights.
- * The weight of each column is 10 times greater than the weight of the column to its right.

In general, the value of any decimal number

$$d_0, d_{0-1}, d_{0-2}, \dots, d_1, d_0$$

Ex :-

Let us consider a number 247.

The number is said to have 3 digits (2, 4 and 7) each one known as decimal digit.

The digit "2" in view of its position has a value equal to 200

Similarly 4 has a value equal to 40 and 7 has a value 7. Hence the value of the given number can be given as,

$$\begin{aligned} 247 &= 2 \times 100 + 4 \times 10 + 7 \times 1 \\ &= 2 \times 10^2 + 4 \times 10^1 + 7 \times 10^0 \end{aligned}$$

The position value of 7 is 1, the positional value of 4 is 10, and 2 is 100.

In general, the value of any mixed decimal number is given by

$$d_n d_{n-1} d_{n-2} \dots d_2 d_1 d_0 \cdot d_{-1} d_{-2} d_{-3} \dots = d_K$$
$$(d_n \times 10^n) + (d_{n-1} \times 10^{n-1}) + \dots + (d_1 \times 10^1) + (d_0 \times 10^0) +$$
$$(d_{-1} \times 10^{-1}) + (d_{-2} \times 10^{-2}) + \dots$$

Ex :-

consider the mixed decimal number 9256.26 using digits 2, 5, 6, 9.

$$9256.26 = 9 \times 1000 + 2 \times 100 + 5 \times 10 + 6 \times 1 + 2 \times (1/10) + 6 \times (1/100)$$
$$= 9 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 6 \times 10^{-2}$$

The positional value of 2 should be less than that of 6 by 10 times (and so equal to 10^1)

The digit 6 is the least significant digit

The digit 9 is the most significant digit

* consider another number 6592.69 using the same

digits 2, 5, 6, 9

Here

$$6592.69 = 6 \times 10^3 + 5 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 + 6 \times 10^{-1} + 9 \times 10^{-2}$$

The digit 9 is the most significant digit

The digit 6 is the least significant digit

Note :- The difference in positional values of the same digits, when placed in different positions.

Binary Number System:

The binary number system is a positional weighted system.

- * The base or radix of this number system is 2 . Hence, it has two independent symbols.
- * The base itself cannot be a symbol. The symbols used are '0' and '1'.
- * A binary digit is called a bit.
- * A binary point separates the integer and fractional parts.
- * The weight of each bit position is one power of 2 greater than the weight of the position to its immediate right.
- * The first bit to the left of the binary point has a weight of 2^0 and that column is called the units column.
- * The second bit to the left has a weight of 2^1 and it is in the 2 's column.
- * The third bit to the left has a weight of 2^2 and it is in the 4 's column, and so on.
- * The first bit to the right of the binary point has a weight of 2^{-1} and it is said to be in the $\frac{1}{2}$'s column, the next right bit with wt of 2^{-2} is in the $\frac{1}{4}$'s column, and so on.

Ex:

Let us consider a binary number 1010

This number consists of 4 binary digits (BITS)

- The decimal value of the binary number is the sum of the products of all its multiplied by the weights of their respective positions.

$$1010 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

- The value of 1010.₁₀ can be given as

$$1010.₁₀ = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}$$

- In this case, the point is known as the binary point.

- The binary digit '0' present after the binary point is now LSB.

- The binary number system is used in digital computers because the switching circuits used in these computers use two state devices such as transistors, diodes etc.

- A transistor can be OFF or ON; a switch can be OPEN or CLOSED, a diode can be OFF or ON etc.

- These devices have to exists in one of the two possible states.

So, these two states can be represented by the symbols 0 and 1 respectively.

Octal Number System:

As the number value increases, the no. of digits to be used in the binary number system increases.

- * In order to make it easier for the user, the octal number system was used.
- * It has a base or radix 8.
- * It consists of digits 0, 1, 2, 3, 4, 5, 6 and 7.
- * In this system, all the numbers are given as combination of these symbols.
- * It is an important system which is often used in microComputers.

Ex :

Let us consider an octal number 352.

This number consists of three octal digits.

- * Each of these digits has a positional value, i.e. weightage to each position will be powers of 8.
- * The right most digit '2' has the least weightage.
- * The left most digit '3' has the highest weightage.
- * The value of the octal number 352, is given by

$$352 = 3 \times 8^2 + 5 \times 8^1 + 2 \times 8^0$$

Hexadecimal number system :-

The hexadecimal number system has a base of 16. This means this system has 16 (Hexadecimal) symbols.

- Since the basic numbers we used to have only ten digits (from 0 to 9).
- The first six letters of the English alphabets are borrowed to make the remaining 6 symbols.
- Hence this system is made up of the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.
- All the numbers are given as combination of these symbols.
- Each of these digits has a positional value.
- The weightage to each position will be a power of 16.

Ex :-

Let us consider a hexadecimal number 5C. This number consists of two hexa decimal digits. The rightmost digit 'C' has the least weightage. The left most digit '5' has the most weightage. The value of the 5C should probably written as

$$5C = 5 \times 16^1 + C \times 16^0$$

C corresponds to the decimal no 12.

$$\text{So, } 5C = 5 \times 16^1 + 12 \times 16^0$$

Conversion of numbers from one radix to another radix.

Decimal to Binary: There are many ways of converting decimal numbers to binary numbers the most commonly used is the double-dabble method.

* To convert decimal number system to binary number system, divide the given number successively by 2 till the quotient is zero and read the remainders from bottom to top for integer.

* To convert the given decimal fraction to binary, successively multiply the decimal fraction by 2 till the product is '0' or till the required accuracy is obtained.

keep the integer in the product as it is and read the integers in the product from top to bottom.

Ex:

1. Convert 52_{10} to binary using double-dabble method.

We divide the given decimal number successively by 2 successive division Remainders

2	52	
2	26	- 0
2	13	- 0
2	6	- 1
2	3	- 0
2	1	- 1
	0	

read the remainders from bottom to top

read the remainders upwards (i.e from bottom to up) to get equivalent binary number.

$$\therefore (52)_{10} = (110100)_2$$

Ex: Convert $(105.15)_{10}$ to binary using dibble-dabble method.

Sol: The given decimal number is a mixed number convert integer and fraction part separately.

Conversion of 105_{10}

successively divide by 2 till the quotient is zero.

and read the remainders from bottom to top.

successive division

$$\begin{array}{r} 2 \overline{)105} \\ 2 \overline{)52 - 1} \\ 2 \overline{)26 - 0} \\ 2 \overline{)13 - 0} \\ 2 \overline{)6 - 1} \\ 2 \overline{)3 - 0} \\ 2 \overline{)1 - 1} \\ 0 \end{array}$$

$$(105)_{10} = (1101001)_2$$

Conversion of 0.15_{10}

Multiply the given fraction by 2

keep the integer in the products as it is and multiply the new fraction in the product by 2.

Continue this process till the product is zero and read the integers in the products from top to bottom.

Given fraction

$$0.15 \text{ by } 2$$

$$0.30 \text{ by } 2$$

$$0.60 \text{ by } 2$$

$$1.20 \text{ by } 2$$

$$0.40 \text{ by } 2$$

$$0.80 \text{ by } 2$$

$$0.15$$

$$0.30$$

$$0.60$$

$$1.20$$

$$0.40$$

$$0.80$$

$$(0.15)_{10} = (0.001001)_2$$

The final result is

$$(105.15)_{10} = (1101001.001001)_2$$

Decimal to octal:

To convert a mixed decimal number to a mixed octal number, convert the integer and fraction parts separately.

- * To convert the given decimal integer number to octal, successively divide the given number by 8 till the quotient is 0. The last remainder is the MSB.
- * The remainders read upwards that gives the equivalent octal integer number.

Ex :-

Convert 378_{10} to octal

Sol We divide the given decimal number by 2 successively and read the remainders from bottom to up.

Successive division remainders

8	378	
8	47	
8	5	
8	0	

↑ 2 7 5

Read the remainders from bottom to up.

Therefore, $378_{10} = 572_8$

* To convert the given decimal fraction to octal, successively multiply the decimal fraction by 8 till the product is '0' or till the required accuracy is obtained. The first integer from the top is the MSB. The integers to the left of the octal point read downwards that give the octal fraction.

Ex :-

Convert $(0.93)_{10}$ to octal

$$\begin{array}{r} 0.93 \times 8 \\ 0.94 \times 8 \\ 0.52 \times 8 \\ 0.16 \times 8 \end{array} \quad \begin{array}{r} 7.44 \\ 3.52 \\ 4.16 \\ \downarrow 1.28 \end{array}$$

Read the integers to the left of the octal point downwards.

$$\therefore (0.93)_{10} = (0.734)_{8}$$

* Conversion of large decimal numbers to binary and large binary numbers to decimal can be conveniently done via octal.

Ex :-

Convert 101111010001_2 to decimal

sol Since the given binary number is large we first convert this number to octal then convert octal number to decimal

$$(1011101000)_2 = (572)_8$$

$$\begin{aligned}(572)_8 &= 5 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 \\ &= 2560 + 448 + 16 + 1\end{aligned}$$

$$(572)_8 = (3025)_{10}$$

Ex 6

Convert $(5497)_{10}$ to binary

~~Since~~ Since the given decimal number is large,
we first convert this number to octal and
then convert the octal number to binary.

Successive division

Remainders

8	5497	
8	687	↑ 1
8	85	1 7
8	10	1 5
8	1	1 2
	0	1

Therefore,

$$(5497)_{10} = (12571)_8$$

$$(12571)_8 = (00101010111001)_2$$

Decimal to Hexadecimal :

To convert decimal to hexadecimal, successively divide the given number by 16 till the quotient is 0.

The last remainder is MSB.

The remainders read from bottom to top give the equivalent hexadecimal integer.

* To convert hexadecimal fraction to decimal,
successively multiply the given decimal fraction and subsequent decimal fraction by 16, till the product is '0' or the required accuracy is obtained, and collect all the integers to the left of the decimal point.

The first integer is the MSB and the read the integers from top to bottom give the hexadecimal fraction. This is known as the hexdabble method.

Ex :-

Convert $(2598.675)_{10}$ to hex

Q) The given decimal number is a mixed number.
Convert the integer and fraction parts separately

Conversion of 2598

16	2598
16	162
16	10
	0

Remainder	
Decimal	Hex
↑ 6	6
↓ 2	2
↓ 10	A

Reading the remainders upwards,

$$2598_{10} = A26_{16}$$

Conversion of 0.675

Given fraction is 0.675

$$0.675 \times 16 \quad | \quad 10.8$$

$$0.800 \times 16 \quad | \quad 12.8$$

$$0.800 \times 16 \quad | \quad 12.8$$

$$0.800 \times 16 \quad | \quad 12.8$$

Reading the integers to the left of hexadecimal point downwards, $0.675_{10} = 0.ACCC_{16}$

$$\therefore 2598.675_{10} = A26.ACCC_{16}$$

Ex:

Convert 49056_{10} to binary
The given number is very large. It is tedious to convert this to binary directly. So, convert this to hex first and then convert the hex to binary.

Remainder

	49056	Decimal	Hex	Binary
16	3066	0	0	0000
16	191	10	A	1010
16	11	15	F	1111
	0	11	B	1011

$$\therefore 49056_{10} = BFA0_{16} = (10111110100000)_2$$

Binary to octal :

To convert a binary number to an octal number, starting from the binary point make groups of 3 bits each, on either side of the binary point, and replace each 3-bit binary group by the equivalent octal digit.

Ex :- Convert 110101.101010_2 to octal

Sol Groups of 3 bits are $\underline{110} \underline{101} \cdot \underline{101} \underline{010}$
Convert each group to $6 \ 5 \cdot 5 \frac{1}{2}$
octal
The result is $(65.52)_8$

2. Convert 1010111001.0111_2 to octal

Sol Groups of 3 bits are $10 \ \underline{\underline{101}} \ \underline{\underline{111}} \ \underline{\underline{001}} \cdot \underline{011} \ \underline{\underline{100}}$
Convert each group to octal
 $2 \ 5 \ 7 \cdot 1 \frac{3}{4}$

∴ The result is $(2571.34)_8$

$$(1010111001.0111)_2 = (2571.34)_8$$

Binary to Decimal :

Binary numbers may be converted to their decimal equivalents by the positional weights method.

* In this method, each binary digit of the number is multiplied by its position weight and the product terms are added to obtain the decimal number.

Ex :

Convert 10101_2 to decimal

Sol Positional weights $2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0$

Binary number 1 0 1 0 1

$$\therefore (10101)_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$= 16 + 0 + 4 + 0 + 1$$

$$= 21_{10}$$

Ex :

Convert 11011.101_2 to decimal

Sol Positional weights $2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \ 2^{-1} \ 2^{-2} \ 2^{-3}$

$$1 1 0 1 1 . 1 0 1 = (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$$

$$= 16 + 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125$$

$$= 27.625_{10}$$

$$\therefore (11011.101)_2 = (27.625)_{10}$$

Binary to Hexadecimal

To convert a binary number to a hexadecimal number, starting from the binary point, make groups of 4 bits each, on either side of the binary point and replace each 4-bit group by the equivalent hex decimal digit.

Ex :

Convert 1011011011_2 to hexadecimal

⇒ Make groups of 4 bits, and replace each 4-bit group by a hex digit.

Given binary number is

Groups of 4 bits are

convert each group to hex

The result is

$\begin{array}{r} 1011011011 \\ 0010 \quad 1101 \quad 1011 \end{array}$

$(2DB)_{16}$

Ex :

Convert $0101111011 \cdot 01111_2$ to hexadecimal

Given binary number is $0101111011 \cdot 01111$

Groups of 4 bits are $0010 \cdot 1111 \quad 1011 \cdot 0111 \quad 1100$

convert each group to hex

to hex

The result is

$(2FB \cdot 7C)_{16}$

Octal to Decimal :

To convert an octal number to a decimal number, multiply each digit in the octal number by the weight of its position and add all the product terms.

The decimal value of the octal number $d_n d_{n-1} \dots d_1 d_0$ d_1, d_2, \dots, d_k is $(d_n \times 8^n) + (d_{n-1} \times 8^{n-1}) + \dots + (d_0 \times 8^0) + (d_1 \times 8^1) + \dots + (d_k \times 8^k)$

Ex:

Convert 4057.06_8 to decimal

$$\begin{aligned}\text{Sol } 4057.06_8 &= 4 \times 8^3 + 0 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 + 0 \times 8^{-1} + 6 \times 8^{-2} \\ &= 2048 + 0 + 40 + 7 + 0 + 0.0937 \\ &= 2095.0937_{10}\end{aligned}$$

Octal to Binary :

To convert a given octal number to binary, just replace each octal digit by its 3-bit binary equivalent.

Ex:

Convert 367.52_8 to binary

Sol Given octal number is 3 6 7 . 5 2
Convert each octal digit to binary
 $011 \quad 110 \quad 111 \cdot 101 \quad 010$

The result is

$$(011110111 \cdot 101010)_2$$

Octal to Hexadecimal :

To convert an octal number to hexadecimal, the simplest way is to first convert the given octal number to binary, and then the binary number to hexadecimal.

Ex :

Convert 756.603_8 to hex

Given octal number is 7 5 6 . 6 0 3
convert each octal digit 111 101 110 . 110 000 011
to binary

Group of four bits are 0001 110 110 . 1100 0001 1000

Convert each four-bit group I E E . C 1 8
to hex

The result is

$(IEE.C18)_{16}$

Hexadecimal to Binary :

To convert a hexadecimal number to binary, replace each hex digit by its 4-bit binary group.

Ex :

Convert $4BAC.80_{16}$ to Binary

Given hex number is 4 B A C . 8 0

convert each hex digit 0100 1011 1010 1100 . 1011 0000

to 4-bit binary.

The result is

$(0100\ 1011\ 1010\ 1100\ .\ 1011\ 0000)_2$

Hexadecimal to Decimal:

To convert a hexadecimal number to decimal, multiply each digit in the hex number by its position weight and add all those product terms.

Ex :

Convert $5C7_{16}$ to decimal

Sol) Multiply each digit of $5C7$ by its position weight and add the product terms.

$$\begin{aligned}5C7_{16} &= (5 \times 16^2) + (12 \times 16^1) + (7 \times 16^0) \\&= 1280 + 192 + 7 = 1479_{10}\end{aligned}$$

Hexadecimal to octal:

To convert a hexadecimal number to octal, the simplest way is to first convert the given octal no to binary and then the binary number to octal.

Ex :

Convert $B9F.AE_{16}$ to octal

Sol) Given hex number is

Convert each hex digit

to binary

Group of three bits are

convert each 3 bit group to octal

The result is $(5627.534)_8$.

Binary Arithmetic:

Arithmetic operations such as addition, subtraction, multiplication and division can be carried out in any number system.

* The method followed is similar to the method followed in the decimal number system.

Binary Addition:

The rules for binary addition are

$$0+0=0 \quad 0+1=1 \quad 1+0=1 \quad 1+1=10 \text{ i.e., } 0 \text{ with carry } 1$$

Ex:

Add the binary numbers 1101.101 and 111.011

Sol.

$$\begin{array}{r} 8 \ 4 \ 2 \ 1 & \frac{-1}{2} \ \frac{-2}{2} \ \frac{-3}{2} \\ 1 \ 1 \ 0 \ 1 & \cdot \ 1 \ 0 \ 1 \\ + 1 \ 1 \ 1 \cdot 0 \ 1 \ 1 \\ \hline 1 \ 0 \ 1 \ 0 \ 1 \cdot 0 \ 0 \ 0 \ 0 \end{array}$$

In the 2^3 's column $1+1=0$, with a carry of 1 to the 2^2 column.

In the 2^2 's column $0+1=1$, with a carry of 1 to the 2^1 column

In the 2^1 's column $1+0+1=0$, with a carry of 1 to the 1's column

In the 1's column $1+1+1=1$, with a carry of 1 to the 2^3 column

In the 2^5 column $0+1+1=0$, with a carry of 1 to the 2^4 column

In the 4^1 's column $1+1+1=1$, with a carry of 1 to the 8^1 column

In the 8^1 's column $1+1=0$, with a carry of 1 to the 16^1 's column

Binary Subtraction:

The binary subtraction is performed in a manner similar to that in decimal subtraction.

The rules for binary subtraction are

$$0-0=0; \quad 0-1=1, \text{ with a borrow of } 1$$

$$1-0=1; \quad 1-1=0.$$

Ex:

Subtract 111.111_2 from 1010.01_2

So

8421	$\begin{matrix} 1 \\ 2 \\ 2 \\ 2 \\ -3 \end{matrix}$	(column numbers)
1010	• 010	
<hr/>		
111 . 111		
<hr/>		
1001		• 011

In the 2^3 column, a '1' can't be subtracted from a '0'.

so, borrow a '1' from the 2^2 column, making the 2^2 column the '1' borrowed from the 2^2 column becomes 10 in the 2^3 column.

$$\therefore \text{In the } 2^3 \text{ column } 10-1=1.$$

$$\text{In the } 2^2 \text{ column } 10-1=1$$

$$\text{In the } 2^1 \text{ column } 1-1=0$$

$$\text{In the } 1's \text{ column } 1-1=0$$

Now, in the 2^5 column, a '1' can't be subtracted from a '0'.

so, borrow a '1' from the 4^5 column. But in the 4^5 column has a '0'.

$$\text{In the } 2^5 \text{ column } 10-1=1$$

$$\text{In the } 4^5 \text{ column } 1-1=0$$

$$\text{In the } 8^5 \text{ column } 0-0=0$$

Hence, the result is. $(0010.011)_2$

Binary Multiplication:

There are two methods of binary multiplication -
The paper method and the computer method.

Both the methods obey the multiplication rules.

$$0 \times 0 = 0; 0 \times 1 = 0; 1 \times 0 = 0; 1 \times 1 = 1$$

Ex:

Multiply 1101_2 by 110_2

Sol

$$\begin{array}{r} 1101 \\ \times 110 \\ \hline 0000 \\ 1101 \\ 1101 \\ \hline 1001110 \end{array}$$

Binary Division:

Like multiplication, division too can be performed by two methods. The paper method, and the computer method.

Ex:

Divide 101101_2 by 110_2

Sol Divisor 110 can't go in the first three bits of the dividend i.e. 101. So, consider the first 4 bits 1011 of dividend.

$$\begin{array}{r)101101} 110 & (111 \\ 110 \\ \hline 1010 \\ 110 \\ \hline 1001 \\ 110 \\ \hline 110 \\ 110 \\ \hline 000 \end{array}$$

$$\text{Therefore, } 101101 \div 110 = 111.1$$

Complements :

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation.

There are two types of complements for each base 'r' system

1. The r's complement and

2. The $(r-1)$'s complement

* when the value of the base 'r' is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and 10's and 9's complement for decimal numbers.

$(r-1)$'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 $r=10$ and $r-1=9$.

So, the 9's complement of n is $(10^{n-1}) - N$

Now, 10^n represents a number that consists of a single 1 followed by numbers.

10^{n-1} is a number represented by n 9's.

* For example, with $n=4$, we have $10^4 = 10000$

$$\text{and } 10^4 - 1 = 9999$$

It follows that the 9's complement of a decimal no is obtained by subtracting each digit from 9.

Ex: Find the 9's complement of 546700

Q) The 9's complement of 546700 is,

$$\begin{array}{r} 999999 \\ - 546700 \\ \hline 453299 \end{array}$$

For binary numbers is defined as follows

$$r=2 \text{ and } r-1=1$$

So, the 1's complement of N is $(2^n-1)-N$

- * 2^n is represented by a binary number that consists of a 1 followed by n 0's.
- * 2^n-1 is a binary number represented by n 1's.

For example,

$$\text{with } n=4, \text{ we have } 2^4 = (10000)_2$$

$$\text{and } 2^4-1 = (1111)_2$$

Thus, the 1's complement of a binary number is obtained by subtracting each digit from 1.

1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's.

- * Binary complements are used to represent the negative numbers.

The subtraction of binary number is carried out

by taking the complement of the number and adding.

Ex:

① I's complement of 1011 is 0100.

② I's complement of 01111 is 10000.

→ Subtract $(1010)_2$ from $(1100)_2$ by using I's complement

So This is explained step by step

Step₁: The I's complement of 1010 is 0101

Step₂: This is added to 1100

which gives 1100 → I's complement of 1010

$$\begin{array}{r} 0101 \\ + 1100 \\ \hline 10001 \end{array}$$

Step₃: It can be noticed that the operands have 4 bits while the result of addition has 5 bits.

* The extra bit (1) is represented at the MSB is known as the end around carry (eac)

* When we get eac, it means that the difference is positive and the actual value is obtained by adding the end carry to the LSB

* In case of end carry is not there, then it means that the difference is negative and the actual value is obtained by taking the I's complement of this result.

$$\begin{array}{r} 1100 \\ 0101 \\ \hline 10001 \\ \xrightarrow{1} \\ 0010 \end{array}$$

Taking the end around carry and adding to LSB '1'

→ Result

Ex: Subtract 1001.101 from 1100.001

sol: Step₁: is complement of 1001.101 is 0110.010.

Step₂: Adding

$$\begin{array}{r} 1100 \cdot 001 \\ 0110 \cdot 010 \\ \hline 0010 \cdot 011 \\ \hline 0010 \cdot 100 \end{array}$$

is complement of 1001.101

Taking the end carry bit and adding to LSB, '1'.

Step₃:

Result

Ex: Subtract 101 from 1101

sol: The length of 1101 is 4 bits.

The length of 101 is 3 bits.

101 is also write as 0101.

Step₁: is complement of 0101 is 1010

Step₂: Adding 1101 and 1010

$$\begin{array}{r} 1101 \\ 1010 \\ \hline 0111 \\ \hline \end{array}$$

1010 is complement of 0101

Taking end around carry and adding to LSB, '1'.

Step₃: Result

Adding leading zeros for the number with less number of bits. so as to make the lengths of both the numbers same before taking Complement.

r's complement:

The r's complement of an 'n'-digit no 'N' in base 'r' is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$.

* Comparing with the (r-1)'s complement, we note that the r's complement is obtained by adding 1 to (r-1)'s complement.

since, $r^n - N = [(r^{n-1}) - N] + 1$

Ex: Find 10's complement for 2389.

Step 1: First find the 9's complement of 2389

$$\begin{array}{r} 9999 \\ - 2389 \\ \hline 7610 \end{array}$$

9's complement of 2389

Step 2: adding 1 to LSB

$$\begin{array}{r} 7610 \\ + 1 \\ \hline 7611 \end{array}$$

The result

Ex: The 2's complement of 101100 is,

Step 1: 1's complement of 101100
replace 0's by 1's and 1's by 0's $\Rightarrow 010011$

$$\begin{array}{r} 010011 \\ + 1 \\ \hline 010100 \end{array}$$

Step 3: we get the result.

Method ②

→ For 10's complement

The 10's complement of 246700 is

Step 1: Leaving the two zeros unchanged

Step 2: Subtracting 7 from 10 and subtracting other digits from 9.

$$\begin{array}{r} \text{Position of 1's digit after decimal point} \\ \begin{array}{r} 9 \ 9 \ 9 \ 10 \\ - 2 \ 4 \ 6 \ 7 \ 0 \ 0 \\ \hline 7 \ 5 \ 3 \ 3 \ 0 \ 0 \end{array} \end{array}$$

Step 3: The result is 753300.

→ For 2's complement

The 2's complement of 1101100 is

Step 1: By leaving all least significant 0's and first 1 unchanged. And then replace all 1's by 0's and all 0's by 1's in all other higher, significant bits.

$$\begin{array}{r} 1101100 \\ \downarrow \downarrow \downarrow \\ 1101100 \\ \sim \sim \sim 100 \\ 00110100 \end{array}$$

Step 2: The 2's complement of 1101100 is

$$0010100.$$

9's complement method of subtraction

To perform decimal subtraction using the 9's complement method,
① obtain the 9's complement of subtrahend and
② add it to the minuend. call this number the intermediate result.

If there is a carry, it indicates that the answer is positive

③ Add the carry to LSD of this result to get the answer.
carry is called end around carry.

→ If there is no carry, it indicates that the answer is negative and the intermediate result is its 9's complement. Take the 9's complement of this result and place a negative sign in front to get the answer.

Ex: Subtract $745.81 - 436.62$ by using the 9's complement method.

Step 1: Here Minuend is $= 745.81$ and Subtrahend is $= 436.62$

obtain the 9's complement of subtrahend.

$$\begin{array}{r} 999.99 \\ - 436.62 \\ \hline 563.37 \end{array}$$

Step 2: add it to the minuend.

$$\begin{array}{r}
 745.81 \\
 + 563.37 \\
 \hline
 1309.18
 \end{array}
 \text{q's complement}$$

→ end around carry.

Step ③: Add end around carry to the LSD of this result.

$$\begin{array}{r}
 1309.18 \\
 + \rightarrow 1 \\
 \hline
 309.19
 \end{array}$$

Intermediate result
End around carry
Result

The carry indicates that the result is positive.

So answer is $[+309.19]$

Ex : ② $436.62 - 745.81$ by using q's complement method.

Step 1: q's complement of subtrahend is

$$\begin{array}{r}
 999.99 \\
 - 745.81 \\
 \hline
 254.18
 \end{array}$$

Step 2: Add 436.62 Minuend

$$\begin{array}{r}
 436.62 \\
 + 254.18 \\
 \hline
 690.80
 \end{array}$$

q's complement of subtrahend

Step 3: There is no carry indicating that the result is negative. So take q's complement of the intermediate result and put a minus sign

$$\begin{array}{r}
 \therefore \text{The q's complement of } 690.80 \text{ is } 999.99 \\
 - 690.80 \\
 \hline
 309.19
 \end{array}$$

∴ The answer is $[-309.19]$

10's complement method of subtraction

To perform decimal subtraction using the 10's complement method,

1. obtain the 10's complement of the subtrahend and
2. add it to the minuend

3. If there is a carry, ignore it.

The presence of carry indicates that the answer is positive. The result obtained is itself the answer.

→ If there is no carry, it indicates that the answer is negative and the result obtained is its 10's complement. obtain the 10's complement of the result and place a negative sign in front to get the answer.

Ex ① Subtract $2928.54 - 416.73$ using 10's complement method.

Step 1: Obtain the 10's complement of subtrahend

$$\begin{array}{r} 9999.99 \\ 0416.73 \\ \hline 9583.26 \end{array} \rightarrow 9's \text{ complement}$$
$$+1 \rightarrow 10's \text{ complement}$$

Step 2: add 2928.54 9583.27 10's complement of 416.73

$$\underline{\underline{12511.81}} \quad \text{Ignore carry}$$

Step 3: There is a carry indicating that the answer is positive. Ignore the carry.

The answer is $\boxed{2511.81}$

Ex 2 Subtract $416.73 - 2928.54$ by using 10's complement method.

Step 1 Step: obtain the 10's complement of subtrahend

$$\begin{array}{r} 9999.99 \\ - 2928.54 \\ \hline 7071.45 \end{array}$$

q's complement

$$\begin{array}{r} + 1 \\ \hline 7071.46 \end{array}$$

10's complement

Step 2: Add

$$\begin{array}{r} 0416.73 \\ + 7071.46 \\ \hline 7488.19 \end{array}$$

minuend

10's complement

Step 3: There is no carry indicating that the result is negative.

So, take 10's complement of the intermediate result and put a minus sign.

\therefore The 10's complement of 7488.19 is

$$\begin{array}{r} 9999.99 \\ - 7488.19 \\ \hline 2511.80 \end{array}$$
$$\begin{array}{r} + 1 \\ \hline 2511.81 \end{array}$$

. The answer is -2511.81

2's complement method of subtraction

1. obtain the 2's complement of the subtrahend
2. Add minuend and 2's complement of the subtrahend.
3. If there is carry, ignore it. Result is positive.
the result obtained is itself the answer.
4. If there is no carry, indicates that result is negative
and result is obtained is its 10's complement.

Subtraction by using 1's and 2's complement

→ Perform the subtraction with the following unsigned binary numbers

- by taking 2's complement of the subtrahend and
- by taking 1's complement of the subtrahend.

i) $11010 - 10000$

Sol a) 2's complement of 10000 is 10000

2) Add

$$\begin{array}{r} 11010 \\ + 10000 \\ \hline 01010 \end{array}$$

2's complement
ignore the carry

$= +01010$ Result is positive

3) There is carry. Ignore it. The MSB is 0. Hence the answer is positive and is in true binary form

so it is $+01010 = +10$

Sol b) 1's complement of 10000 is 01111

2) Add

$$\begin{array}{r} 11010 \\ + 01111 \\ \hline 01001 \end{array}$$

is complement
end around carry

3) There is carry. It is added to MSB of intermediate result. The MSB is 0.

$$\begin{array}{r} 01001 \\ \swarrow \uparrow \\ 01010 \end{array}$$

Hence the result is positive and is in binary form

so it is $+1010 = +10$

$$\text{i)} 1010100 - 1100000$$

sd) 2's complement method

i) obtain 2's complement of 1100000

$$\begin{array}{r} \text{old 1011 1's complement} \\ + \quad 1 \quad \text{Adding 1} \\ \hline \text{110100 2's complement} \end{array}$$

ii)

3) Ignore the carry.

The MSB is 0.

Hence the answer is positive.

$$\begin{array}{r} 1010100 \\ 1010100 \\ \hline 11000000 \end{array}$$

$$\text{ii)} 11010 - 1101000$$

sd) 2's complement Method

i)

$$\begin{array}{r} 11010 \\ + 10011 \quad \text{2's complement} \\ \hline 1101101 \quad \text{Ignore carry} \end{array} = 26$$

$$= +01101 = +13.$$

There is carry. Ignore it. The MSB is 0. Hence the answer is positive and is in true binary form

$$\text{so it is } +01101 = +13$$

1's complement method

i)

$$\begin{array}{r} 11010 \\ + 10000 \quad \text{1's complement} \\ \hline 1101010 \quad \text{Add carry} \end{array} = 26$$

Result positive = +13.

$$\text{so. it is } +1101 = +13$$

Boolean Algebra:

Boolean Algebra is the algebra of truth tables and operations performed on them.

* In the year 1938, Claude Elwood Shannon developed Boolean algebra for the analysis of two valued switching functions. Shannon was the first to apply Boolean algebra to digital circuitry.

- * Every circuit in a computer or any other electronic device can be designed by using the rules of boolean algebra.
- * It is used in designing circuits which perform different tasks.
- * Boolean algebra performs logical addition & multiplication. No subtraction and division operations are available here.
- * There are also no negative or fractional numbers in boolean algebra.
- * Therefore, boolean algebra reduces complex logic circuits into simpler ones.
- * Binary '0' represents low voltage level while binary '1' represents high voltage level.
- * Symbols '0' and '1' are used to represent open and closed contacts respectively. Hence two element boolean algebra also known as switching algebra can be constructed.

Boolean variable:

In algebra, we define a variable as the one that can take different values at different instants of time.

- * A boolean variable is a variable that is capable of taking only two states or values. These states are represented by 1 or 0. (True or False)

Boolean Expression:

Boolean Expressions are similar to algebraic expressions containing the relationship among several terms. These terms will be have boolean variables and logical operators.

Truth Table:

- * A truth table is the pictorial representation of a boolean expression containing boolean variables for showing results of all possible combinations of input.

- * The input boolean variables will have one of two states '0' or '1', True or False.
- * The combinations of all such input variable states will result into an output value to be zero or one for that Boolean expression.
- * The study of truth table is very useful in simplification of boolean expressions and in designing of electronic circuit.

Laws of Boolean Algebra: Properties

Axioms or postulates of Boolean algebra are a set of logical expressions that we accept without proof and upon which we can build a set of useful theorems.

AND operation

$$0 \cdot 0 = 0 \quad 0 \cdot 1 = 0 \quad 1 \cdot 0 = 0 \quad 1 \cdot 1 = 1$$

OR operation

$$0 + 0 = 0 \quad 0 + 1 = 1 \quad 1 + 0 = 1 \quad 1 + 1 = 1$$

NOT operation

$$\bar{0} = 1 \quad \bar{1} = 0$$

Complementation laws:

Law1: If $A = 0$, then $\bar{A} = 1$

Law2: If $A = 1$, then $\bar{A} = 0$

Law3: $\bar{\bar{A}} = A$

Commutative laws:

Law1: $A + B = B + A$

Law2: $A \cdot B = B \cdot A$

Associative laws:

Law1: $(A + B) + C = A + (B + C)$

Law2: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

Distributive laws:

$$\text{Law}_1 : A \cdot (B+C) = AB + AC$$

$$\text{Law}_2 : A + BC = (A+B)(A+C)$$

$$\text{Law}_3 : A \cdot \overline{AB} = A+B$$

Idempotent laws:

$$\text{Law}_1 : A \cdot A = A$$

$$\text{Law}_2 : A + A = A$$

Identity laws:

$$\text{Law}_1 : A \cdot 1 = A$$

$$\text{Law}_2 : A + 1 = 1$$

Null laws:

$$\text{Law}_1 : A \cdot 0 = 0$$

$$\text{Law}_2 : A + 0 = A$$

Absorption laws:

$$\text{Law}_1 : A + A \cdot B = A$$

$$\text{Law}_2 : A(A+B) = A$$

$$\text{Law}_3 : A(\overline{A}+B) = AB$$

$$\text{Law}_4 : A \cdot B + \overline{B} = A + \overline{B}$$

$$\text{Law}_5 : A \cdot \overline{B} + B = A + B$$

DeMorgan's laws:

$$\text{Law}_1 : \overline{A+B} = \overline{A}\overline{B}$$

$$\text{Law}_2 : \overline{A \cdot B} = \overline{A} + \overline{B}$$

Basic Theorems of Boolean algebra

1. Consensus Theorem (Included factor Theorem)

Theorem 1 :

$$AB + \bar{A}C + BC = AB + \bar{A}C$$

• proof :

$$\begin{aligned} L.H.S &= AB + \bar{A}C + BC \\ &= AB + \bar{A}C + BC (A + \bar{A}) && [\because A + \bar{A} = 1] \\ &= AB + \bar{A}C + ABC + \bar{A}BC && \text{Identity law} \\ &= AB(1+C) + \bar{A}C(1+B) && [1+C = 1] \\ &= AB + \bar{A}C && [1+B = 1] \\ &= R.H.S \end{aligned}$$

∴ Hence proved

* This theorem can be extended to any number of variables.

$$AB + \bar{A}C + BCD = AB + \bar{A}C$$

$$L.H.S = AB + \bar{A}C + BCD$$

$$\begin{aligned} &= AB + \bar{A}C + BC + BCD \\ &= AB + \bar{A}C + BC(1+D) \end{aligned}$$

$$= AB + \bar{A}C + BC$$

$$= AB + \bar{A}C$$

$$= R.H.S$$

$$\therefore AB + \bar{A}C + BCD = AB + \bar{A}C$$

Theorem 2 :

$$(A+B)(\bar{A}+C)(B+C) = (A+B)(\bar{A}+C)$$

Proof :

$$L.H.S = (A+B)(\bar{A}+C)(B+C)$$

$$= (A\bar{A} + AC + \bar{A}B + BC)(B+C)$$

$$= A\bar{A}B + ABC + \bar{A}BB + BBC + A\bar{A}C + ACC + \bar{A}BC + BCC$$

$$\left[\because A \cdot \bar{A} = 0 \quad BB = B \quad CC = C \right]$$

$$= 0 + ABC + \bar{A}B + BC + 0 + AC + \bar{A}BC + BC$$

$$= ABC + \bar{A}B + BC + AC + \bar{A}BC \quad \left[\because BC + BC = BC \right]$$

$$= BC(A + \bar{A}) + BC + \bar{A}B + AC$$

$$= BC + BC + \bar{A}B + AC \quad \left[\because A + \bar{A} = 1 \right]$$

$$= BC + \bar{A}B + AC \quad \left[\because BC + BC = BC \right]$$

$$R.H.S = (A+B)(\bar{A}+C)$$

$$= A\bar{A} + AC + \bar{A}B + BC$$

$$= 0 + AC + \bar{A}B + BC$$

$$= BC + \bar{A}B + AC$$

$$\therefore L.H.S = R.H.S$$

This theorem can be extended to any no. of variables.

For example,

$$(A+B)(\bar{A}+C)(B+C+D) = (A+B)(\bar{A}+C)$$

Transposition Theorem:

Theorem:

$$AB + \bar{A}C = (A+C)(\bar{A}+B)$$

proof:

$$AB + \bar{A}C = (A+C)(\bar{A}+B)$$

$$= A\bar{A} + AB + \bar{A}C + BC$$

$$= 0 + AB + \bar{A}C + BC (A+\bar{A}) \quad [A\bar{A} = 0]$$

$$= AB + \bar{A}C + BC (A+\bar{A}) \quad [A+\bar{A} = 1]$$

$$= AB + \bar{A}C + ABC + \bar{A}BC$$

$$= AB(1+C) + \bar{A}C(1+B)$$

$$[1+C = 1]$$

$$[1+B = 1]$$

Identity law

$$\therefore AB + \bar{A}C = (A+C)(\bar{A}+B)$$

Duality:

In Boolean Algebra we can produce dual by changing all "+" signs to "·" signs
all "·" signs to "+" signs and complementing all '0's and '1's.

The variables are not complemented.

Simplifying the following Boolean Expressions.

① $A\bar{B}C + B + B\bar{D} + A\bar{B}\bar{D} + \bar{A}C$

Eq $A\bar{B}C + B + B\bar{D} + A\bar{B}\bar{D} + \bar{A}C$

$$\Rightarrow A\bar{B}C + B(1 + \bar{D} + A\bar{D}) + \bar{A}C$$

$$[\because 1 + \bar{D} + A\bar{D} = 1]$$

$$\Rightarrow A\bar{B}C + B + \bar{A}C$$

$$1 + \bar{D} + A\bar{D}$$

$$1 + \bar{D}(1 + A)$$

$$1 + \bar{D} = 1$$

$$\Rightarrow C(A\bar{B} + \bar{A}) + B$$

$$\Rightarrow C(\cancel{A\bar{B}} + \cancel{\bar{A}}) + C(\bar{A} + A)(\bar{A} + B) + B$$

$$[\because A\bar{B} + \bar{A} = \bar{A} + \bar{B}]$$

$$\Rightarrow \bar{A}C + \bar{B}C + B \Rightarrow C(\bar{A} + \bar{B}) + B$$

$$\text{distributive law}$$

$$\Rightarrow (B + C)(B + \bar{B}) + C\bar{A}$$

$$\bar{A}\bar{A} + \bar{A}\bar{B} + A\bar{A} + AB$$

$$\bar{A} + \bar{A}\bar{B} + 0 + AB$$

$$\bar{A}(1 + \bar{B}) + AB$$

$$\bar{A} + AB = \bar{A} + \bar{B}$$

$$\Rightarrow B + C$$

Reduce the expression $AB + A\bar{B}C + B\bar{C}$

Eq

$$AB + A\bar{B}C + B\bar{C} = A(B + \bar{B}C) + B\bar{C}$$

$$(B + \bar{B}C = (B + \bar{B})(B + C))$$

$$= A(B + \bar{B})C + B\bar{C}$$

$$= AB + AC + B\bar{C}$$

$$= ABC + AB\bar{C} + AC + B\bar{C}$$

$$= AC(1 + B) + B\bar{C}(1 + A)$$

$$= AC + BC$$



Logic Gates :

Logic gates are the fundamental blocks of digital system.

- * It consists of several inputs and only one output.
- * The output level (logic HIGH or logic LOW) depends upon the combinations of input levels.
- * There are just three basic types of gates -
 1. AND Gate
 2. OR Gate
 3. NOT Gate

* The interconnection of gates to perform a variety of logic operations is called "logic design".

THE UNIVERSAL GATES

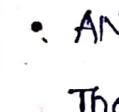
Though logic circuits of any complexity can be realized by using only the three basic gates, there are two universal gates

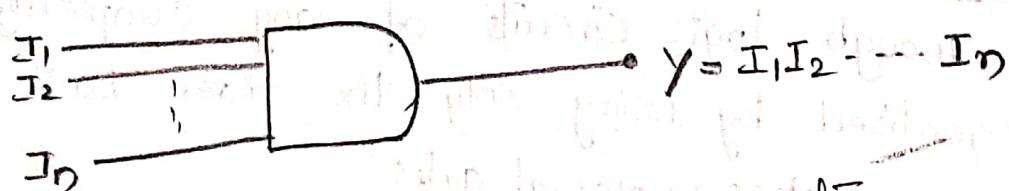
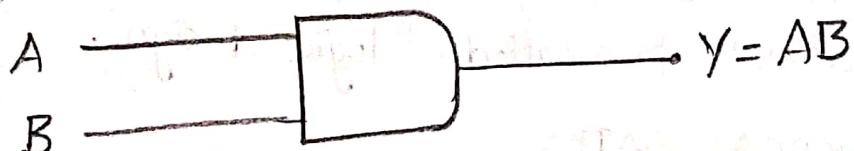
1. NAND Gate
2. NOR Gate

- each of which can also realize logic circuits single handedly.
- The NAND and NOR gates are called Universal Building blocks.
- Both NAND and NOR gates perform all the 3 basic logic functions (AND, OR and NOT).

The AND Gate:

An AND gate has two or more inputs but only one output.

- * The output assumes the logic 1 state, only when each one of its inputs at logic 1 state.
 - * The output assumes the logic 0 state even if one of its inputs is at logic 0 state.
 - * The AND gate is defined as a device whose output is 1, if and only if all its inputs are 1. Hence, the AND gate is also called as all or nothing gate.
 - AND logic gate is denoted by the symbol  (dot)
- The logic diagram for AND gate is,



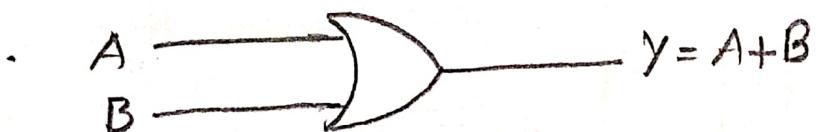
logic symbol of n input AND gate

Input	Output	
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Truth table of AND gate

The OR Gate:

- An OR gate has two or more inputs but only one output.
- * Its output assumes the logic 1 state, even if one of its inputs is in logic 1 state.
 - * Its output assumes the logic 0 state, only when each of its inputs is in logic 0 state.
 - * An OR gate can be defined as a device whose output is 1, even if one of its inputs is 1. Hence an OR gate is also called an any or all gate.
 - * It can also be called an inclusive OR gate.
 - * Logic symbol for the OR operation is



logic symbol of OR gate

The truth table for OR gate is,

Input		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

The NOT Gate : (Inverter)

A NOT gate, also called an inverter, has only one input and only one output.

* It is a device whose output is always the complement of its input.

i.e., the output of a NOT gate assumes the logic 1 state when its input is in logic 0 state and the logic 0 state when its input is in logic 1 state.

* The logic symbol of Inverter is,



logic symbol of NOT gate

* TruthTable for NOT gate is,

INPUT	OUTPUT
A	y
0	1
1	0

TruthTable of NOT gate

* The symbol for NOT operation is '' bar'.

* When the input variable to the NOT gate is represented by A and the output variable by X, the output expression for the output is $X = \bar{A}$.

This is read as x is equal to A bar.

* A discrete NOT gate may be realized using a transistor.

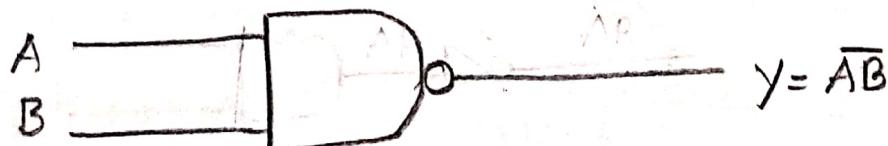
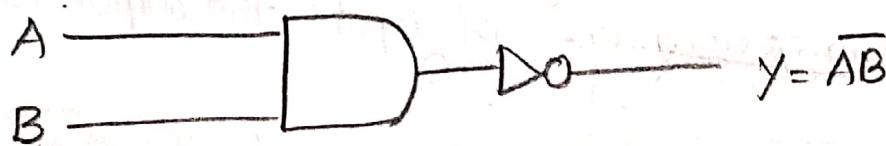
The NAND Gate:

NAND means NOT AND, i.e the AND output is NOTed, so, a NAND gate is a combination of an AND gate and a NOT gate.

* NAND is a contraction of the word NOT-AND.

NAND operation = AND operation + NOT operation

- * The output is logic 0 level, only when each of the inputs assumes a logic 1 level.
- * NAND indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle.
- * Logic diagram for NAND gate is,



- * Truth table for NAND gate is,

Inputs		Output
A	B	y
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate :

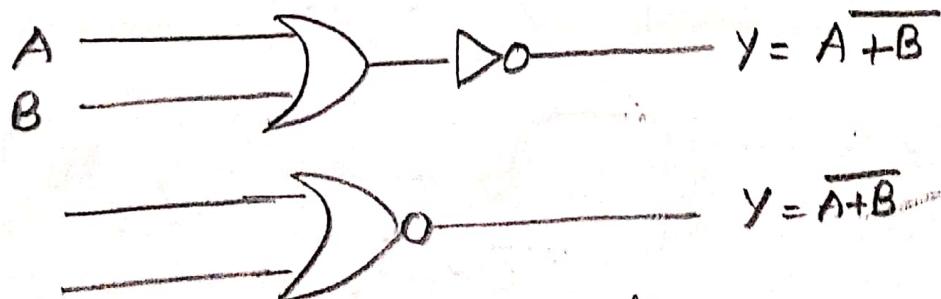
NOR means NOT OR, i.e. the OR output is NOTed.

- * So, NOR gate is a combination of an OR gate and a NOT gate.
- * In fact NOT is a contraction of the word NOT-OR.

$$\boxed{\text{NOR gate} = \text{OR gate} + \text{NOT gate}}$$

- * A NOR gate can also be used as an inverter, by tying all its input terminals together and applying the signal to be converted to the common terminal.
- * NOR gate uses an OR graphic symbol followed by a small circle.
- * The output is logic 1 level, only when each one of its inputs assumes a logic 0 level.

For other combinations of inputs, the output is logic 0 level.



Logic symbol of (NOR gate)

Truth table for NOR gate is,

Inputs		Output
A	B	y
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR Gate :

The exclusive-OR gate has a graphic symbol similar to the OR gate, except for the additional curved line on the input side.

- * The X-OR gate has two inputs but one output.
- * The output is logic high when the inputs are not equal.
- * The output is logic low when the inputs are equal to either logic HIGH or logic LOW.
- * X-OR gate is also called as inequality detector or anti-coincidence gate.
- * X-OR gate is a logic gate that performs modulo addition of its inputs.
- * X-OR operator is denoted by the symbol \oplus .



$$Y = A \oplus B = AB + \bar{A}\bar{B}$$

Logic symbol of 2-input X-OR gate

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Truth table of X-OR gate

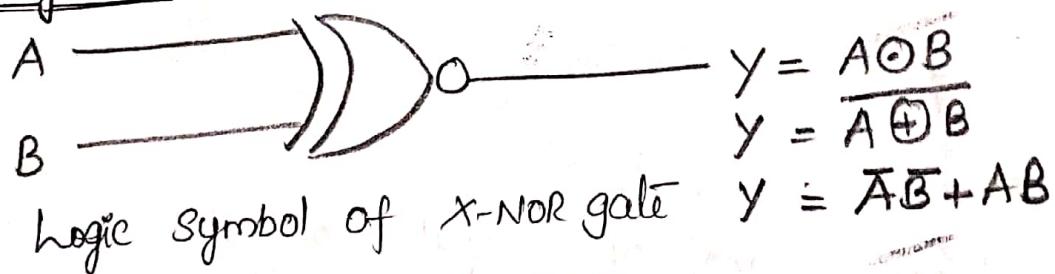
Exclusive-NOR Gate (X-NOR)

XNOR is a logic gate that performs the negation of X-OR operation.

$$\text{XNOR operator} = \text{X-OR operator} + \text{NOT operator}$$

- * The output of the X-NOR gate is logic 1 (i.e high) when both the inputs have the same values i.e. either both are high or both are low
- * The output of X-NOR logic gate is low (i.e logic 0) if any one of the inputs is either low(0) or high(1)
- * X-NOR is also called as equality detector.
- * X-NOR is indicated by an X-OR graphic symbol followed by a small circle.
- * X-NOR operator is denoted by the symbol ' \ominus '.

Logic symbol



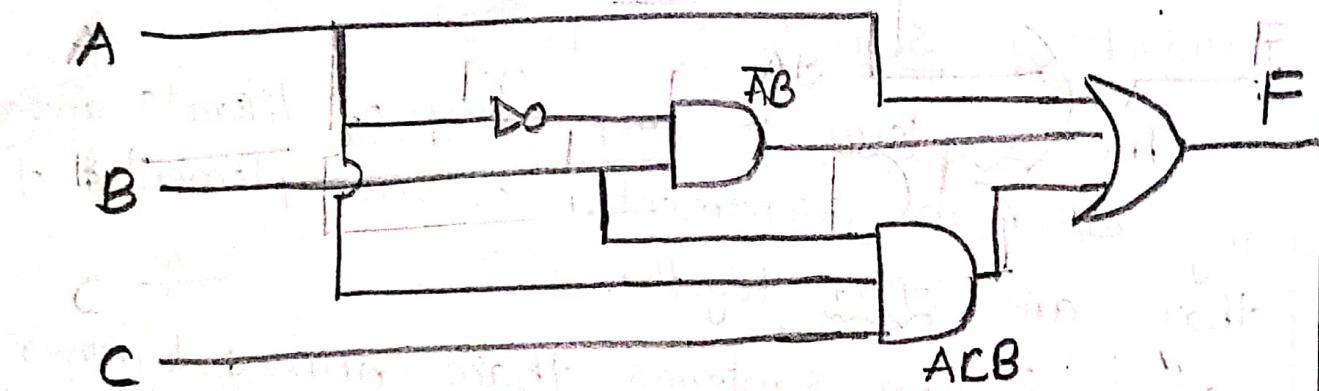
Inputs		Output
A	B	y
0	0	1
0	1	0
1	0	0
1	1	1

Truth table of X-NOR gate

Canonical and Standard Forms :

$F = A + \overline{A}B + ACB$, in this A, B, C are known as literals.

- * A Literal is complemented or uncomplemented variable
- * To Complement any Boolean expression using logic gates the literals are designated as inputs to the logic gate.
- * The implementation of $F = A + \overline{A}B + ACB$ using logic gates as,



- * To implement a logic function with less number of gates we have to minimize literals and the number of terms.

usually, literals and terms are arranged in one of the two standard forms.

1. Sum of products form
2. product of sum form.

Sum of products form (Sop) :

- * The product term is any group of literals appearing either in complemented or uncomplemented form, that are ANDed together.
- * Two or more product terms are ORed together, to form a sum of products expression.

Ex :-

$$1. F = A\bar{B}C + \bar{B}CD + \bar{A}CD + A\bar{B}\bar{D}$$

$$2. F = A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{D} + ABC + BCD$$

$$3. F = AB + \bar{A}CB + AC + BD$$

where, $AB, \bar{A}CB, AC, BD$ are product terms.

Product of sum form (Pos) :

- The sum term is any group of literals appearing either in complemented or uncomplemented form, that are ORed together.

- * Two or more sum terms are ANDed together to form a product of sums expression.

Ex :-

$$1. F = (A + \bar{B} + \bar{C}) (A + B + \bar{C} + \bar{D}) (C + \bar{D})$$

$$2. F = (B + C) (\bar{A} + \bar{C} + D) (A + \bar{B} + D)$$

$$3. F = (\bar{A} + B) (A + B + C) (\bar{C} + B) (\bar{B})$$

where, $\bar{A} + B, A + B + C, \bar{C} + B, \bar{B}$ are sum terms.

Canonical sum of products :

A sum of products expression is referred to as Canonical sum of products or standard sum of products expression if every product term involves every literal or its complement.

Ex:

$$1. F = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D}$$

$$2. F = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + A\bar{B}\bar{C}\bar{D}$$

$$3. F = \bar{A}BC + \bar{A}\bar{B}C + AC\bar{B} + ABC \text{ where}$$

$\bar{A}BC, \bar{A}\bar{B}C, AC\bar{B}, ABC$ are product terms involves all the literals A, B and C.

Canonical product of sum :

The product of sum expression is referred to as canonical product of sums or standard product of sums expression. If every sum term involves every literal or its complement.

Ex:

$$1. F = (\bar{A} + \bar{B} + \bar{C} + \bar{D}) (\bar{A} + B + C + D) (A + B + \bar{C} + D)$$

$$2. F = (A + \bar{B} + \bar{C} + \bar{D}) (\bar{A} + B + \bar{C} + \bar{D}) (\bar{A} + B + C + D)$$

$$3. F = (A + \bar{B} + C + D) (\bar{A} + B + \bar{C} + \bar{D}) (A + \bar{B} + \bar{C} + D) (A + B + C + D)$$

Where, $\bar{A} + \bar{B} + C + D$, $\bar{A} + B + \bar{C} + \bar{D}$, $A + \bar{B} + \bar{C} + D$,

$A + B + C + D$ are sum terms involving all literals A, B, C and D.

Minterm:

A product term which contains each of 'n' variables as factors in either complemented or uncomplemented form is called a Minterm.
* It is represented by ' m '.

Maxterm:

A sum term which contains each of 'n' variables as factors in either complemented or uncomplemented form is called a Maxterm.
* It is represented by ' M '.

Conversion from POS to standard POS:

1. Examine each term
2. In each term, check the variables that do not occur, then add the "variables and its complement".

Ex: In ' $x+y$ ', ' z ' is not present.

Add $z\bar{z}$ to term and multiply it

$$\Rightarrow (x+y) + z\bar{z}$$

$$\Rightarrow x+y+z\bar{z}$$

$$\Rightarrow (x+y+z)(x+y+\bar{z})$$

3. Eliminate redundant terms.

Convert pos into canonical pos. form

1. $f = (A+B)(A+C)(B+\bar{C})$

Given that,

$$f = (A+B)(A+C)(B+\bar{C})$$

We know that

$$A\bar{A} = 0, B\bar{B} = 0, C\bar{C} = 0$$

$$\therefore f = (A+B+C\bar{C})(A+\bar{B}+C)(A\bar{A}+B+\bar{C})$$

$$= (A+B+C)(A+\bar{B}+C)(A+\bar{B}+C)(A+B+\bar{C})(\bar{A}+B+\bar{C})$$

$$\therefore f = (A+B+\bar{C})(A+B+C)(A+\bar{B}+C)(\bar{A}+B+\bar{C})$$

2. $f = (\bar{A}+\bar{B})(A+\bar{B})(A+B)$

Given that,

$$f = (\bar{A}+\bar{B})(A+\bar{B})(A+B)$$

We know that

$$A\bar{A} = 0, B\bar{B} = 0, C\bar{C} = 0$$

$$\therefore f = (\bar{A}+\bar{B}+C\bar{C})(A+\bar{B}+C\bar{C})(A+\bar{B}+C\bar{C})$$

$$= (\bar{A}+\bar{B}+C)(\bar{A}+\bar{B}+C)(A+\bar{B}+C)(A+\bar{B}+C)(A+B+\bar{C})$$

$$(A+B+\bar{C})$$

$$\therefore f = (\bar{A}+\bar{B}+C)(\bar{A}+\bar{B}+C)(A+\bar{B}+C)(A+\bar{B}+C)$$

$$(A+B+C)(A+B+\bar{C})$$

Conversion from SOP to standard SOP form:

To convert SOP to canonical SOP form the following steps are required.

1. Examine each term, if it is a minterm, retain it and continue to the next term.
2. In each product which is not a minterm, check the variables that do not occur, for each x_i that does not occur, multiply by product by $(x_i + \bar{x}_i)$. For ex, for a 3-variable function (x, y, z) if a term contains only x, y which is not a minterm multiply xy with $(z + \bar{z}) = xyz + xy\bar{z}$.
3. Multiply out all products and eliminate redundant terms.

Minimization of switching functions:

- A switching function can usually be represented by a no. of expressions.
- * Our intention is to develop a procedure for obtaining minimal expression.
 - * We did simplifications, by using algebraic manipulations. Even for 4 or 5 variables this method is ineffective.
 - * The presented map method is very effective for hand simplifications of expressions upto 5 or 6 variables, while the tabulation procedure is suitable for machine computation and yields minimal expressions.

Sop into canonical sop form

→ Convert Sop into canonical Sop form:

1. $f = \bar{A}\bar{B} + \bar{C} + ABC$.

sd We know that $A + \bar{A} = 1$; $B + \bar{B} = 1$; $C + \bar{C} = 1$

In $\bar{A}\bar{B}$ term 'c' is not present,

In \bar{C} term A, B are not present

so, add these literals.

$$f = \bar{A}\bar{B}(C + \bar{C}) + (A + \bar{A})(B + \bar{B})\bar{C} + ABC$$

$$= \bar{ABC} + \bar{ABC}\bar{C} + ABC\bar{C} + \bar{ABC} + \bar{ABC}\bar{C} + ABC$$

$$= \bar{ABC} + \bar{ABC}\bar{C} + ABC\bar{C} + A\bar{B}\bar{C} + \bar{ABC} + ABC$$

Eliminate redundant term \bar{ABC} .

$$\therefore f = \bar{ABC} + \bar{ABC}\bar{C} + ABC\bar{C} + A\bar{B}\bar{C} + \bar{ABC} + ABC$$

2. $f = A\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}$.

sd We know that $A + \bar{A} = 1$, $B + \bar{B} = 1$, $C + \bar{C} = 1$

In $A\bar{C}$ term, B literal is not present

In $\bar{A}\bar{B}\bar{C}$ term, C literal is not present

In $\bar{A}\bar{B}$ term, C literal is not present

So, add missing literals

$$f = A(B + \bar{B})\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}(C + \bar{C})$$

$$= \bar{ABC} + \bar{ABC}\bar{C} + \bar{ABC} + \bar{ABC}\bar{C} + \bar{ABC}$$

$$\therefore f = \bar{ABC} + \bar{ABC}\bar{C} + \bar{ABC} + \bar{ABC}\bar{C}$$

Difference between sop and pos

Sum of products

1. A way of representing boolean expression as sum of product terms

2. sop uses minterms. Minterm is a product of boolean variables either in normal form or complemented form.

3. It is sum of minterms. Minterms are represented as 'm'

4. sop is formed by considering all the minterms, whose output is HIGH(1) while writing minterms for sop,

5. while writing minterms for sop, input with value 1 is considered as the variable itself and input with value 0 considered as complement of the input.

product of sum

1. A way of representing boolean expressions as product of sum terms

2. pos uses maxterms. Maxterm is a sum of boolean variables either in normal form or complemented form.

3. It is product of maxterms. Maxterms are represented as 'M'

4. pos is formed by considering all the maxterms, whose output is Low(0).

5. While writing maxterms for pos, input value 1 is considered as the complement and input with value 0 is considered as the variable itself.

Karnaugh Mapping:

Karnaugh mapping is a method of plotting a boolean function and is used to simplify the expression making use of a truth-table.

- * To draw a Karnaugh map, a Boolean function must be written in a sum of products form or in the Minterm form.
- * If it is not in this form, then we should use the truthtable to convert it to the sum of the products form.

How to plot a Karnaugh Mapping?

A Karnaugh mapping is a diagram consists of squares. Each square of the map represents a minterm. i.e a term in the sum of the product form.

- * Any logic expression can be written as a sum of products, i.e sum of minterms. Therefore, a logic expression can easily be represented on a Karnaugh map.

A Karnaugh map for n variables is made up of 2^n squares.

Each square designates a product term of a Boolean Expression.

For product terms which are present in the expression 1's are written in the corresponding squares.

0's are written in those squares which correspond to product terms not present in the expression. For clarity of the map, writing of 0's can be omitted.

so, blank squares indicate that they contain 0's.

TWO-variable K-map :

A two-variable expression can have $2^2 = 4$ possible combinations of the input variables A and B.

- Each of these combinations, $\bar{A}\bar{B}$, $\bar{A}B$, $A\bar{B}$ and AB (in the SOP form) is called a minterm.
- Instead of these, representing the minterms in terms of input variables, using the notation the minterms may be represented in terms of their decimal designations.

m_0 for $\bar{A}\bar{B}$

m_1 for $\bar{A}B$

m_2 for $A\bar{B}$

m_3 for AB

Assuming that A represents the MSB. The letter 'm' stands for minterm and the subscript represents the decimal designation of minterm.

	B 0	1	
A 0	m_0 $\bar{A}\bar{B}$	m_1 $\bar{A}B$	
1	m_2 $A\bar{B}$	m_3 AB	

Fig: The minterms of a two variable K-map.

- Each sum term in the standard POS expression is called a maxterm. A function in two variables (A, B) has $2^2 = 4$ possible maxterms, $A+B$, $A+\bar{B}$, $\bar{A}+B$ and $\bar{A}+\bar{B}$. They are represented

M_0 for $A+B$

M_1 for $A+\bar{B}$

M_2 for $\bar{A}+B$

M_3 for $\bar{A}+\bar{B}$

The uppercase letter M stands for maxterm and its subscript denotes the decimal designation of that maxterm.

- For mapping a POS expression on to the K-map, 0's are placed in the squares corresponding to the maxterms which are present in the expression. 1's are placed (or no entries are made) in the squares corresponding to the maxterms which are not present in the expression.
- The decimal designation of the squares for maxterms is the same as that for the minterms.

		0	1
A	B	m_0	m_1
		$A+B$	$A+\bar{B}$
1	0	m_2	m_3
	1	$\bar{A}+B$	$\bar{A}+\bar{B}$

Maxterms of a 2-variable K-map.

Ex: Reduce the expression $f = \bar{A}\bar{B} + \bar{A}B + AB$ using mapping

Sol: The Given expression is

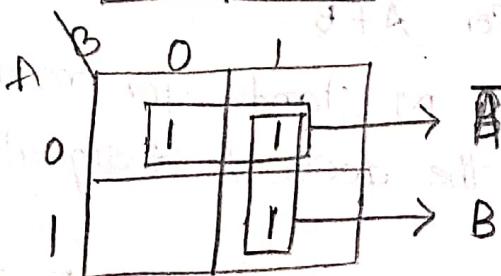
$$f = \bar{A}\bar{B} + \bar{A}B + AB$$

Expressed in terms of minterms is

$$f = \Sigma m(0, 1, 3)$$

k-map for this function is

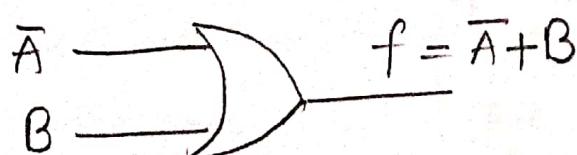
A\B	0	1
0	m ₀	m ₁
1	m ₂	m ₃



In one square, A is constant as a '0' but B varies from a '0' to a '1', and in the other 2-square, B is constant as a '1' but A varies from a '0' to a '1'.

So, the required expression is $\bar{A} + B$.

It requires two gate inputs for realization.



Logic diagram for $f = \bar{A} + B$

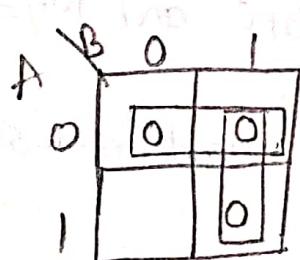
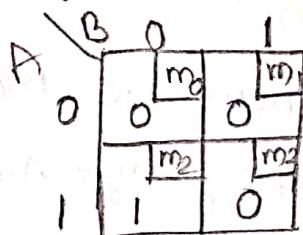
→ Reduce the expression $f = (A+B)(A+\bar{B})(\bar{A}+\bar{B})$ using mapping

Given expression is $f = (A+B)(A+\bar{B})(\bar{A}+\bar{B})$

Expressed in terms of minterms PS

$$f = \prod M(0,1,3)$$

The K-map for f and its reduction is



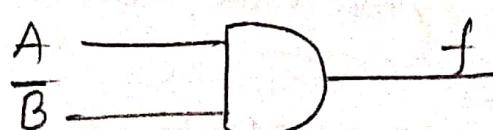
In the given expression, the minterm M_2 is absent. This is indicated by a '0' on the K-map.

So, the required expression is $\bar{A}\bar{B}$.

The corresponding SOP expression is $\sum m_2$ or $\bar{A}\bar{B}$.

This realization is the same as that of the POS form.

It requires two gate inputs, for re



Three-Variable K-map:

A function in three variables (A, B, C) expressed in the standard SOP form have $2^3 = 8$ possible combinations.

$\bar{A}\bar{B}\bar{C}$, $\bar{A}\bar{B}C$, $\bar{A}BC$, $A\bar{B}\bar{C}$, $A\bar{B}C$, $AB\bar{C}$ and ABC .

Each of these combinations designated by $m_0, m_1, m_2, m_3, m_4, m_5, m_6$ and m_7 respectively, is called a minterm.

- A is the MSB of the minterm designator and C is the LSB.

In the standard POS form,

the 8 possible combinations are $M_0 = A + B + C$,

$M_1 = A + B + \bar{C}$, $M_2 = A + \bar{B} + C$, $M_3 = A + \bar{B} + \bar{C}$, $M_4 = \bar{A} + B + C$

$M_5 = \bar{A} + B + \bar{C}$, $M_6 = \bar{A} + \bar{B} + C$ and $M_7 = \bar{A} + \bar{B} + \bar{C}$.

is called a maxterm.

- A is the MSB of the maxterm designator and C is the LSB.

A 3-variable K-map, has $8 = 2^3$ squares or cells, each square on the map represents a minterm or maxterm. A small number on the top right corner of each cell indicates the minterm or maxterm designation.

		BC	00	01	11	10
		A	0	1	3	2
0	0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$A\bar{B}\bar{C}$	
	1	$\bar{A}\bar{B}C$	$A\bar{B}\bar{C}$	ABC	$A\bar{B}C$	

Minterms

		BC	00	01	11	10
		A	0	1	3	2
0	0	$A + B + C$	$A + B + \bar{C}$	$A + \bar{B} + C$	$A + \bar{B} + \bar{C}$	
	1	$\bar{A} + B + C$	$\bar{A} + B + \bar{C}$	$\bar{A} + \bar{B} + C$	$\bar{A} + \bar{B} + \bar{C}$	

Maxterms

Ex: Map the expression.

$$f = \bar{A}\bar{B}C + A\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + ABC$$

Sol: Given expression is.

$$f = \bar{A}\bar{B}C + A\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + ABC$$

In the given expression, the minterms are

$$\bar{A}\bar{B}C = 001 = m_1$$

$$A\bar{B}C = 101 = m_5$$

$$\bar{A}BC = 010 = m_2$$

$$A\bar{B}\bar{C} = 110 = m_6$$

$$ABC = 111 = m_7$$

so, the expression $f = \sum m(1, 2, 5, 6, 7)$

The corresponding K-map is.

		BC	00	01	11	10
		A	0	1	3	2
0	0	(m ₀)	(m ₁)	(m ₃)	1(m ₂)	
	1	0 ₄ (m ₄)	1 ₅ (m ₅)	1 ₇ (m ₇)	1 ₆ (m ₆)	

		BC	00	01	11	10
		A	0	1	1	1
0	0					
	1			1	1	1

K-map in SOP form.

→ Map the expression.

$$f = (A+B+C)(\bar{A}+B+\bar{C})(\bar{A}+\bar{B}+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+\bar{B}+C)$$

∴ The Given expression is,

$$f = (A+B+C)(\bar{A}+B+\bar{C})(\bar{A}+\bar{B}+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+\bar{B}+C)$$

Maxterms are,

$$A+B+C = 000 = M_0$$

$$\bar{A}+B+\bar{C} = 101 = M_5$$

$$\bar{A}+\bar{B}+\bar{C} = 111 = M_7$$

$$A+\bar{B}+\bar{C} = 011 = M_3$$

$$\bar{A}+\bar{B}+C = 110 = M_6$$

So, the expression $f = \prod M(0, 3, 5, 6, 7)$

The mapping of the expression is,

		BC	00	01	11	10	
		A	0	M_0	M_1	M_3	M_2
		1	M_4	M_5	M_7	M_6	

		BC	00	01	11	10	
		A	0	0		0	
		1		0	0	0	0

K-map in pos form

Four variable K-map :

A four variable (A, B, C, D) expression can have $2^4 = 16$ possible combinations of input variables such as $\bar{A}\bar{B}\bar{C}\bar{D}$, $\bar{A}\bar{B}\bar{C}D$, --- $A\bar{B}CD$ with minterm designations m_0, m_1, \dots, m_{15} respectively, in the SOP form and $A+B+C+D$, $A+B+C+\bar{D}$, --- $\bar{A}+\bar{B}+\bar{C}+D$ with maxterm designations M_0, M_1, \dots, M_{15} respectively, in the POS form.

- A 4-variable K-map has $2^4 = 16$ squares or cells and each square on the map represents either a minterm or a maxterm.

$AB \backslash CD$	00	01	11	10
00	$\bar{A}\bar{B}\bar{C}\bar{D}$ (m_0)	$\bar{A}\bar{B}\bar{C}D$ (m_1)	$\bar{A}\bar{B}CD$ (m_3)	$\bar{A}\bar{B}C\bar{D}$ (m_2)
01	$\bar{A}B\bar{C}\bar{D}$ (m_4)	$\bar{A}B\bar{C}D$ (m_5)	$\bar{A}BCD$ (m_7)	$\bar{A}BC\bar{D}$ (m_6)
11	$A\bar{B}\bar{C}\bar{D}$ (m_{12})	$A\bar{B}\bar{C}D$ (m_{13})	$ABC\bar{D}$ (m_{15})	$ABC\bar{D}$ (m_{14})
10	$A\bar{B}\bar{C}\bar{D}$ (m_8)	$A\bar{B}\bar{C}D$ (m_9)	$A\bar{B}CD$ (m_{11})	$A\bar{B}C\bar{D}$ (m_{10})

Minterms of a 4-variable K-map.

The binary number designations of the rows and columns are in the Gray code.

Here 11 follows 01 and 01 follows 10 follows 11.
This is called adjacency ordering.

- The binary numbers along the top of the map indicate the conditions of C and D along any column and binary numbers along the left side indicate the conditions of A and B along any row.
- The numbers in the top right corners of the squares indicate the minterm or maxterm designations as usual.

AB CD	00	01	11	10
00	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$
01	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$
11	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$
10	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$	$A+B+C+D$

maxterms of a four-variable K-map.

- squares which are physically adjacent to each other or which can be made adjacent by wrapping the map around from left to right or top to bottom can be combined to form bigger squares.
- The bigger squares (2 squares, 4 squares, 8 squares, etc.) must form either a geometric square or rectangle.

→ Using K-map, find the minimal sum of products and product of sums that are equivalent to $E(w, x, y, z) = \prod M(1, 3, 4, 7, 10, 13, 14, 15)$

Sol Given that,

$$E(w, x, y, z) = \prod M(1, 3, 4, 7, 10, 13, 14, 15)$$

Given function is in pos-form (product of sum)

K-map for this function is,

w\z	00	01	11	10	
00	0	0			$w+x+\bar{z}$
01	0	0			$\bar{x}+\bar{y}+\bar{z}$
11	0	0			$w+\bar{x}+z$
10	0				$\bar{w}+\bar{y}+z$

$$f = (w+x+\bar{z})(w+\bar{x}+y+z)(\bar{x}+\bar{y}+\bar{z})(\bar{w}+\bar{x}+\bar{z})(\bar{w}+\bar{y}+z)$$

Sop form is:

$$E(w, x, y, z) = \sum m(0, 2, 5, 6, 8, 9, 11, 12)$$

w\z	00	01	11	10	
00	1				$\bar{w}\bar{x}\bar{z}$
01		1			$\bar{w}y\bar{z}$
11		1			$w\bar{y}\bar{z}$
10	1		1	1	$w\bar{x}z$

$$\therefore f = \bar{w}\bar{x}\bar{z} + \bar{w}x\bar{y}z + \bar{w}y\bar{z} + w\bar{y}\bar{z} + w\bar{x}z.$$

Reduce using mapping the expression
 $f = \text{TM}(2, 8, 9, 10, 11, 12, 14)$ and implement the real
 minimal expression in Universal logic.

Sol: The given expression is in the form of SOP

$$f = \Sigma m(0, 1, 3, 4, 5, 6, 7, 13, 15)$$

K-map for SOP form is,

		CD		00	01	11	10
		AB		00	1	1	1
		00		1	1	1	
		01		1	1	1	1
11				1	1		
10							

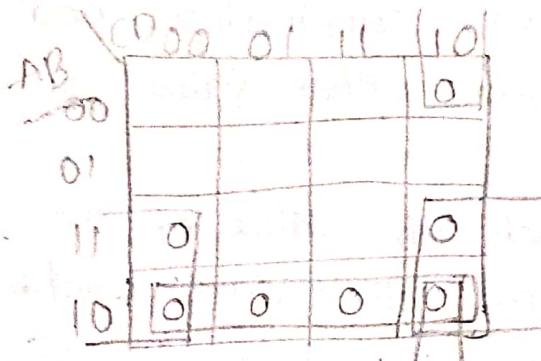
$$f_{\min} = \overline{AC} + \overline{AD} + \overline{AB} + BD$$

f_{\min} SOP form requires 12 gate inputs.

1. There are no isolated 1's
2. There are no 1's which can be combined only into 2-squares - So make no 2-squares.
3. m_6 can form a 4-square with m_4, m_5, m_7 .
Make it and read it as \overline{AB} .
4. m_{15} can form a 4-square with m_5, m_7, m_{13} .
Make it and read it as BD .
5. m_0 can form a 4-square with m_1, m_3, m_5 .
Make it and read it as \overline{AC} .
6. Only m_9 is left. It can make a 4-square with m_1, m_5, m_7 . Make it and read it as \overline{AD} .
7. Write all the product terms in SOP form.

So, the minimal SOP expression is

$$f_{\min} = \bar{A}B + BD + \bar{A}\bar{C} + \bar{A}D.$$



pos K-map

$$f_{\min} = (\bar{A}+B)(\bar{A}+D)(B+\bar{C}+D)$$

In the pos K-map,

1. There are no isolated 0's.

2. M₂ can form a 2-square with only M₁₀.

Make it and read it as (B+̄C+D)

3. M₀ and M₈ can form a 4-square with M₈ M₁₀

Make it and read it as (̄A+D)

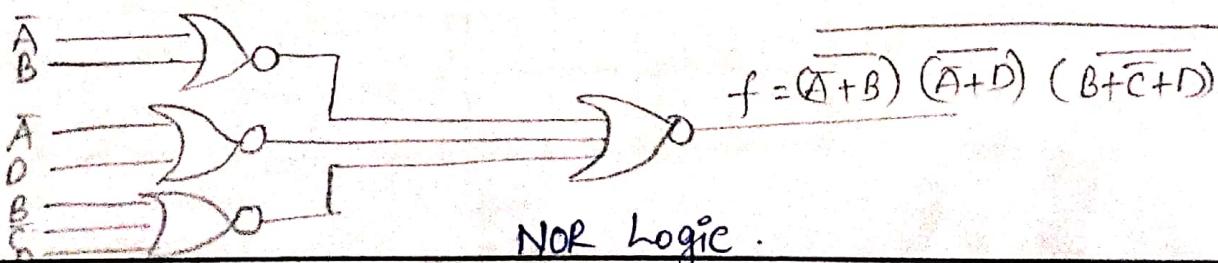
4. M₉ and M₁₁ can form a 4-square with M₈ M₁₀

Make it and read it as (̄A+B)

5. Write all the sum terms in pos form

The implementation of the minimal expression using

NOR gates is $f_{\min} = \underline{(\bar{A}+B)} \underline{(\bar{A}+D)} \underline{(B+\bar{C}+D)}$
 $= \underline{(\bar{A}+B)} \underline{(\bar{A}+D)} \underline{(B+\bar{C}+D)}$



Don't Care Combinations:

So far the functions considered have been completely specified for every combination of the variables. There exist situations, however, where while a function is to assume a value 1 for some combinations and the value '0' for others, it may assume either value for a number of combinations.

- * Combinations for which the value of the function is not specified are called don't care combinations.
- * The value of function for such combinations is denoted by a ' ϕ ' or 'd'.

For example,

The d's can be used as 1's to form a subcube consisting of four 1's.

W ^x	Y ^z	00	01	11	10
00	1	d			
01	1	d			
11					
10					

→ whereas the d's need not to be used.

W ^x	Y ^z	00	01	11	10
00	1	d		1	
01	1	d		1	
11					
10					

Reduce using mapping the expression

$f = \Sigma m(0, 1, 2, 3, 5, 7, 8, 9, 10, 12, 13)$ and implement the real minimal expression in universal logic.

The given expression function is

$$f = \Sigma m(0, 1, 2, 3, 5, 7, 8, 9, 10, 12, 13)$$

K-map for given function is (Sop K-map)

		CD	00	01	11	10
		AB	00	11	11	11
		00	1			
		01		1	1	
		11	1	1		
		10	1	1	0	1

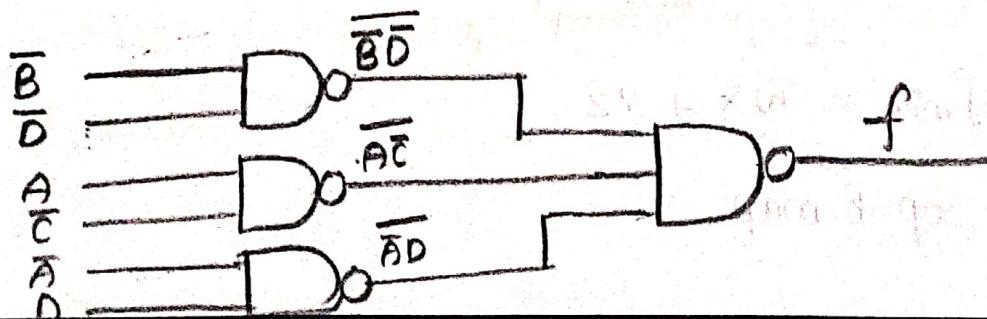
$$f_{min} = \overline{AD} + A\overline{C} + \overline{BD} = \overline{AD} + A\overline{C} + \overline{BD}$$

K-map for pos form is,

$$f = \prod M(4, 6, 11, 14, 15)$$

		CD	00	01	11	10
		AB	00			
		00		○		
		01			○	○
		11			○	○
		10			○	

$$f_{min} = (A + \overline{B} + D) (\overline{A} + \overline{B} + \overline{C}) (\overline{A} + \overline{C} + \overline{D})$$

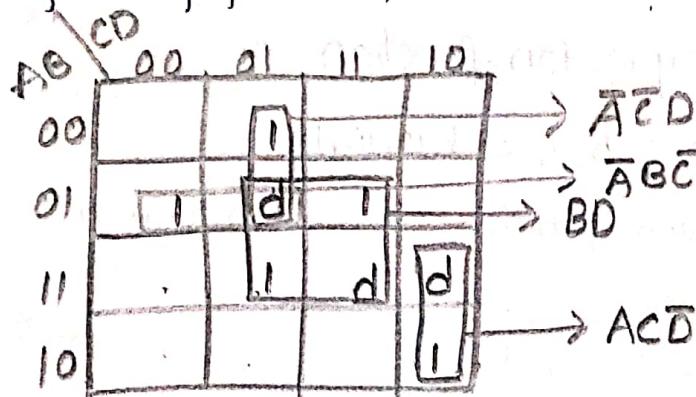


Minimize the following expressions using K-map:

1) $F(A, B, C, D) = \sum m(1, 4, 7, 10, 13) + \sum d(5, 14, 15)$

Sol The given expressions are in SOP form.

K-map for SOP form is,



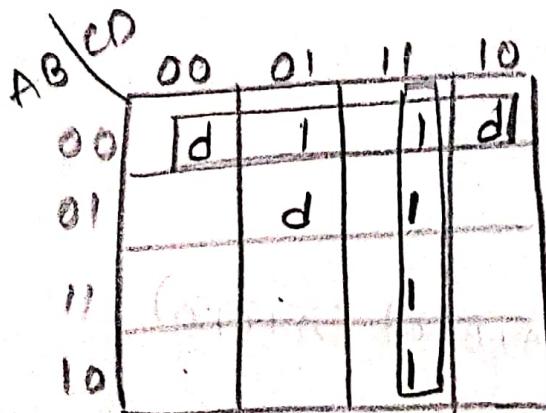
(K-map) SOP k-map

$$f_{min} = \bar{A}\bar{C}D + \bar{A}\bar{B}\bar{C} + BD + A\bar{C}\bar{D}$$

2) $F(w, x, y, z) = \sum m(1, 3, 7, 11, 15) + \sum d(0, 2, 5)$

Sol The given expressions are in SOP form

K-map for this SOP form is,



$$f_{min} = \bar{w}\bar{x} + yz$$

SOP k-map

Five-variable K-map :

A five-variable (A, B, C, D, E) expression can have $2^5=32$ possible combinations of input variables such as $\bar{A}\bar{B}\bar{C}\bar{D}\bar{E}$, $\bar{A}\bar{B}\bar{C}\bar{D}E$, ---, $ABCDE$, with minterm designations m_0, m_1, \dots, m_{31} respectively, in SOP form.

- $A+B+C+D+E$, $A+B+C+D+E$, ---, $\bar{A}+\bar{B}+\bar{C}+\bar{D}+\bar{E}$, with maxterms designations M_0, M_1, M_{31} - respectively, in POS form.
- The 32 squares of the K-map are divided into 2 blocks of 16 squares each.
- Left block represents minterms from m_0 to m_{15} in which A is a '0' and right block represents minterms from m_{16} to m_{31} in which A is 1.
- The five-variable K-map may contain 2-squares, 4-squares, 8-squares, 16-squares or 32-squares involving these two blocks.
- Squares are also considered adjacent in these two blocks, if when superimposing one block on top of another, the squares coincide with one another.

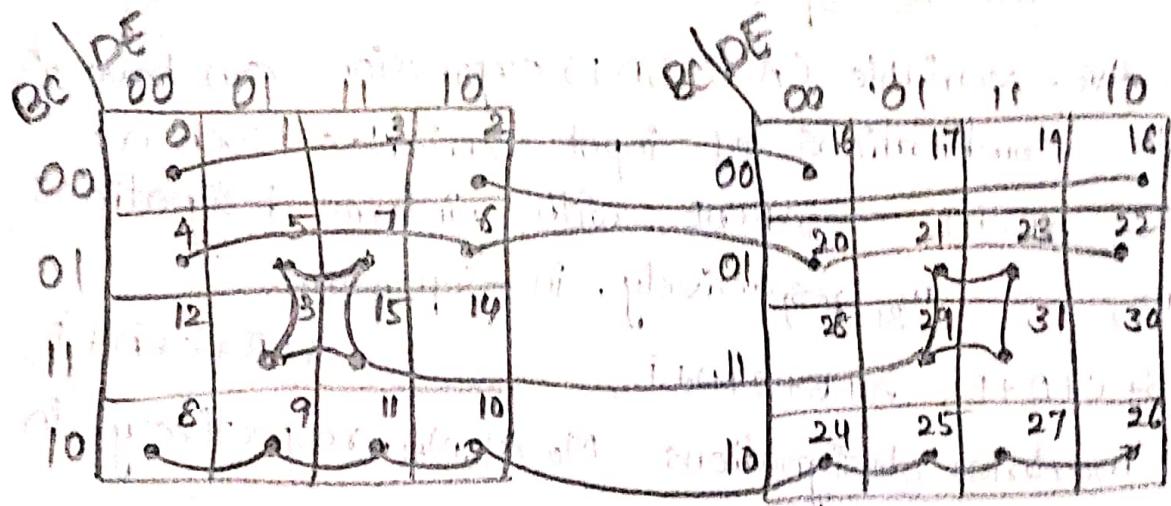
$$a) m_0, m_{16} = \bar{B}\bar{C}\bar{D}\bar{E}$$

$$b) m_2, m_{18} = \bar{B}\bar{C}DE$$

$$c) m_4, m_6, m_{20}, m_{22} = \bar{B}CE$$

$$d) m_5, m_7, m_{13}, m_{15}, m_{21}, m_{23}, m_{27}, m_{31} = CE$$

$$e) m_8, m_9, m_{10}, m_{11}, m_{24}, m_{25}, m_{26}, m_{27} = \bar{B}\bar{C}$$



Some possible groupings in a five variable K-map.

Ex :

Reduce the following expression to Sop and pos forms using mapping :

$$f = \sum(0, 2, 3, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21, 26, 27)$$

Sol: The given expression in pos form is

$$f = \prod(1, 4, 5, 6, 7, 8, 9, 14, 15, 22, 23, 24, 25, 28, 29, 30, 31)$$

The real minimal expression is the minimal of the Sop and pos forms.

In the Sop form

1. There are no isolated 1's.

2. m_{12} can go only with m_{13} , form a square which is read as $\bar{A}BC\bar{D}$.

Subject:
Faculty:
Topic:

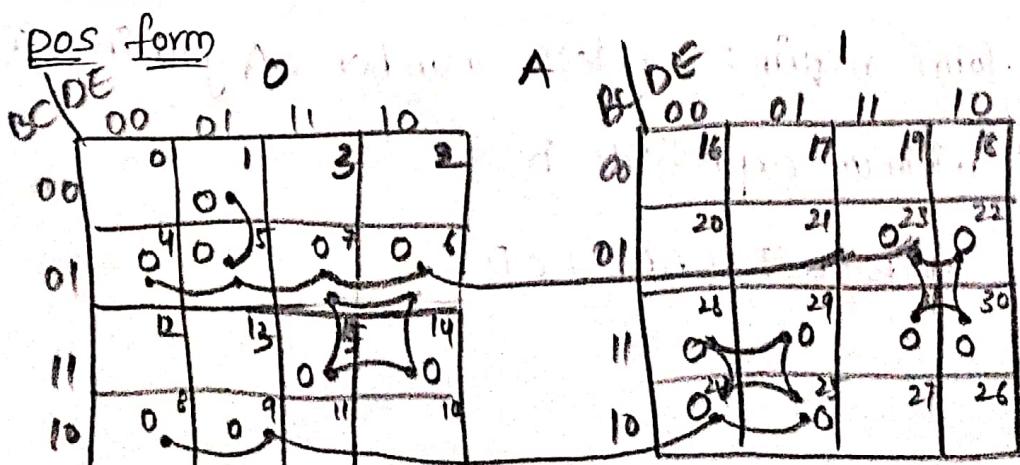
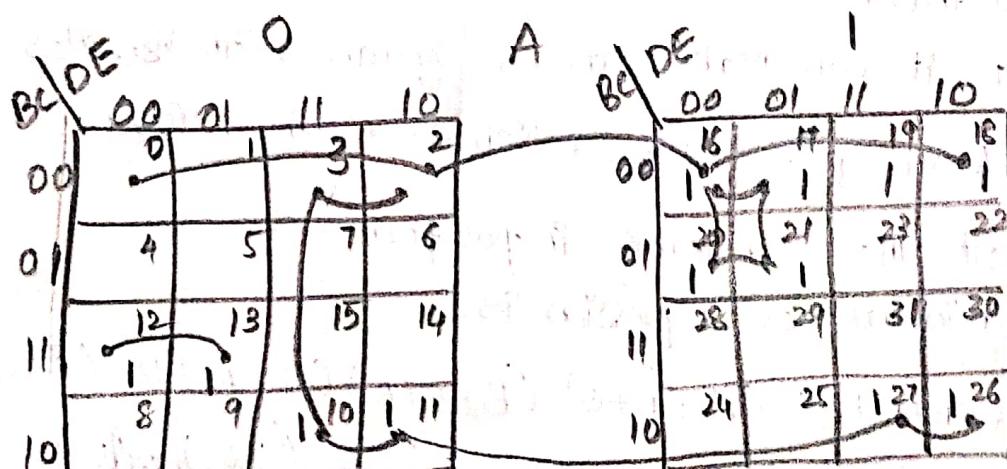
Class Notes

Unit No:
Lecture No:
Link to Session
Planner (SP): S.No.... of SP
Book Reference:
Date Conducted:
Page No: 40

- 3) m_0 can go with m_2 , m_{16} and m_{18} so, form a 4-square, which is read as \overline{BCE}
- 4) m_{20}, m_1, m_7 and m_{16} form a square, which is read as \overline{ABD}
- 5) $m_2, m_3, m_{18}, m_{19}, m_{10}, m_{11}, m_{26}$ and m_{27} form an 8-square which is read as \overline{CD} .
- 6) write all the product terms in SOP form.

So, the minimal SOP expression is

$$f_{min} = \overline{ABCD} + \overline{BCE} + \overline{ABD} + \overline{CD} \quad (16 \text{ inputs})$$



In the pos K-map, the reduction is done as per these steps.

1. There are no isolated 0's.
2. M_1 can go only with M_5 or M_9 . Make a 2-square with M_5 , which is read as $(A+B+C+D+E)$.
3. M_4 can go with M_5 , M_7 and M_{16} to form a 4-square, which is read as $(A+B+C)$.
4. M_8 can go with M_9 , M_{24} and M_{25} to form a 4-square, which is read as $(\bar{A}+\bar{B}+D)$ (or) $(\bar{B}+C+D)$.
5. M_{28} can go with M_{29} , M_{24} and M_{25} to form a 4-square which is read as $(\bar{A}+\bar{B}+D)$.
6. M_{30} can make a 4-square with M_{31} , M_{29} and M_{28} or with M_{31} , M_4 and M_{15} or with M_{20} , M_{22} and M_{23} .
Don't do that.

Note that it can make an 8-square with M_{31} , M_{23} , M_{22} , M_6 , M_7 , M_{14} and M_{15} , which is read as $(\bar{C}+\bar{D})$.

7. Write all the sumterms in pos form.
So, the minimal pos expression is

$$F_{min} = (A+B+D+E) (A+B+\bar{C}) (\bar{B}+C+D) (\bar{A}+\bar{B}+D) (\bar{C}+\bar{D})$$

(20 inputs)

The sop form requires a less number of gate inputs.

The real minimum expression is.

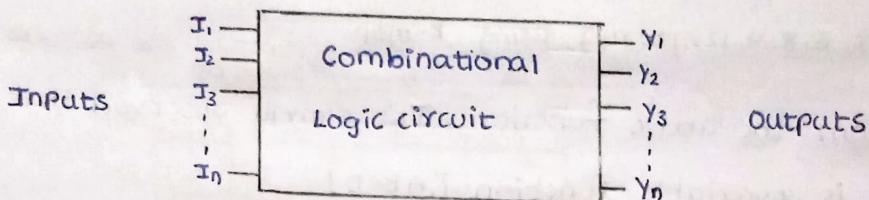
$$f_{min} = \bar{A}BC\bar{D} + \bar{B}\bar{C}\bar{E} + A\bar{B}\bar{D} + \bar{C}D.$$

2. COMBINATIONAL AND SEQUENTIAL LOGIC CIRCUIT

For digital system consist of combinational circuit and Sequential circuits.

COMBINATIONAL CIRCUIT

THE Combinational circuit output depends on only present inputs only.



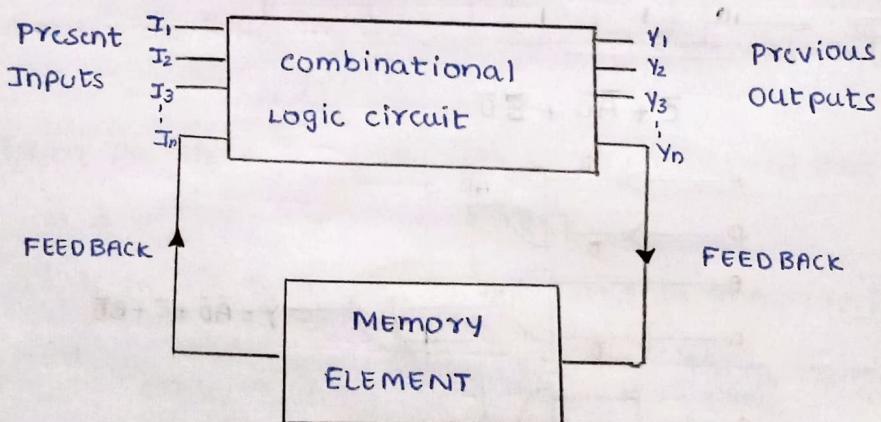
Here $I_1, I_2, I_3 \dots I_n$ are inputs. $Y_1, Y_2, Y_3 \dots Y_n$ are outputs.

Combinational logic circuits may be adder or subtractor, Encoded, decoded, multiplexer, de-multiplexer,

In a combinational logic circuit n input. 2^n outputs.

SEQUENTIAL CIRCUIT

It output depends on present input and previous output.



Here $I_1, I_2, I_3 \dots I_n$ are inputs. $Y_1, Y_2, Y_3 \dots Y_n$ are outputs. Here memory element might be flipflop

Sequential logic circuits may be Register, counter, flipflop.

Differences between Combinational and Sequential logic circuit

COMBINATIONAL LOGIC CIRCUIT	SEQUENTIAL LOGIC CIRCUIT
In combinational circuit the output variable at any instant of time or dependent only on the present input variables.	In sequential circuit the output variable at any instant of time or dependent on the previous output, not only on the present input but also present stage or previous output.
Memory element is not required.	Memory element is required to store the previous history of the input variable.
Combinational circuit are easy to design.	It is are comparatively hard to design.
Memory element or feedback path is not present	Memory element or feedback path is present
It is a simple circuit	It is a complex circuit
It is are faster because the delay between there input and output is low.	It is slower because the output is due to propagation delay along with feedback path delay.

Classification of Sequential Circuits.

1. Synchronous
2. Asynchronous.

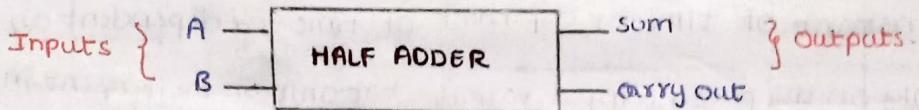
ADDERS

Digital computer performs various arithmetic operations. In most basic operation is adding of two binary digits.

HALF ADDER

A combinational circuit that performs the addition of two bits is called a half adder.

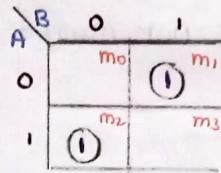
It adds two inputs [A,B] which as single bit. It doesn't take carry from previous sum.



TRUTH TABLE

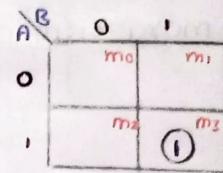
Inputs		Outputs	
A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

K-map



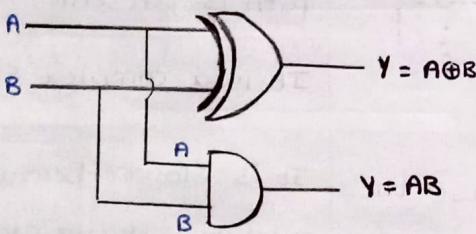
$$\text{Sum} = A\bar{B} + \bar{A}B$$

$$\text{Sum} = A \oplus B$$



$$\text{Carry out} = AB$$

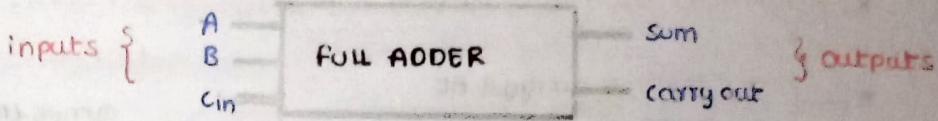
Logic diagram.



FULL ADDER

It is a combinational circuit that performs the addition of three bit is called a full adder.

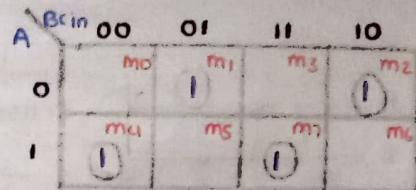
It is designed to add more than two bit. It can add two single bit number along with a carry in [A, B, cin]



TRUTH TABLE

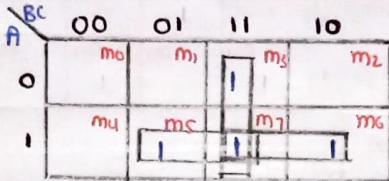
Inputs			Outputs	
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

K-map for sum



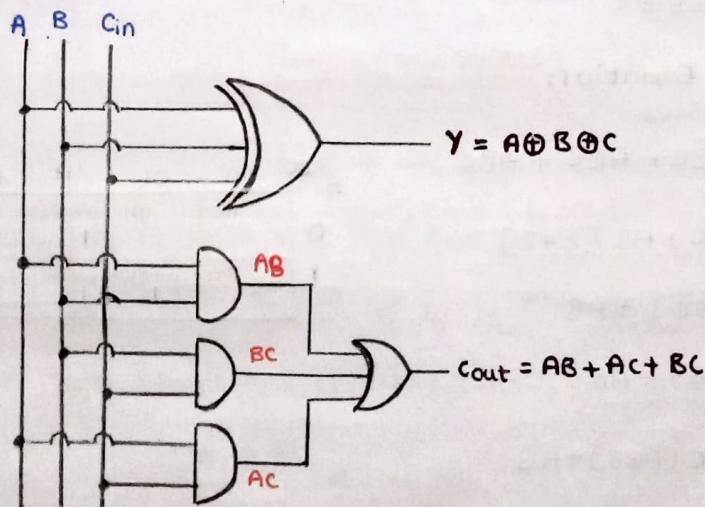
$$\begin{aligned}
 \text{Sum} &= \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC + \bar{A}B\bar{C} \\
 &= \bar{B}[\bar{A}C + A\bar{C}] + B[AC + \bar{A}\bar{C}] \\
 &= \bar{B}[A \oplus C] + B[\bar{A} \oplus \bar{C}] \\
 &= \bar{B}[x] + B[\bar{x}] \\
 &= B \oplus x \Rightarrow A \oplus B \oplus C
 \end{aligned}$$

K-map for carry out

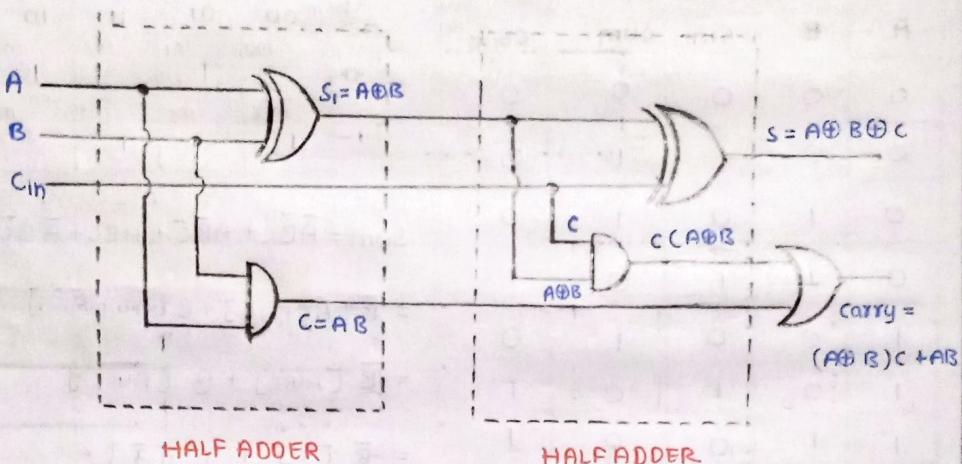
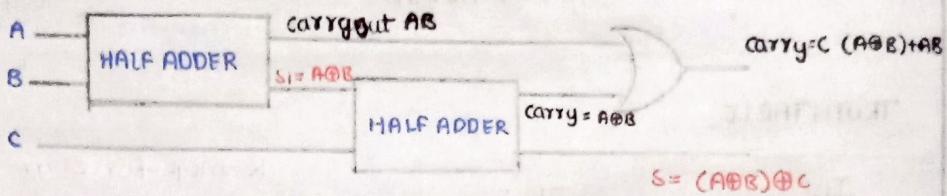


$$\text{Carry} = AC + BC + AB$$

Logical diagram



REALIZATION OF FULL-ADDER USING TWO HALF ADDERS



Sum Equation

$$\bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

$$C[\bar{A}\bar{B} + AB] + \bar{C}[\bar{A}B + A\bar{B}]$$

$$C\bar{X} + \bar{C}X$$

$$\therefore X = A \oplus B = \bar{A}B + \bar{B}A$$

$$\text{Sum} = X \oplus C$$

$$\bar{X}_1 = \overline{A \oplus B} = \bar{A}\bar{B} + A\bar{B}$$

$$\text{Sum} = A \oplus B \oplus C$$

		BC	00	01	11	10
		A	m ₀	m ₁	m ₃	m ₂
0	0		1		1	1
	1		m ₄	m ₅	m ₇	m ₆

Carry Equation

$$\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

$$\bar{A}BC + A\bar{B}C + AB[C + \bar{C}]$$

$$C[\bar{A}B + A\bar{B}] + AB$$

$$C[A \oplus B] + AB$$

$$\text{Carry} = C[A \oplus B] + AB$$

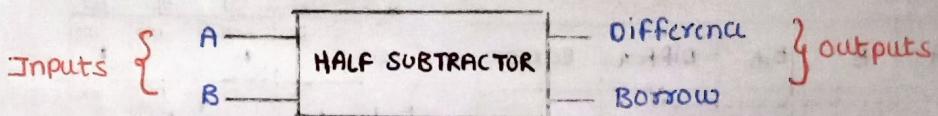
		BC	00	01	11	10
		A	m ₀	m ₁	m ₃	m ₂
0	0		1		1	1
	1		m ₄	m ₅	m ₇	m ₆

HALF SUBTRACTOR

A combinational circuit that performs the subtraction of two bits is called a half subtractor.

It subtracts two inputs [A, B] which as single bit.

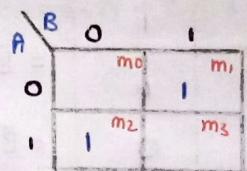
It doesn't take carry from previous subtractor.



TRUTH TABLE

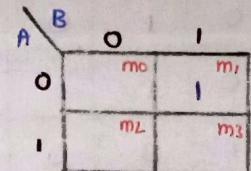
Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map



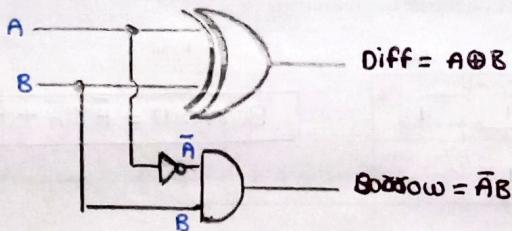
$$\text{Sum} = A\bar{B} + \bar{A}B$$

$$\text{Diff} = A \oplus B$$



$$\text{Borrow} = \bar{A}B$$

LOGIC DIAGRAM

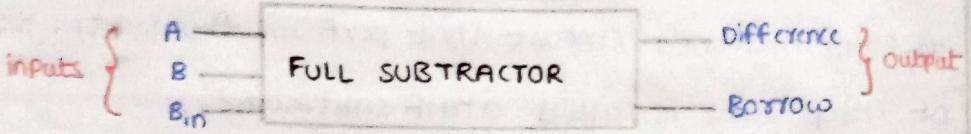


FULL SUBTRACTOR

It is a combinational circuit that performs the subtraction of three bits is called a full subtractor.

It is designed to subtract more than two bits. It can add and subtract two single bit numbers along with a Borrow in

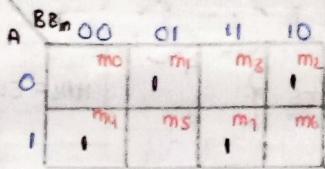
[A, B, Bin]



TRUTH TABLE

Inputs			Outputs	
A	B	B _{in}	Difference	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K-map for Difference



SIMPLIFIED EXPRESSION

$$\text{Difference} = \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + AB\bar{B}_{in} + A\bar{B}\bar{B}_{in}$$

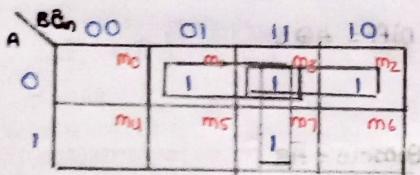
$$= \bar{B}[\bar{A}B_{in} + AB_{in}] + B[A\bar{B}_{in} + \bar{A}\bar{B}_{in}]$$

$$= \bar{B}[A \oplus B_{in}] + B[\bar{A} \oplus B_{in}]$$

$$= \bar{B}[x] + B[\bar{x}]$$

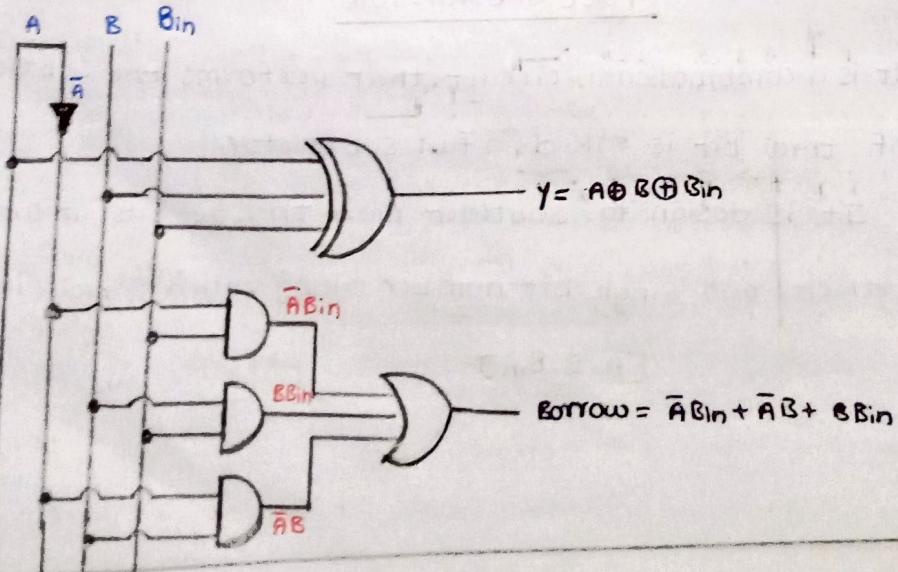
$$= \bar{B} \oplus x \Rightarrow \boxed{\text{Difference} = A \oplus B \oplus B_{in}}$$

K-map for Borrow



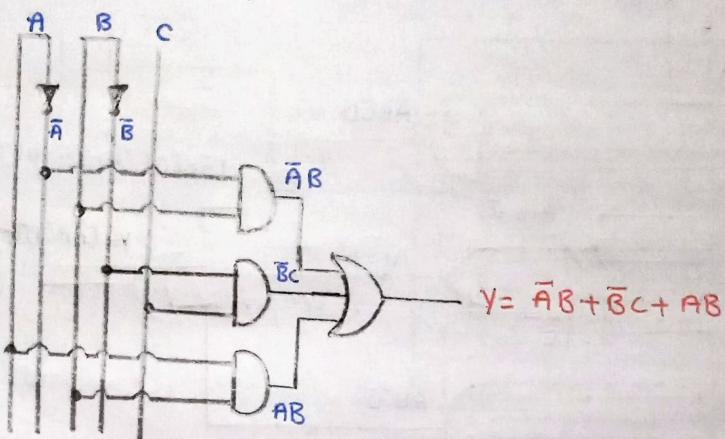
$$\text{Borrow} = \bar{A}B_{in} + \bar{A}B + BB_{in}$$

Logic DIAGRAM

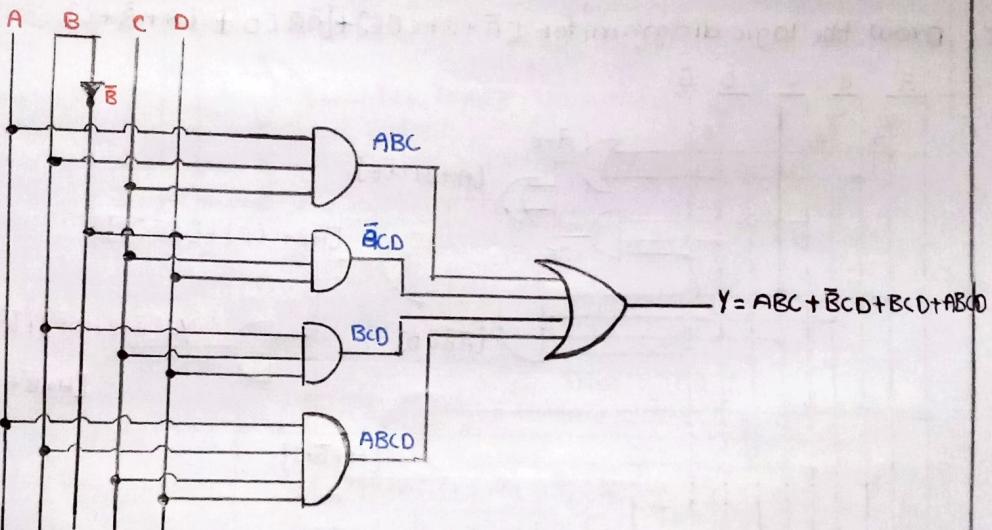


HOME WORK

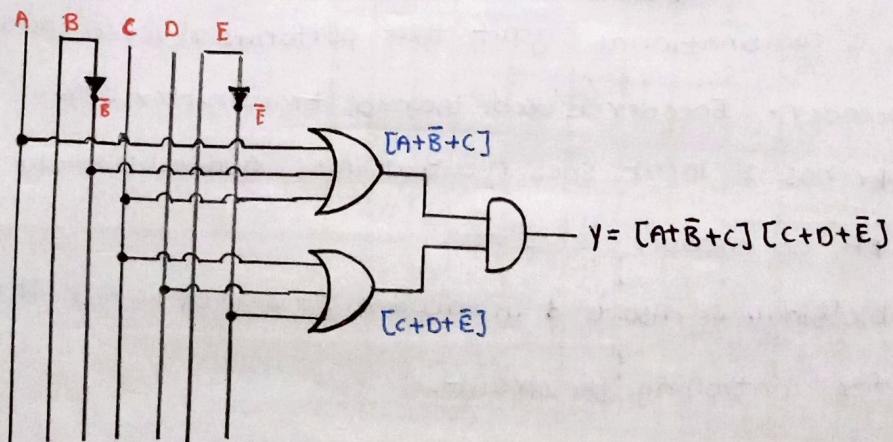
1 Draw Logic diagram of $\bar{A}B + \bar{B}C + AB$



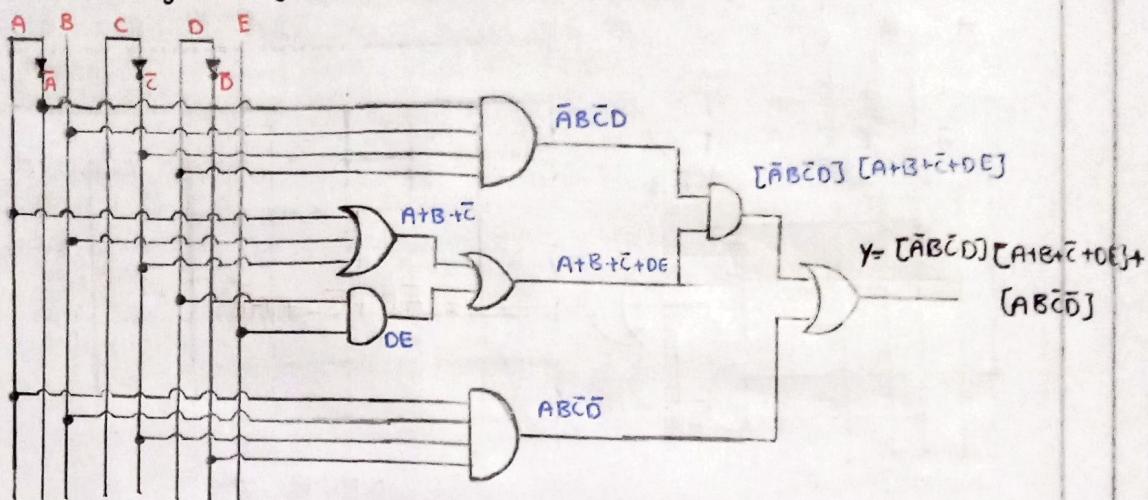
2 Draw Logic diagram of $ABC + \bar{B}CD + BCD + ABCD$



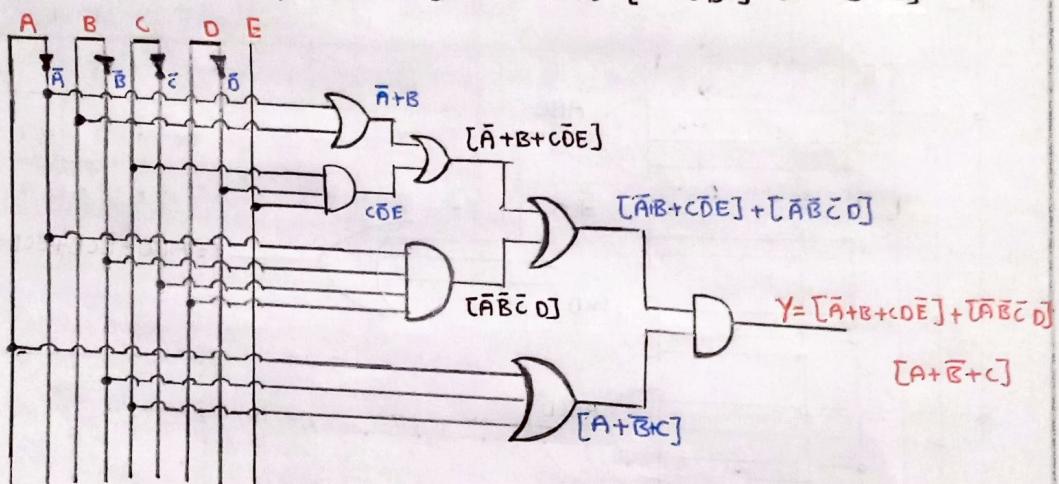
3 Draw logic diagram for $[A + \bar{B} + C] [C + D + \bar{E}]$



4 Draw a logic diagram for $[\bar{A}\bar{B}\bar{C}D] \cdot [A+B+\bar{C}+DE] + [\bar{A}B\bar{C}\bar{D}]$



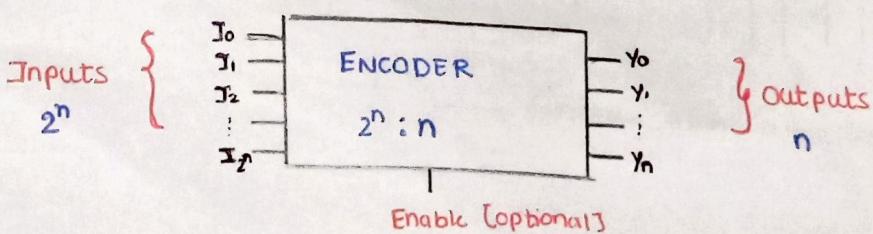
5 Draw the logic diagram for $[\bar{A}+B+C\bar{D}E]+[\bar{A}\bar{B}\bar{C}D] \cdot [A+\bar{B}+C]$



ENCODER

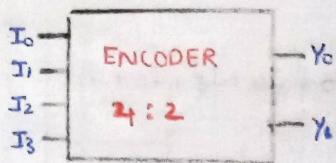
It is a combinational circuit that performs reverse operation of Decoder. Encoder is used toward transmitter side. Encoder has 2^n input lines n output lines. General formula of an encoder is $2^n \times n$.

Enable Single is also used in encoder. It is an optional. It is used for controlling the circuit.



DESIGNING OF 4:2 ENCODER

TRUTH TABLE



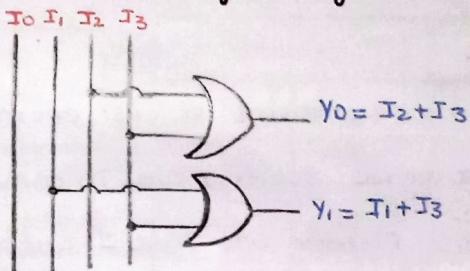
Output Equations:

$$Y_0 = I_2 + I_3$$

$$Y_1 = I_1 + I_3$$

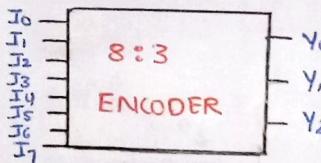
INPUTS						OUTPUTS	
E	J ₃	J ₂	J ₁	J ₀		Y ₀	Y ₁
0	x	x	x	x		0	0
1	0	0	0	1		0	0
1	0	0	1	0		0	1
1	0	1	0	0		1	0
1	1	0	0	0		1	1

Logic diagram.



DESIGNING OF 8:3 ENCODER

TRUTH TABLE



Output Equations:

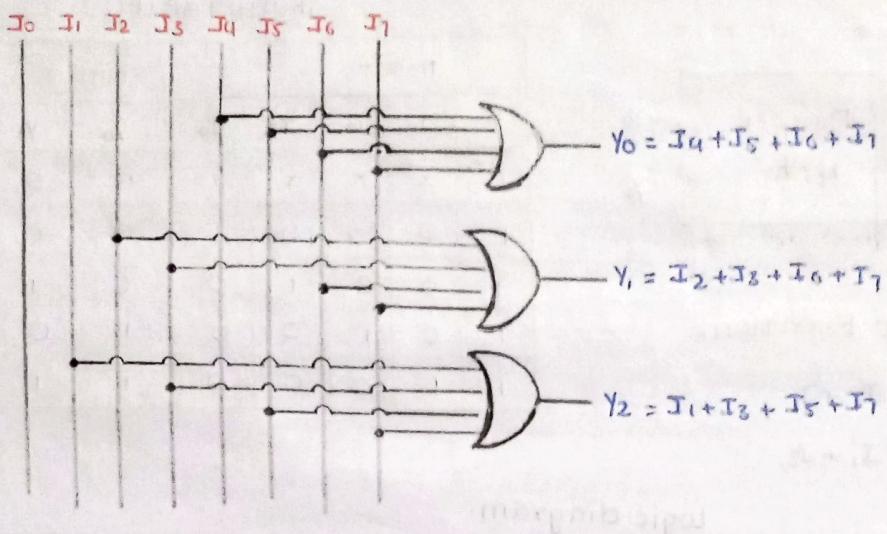
$$Y_0 = I_4 + I_5 + I_6 + I_7$$

$$Y_1 = I_2 + I_3 + I_6 + I_7$$

$$Y_2 = I_1 + I_3 + I_5 + I_7$$

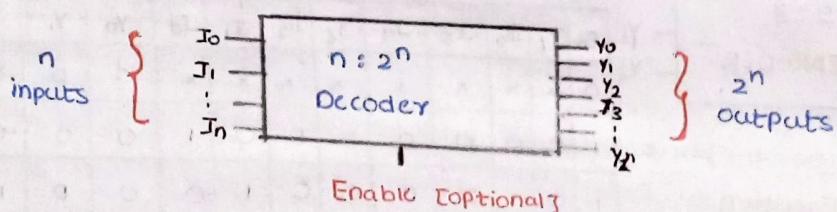
Inputs									Outputs		
E ₀	I ₇	I ₆	I ₅	I ₄	I ₃	I ₂	I ₁	I ₀	Y ₀	Y ₁	Y ₂
0	x	x	x	x	x	x	x	x	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	1	0	1
1	1	0	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	1	1	1

Logic diagram

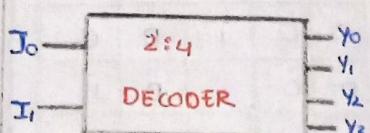


DECODER

It is combinational circuit that performs reverse operation of Encoder. Decoder are used towards receiver side in communication to decode the encoded data. Decoder are have 2^n output lines and n inputs. General formula of Decoders is $n \times 2^n$. Enable signal is used in Decoder. It is an optional. It is used to control the circuit.



DESIGNING OF 2:4 DECODER



output equation

$$Y_0 = \bar{I}_0 \bar{I}_1$$

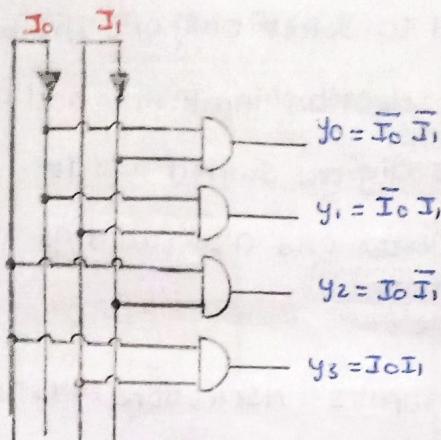
$$Y_1 = \bar{I}_0 I_1$$

$$Y_2 = I_0 \bar{I}_1$$

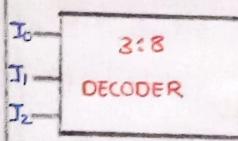
$$Y_3 = I_0 I_1$$

Inputs			Outputs			
En	I_1	I_0	Y_0	Y_1	Y_2	Y_3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Logic diagram.



DESIGNING OF 3:8 DECODER



output equation:

$$y_0 = \bar{I}_2 \bar{I}_1 \bar{I}_0$$

$$y_1 = \bar{I}_2 \bar{I}_1 I_0$$

$$y_2 = \bar{I}_2 I_1 \bar{I}_0$$

$$y_3 = \bar{I}_2 I_1 I_0$$

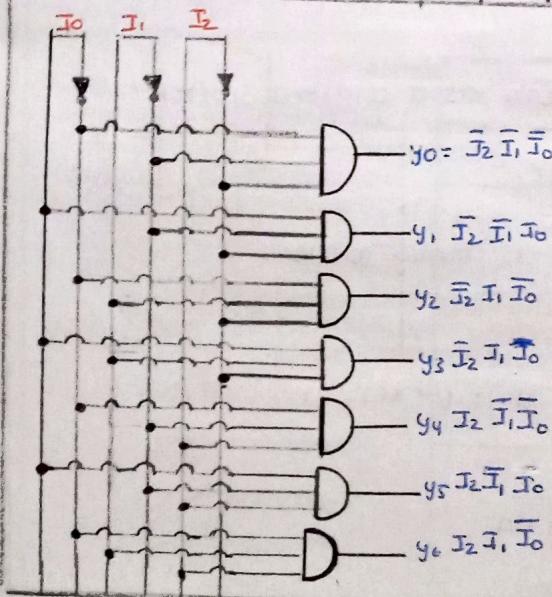
$$y_4 = I_2 \bar{I}_1 \bar{I}_0$$

$$y_5 = I_2 \bar{I}_1 I_0$$

$$y_6 = I_2 I_1 \bar{I}_0$$

$$y_7 = I_2 I_1 I_0$$

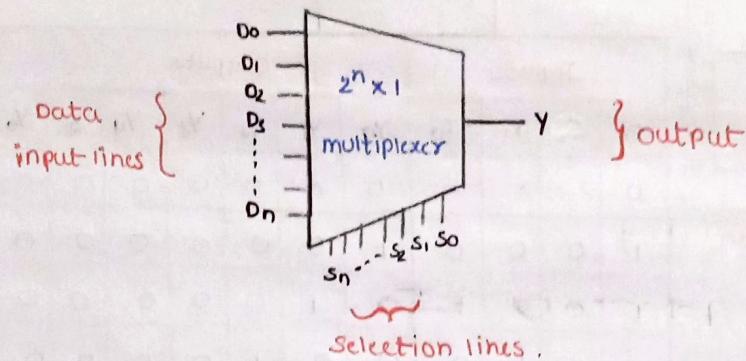
Inputs			Outputs								
E _n	I ₂	I ₁	I ₀	y ₀	y ₁	y ₂	y ₃	y ₄	y ₅	y ₆	y ₇
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1



MULTIPLEXER

It is a combinational circuit used to select only one input among several inputs based on selection input lines and it is MSI $2^n \times 1$. This can act as digital switch and it is denoted by $2^n \times 1$, data selector and also called as Selector variable or Selector line.

for a mux there can be 2^n inputs, n Selection lines and only one output is possible.



Advantages:

1. It reduces number of wires
2. It reduce circuit complexity.
3. It reduce cost and reduces number of lines.

Application

1. It is used in communication as a digital switch
2. It is used as data selector.

NOTE: multiplexer is known as many to one

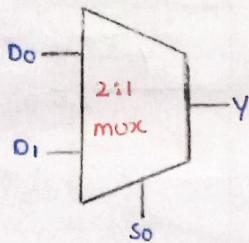
Types

There are many types of multiplexer

1. 2x1
2. 4x1
3. 8x1
4. 16x1 ... so on.

DESIGNING OF 2:1 MULTIPLEXER

2x1 mux will accept 2 inputs and gives 1 output and it is one selection line input and it has also one enable and it is optional.



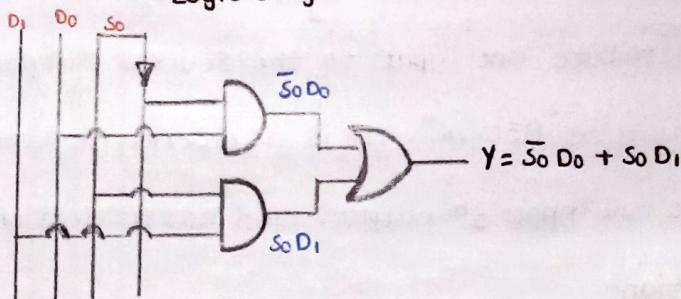
Inputs		Output
E	S0	Y
0	X	0
1	0	D0
1	1	D1

output equation:

$$Y = \bar{S}_0 D_0$$

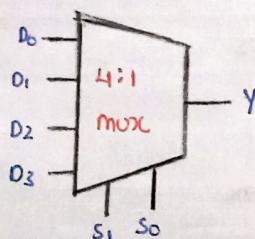
$$Y = S_0 D_1 \quad Y = \bar{S}_0 D_0 + S_0 D_1$$

Logic diagram.



DESIGNING OF 4:1 MULTIPLEXER

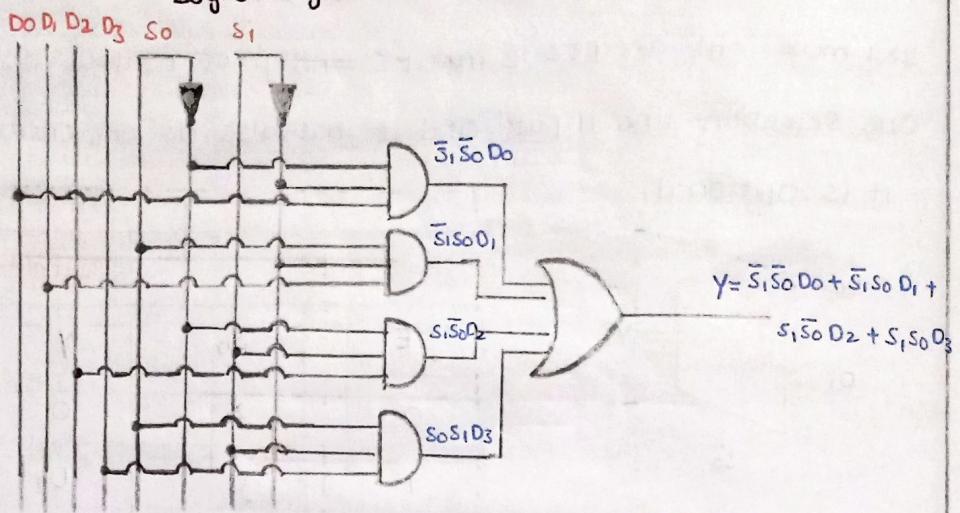
4x1 mux will accept 4 inputs and gives 1 output and it has two selection line input and it also has one enable and its optional



$$Y = \bar{S}_0 \bar{S}_1 D_0 + S_0 \bar{S}_1 D_1 + \bar{S}_0 S_1 D_2 + S_0 S_1 D_3$$

Inputs			Outputs
E	S1	S0	Y
0	X	X	0
1	0	0	D0
1	0	1	D1
1	1	0	D2
1	1	1	D3

Logic diagram.



DE MUX

It is a combinational circuit which is used in communication.

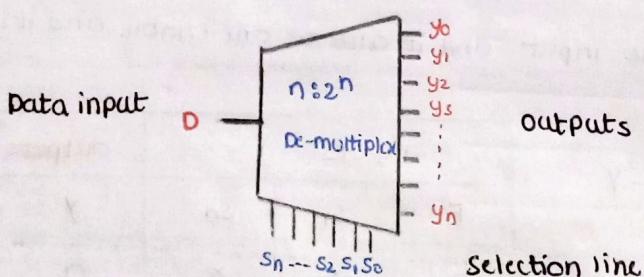
It can distribute one input to the several output. Demultiplex is also called as "DEMUX". It is as a serial to parallel converter.

It accepts one input 2^n outputs and has n selection lines.

Applications

It is used as distribution in communication.

It is as a serial to parallel converter.



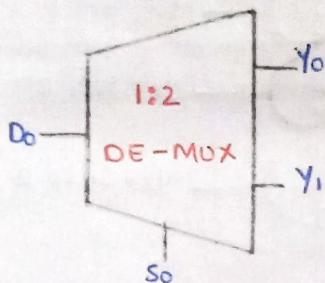
TYPES OF DE-MUX

There are many types of de multiplexers

- 1. 1x2 3. 1x8
- 2. 1x4 4. 1x16

DESIGNING OF 1:2 DE-MULTIPLEXER

It is a combinational circuit it will accept one input data line and distribute it to the output lines [y_0, y_1] based on selection input lines]

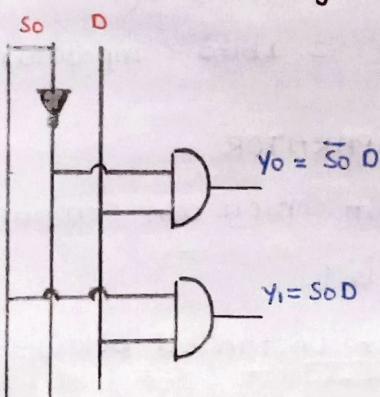


Input		outputs	
E	S_0	y_0	y_1
0	x	0	0
1	0	0	1
1	1	1	0

Output equation:

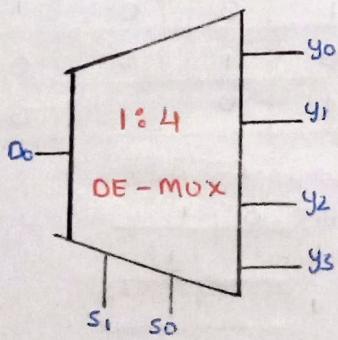
$$y_0 = \bar{S}_0 D \quad y_1 = S_0 D$$

Logical diagram



DESIGNING OF 1:4 DE-MULTIPLEXER

It is a combinational circuit it will accept one input data line and distribute it to the output lines [y_0, y_1, y_2, y_3] based on selection input lines].



Input			outputs			
E	S_1	S_0	y_0	y_1	y_2	y_3
0	x	x	0	0	0	0
1	0	0	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	1	0	0	0	0

output equation:

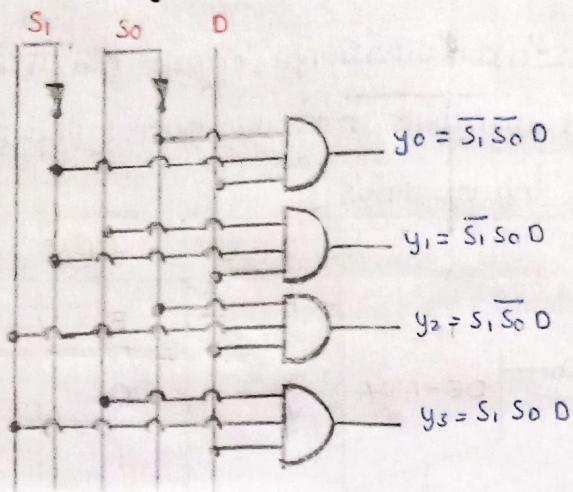
$$y_0 = \bar{s}_1 \bar{s}_0 D$$

$$y_1 = \bar{s}_1 s_0 D$$

$$y_2 = s_1 \bar{s}_0 D$$

$$y_3 = s_1 s_0 D$$

LOGIC DIAGRAM



COMPARATOR

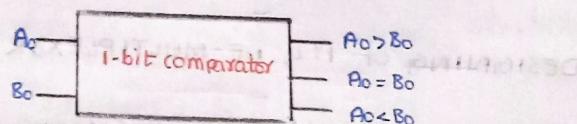
Comparator is a combinational circuit which is used for comparing numbers $[A, B]$. There are many types

- 1. 1bit comparator 2. 2bits comparator ... Soon

DESIGNING OF 1 BIT COMPARATOR

It is a combinational circuit which can compare two numbers $[A, B]$ of single bit $[A_0, B_0]$.

Comparison can be better understand below logic



1. $A_0 = 1, B_0 = 0$ $A_0 > B_0$ [Greater]

2. $A_0 = 1, B_0 = 1$ $A_0 = B_0$ [Equal]

3. $A_0 = 0, B_0 = 1$ $A_0 < B_0$ [Lesser]

4. $A_0 = 0, B_0 = 0$ $A_0 = B_0$ [Equal]

Inputs		Outputs		
A_0	B_0	$A > B_0$	$A = B$	$A < B_0$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

K-map for $A > B$

A_0	B_0	0	1
0	m ₀	m ₁	
1	m ₂	m ₃	

$A_0 \bar{B}_0$

K-map for $A = B$

A_0	B_0	0	1
0	m ₀	m ₁	1
1	1	m ₂	m ₃

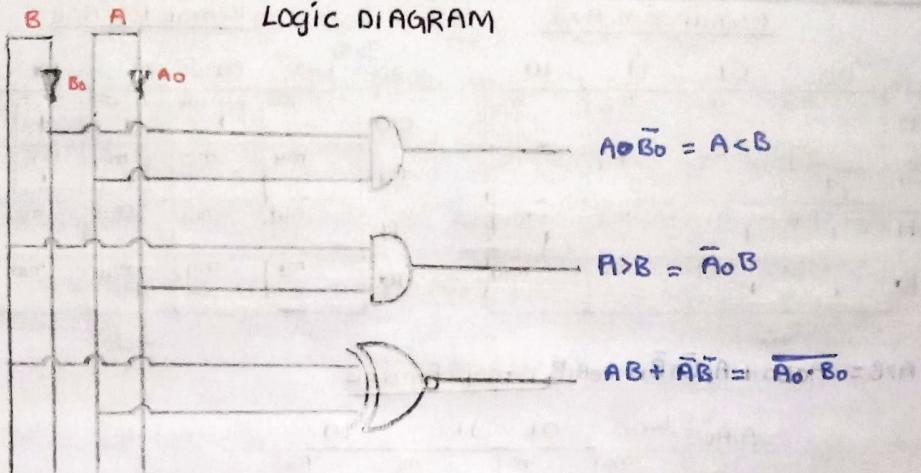
$A_0 + \bar{A}_0 \bar{B}_0$

K-map for $A < B$

A_0	B_0	0	1
0	m ₀	1	m ₃
1	m ₂		m ₃

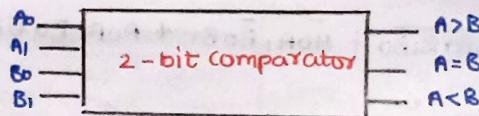
$\bar{A}_0 B_0$

LOGIC DIAGRAM



DESIGNING OF 2 BIT COMPARATOR

- It is a combinational circuit which can compare 2 bits (A, B) of 2 bits. $[A = A_0, A_1]$ $[B = B_0, B_1]$
- 2 bit comparator logic can be better understood by below truth table.



inputs				outputs		
A_1	A_0	B_1	B_0	$A > B$	$A = B$	$A < B$
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	0	0

K-map for $A > B$

		00	01	11	10
		m_0	m_1	m_3	m_2
		m_4	m_5	m_7	m_6
		m_8	m_9	m_{15}	m_{14}
		m_{10}	m_{11}	m_{13}	m_{12}

K-map for $A < B$

		00	01	11	10
		m_0	m_1	m_3	m_2
		m_4	m_5	m_7	m_6
		m_{12}	m_{13}	m_{15}	m_{14}
		m_8	m_9	m_{11}	m_{10}

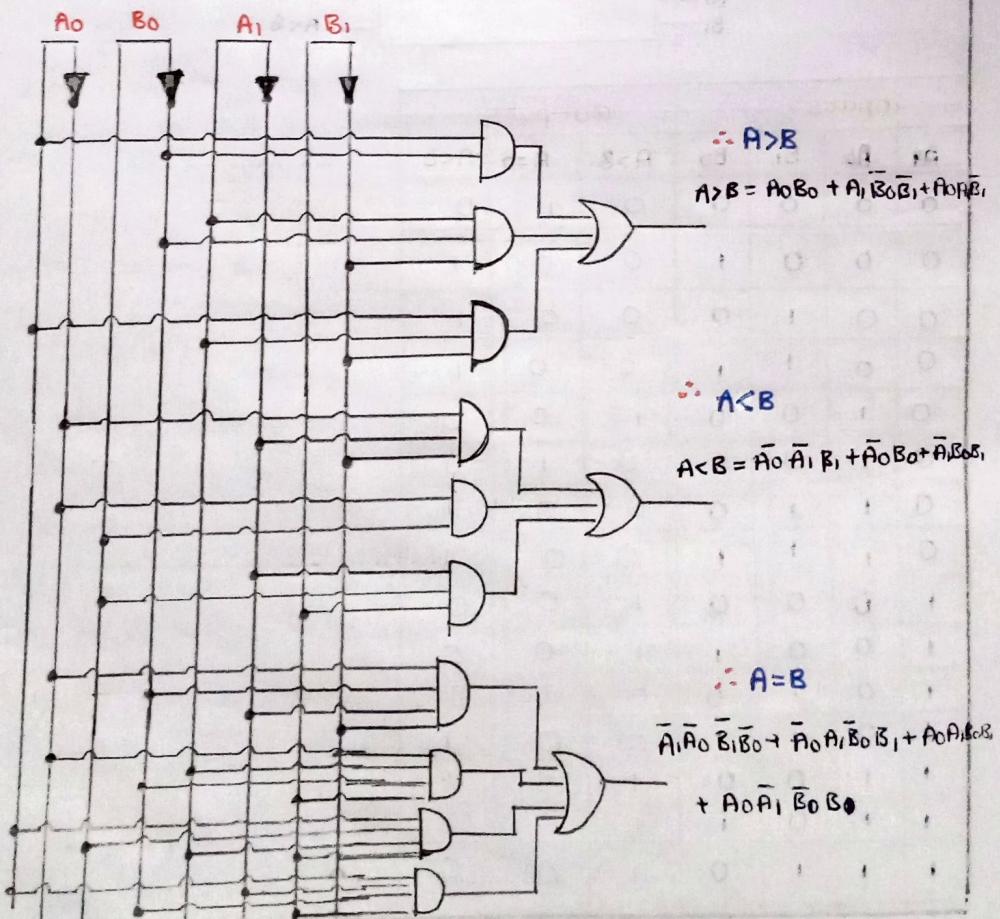
K-map for $A = B$

		00	01	11	10
		m_0	m_1	m_3	m_2
		m_4	m_5	m_7	m_6
		m_2	m_{13}	m_{15}	m_{14}
		m_8	m_9	m_{11}	m_{10}

$$\therefore A > B = \bar{A}_0 \bar{B}_0 + A_1 \bar{B}_0 \bar{B}_1 + \bar{A}_0 A_1 \bar{B}_1$$

$$\therefore A < B = \bar{A}_0 \bar{A}_1 \bar{B}_1 + \bar{A}_0 B_0 + \bar{A}_1 B_0 \bar{B}_1$$

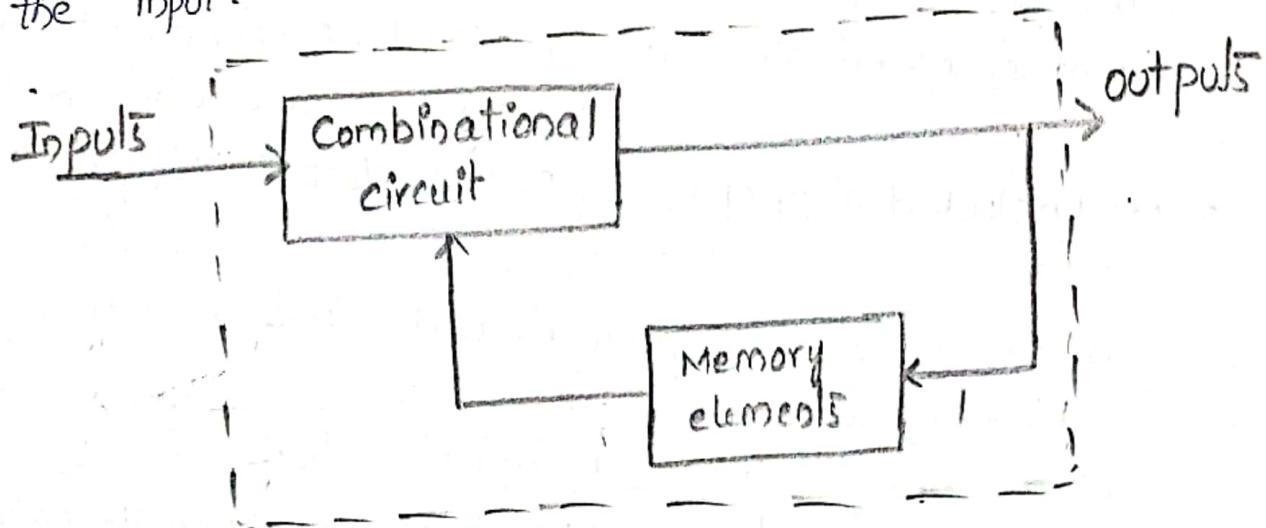
$$A = B = \bar{A}_1 \bar{B}_1 \bar{B}_0 + \bar{A}_0 A_1 \bar{B}_0 \bar{B}_1 + \bar{A}_0 A_1 B_0$$



Sequential logic circuits:

The design of logic circuit using memory and elements and combinational circuit is known as sequential logic circuit.

- Sequential logic circuit generates output in accordance with the sequence in which the input signals are received.
- The outputs always depends on the present inputs and on past outputs, because output is connected back to the input.



Block diagram of sequential circuit

Thus we can specify the sequential circuit by a time sequence of external inputs, internal states (present states and next states) and outputs.

Sequential circuits can be classified into

1. Synchronous Sequential circuits and
2. Asynchronous Sequential circuits depending on the timing of their signals.

- In asynchronous sequential circuits, change in input signals can affect memory elements at any instant of time.
- In synchronous sequential circuits, signals can affect the memory elements only at discrete instants of time.

Asynchronous Sequential Circuits

1. Input signals can affect memory elements at any instant of time.
2. More difficult to design
3. Clock is not required as one of the input
4. Speed of operation is high
5. Ex: Unclocked flipflops

Synchronous Sequential Circuits

1. Input signals can effect memory elements at discrete instant of time.
2. Easier to design.
3. Clock is required as one of the input.
4. Speed of operation is limited by the time delays involved.
5. Ex: Clocked flipflops

Comparison between Sequential and Combinational Circuits

Combinational logic circuits

1. output depends only on the present input
2. Easier to design
3. Speed of operation is high
4. Memory unit is not required
5. Ex: parallel adder

Sequential logic circuits

1. output depends on the present input and past output also
2. comparatively harder to design
3. speed of operation is comparatively low.
4. Memory unit is required to store the past outputs.
5. Ex: Serial adder.

Latch

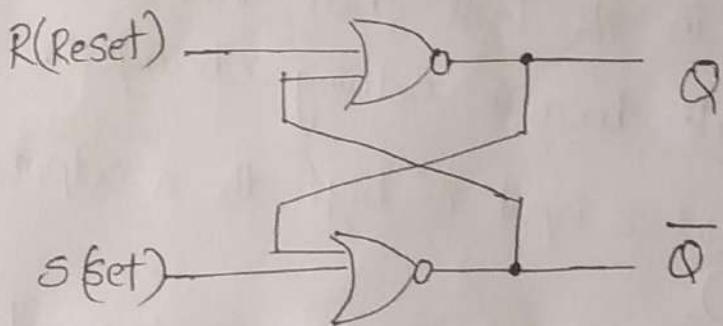
Latches and flipflops both are bistable elements. These are the basic building blocks of most sequential circuits.

- The main difference between latches and flipflop is in the method used for changing their state.
- Many times enable signal is provided with the latch. When enable signal is active output changes occur as input changes.
But when enable signal is not activated input changes don't affect output.

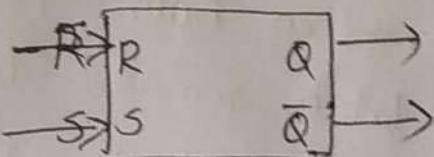
S-R latch:

The simplest type of latch is set-reset (SR) latch. It can be constructed from either two NAND gates or two NOR gates.

SR latch using NOR gates



logic symbol



logic diagram

The two NOR gates are cross coupled so that the output of NOR gate 1 is connected to one of the

Inputs of NOR gate 2 and vice versa.

The latch has two inputs Q and \bar{Q} , and two inputs Set and Reset.

Case 1 : $S=0$ and $R=0$

case 2 : $S=0$ and $R=1$

Case 3 : $S=1$ and $R=0$

Case 4 : $S=1$ and $R=1$

Truth Table

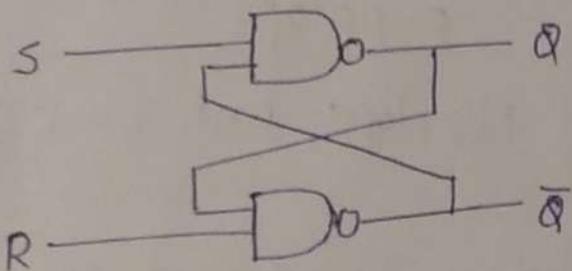
S	R	Q_n	Q_{n+1}	state
0	0	0	0	No change
0	0	1	1	
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	x	Indeterminate
1	1	1	x	

operation :

- When both inputs low, the output does not change and latch remains latched in its last state. This condition is called inactive state because nothing changes.
- When R input is low and S input is high, the Q output of latch is set (at logic 1).
- When R input is high & S input is low, the o/p is Reset.
- When R and S inputs both are high, output is unpredictable. This is called indeterminate condition.

SR latch using NAND gates.

logic diagram



The operation of this latch is the reverse of the operation of NOR gate latch.

Truth table

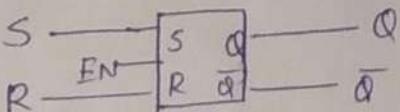
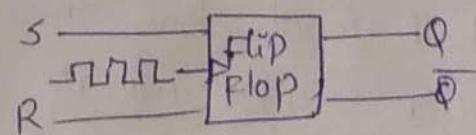
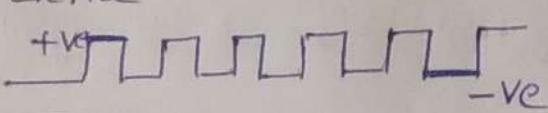
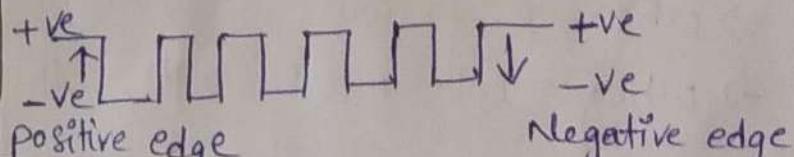
S	R	Q_n	Q_{n+1}	state
0	0	0	X	Indeterminate
0	0	1	X	Indeterminate
0	1	0	∅	Set
0	1	1	∅	Set
1	0	0	0	Reset
1	0	1	0	Reset
1	1	0	0	No change
1	1	1	1	No change

operation

- When both inputs low, the output is unpredictable . this is called Indeterminate condition.
- When R is high, S is low the Q output of latch is set
- When R is low, S is high the Q output of latch is Reset
- When both inputs high, the output does not change.

Differences between Latch and FlipFlop

Latches and FlipFlops both are building blocks of sequential circuits

Latches	FlipFlops
1. Latches do not require clock signal	FlipFlops have clock signals.
2. A power requirement of a latch is less	Power requirement of a flip-flop is more.
3. A latch works based on the enable signal	A flip flop works based on the clock signal.
4. The operation of a latch is faster as they do not have to wait for any clock signal	FlipFlops are comparatively slower than latches due to clock signal
5. A latch is an asynchronous device	A flip flop is a synchronous device.
6. Latch can be built by logic gates	Flip flop can be built by latch with an additional control input.
7. <u>Logic Symbol</u>	<u>Logic Symbol</u>
	
8. Latch is a level triggered device	Flip flop is a edge triggered device
	

Flip Flops:

The storage elements employed in clocked sequential circuits are called flip flops.

- A flip-flop is a binary cell capable of storing one bit of information.
- It has two outputs, one for normal value and one for the complement value of the bit stored in it.
- A flip flop maintains a binary state until directed by a clock pulse to switch states.
- A flip flop is a bistable electronic circuit that has two stable states i.e., the output is either 0 or 1.
- Flip flop is the smallest storage unit of a computer since what it stores is a single bit (0 or 1).
- There are many types of flip flops.
The major difference between them are in the number of inputs applied and a manner how the inputs change the outputs.

1. SR flip flop

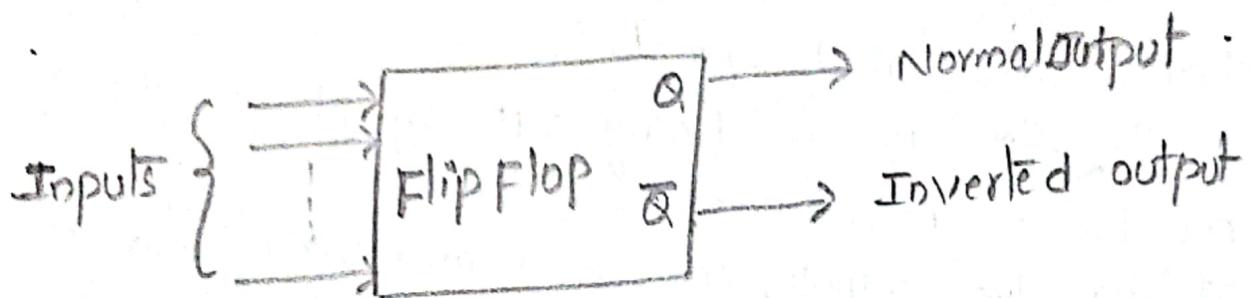
2. JK flip flop

3. D flip flop and

4. T flip flop

- SR flip flop is the commonest of them.

- A flipflop can have one or more inputs. The input signals which command the flipflop to change state are called excitations.



General flipflop symbol.

Applications of flip flops:

There are number of applications of flip flops.

1. The flipflop serves as a storage device
2. It stores a '1' when its Q output is '1', and stores a '0' when its Q output is a '0'.
3. flipflops are the fundamental components of shift registers and counters.

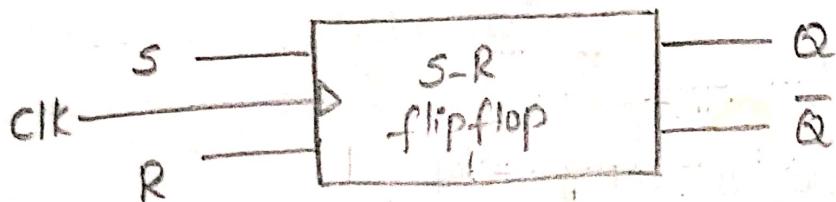
S-R FlipFlop :

In digital circuits, many operations have to be carried out in a proper sequence at the appropriate time. These operations are controlled by clock pulses.

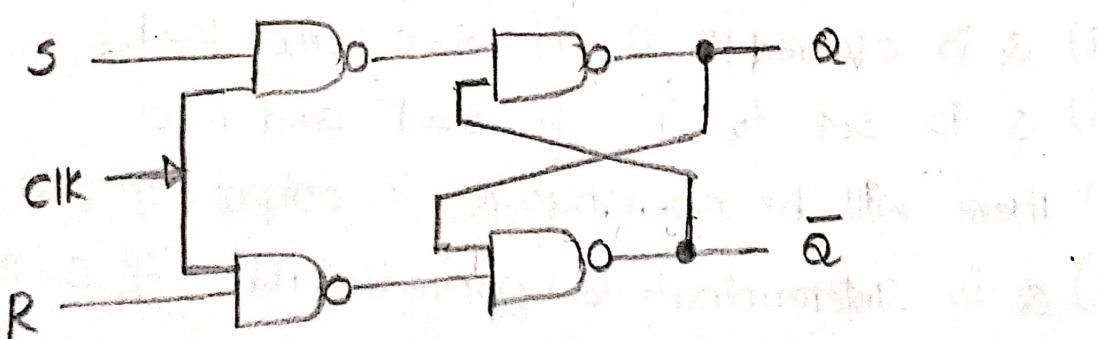
- The flip-flop output is controlled i.e., neither be set nor reset without the preference of the clock pulse.

Such a flip flop is the clocked SR flip flop.

- The SR flip flop can be represented using a graphic symbol.



Graphic symbol of S-R flipflop



Logic diagram of S-R flipflop

- It consists of inputs S, R and clk to set, Reset and for clock respectively.
- The clk input, that is represented with an arrowhead shape is dynamic, as it responds only when there is a

Positive transition (i.e from 0 to 1)

- The output can be Q or \bar{Q} (Q 's complement)

Truth table for S-R flip flop is,

S	R	On	Qn+1 state
0	0	0	0
0	0	1	1 No change
0	1	0	0
0	1	1	0 Reset
1	0	0	1
1	0	1	1 Set
1	1	0	x Indeter-
1	1	1	x minate

Truth table of S-R flip flop

- Whenever the clock input changes from 0 to 1
 - Q is cleared to '0' if $S=0$ and $R=1$
 - Q is set to '1' if $S=1$ and $R=0$
 - There will be no change in output, if $S=0$ and $R=0$
 - Q is Indeterminate (or) unpredictable, if $S=R=1$.
- When the clock 'clk' has no signal, then there will be no change in output.

Q_n	S	R	Q_{n+1}
0	0	0	0 Q_n
0	0	1	0 Q_n
0	1	0	1
0	1	1	X
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	X

Characteristic table of SR flipflop.

K-map for characteristic table is

$Q_n \backslash S$	00	01	11	10
0			X 1	
1	1		X 1	

Characteristic eq n is $Q_{n+1} = S + Q_n R$

Excitation table for SR flipflop is

present state Q_n	Next state Q_{n+1}	Required inputs	
		S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

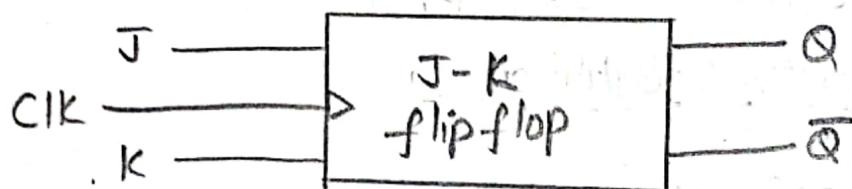
J-K flipflop :

In an S-R flipflop, the state of the output is not predictable when $R=1$ and $S=1$.

The J-K flipflop allows inputs $J=K=1$.

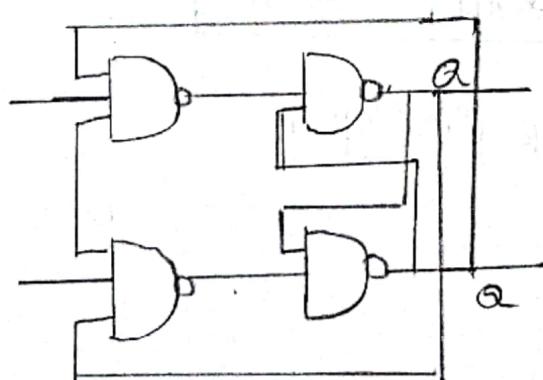
In this situation, the state of the output is changed.

- In J-K, the letter J is for set and the letter K is for Reset (clear).
- When inputs are applied to both J and K simultaneously as 1, the flipflop switches to its complement state i.e $Q=1$, it switches to $Q=0$ and viceversa.



Graphical representation of J-K flip flop

logical diagram for J-K flipflop



Truthtable

J	K	Q_n	Q_{n+1}	state
0	0	0	0	
0	0	1	1	Nochange
0	1	0	0	
0	1	1	0	Reset
1	0	0	1	
1	0	1	1	set
1	1	0	1	Toggle
1	1	1	0	(Complementary)

Characteristic table

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

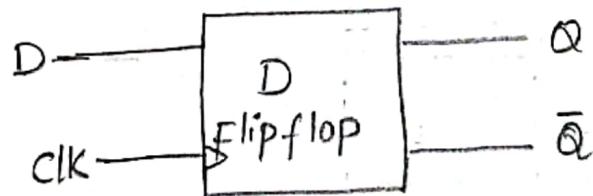
Excitation table

present state	Q_{n+1}	Required inputs	
		J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

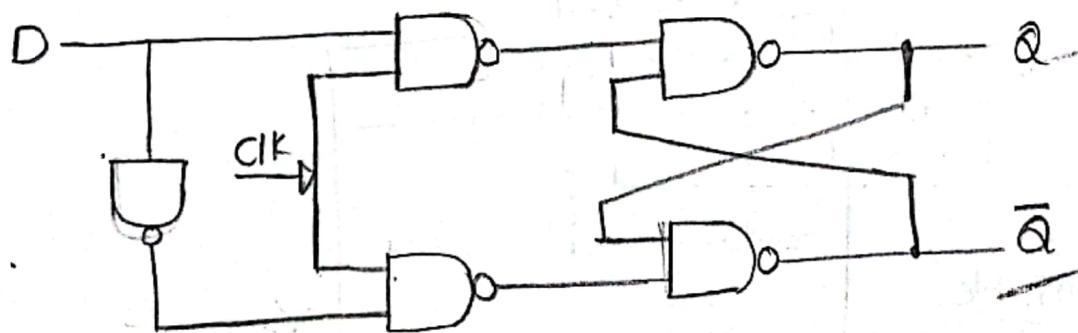
D Flip-Flop:

The D flip-flop is also known as Delay flip-flop (or) Data flip-flop.

- The D flip-flop is a slight modification of the SR flip-flop.
- An SR flip-flop is converted to a D flip-flop by inserting an inverter between S and R and assigning the symbol D to the single input.
- The D input is sampled during the occurrence of a clock transition from 0 to 1.
- D flip-flop is used to either store the data or introduce a delay.



Graphic symbol of D flipflop



Circuit diagram of D flipflop

operation . Truth table for D flipflop

If $D=1$, the output of flipflop goes to the 1 state
but if $D=0$, the output of flipflop goes to the 0 state.

TruthTable

D	Q_n	Q_{n+1}	state
0	0	0	Reset
	1	0	
1	0	0	Set
	1	1	

TruthTable of D-flipflop

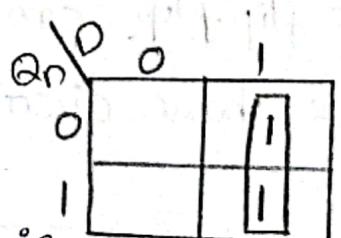
The output terminals of D-flipflop are same as that of S-R flipflop.

- D-flipflop is said to be an extension or improved version of S-R flipflop.
- The output values are applied to the input terminal. The input values and its corresponding outputs are represented in the characteristic table.

Characteristic table :

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

K-map for characteristic equation is



Characteristic equation for D flipflop is,

$$Q_{n+1} = D$$

Excitation table for D-flipflop

<u>PS</u> Q_n	<u>NS</u> Q_{n+1}	<u>Required J Input</u> D
0	0	0
0	1	1
1	0	0
1	1	1

For a D flip flop, the next state is always equal to the D input and it is independent of the present state.
 \therefore D must be 0 if Q_{n+1} has to be 0, and
D must be 1 if Q_{n+1} has to be 1
regardless of the value of Q_n .

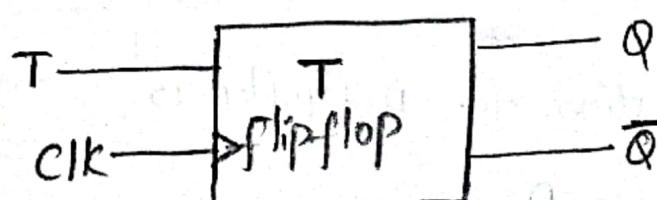
T-flipflop:

A T flipflop has a single control input, labelled T for toggle.

Whenever a pulse is given to the T input, the output changes state.

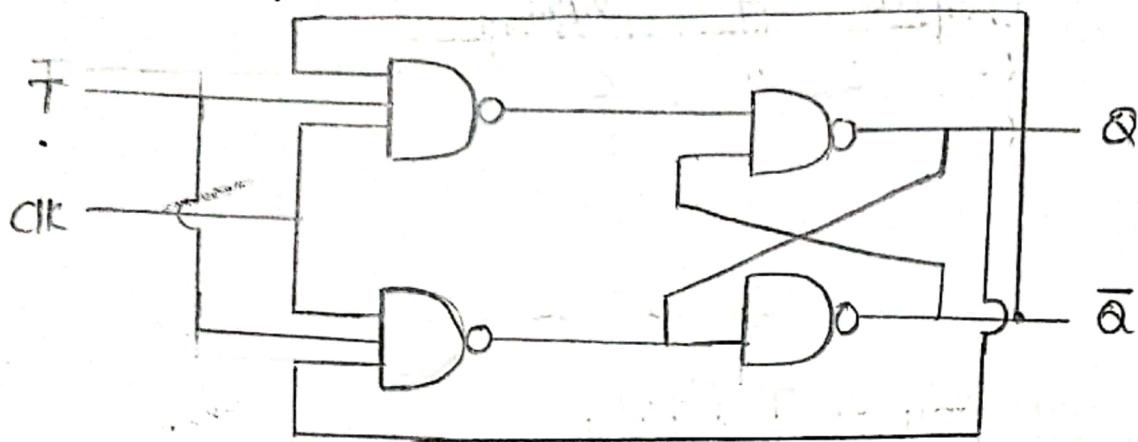
T flipflop can be used as a frequency divider.

The basic circuit diagram for T flipflop is



Circuit diagram

Circuit diagram:



It is easy to convert a J-K flipflop to the functional equivalent of a T flipflop by just connecting J and K together and labelling the common connection as T.

Truth table

T	Qn	Qn+1	state
0	0	0	No change
0	1	1	No change
1	0	1	Toggle
1	1	0	Toggle

Truth table of T flipflop

operation

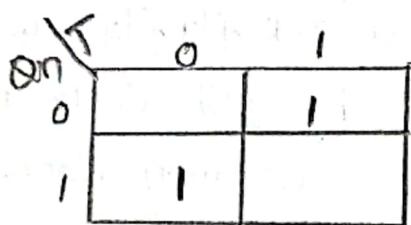
When $T=0$, there is no change of state

When $T=1$, the flipflop toggles.

Characteristic table for T-flipflop :

<u>Present state</u>	<u>Input</u>	<u>Next state</u>	
Q_n	T	Q_{n+1}	
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

K-map for Q_{n+1} of T flipflop



The characteristic equation of T-flip flop is

$$Q_{n+1} = \overline{Q_n}T + Q_n\overline{T}$$

Excitation table of T flipflop

<u>present state</u>	<u>next state</u>	<u>Required Input</u>
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Counters:

Counters are one form of registers, which can be used to count the number of clock pulses applied at input over a period of time.

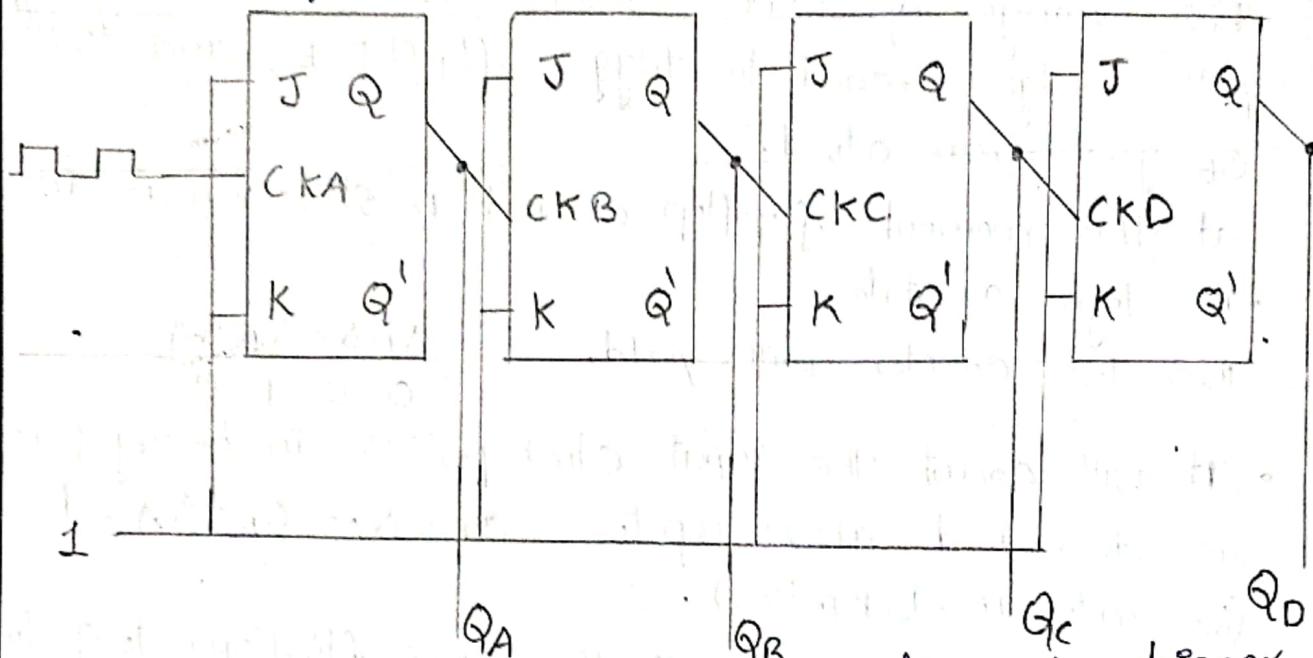
- Counters are found in almost all equipment containing digital logic.
- Counters are mostly constructed using JK flipflops in toggle mode or T flipflops.
- Flipflop programmed as counters are used in a wide variety of counting applications in scientific instruments, industrial controls, computers and communication equipments, as well as in many other areas.
- A counter that follows the binary number sequence is called a binary counter.
- There are two types of counters,
 1. Synchronous counter
 2. Asynchronous counter.
- In synchronous counter, the common clock input is connected to all of the flipflop and they are clocked simultaneously.
- In Asynchronous counter, commonly called ripple counters, the first flipflop is clocked by the external clock pulse and then each successive flipflop is clocked by the output of the previous flipflop.

Asynchronous or Ripple Counter:

Ripple counters are simplest possible counters.

- The circuit diagram of a ripple counter is constructed using JK flipflop.
- The J and K of all the flipflops are connected to high
- (1). so that all the flipflops are in toggle mode.
- Flipflops considered here are all negative edge triggered.
- Ripple counter is a cascaded arrangement of flipflops where the output of one flipflop drives the clock input of the following flipflop.
- A n-bit ripple counter can count up to 2^n states.
- It is also known as MOD n counter.
- Some of the features of ripple counter are
 1. It is an asynchronous counter
 2. Different flipflops are used with a different clock pulse.
 3. All the flipflops are used in toggle mode.
 4. only one flipflop is applied with an external clockpulse and another flipflop clock is obtained from the output of the previous flipflop.
 5. The flipflop applied with an external clock pulse act as LSB (Least Significant Bit) in the counting sequence.

A simple 4-bit binary ripple counter that uses four J-K flipflops.



- Suppose that the flipflops are negative edge-trigger type.
Initially all flipflops are in logic 0 state ($Q_A = Q_B = Q_C = Q_D = 0$).
A clock pulse is applied to the clock input of flipflop A, causing Q_A to change from logic 0 to logic 1.
At this moment, flipflop B, C and D do not change state (i.e. remain in the logic 0 state)
since there is no negative going edge of the clock pulse present at their clock input.
If Q_A and Q_D denote a LSB and MSB, respectively,
after the first clock pulse is applied to the clock input to flipflop Q_A ,

the counter will read

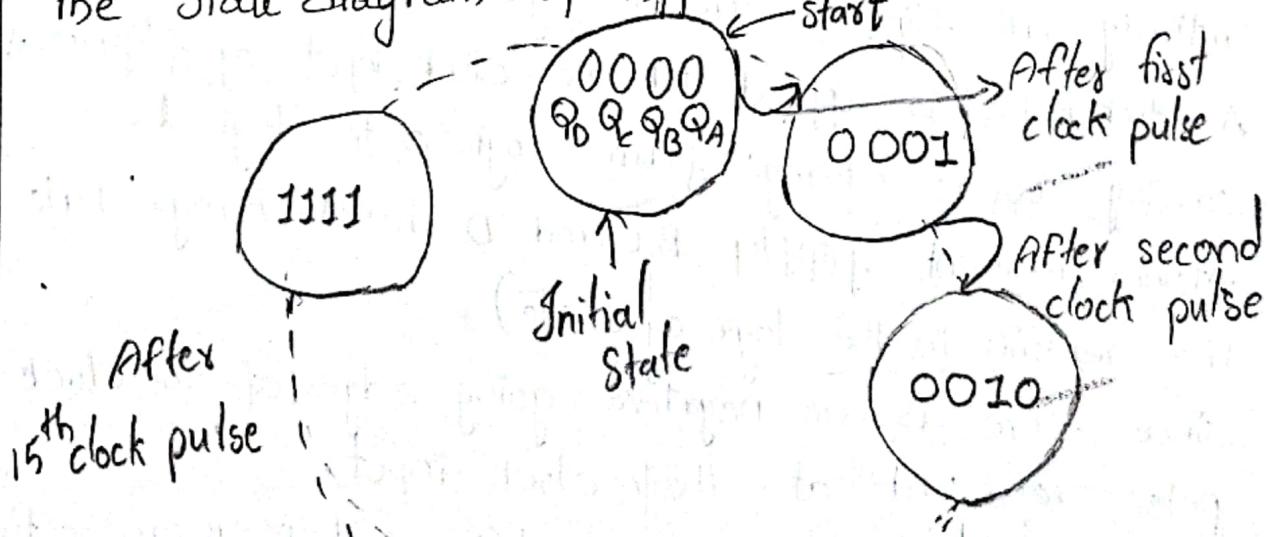
$$\begin{array}{l} Q_D \quad Q_C \quad Q_B \quad Q_A \\ \hline 0 \quad 0 \quad 0 \quad 1 \end{array}$$

- with the arrival of the second clock pulse to flipflop A, Q_A goes from 1 to 0. This change of state creates the negative-going pulse edge needed to trigger flipflop B, and thus Q_B goes from 0 to 1. At this moment flipflop C and D still remain in the logic 0 state. Thus the counter will yield.

$$\begin{array}{l} Q_D \quad Q_C \quad Q_B \quad Q_A \\ \hline 0 \quad 0 \quad 1 \quad 0 \end{array}$$

- It will count the input clock pulses in binary form as described above up to $Q_D = Q_C = Q_B = Q_A = 1$ (i.e upto 15 clock pulses). Clock pulse 16 causes all the four flipflops to go to state 0—that is the counter recycles.

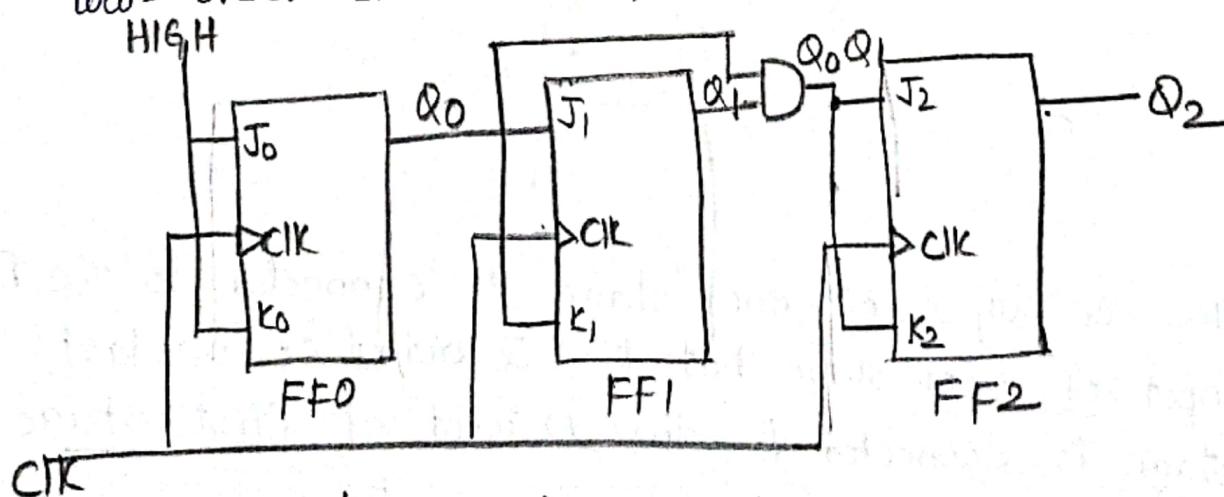
The state diagram of ripple counter is,



Synchronous counter:

To increase the speed of the counter the clock pulse were to be applied to all the flipflop at the same time affecting the change in all the flipflops simultaneously. Such counters are known as synchronous counters.

- In synchronous counters all the four flipflops are controlled by the same clock.
- The J and K inputs of the first flipflop are connected to High (1). Therefore the first flipflop would toggle for each clockpulse.
- The counter is based on the principle that in the sequence 0000; 0001, 0010, 0011, 0100, 0101, 0110 etc, the low order bit is complemented after every count and every other bit is complemented if the low-order bits are equal to 1.



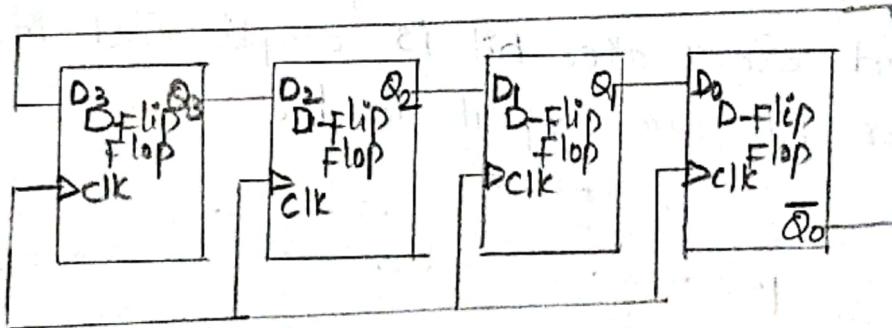
A 3-bit synchronous binary Counter

Twisted Ring counter (Johnson Counter)

Johnson counter also known as creeping counter, is an example of synchronous counter.

- In Johnson counter, the complemented output of last flipflop is connected to input of first flipflop and to implement n -bit Johnson counter we require ' n ' flipflops.
- It is one of the most important type of shift register counter.
- It is informed by the feedback of the output to its own input.
- Other names of Johnson counter are: bit counter, creeping counter, twisted ring counter, walking counter, mobile counter and switch tail counter.

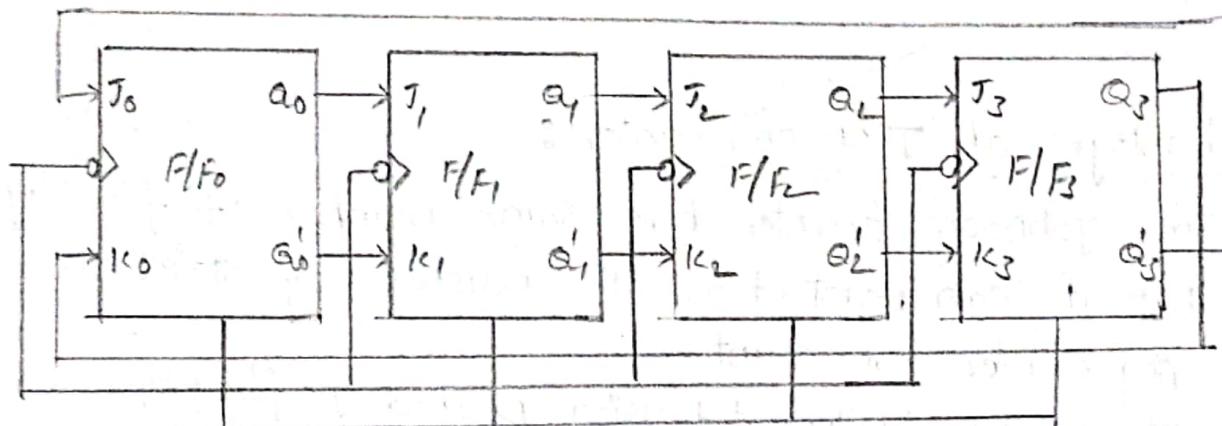
The logic diagram of 4-bit Johnson counter using D flip flops is,



- The Q output of each stage is connected to the D input of next stage but the Q output of the last stage is connected to the D input of first stage, therefore the name twisted ring counter.

This feed back arrangement produces a unique sequence of states.

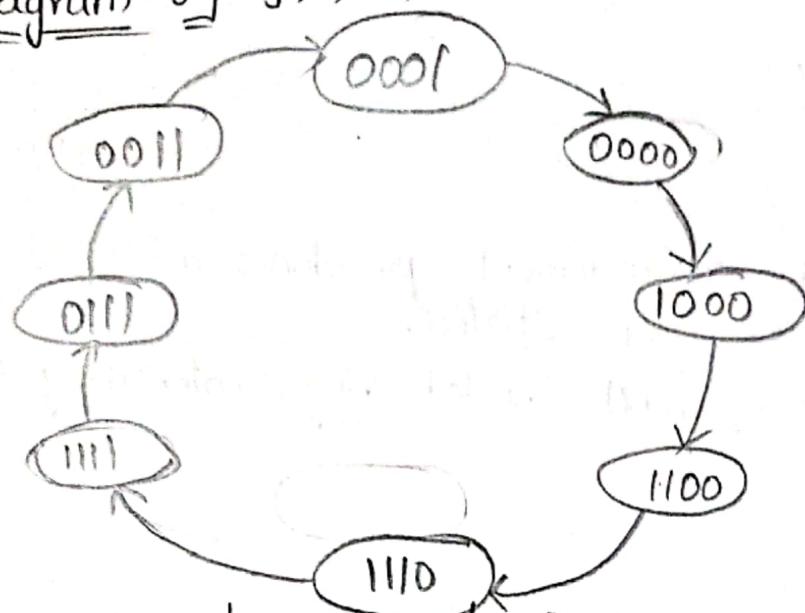
logic diagram of 4-bit twisted ring counter using JK flipflop



Truth table for Johnson counter

State	Q ₀	Q ₁	Q ₂	Q ₃
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1
8	0	0	0	0

State diagram of Johnson Counter



Advantages of Johnson Counter:

1. The Johnson counter has same number of flipflop but it can count twice the number of states the ring counter can count.
2. It can be implemented using D and JK flipflops
3. Johnson ring counter is used to count the data in a continuous loop.
4. Johnson counter is a self-decoding circuit.

Disadvantages of Johnson counter:

1. Johnson counter doesn't count in a binary sequence.
2. In Johnson counter more number of states being utilized.
3. The number of flipflops needed is one half the number of timing signals.
4. It can be constructed for any number of timing sequence.

Comparison between Asynchronous and Synchronous counter

Asynchronous Counter	Synchronous Counter
<ol style="list-style-type: none">These are serial countersi.e In these counters clock is connected to the first flip flop and its output acts as a clock input to the next flip flop and so onlogic circuit is very simple even for more number of statesspeed of operation is very low	<ol style="list-style-type: none">These are parallel countersIn these counters clock is connected simultaneously to all the flip flops.logic circuit becomes complex as number of states increases.speed of operation is relatively high.

Applications of counters:

Basically counters are useful for generating timing variables to sequence and control the operations in a digital system.

Hence the counters are very useful and versatile devices that are found in many applications.

1. Digital clock
2. Frequency dividers
3. parallel to serial data conversion (Multiplexing)
4. Auto parking control
5. Industrial digital control system etc.

Design of Synchronous counter:

The following steps are to be followed to design a simple synchronous counters.

- ① Determine the number of flip-flops needed using the formula $2^n \geq N$.
where n is the number of flipflops and
 N is the number of clock pulses to be counted.
2. select the type of flipflop to be used.
3. prepare the statetable from the given circuit information and derive the circuit excitation table.
4. Simplify the expressions for the flipflop inputs using any one of the simplification method (K-map or tabulation or algebraic)
5. Draw the logic diagram.

Design of Asynchronous counters:

To design an asynchronous counter the following steps are to be followed.

1. write the counting sequence
2. Tabulate the values of reset signal R for various states of the counter.
3. obtain the minimal expression for R or \bar{R} using K-map or any other method.
4. provide a feedback such that R or \bar{R} resets all the flipflops after the desired count.

Register :

Register is a series of flipflops. (or)
A register is a group of flipflops with each flipflop capable of storing one bit of information.
can store any discrete quantity

- A register with n flipflop of information contains n bits.

- The flipflop hold the binary information and the gates control when and how new information is transferred into the register.

- The transfer of new information into a register is referred to as loading the register.

- The registers can be classified into four possible modes of operation.

They are

1. serial in - serial out

2. serial in - parallel out

3. parallel in - serial out

4. parallel in - parallel out

Applications of registers :

The registers are found in many applications, a few of them are:

1. Time delay 2. Keyboard encoder

3. Ring counter

4. Serial to parallel and parallel to serial data converter.

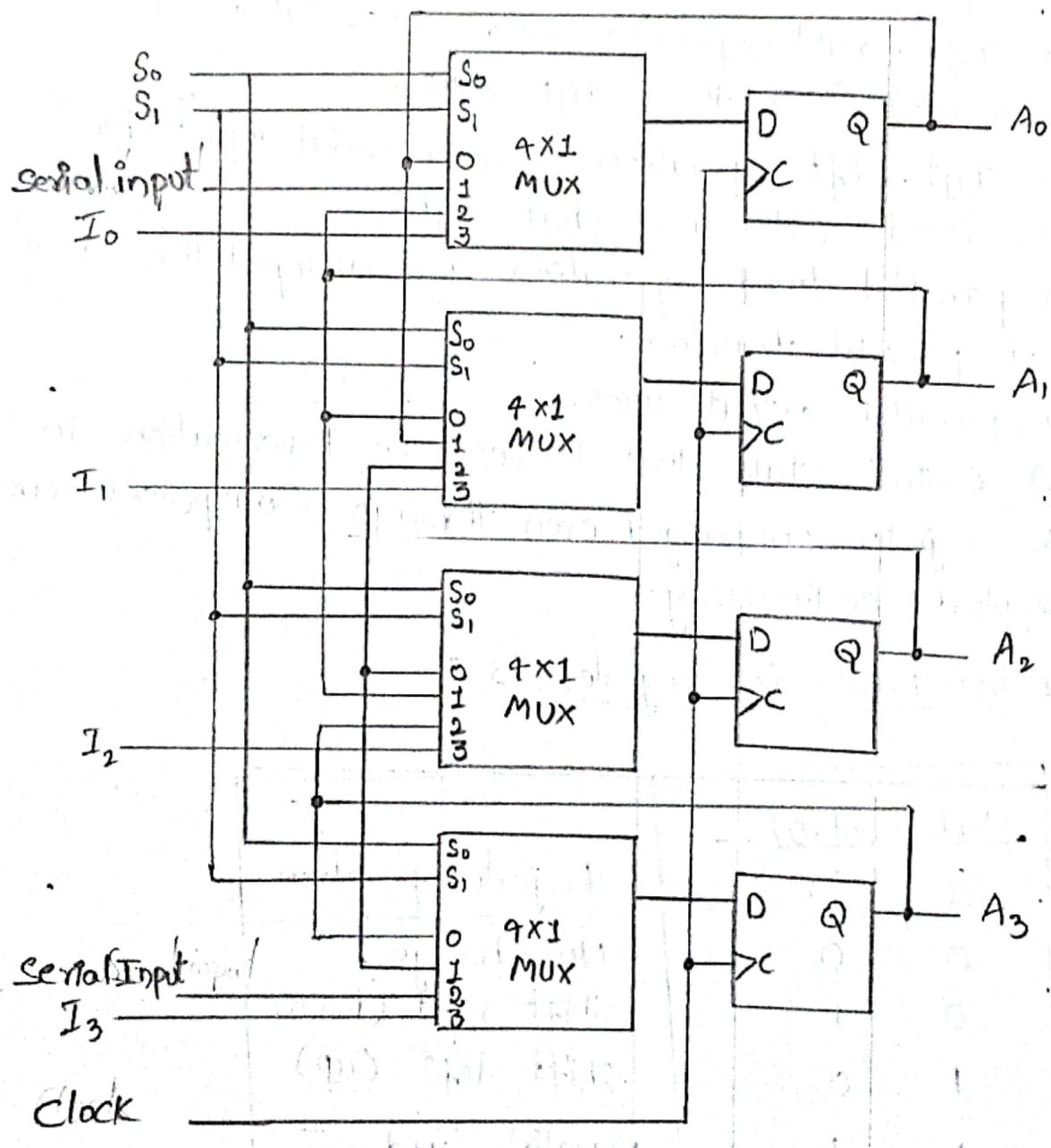
Shift Registers :

A register capable of shifting its binary information in one or both directions is called a shift register.

- The logical configuration of a shift register consists of a chain of flipflops in cascade, with the output of one flipflop connected to the input of the next flipflop.
- All flipflops receive common clock pulses that initiate the shift from one stage to the next.
- The simplest possible shift register is one that uses only flip-flops.
- The output of a given flipflop is connected to the D input of the flip-flop at its right.
- The clock is common to all flip-flops.
- The serial input determines what goes into the left most position during the shift.
- The serial output is taken from the output of the rightmost flipflop.
- sometimes it is necessary to control the shift so that it occurs with certain clock pulses but not with others.
- This can be done by inhibiting the clock from the input of the register if we do not want it to shift.
- When the shift register is used, the shift can be controlled by connecting the clock to the input of an AND gate and the 2nd output of the AND gate

can then control the shift by inhibiting the clock.
it is also possible to provide extra circuits to control the shift operation through the D inputs of the flip flops rather than the clock input.

Bidirectional shift Register with parallel load



A register capable of shifting in one direction only is called a unidirectional shift register.

- A register that can shift in both directions is called a bidirectional shift register.
- The most general shift register has all the capabilities. Others may have some of these capabilities, with at least one shift operation.
- 1. An input for clock pulses to synchronize all operations.
- 2. A shift-right operation and a serial input line associated with the shift-right.
- 3. A shift-left operation and a serial input line associated with the shift-left.
- 4. A parallel load operation and n input lines associated with parallel transfer.
- 5. n parallel output lines.
- 6. A control state that leaves the information in the register unchanged even though clock pulses are applied continuously.

Function table for Register is :

Mode control		Register operation
S_1	S_0	
0	0	No change
0	1	Shift right (down)
1	0	Shift left (up)
1	1	parallel load

Computer organization

Computer organization and Architecture is used to design computer system.

Computer Architecture: it is considered

Computer Architecture refers to the end to end structure of a computer system that determines how its components interact with each other in helping to execute the machine's purpose (i.e., processing data), often avoiding any reference to the actual technical implementation.

- Computer Architecture defines the system in an abstract manner, it deals with what does the system do.

Computer organization:

computer organization is realization of what is specified by the computer architecture.

- It deals with how operational attributes are linked together to meet the requirements specified by computer architecture.
- Some organizational attributes are hardware details
Control Signals and
Peripherals .

The basic minimum features of all digital computers contains the following components:

- (1) Internal registers
- (2) The main memory
- (3) A set of instruction
- (4) The timing and control structure.

- A program is a set of instructions that specify the operations, operands and the sequence by which processing has to occur.
- An instruction is a binary code that specifies a sequence of microoperations. (or)
A computer instruction refers to a binary code that controls how a computer performs micro-operations in a series.
- Instructions together with the 'information', are saved in memory. i.e Instructions and data are stored in memory.
- Every computer has its own set of instructions.
- The bits of an instruction can be grouped into parts called fields.

Memory



Instruction code

An instruction is a command given to a computer to perform an operation on data.

- An instruction code is a group of bits that instruct the computer to perform a specific operation.

The operation code of an instruction is a group of bits that define operations such as addition, subtraction, shift, complement etc.

- An instruction consists of 3 parts these three parts of an instruction are called fields.
- In a computer, the instruction format usually consists of 3 fields.

These are

operation code (or) opcode field: The value of opcode specifies the operation, such as addition, subtraction... etc to be performed on the operands.

Mode field: An I bit, which is a single bit specifying the addressing mode

Address field: Address field specifies the address of the main memory or the register reference operation or the input/output (I/O) operation.

Mode	opcode	operand/address of operand
------	--------	----------------------------

Computer Registers :

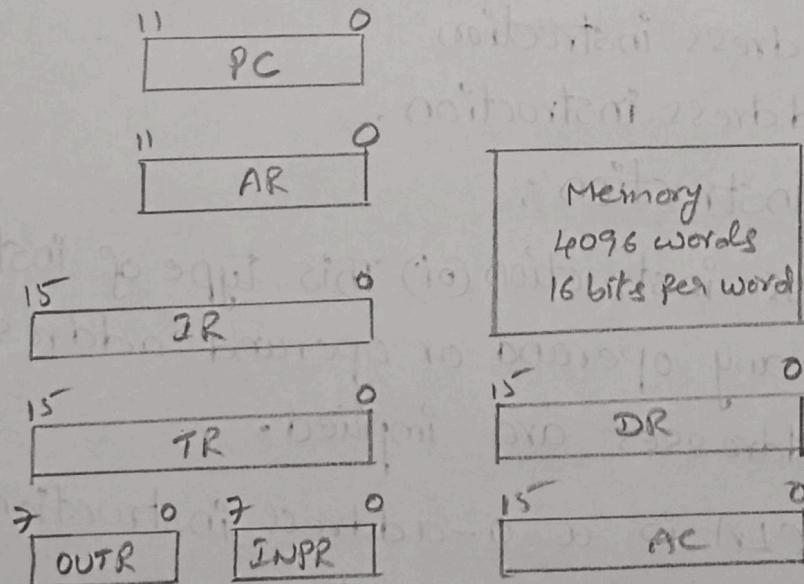
Registers are temporary storage units of a computer that keep both data and instructions in a binary form.

- Registers are electronic device that temporarily hold values in the form of 1 and 0.
- In computer organisation; the registers are a type of computer memory used to accept, store and transfer data and instructions used by the CPU right way.
- During the execution of program, registers are used to store data temporarily.

Types of Registers :

1. one register for holding data called - Data Register (DR)
2. One register for storing instruction called - Instruction Register (IR)
3. A register for holding and address of memory word called - Address Register (AR)
4. A register for holding temporary data generated during processing. This register is named as - Temporary Register. (TR)
5. A register is required for doing operations on data. This processor register holds data on which addition, subtraction, multiplication, shift and logical operations are to be carried out - A processor Register (Accumulator - Ac) (PR)

6. A register that will act as a counter and will hold the address of next instruction.
such a register is named as - program counter (PC)
7. Registers for inputting and outputting data.
INPR Input Register will hold data obtained from user through input devices like Keyboard.
OUTR Output Register will hold data that need to be sent to output devices like monitor, printer etc.



Basic Computer Registers and Memory

A computer instruction refers to a binary code that controls how a computer performs micro-operations in a series.

Every computer has its own set of instructions. In computer operation codes or opcodes and addresses are the two elements that they are divided into.

- Instructions are of different types depending on the number of operands they contain or require, to operate on.

These are.

1. zero-address instruction
2. one-address instruction
3. Two-address instruction
4. Three-address instruction.

zero - address instruction :

zero - address instruction (or) This type of instructions do not contain any operand or operand address.

- The operand addresses are implied.
- The instruction CLA is a 0-address instruction.
- CLA stands for clear accumulator.
- Here the instruction itself specifies that the operation "clear" is to be performed on Accumulator.

One-address instruction:

In this type of instruction, a single operand address is specified. The other operand lies in the accumulator and the result is also stored back in the accumulator.

Ex: Some examples of 1-address instruction.

LDA	A
STA	A
ADD	A

Two-address instruction:

In this type of instructions, addresses of two operands are specified. The result of the operation is stored in one of the given operand addresses.

Ex: MOV R₁, R₂
ADD R₁, R₂

Three-address instruction:

In three address instructions, two addresses are specified and a third address is provided for storing the result.

Ex: ADD R₁, A, B

leads to addition of operands at locations A and B and the sum of the two operands are stored in register R₁.

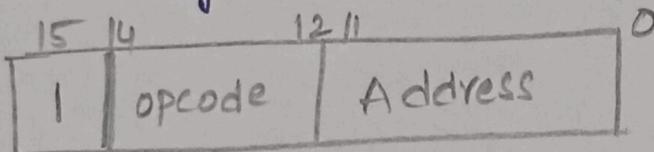
Instruction code formats (COMPUTER INSTRUCTIONS):

A basic computer has three instruction code formats which are

1. Memory - reference instruction
2. Register - reference instruction
3. Input - output instruction.

Memory reference instruction:

In memory reference instruction, 12 bits of memory is used to specify an address and one bit to specify the addressing mode 'I'.

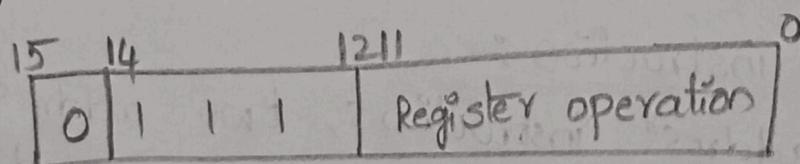


(opcode = 000 through 110)

Register reference instruction:

In register reference instructions are represented by the opcode III with a 0 in the leftmost bit (bit 15) of the instruction.

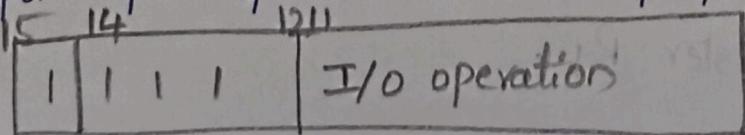
A register reference instruction specifies an operation on or a test of the AC (Accumulator) register.



(opcode = III, I=0)

Input-output instruction:

Just like register reference instruction, an input-output instruction does not need a reference to memory and is recognized by the operation code III with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.



Addressing Modes:

Instructions that define the address of a definite memory location are known as memory reference instructions. The method in which a target address or effective address is recognized within the instruction is known as addressing mode.

The address field for instruction can be represented in two different ways are as follows -

1. Direct Addressing - It uses the address of the operand.

2. Indirect Addressing - It facilitates the address of a pointer to the operand.

The address of the operand or the target address is called the effective address.

Effective Address: It defines the address that can be executed as target address for a branch type instruction or the address that can be used directly to create an operand for a computation type instruction, without creating any changes.

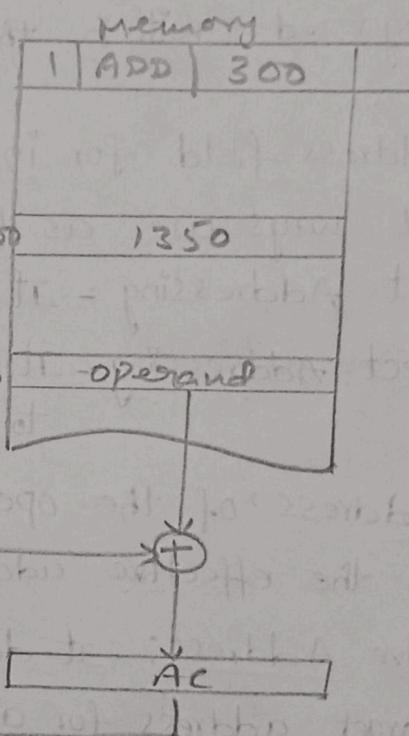
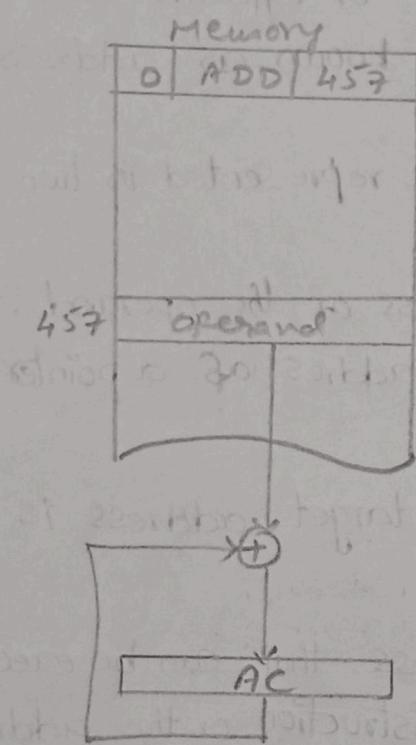
Address:

The address is represented as the locator where a specific instruction is constructed in the memory.

The address bits of an instruction code is used as an operand and not as an address.

In such methods, the instruction has an immediate operand. If the second part has an address, the instruction is referred to have a direct address.

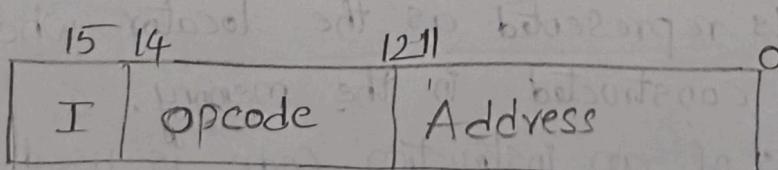
- There is another possibility in the second part including the address of operand. This is referred to as an indirect address. In the instruction code, one bit can signify if the direct or indirect address is executed.



Direct Address

Indirect Address

Demonstration of Direct and Indirect Address



Instruction format

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 flipflops and registers in the system, including the flipflops and registers in the control unit.

The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers and microoperations for the accumulator.

There are two major types of control organization:

1. Hardwired control :

The control logic is implemented with gates, flip-flops, decoders, and other digital circuits.

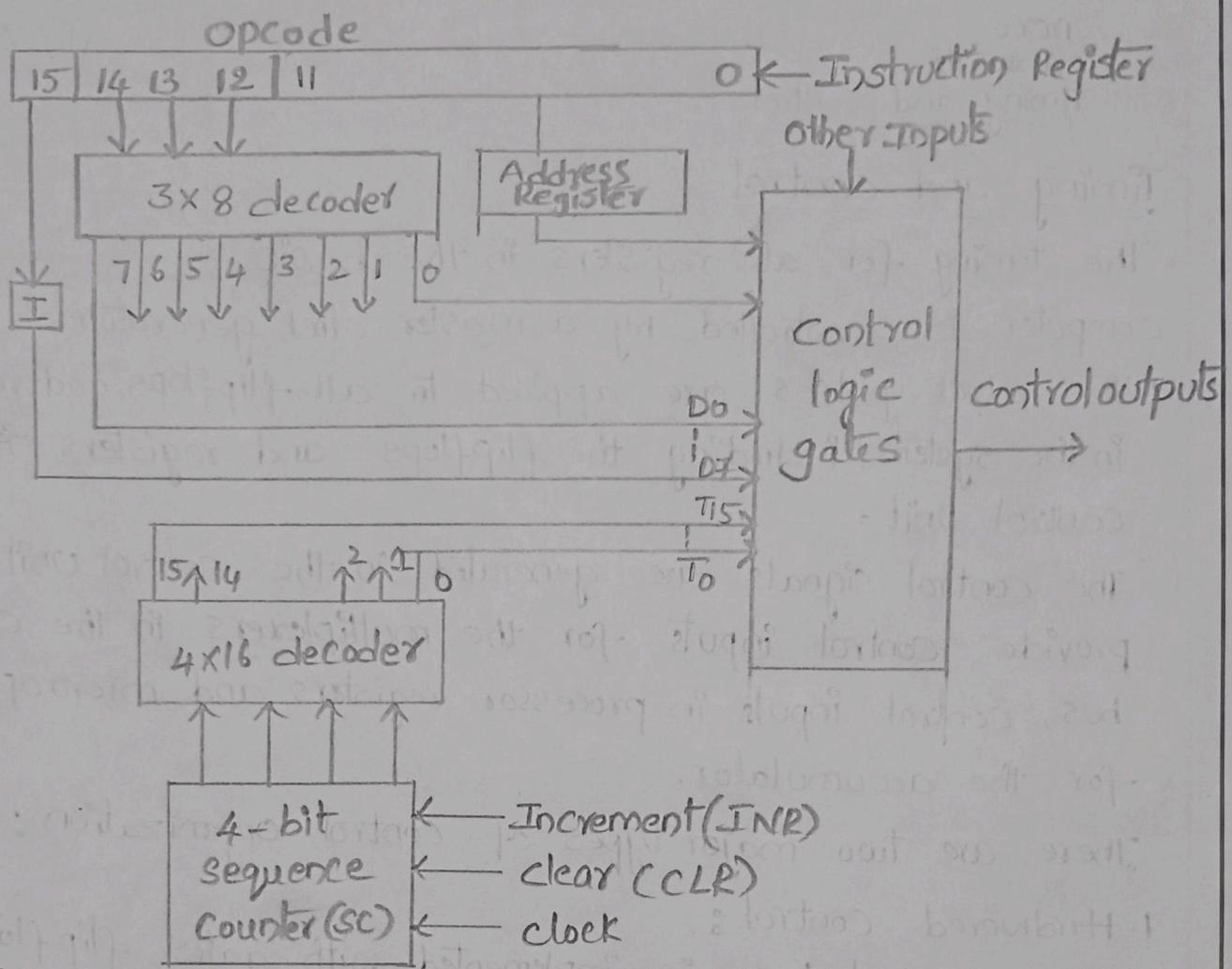
It has the advantages that it can be optimized to produce a fast mode of operation.

2. Microprogrammed control :

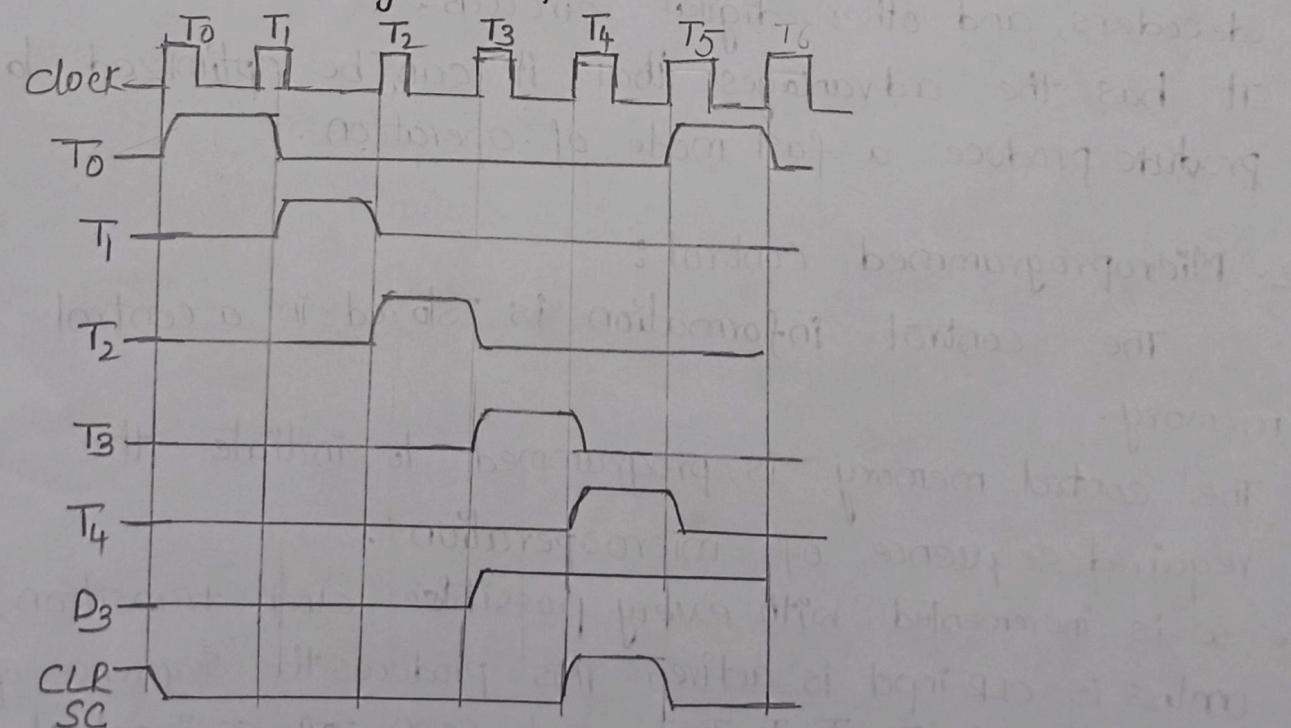
The control information is stored in a control memory.

The control memory is programmed to initiate the required sequence of microoperations.

- SC is incremented with every possible clock transition, unless its CLR input is active. This produces the sequence of timing signals T₀, T₁, T₂, T₃, T₄ and so on, if SC is not cleared, the timing signals will continue with T₅, T₆ upto T₁₅ and back to T₀.



The Block diagram of control unit



Example of control timing Signals.

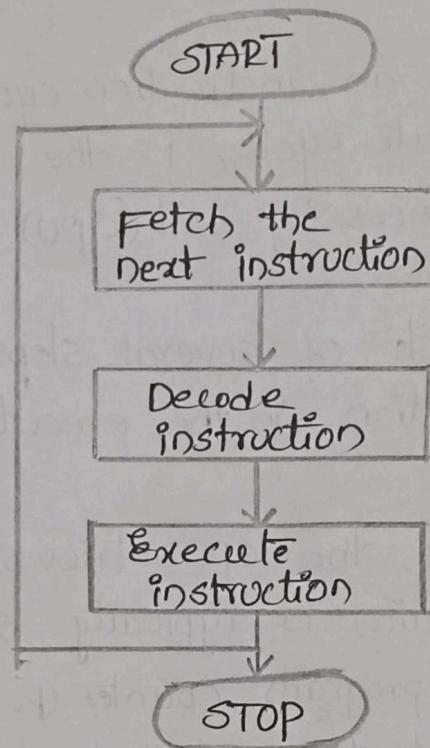
Instruction cycle

In computer organization, an instruction cycle, also known as a fetch-decode-execute cycle, is the basic operation performed by a central processing unit (CPU) to execute an instruction.

The instruction cycle consists of several steps, each of which performs a specific function in the execution of the instruction.

1. Fetch: In the fetch cycle, the CPU retrieves the instruction from memory. The instruction is typically stored at the address specified by the program counter (PC). The PC is then incremented to point to the next instruction in memory.
 2. Decode: In the decode cycle, the CPU interprets the instruction and determines what operation needs to be performed. This involves identifying the opcode and any operands that are needed to execute the instruction.
 3. Execute: In the execute cycle, the CPU performs the operation specified by the instruction. This may involve reading or writing data from or to memory, performing arithmetic or logic operations on data, or manipulating the control flow of the program.
- These cycles are the basic building blocks of the CPU's operation and are performed for every instruction executed by the CPU.

By optimizing these cycles, CPU designers can improve the performance and efficiency of the CPU, allowing it to execute instructions faster and more efficiently.



The advantages and disadvantages of the instruction cycle depend on various factors, such as the specific CPU architecture and the instruction set used.

Advantages of instruction cycle

1. Standardization
2. Efficiency
3. Pipelining

Disadvantages of instruction cycle

1. Overhead
2. Complexity
3. Limited parallelism.

This process continues indefinitely unless a HALT instruction is encountered. Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0 . After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2 , and so on.

The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$T_0 : AR \rightarrow$

$T_0 : AR \leftarrow PC$

$T_1 : IR \leftarrow M[AR], PC = PC + 1$

$T_2 : D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

It is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. place the content of memory onto the bus by making $S_2, S_1, S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR
4. Increment PC by enabling the INR input of PC.

Memory Reference Instructions :

Memory reference instructions are those instructions where the operand to be operated upon is fetched from the memory.

We have different memory reference instruction to a basic computer.

We shall use these symbols as mnemonics in assembly language programs later.

List of memory Reference Instructions .

Symbol	Description
--------	-------------

AND A - Do the logic AND operation on data stored at location A with the contents of AC .

ADD A - A + contents of AC

LDA A - load operand at A to AC

STA A - store value of AC at location A

BUN A - Branch unconditionally to address A

BSA A - Branch to A and save return address

ISZ A - Increment the value at A and skip if result is zero .

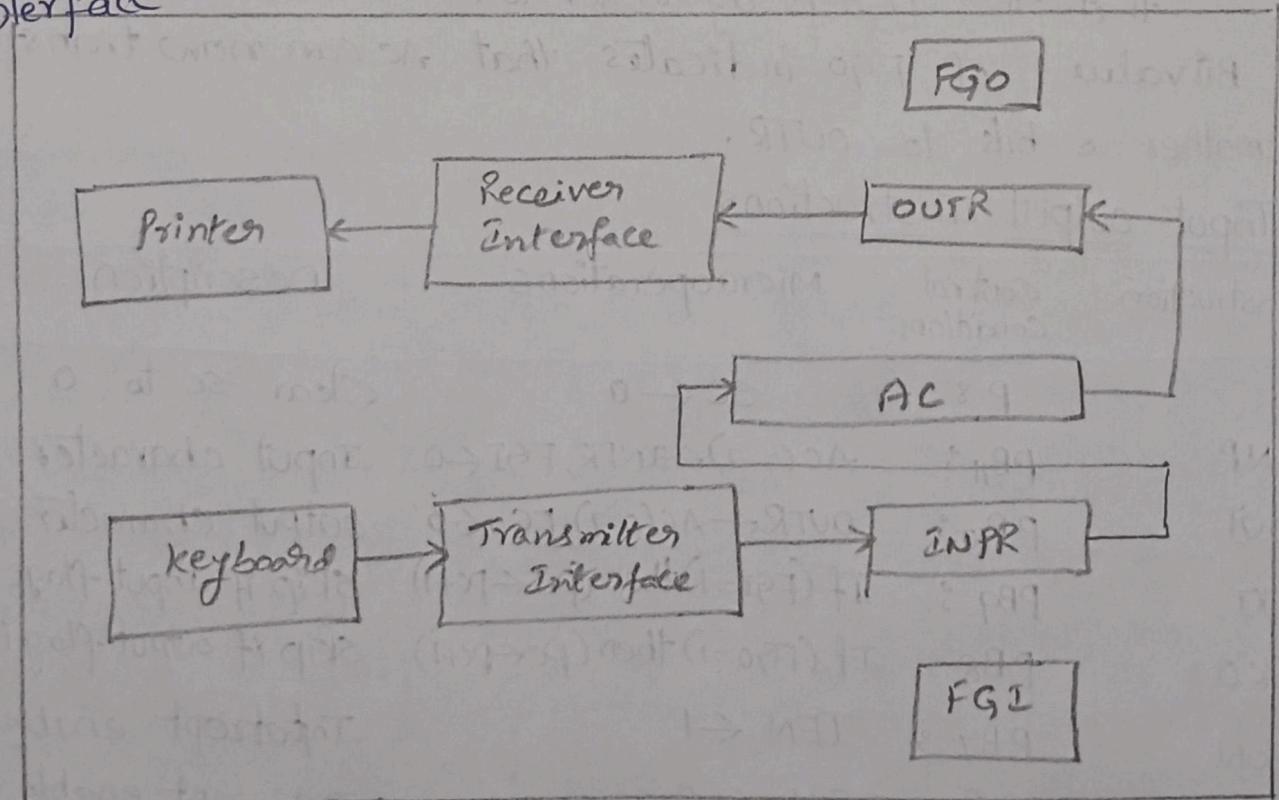
Input-output organization

Computer needs to communicate with the peripherals to take data or to transfer processed data . Instructions and data stored must come from some input device .

Similarly, the results of processing on data, must be

sent to user through some output device.

- A commercial computer will be having multiple Input-output (I/O) devices, for simplicity, we will consider a keyboard as an input device and monitor (screen) as an output device.
- Keyboard is connected to the Input Register (INPR) through a transmitter interface. The register INPR accepts 8 bits from a keyboard at a time and sends them to AC.
- AC is connected to the OUTR, which sends 8 bits at a time to the monitor or printer through a receiver interface.



Input-output configuration.

- Two single bit flipflops namely FGI and FG0 control the transfer of data between memory and I/O devices.

→ When a new information is available in input device, then the single bit FGI flipflop is set to 1. This information is transferred to AC.

After transfer has been done, FGI is set back to 0. This indicates that INPR is now free and can accept another 8 bits of input data.

No data are transferred from keyboard to INPR as long as FGI contains the bit value 1.

- Another single bit flipflop FG0 controls transmission of data between AC and OUTPR.

If FG0 is set to 1, AC transmits data to OUTPR.

If FG0 is set to 0, it is only after transmission is completed, that the FG0 is again set to 1.

Bit value 1 of FG0 indicates that AC can now transfer another 8 bits to OUTR.

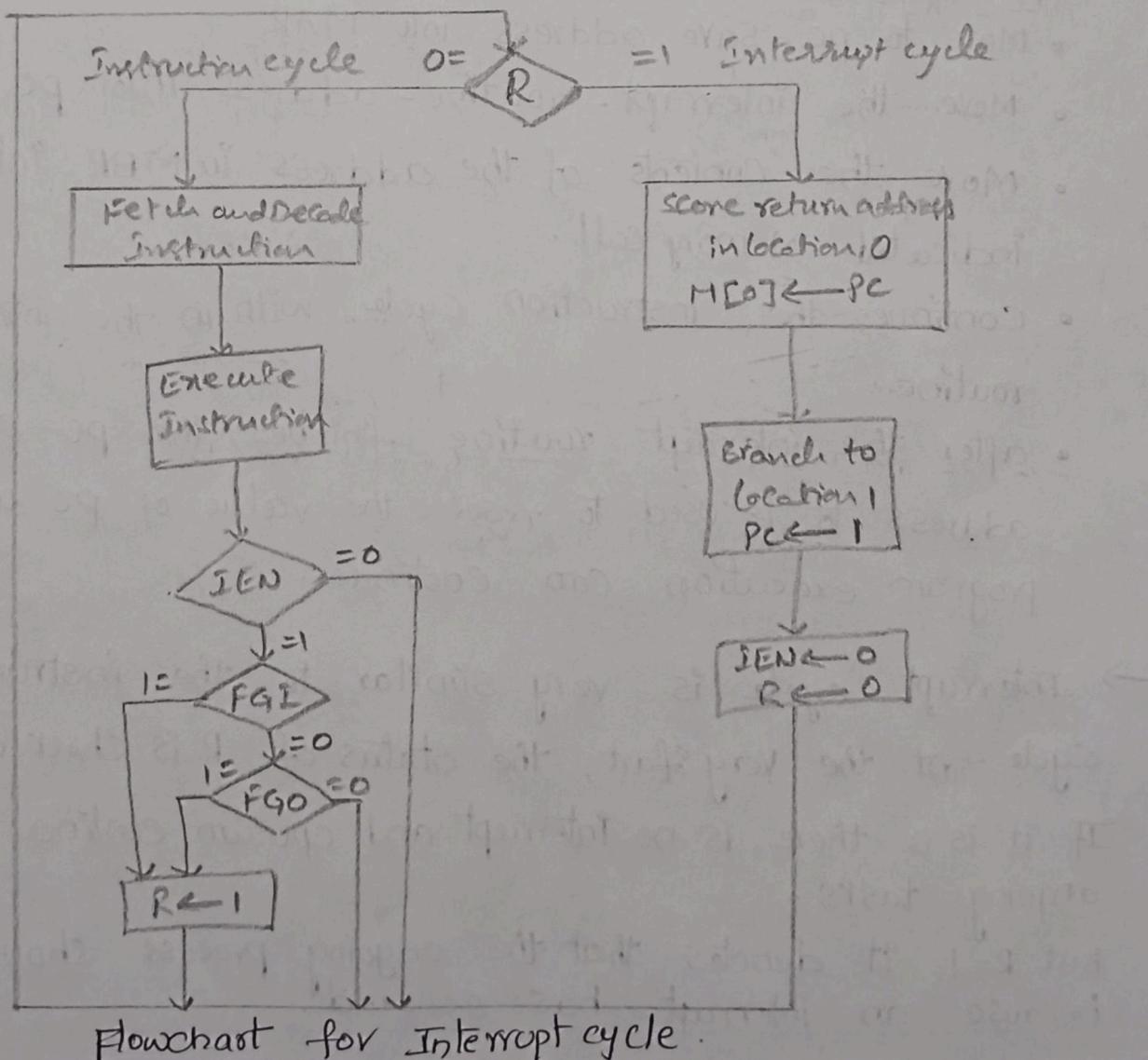
Input-output Instructions

Instruction	Control condition	Microoperations	Description
INP	P : PB11 :	SC \leftarrow 0 AC(0-7) \leftarrow INPR, FGI \leftarrow 0	Clear SC to 0 Input character
OUT	PB10 :	OUTR \leftarrow AC(0-7), FG0 \leftarrow 0	Output character
SKI	PB9 :	If (FGI = 1) then (PC \leftarrow PC + 1)	Skip if Input flag is on
SKO	PB8 :	If (FG0 = 1) then (PC \leftarrow PC + 1)	Skip if output flag is on
ION	PB7 :	IEN \leftarrow 1	Interrupt enable on
IOP	PB6 :	IEN \leftarrow 0	Interrupt enable off

Interrupt cycle

It is the process by which a Computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions.

This cycle is repeated continuously by the central processing unit (CPU), from bootupto when the computer is shutdown.



- When IEN it checks whether "FGI" or "FGo" are set to 1 then an interrupt cycle happens, but when FGI=1 it means that information in INPR cannot be changed, and FGo is the reverse to that which means that when FGo is set to 1 then the information in AC will be transferred to OUTR, OUTR can be changed.
- After the execute cycle is completed, a test is made to determine if an interrupt was enabled
(ex: so that another process can access the CPU)
- If not, instruction cycle returns to the fetch cycle.
- If so, the interrupt cycle might performs the following tasks .
 - move to the current value of PC into MBR
 - Move to PC-save address into MAR.
 - Move the interrupt-routine-address into PC.
 - Move the contents of the address in MBR into indicated Memory cell.
 - continue the instruction cycle with in the interrupt routine.
 - after the interrupt routine finishes, the PC-save address is used to reset the value of PC and program execution can continue.

→ Interrupt cycle is very similar to the instruction cycle. At the very start, the status of R is checked .

If it is 0 there is no interrupt and CPU can continue its ongoing tasks .

But R=1, it denotes that the ongoing process should halt because an interrupt has occurred.

Stack organization

A stack is a data storage structure in which the most recent thing deposited is the most recent item retrieved. It is based on the LIFO (Last In First Out) concept.

- The stack is a collection of memory locations containing a register that stores the top of element address in digital computers.

stack's operations are :

push : Adds an item to the top of the stack.

pop : Removes one item from the stack's top.

LIFO The Last In First Out list is another name for stack.

- It is the CPU's most crucial feature
- It saves information so that the last element saved is retrieved first.
- A memory space with an address register is called a stack. This register known as the stack pointer, affects the stack's address (sp).
- The Address of the element at the top of the stack is continuously influenced by the stack pointer.

Implementation of Stack

The stack can be implemented using two ways .

- Register stack
- Memory stack.

Register stack :

A stack of memory words or registers may be placed on top of each other.

Consider a 64-word register stack. A binary number which is the address of the element at the top of the stack, stored in the stack pointer register.

- The stack has the three elements A, B and C.
- The stack pointer holds C's address, which is 3.

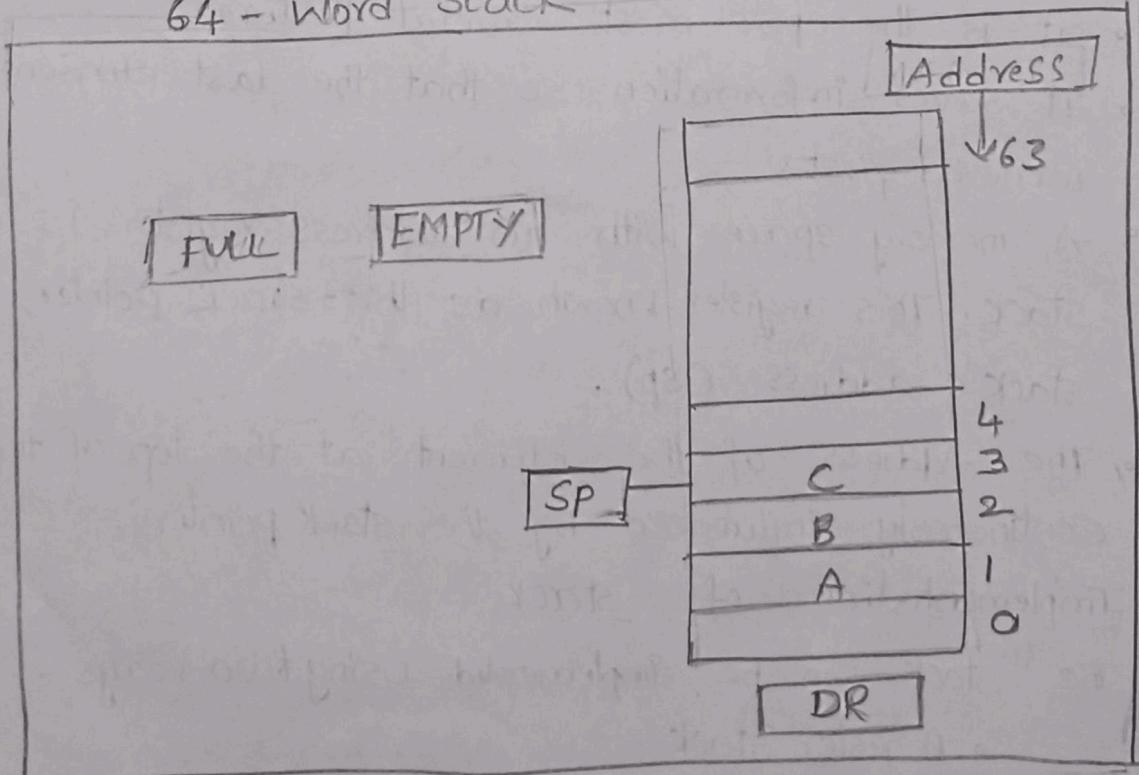
C is at the top of the stack.

The top element is removed from the stack by reading the memory word at address 3 and decreasing the stack pointer by one.

- As a result, B is at the top of the stack, and the SP is aware of B's address, which is 2.

It may add a new word to the stack by increasing the stack pointer and inserting a word in the newly increased location.

64 - Word stack



When a stack is created the pointer will point to the address 0 and the stack will be empty and not full.

Accordingly, the registers SP, EMPTY and FULL will be hold 0, 1 and 0 values respectively.

- When the FULL Register holds '0', new elements can be inserted into the stack using the push operation that involves the following sequence of microoperations.

$SP \leftarrow SP + 1$: stack pointer is incremented

$M[SP] \leftarrow DR$: Element is inserted on top of the stack

$if [SP=0] then (FULL \leftarrow 1)$: Check whether the stack is full or not

$EMPTY \leftarrow 0$: Indicate stack is not empty

- The top element of the stack can be deleted by performing pop operation only when the stack is not empty. The sequence of microoperations to pop the topmost element is listed below.

$DR \leftarrow M[SP]$: Topmost element of stack is read in DR

$SP \leftarrow SP - 1$: stack pointer is decremented

$if (SP=0) then (EMPTY \leftarrow 1)$: Check whether the stack is empty

$FULL \leftarrow 0$: To indicate that stack is not full

Memory stack

Stack operation can also be implemented in computer memories.

This can be done by assigning few segments of the main memory to store the stack, data and program instructions.

- Then the registers such as stack pointer (sp), Array Register (AR) and program counter (pc) will hold the address of topmost element of the stack, data and the instructions of the program, respectively.
- All these registers are connected to a common bus system. An additional Data Register (DR) is also used which help in communicating with the stack.
To push new element at the top of the stack, the following micro operations are performed.

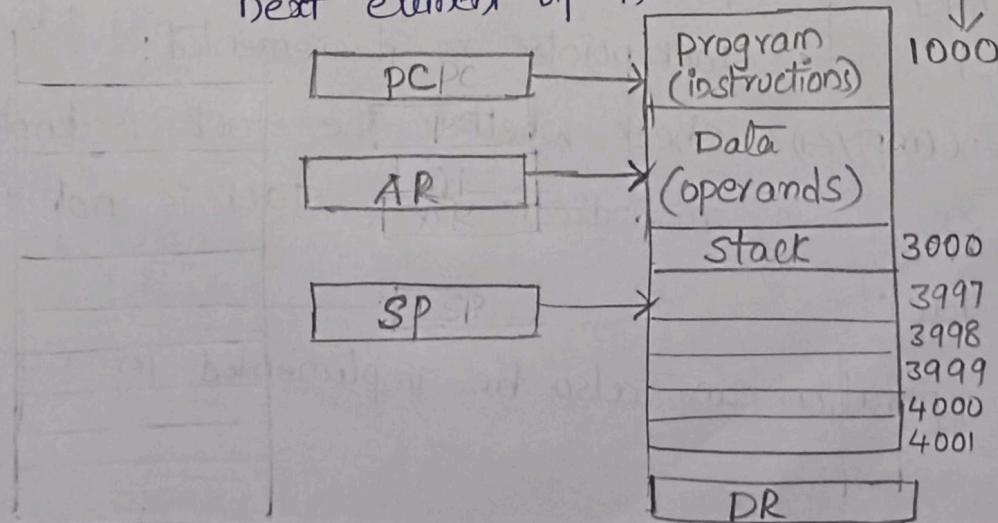
$SP \leftarrow SP - 1$: stack pointer is decremented to point to the newly inserted element

$M[SP] \leftarrow DR$: The element present in DR is inserted at top of stack

To pop the topmost element of the stack, the following micro operations are performed.

$DR \leftarrow M[SP]$: The topmost element of stack stored in DR

$SP \leftarrow SP + 1$: stack pointer is incremented to point to the next element of the stack.



polish notation:

Generally, the arithmetic expressions are written in infix notation as,

$P * Q - R * S$

In this notation, each operator is placed in between the operands.

The drawbacks while using this notation is that, it requires scanning of expression back and forth in order to find the next operation to be performed.

- The polish notation is actually the prefix notation and can be used to overcome the drawback of the infix notation,

In this notation the operator is placed before the operands

$- * P Q * R S$

Reverse polish Notation in stack

The reverse polish notation in the stack is also known as postfix expression.

Here, we use stack to solve the postfix expression.

- From the postfix expression, when some operand is found, we push it into the stack, and when some operator is found, we pop elements from the stack, and after that, the operation is performed in the correct sequence, and the result is also stored in the stack.

For example,

Given arithmetic expression is $P \times Q - R \times S$.

Postfix notation for this is $PQ \times RS \times -$

Post-fix notation can be used in place of Infix notation to overcome its drawback.

Advantages of stack organization

- Complex arithmetic statements may be rapidly calculated.
- Instruction execution is rapid because operand data is stored in consecutive memory areas
- This instructions are minimal since they don't contain an address field.

Disadvantages of stack organization

- The size of the program increases when we use a stack
- It's in memory, and memory is slower in several ways than CPU registers.
It generally has a lesser bandwidth and a longer latency.

Memory accesses are more difficult to accelerate.

Addressing Modes:

Addressing Modes in computer architecture plays a vital role in enabling efficient and flexible memory access and operand manipulation.

- Addressing modes define the rules and mechanisms by which the processor calculates the effective memory address or operand location for data operations.
- Each addressing mode has its own set of rules and mechanisms, allowing programmers to optimize memory utilization and enhance overall system performance.
- The different ways of specifying the location of an operand in an instruction are called addressing modes.

In Computer Architecture, there are following types of addressing modes -

1. Implied Addressing Mode
2. Immediate Addressing Mode
3. Direct Addressing Mode
4. Indirect Addressing Mode
5. Register direct Addressing Mode
6. Register Indirect Addressing Mode
7. Relative Addressing Mode
8. Indexed Addressing Mode
9. Base index Addressing Mode
10. Auto increment Addressing Mode
11. Auto decrement Addressing Mode

Implied Addressing Mode:

In this addressing mode,
The definition of the instruction itself specify the operands implicitly.

It is also called as implicit addressing mode.

- AS the operands are directly specified instead of their address.

Ex: Increment Accumulator.

The operand is the data present in the accumulator on which the increment operation is performed.

Immediate Addressing Mode:

In this addressing mode,
the operand is a part of the instruction instead of the contents of a register or memory location.

- The operand is specified in the instruction explicitly.

opcode	Immediate operand
--------	-------------------

Ex: ADD 10 will increment the value stored in AC by 10.

- Immediate Addressing mode has an operand field rather than address field.

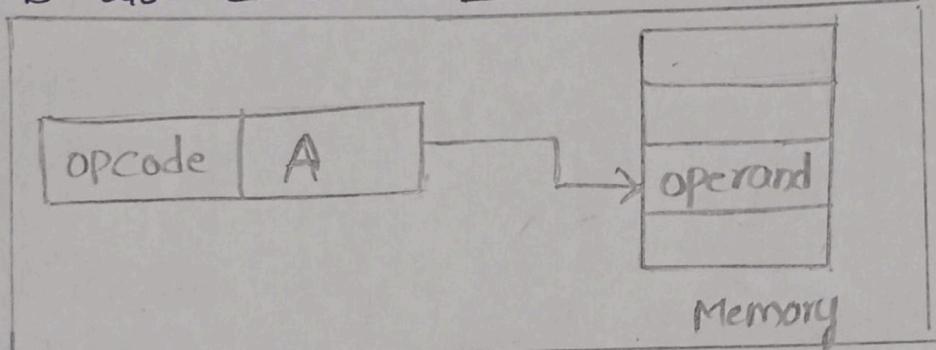
since the data are encoded directly onto the instruction, immediate operands normally represent constant data.

Direct Addressing Mode:

In this addressing mode,

- The address field of the instruction contains the effective address of the operand.

- only one reference to memory is required to fetch the operand.
- It is also called as absolute addressing mode.



Ex: ADD X

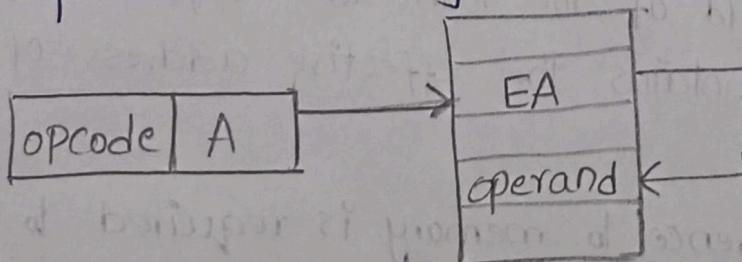
will increment the value stored in the Ac (accumulator) by the value stored at memory location X.

$$Ac \leftarrow Ac + [x]$$

Indirect Addressing Mode

In this addressing mode,

- The address field of the instruction specifies the address of memory location that contains the effective address of the operand.
- Two references to memory are required to fetch the operand.



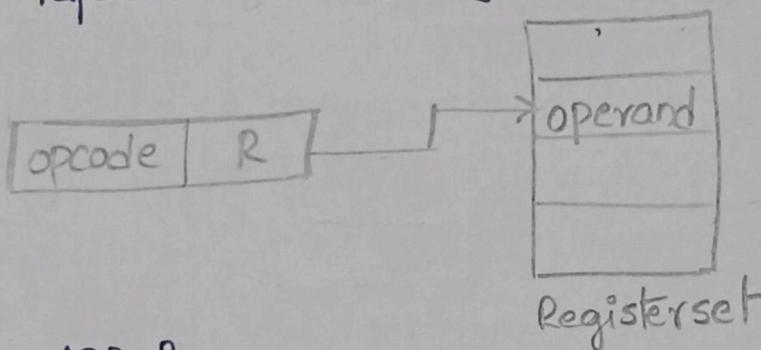
Ex: ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X

$$Ac \leftarrow Ac + [r_x]$$

Register Direct Addressing mode

In this addressing mode,

- The operand is contained in a register set.
- The address field of the instruction refers to a CPU register that contains the operand.
- No reference to memory is required to fetch the operand.



Ex: ADD R

will increment the value stored in the accumulator by the content of register R.

$$AC \leftarrow AC + [R]$$

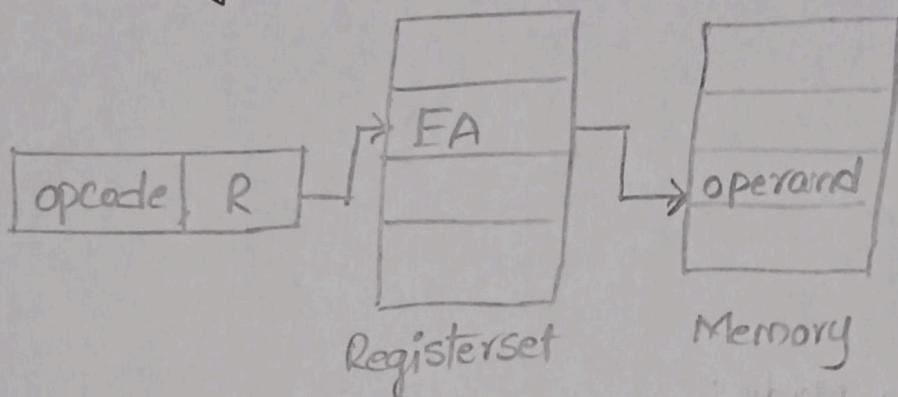
Note: Register Direct Addressing Mode is similar to Direct addressing mode.

The only difference is address field of the instruction refers to a CPU register instead of main memory.

Register Indirect Addressing mode -

In this addressing mode,

- The address field of the instruction refers to a CPU register that contains the effective address of the operand.
- only one reference to memory is required to fetch the operand.



Ex: ADD R

Will increment the value stored in the accumulator by the content of memory location specified in Register R

$$AC \leftarrow AC + [R]$$

NOTE: Register Indirect mode is similar to indirect addressing mode. The only difference is address field of the instruction refers to a CPU register.

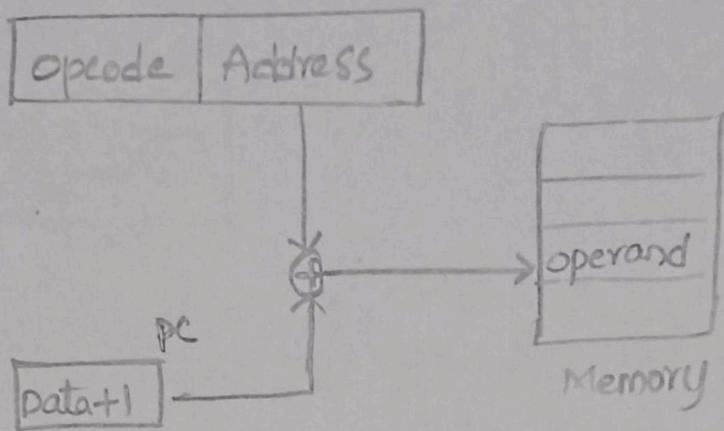
Relative Addressing mode:

In this addressing mode,

Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.

$$\text{Effective Address} = \text{Content of program counter} + \text{Address part of the instruction}$$

- program counter (pc) always contains the address of the next instruction to be executed.
- After fetching the address of instruction, the value of program counter immediately increases
- The value increases irrespective of whether the fetched instruction has completely executed or not.



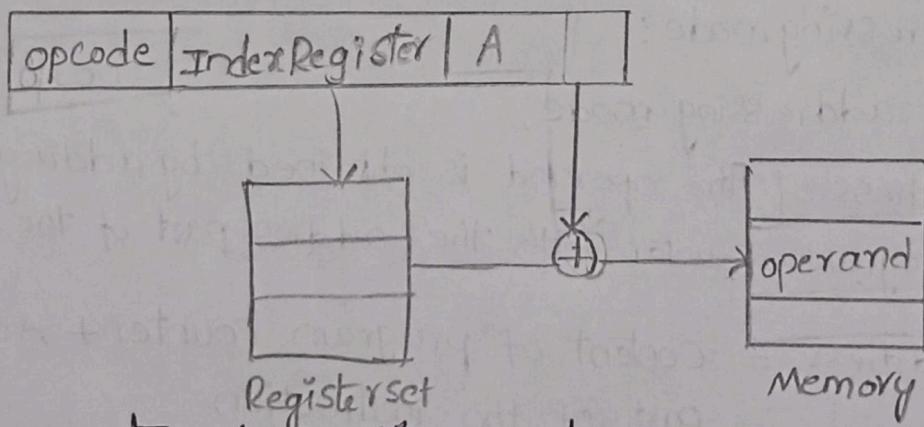
Indexed Addressing Mode:

In this addressing mode,

the instruction contains an address.

- The address is added to the data present in the index register to get the effective address.

Effective address = Content of Index Register [IR] + Address specified in the instruction.



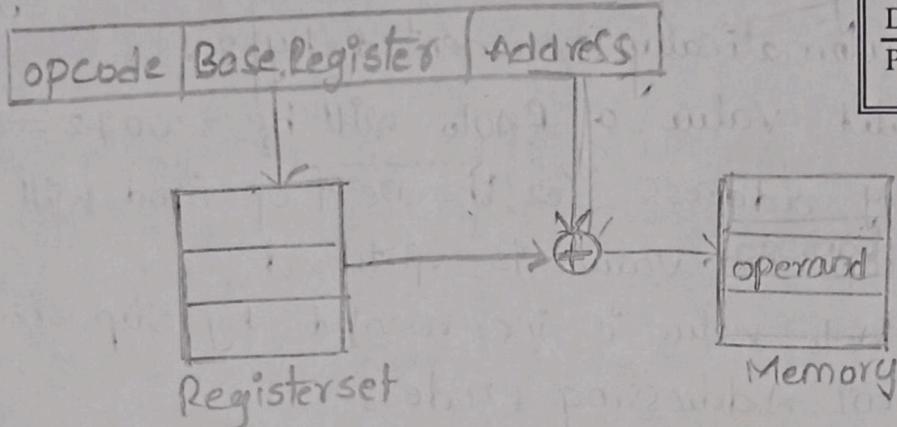
Base Register Addressing Mode:

In this addressing mode,

the instruction contains an address.

- This address is added to the data present in the base register to get the effective address

Effective address = Content of Base Register [BR] + Address specified in the instruction



AutoIncrement Addressing mode:

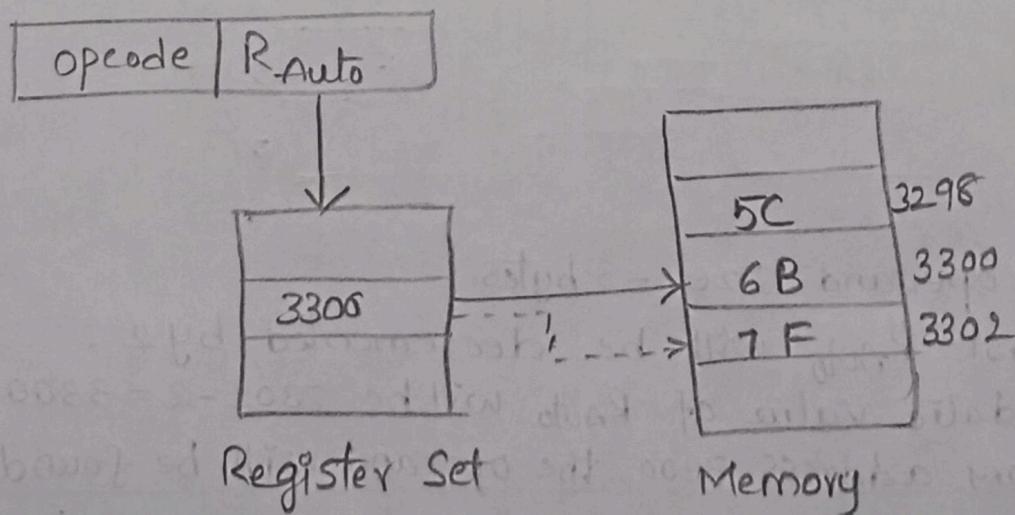
This addressing mode is a special case of Register-indirect Addressing mode.

Where

Effective Address of the operand = content of Register

In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size 'd'.
- Step size 'd' depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.



Assume operand size = 2 bytes

Here,

- After fetching the operand 6B, the instruction register R_{Auto} will be automatically incremented by 2.
- Then, updated value of R_{Auto} will be $3300 + 2 = 3302$
- At memory address 3302, the next operand will be found.

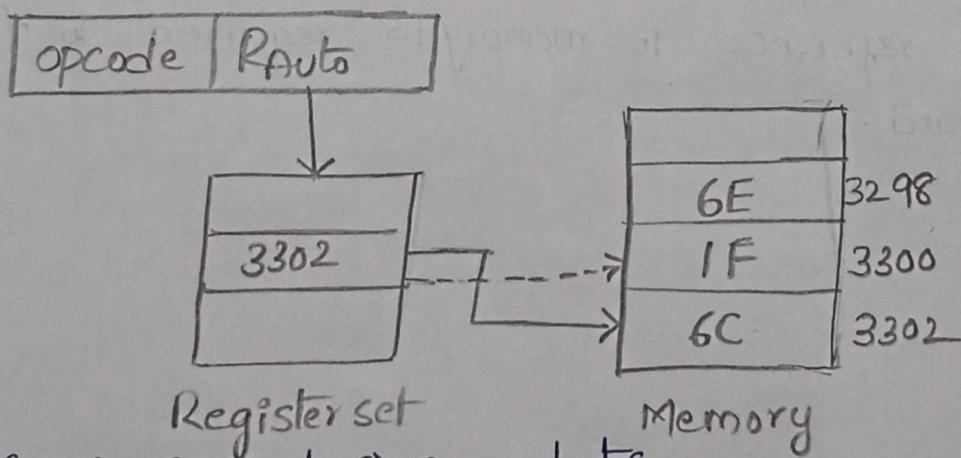
NOTE: 1) First operand value is fetched
2) Then R_{Auto} value is incremented by step size 'd'.

Auto-Decrement Addressing Mode :-

In this addressing mode,

Effective Address of the operand = content of Register - stepsize.

- First, the content of the register is decremented by step size 'd'.
- Step size 'd' depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.



Assume operand size = 2 bytes

Here, First R_{Auto} will be decremented by 2.

Then, updated value of R_{Auto} will be $3302 - 2 = 3300$

At Memory address 3300, the operand will be found.

Data transfer Instructions

Data transfer Instructions transfer the data between memory and processor registers, I/o devices and processor registers and from one processor register to another.

- There are 8 commonly used data transfer instructions.
Each instruction is represented by a mnemonic symbol.

Instruction Name	symbol used	Description
push	PUSH	It is used to direct a word of data on to the top of a stack.
pop	POP	It causes data word to be removed from the top of a stack
Input	IN	data to be loaded from a given source (I/o) into the memory or registers.
output	OUT	data to be transmitted from given memory or registers to the destination
load	LD	(I/o port, I/o devices etc.) transfer of data from given memory location to the processor for further computations
store	ST	reverse of load operation i.e data gets stored in memory
Move	Mov	transfer of data from source to destination.
Exchange	XCH	It causes swapping operation i.e the contents of source is copied into the destination and viceversa.

Data Manipulation Instructions:

Data Manipulation Instructions perform operations on data and provide computational capabilities for the computer.

- The data manipulation instructions in a typical computer are usually divided into three basic types.

1. Arithmetic instructions

2. Logical and bit manipulation instructions

3. Shift instructions

Arithmetic instructions:

As the name suggests arithmetic instructions cause arithmetic operations (add, subtract, division and multiplication etc) to be performed on a given set of data.

Instruction Name	Mnemonic Symbol	Description
Addition	ADD	It adds the values of two operands
Subtraction	SUB	It subtracts the values of two operands
Multiplication	MUL	It multiplies the two operands
Division	DIV	It divides the two operands by considering one as dividend & other as divisor and the result of this operation is the quotient.
Increment	INC	It increments the value of given operand by '1'
Decrement	DEC	It decrements the value of given operand by '1'.
Add with carry	ADDC	the addition operations b/w two operands and considers even its carry bit.
Subtract with borrow	SUB B	the subtraction operation b/w two borrow operands and considers even the borrow bit
Negation	NEG	It simply changes the sign of a given operand value.

logical and Bit Manipulation Instructions

logical instructions perform binary operations on strings of bits stored in registers. They are helpful for manipulating individual bits or a group of bits.

Instruction Name	Mnemonic Symbol	Description
logical AND	AND	It performs logical AND operation between the values of two registers.
logical OR	OR	It performs logical OR operation between the values of two registers.
logical exclusive OR	XOR	It performs logical ex-OR operation between the values of two registers.
clear carry	CLRC	It clear or vacant the register maintaining the carry bit.
set carry	SETC	It enables the carry register
clear	CLR	It empties or vacant the given register
complement	COM	It inverts the bits of a given register
Enable Interrupt	EI	It enables an interrupt
Disable Interrupt	DI	It disables an interrupt
complement carry	COMC	It complements to invert the carry bit.

Shift Instruction

shift operations in which the bits of a word are moved to the left or right.

shift instructions may specify either logical shifts, arithmetic shifts or rotate type operations.

Instruction Name	Mnemonic symbol	Description
logical shift left	SHL	shifted to 1 bit position towards its left excluding the sign bit.
logical shift right	SHR	shift right to be shifted 1 bit position towards its right excluding the sign bit.
Arithmetic shift right	SHRA	instructions to be shifted 1 bit position towards its right including the sign bit. But the sign bit remains unchanged
Arithmetic shift left	SHLA	shifted 1 bit pos towards its left, including the sign bit. But sign bit remains unchanged.
Rotate right	ROR	shifted 1 bit position towards its right in a circular manner i.e, the last bit of the register forming the 1 st bit and rest of bits shifting towards their right by 1 bit position.
Rotate left	ROL	shifted 1 bit position towards their left in a circular manner.
Rotate right through carry	RORC	same as ROR, except the fact that, it even carry includes the sign bit.
Rotate left carry through	ROL C	Same as ROL, except the fact that, it even carry includes the sign bit.

CHAPTER FOUR

Register Transfer and Microoperations

IN THIS CHAPTER

- 4-1** Register Transfer Language
- 4-2** Register Transfer
- 4-3** Bus and Memory Transfers
- 4-4** Arithmetic Microoperations
- 4-5** Logic Microoperations
- 4-6** Shift Microoperations
- 4-7** Arithmetic Logic Shift Unit

4-1 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an 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 register. Examples of microoperations are shift, count, clear, and load. Some of the digital components introduced in Chap. 2 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the micro-

microoperation

operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The internal hardware organization of a digital computer is best defined by specifying:

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.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4-2 Register Transfer

registers

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR*. Other designations for registers are *PC* (for program counter), *IR* (for instruction register, and *R1* (for processor register). The individual flip-flops in an n -bit register are numbered in sequence from 0 through $n - 1$, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is *PC*. The symbol *PC(0-7)* or *PC(L)* refers to the low-order byte and *PC(8-15)* or *PC(H)* to the high-order byte.

register transfer

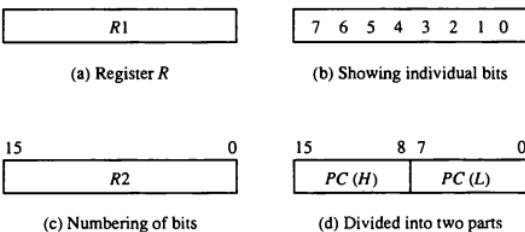
Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$R2 \leftarrow R1$

denotes a transfer of the content of register *R1* into register *R2*. It designates a replacement of the content of *R2* by the content of *R1*. By definition, the content of the source register *R1* does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Nor-

Figure 4-1 Block diagram of register.



mally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an *if-then* statement.

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

control function

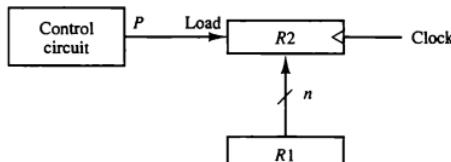
where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a *control function*. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$P: R2 \leftarrow R1$

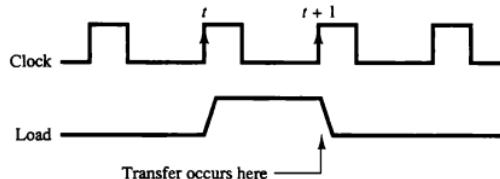
The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $P = 1$.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load input that is activated by the control variable P . It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown

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



(a) Block diagram



(b) Timing diagram

in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t . The next positive transition of the clock at time $t + 1$ finds the load input active and the data inputs of $R2$ are then loaded into the register in parallel. P may go back to 0 at time $t + 1$; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time t , the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time $t + 1$.

The basic symbols of the register transfer notation are listed in Table 4-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

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

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.

TABLE 4-1 Basic Symbols for Register Transfers

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

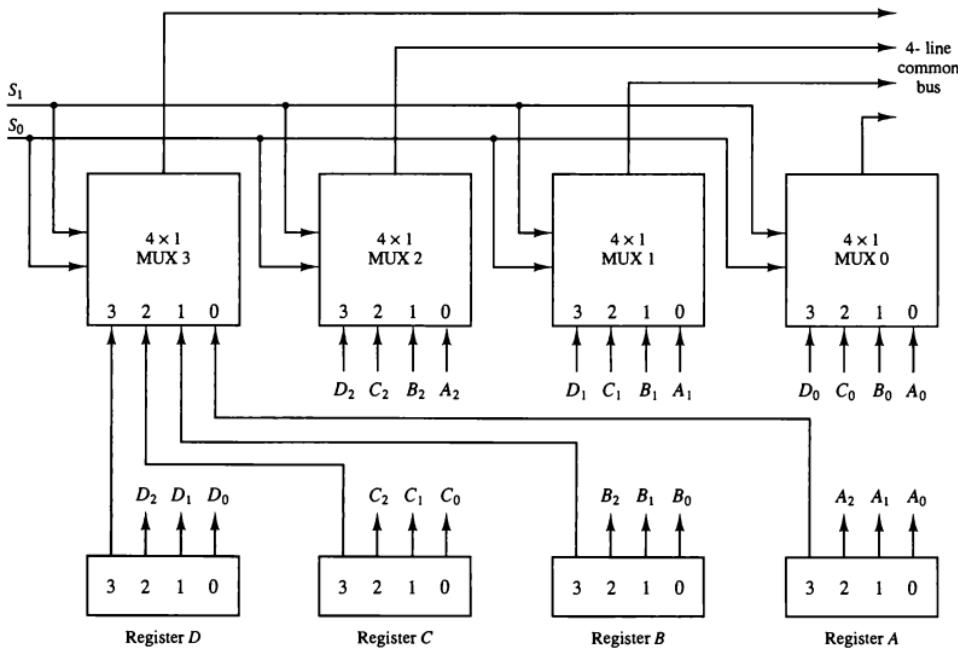
4-3 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals

determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 4-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4×1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0 . In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A_1 . The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

Figure 4-3 Bus system for four registers.



bus selection

The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if $S_1S_0 = 01$, and so on. Table 4-2 shows the register that is selected by the bus for each of the four possible binary value 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

In general, a bus system will multiplex k registers of n bits each to produce an n -line common bus. The number of multiplexers needed to construct the bus is equal to n , the number of bits in each register. The size of each multiplexer must be $k \times 1$ since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one 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 the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, \quad R1 \leftarrow BUS$$

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

Three-State Bus Buffers

three-state gate

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 as in a conventional gate. The third state is a *high-impedance state*. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in Fig. 4-4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

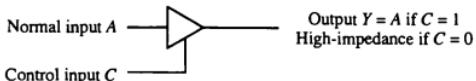
bus system

The construction of a bus system with three-state buffers is demonstrated in Fig. 4-5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any 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 a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 4-5 is another way of constructing a 4×1 multiplexer since the circuit can replace the multiplexer in Fig. 4-3.

To construct a common bus for four registers of n bits each using three-

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



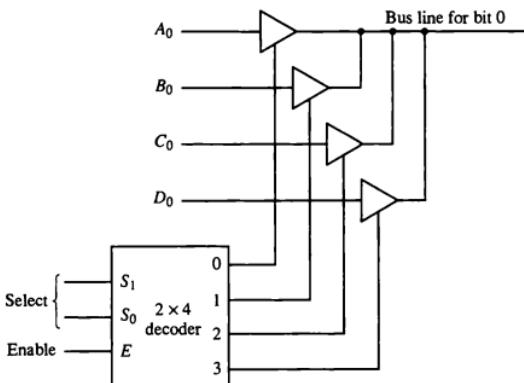


Figure 4-5 Bus line with three state-buffers.

state buffers, we need n circuits with four buffers in each as shown in Fig. 4-5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four registers.

Memory Transfer

The operation of a memory unit was described in Sec. 2-7. The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter M . The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M .

Consider a memory unit that receives the address from a register, called the address register, symbolized by AR . The data are transferred to another register, called the data register, symbolized by DR . The read operation can be stated as follows:

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

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

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register $R1$

memory read

memory write

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

$\overline{R2}$ is the symbol for the 1's complement of $R2$. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of $R1$ to the 2's complement of $R2$ is equivalent to $R1 - R2$.

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

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

The arithmetic operations of multiply and divide are not listed in Table 4-3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). 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 con-

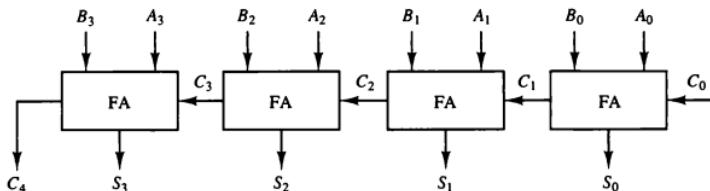


Figure 4-6 4-bit binary adder.

nected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4 . The S outputs of the full-adders generate the required sum bits.

An n -bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as $R1$), and the n data bits for the B inputs come from another register (such as $R2$). The sum can be transferred to a third register or to one of the source registers ($R1$ or $R2$), replacing its previous content.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

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-subtractor circuit is shown in Fig. 4-7. The mode input M controls the operation. When $M = 0$ the circuit is an adder and when $M = 1$ the circuit becomes a 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 performs 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

full-adder

adder-subtractor

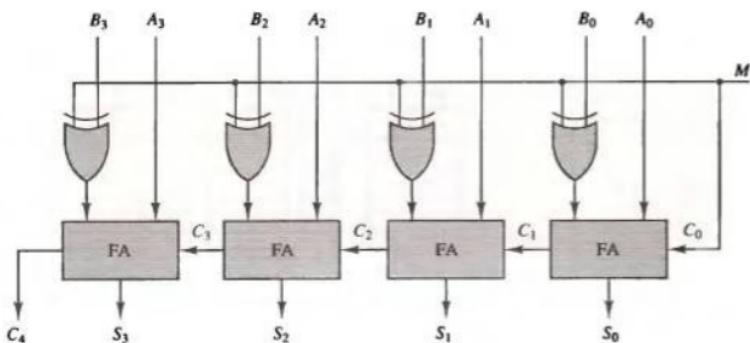


Figure 4-7 4-bit adder-subtractor.

2's complement of B . For unsigned numbers, this gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$ provided that there is no overflow.

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. This microoperation is easily implemented with a binary counter (see Fig. 2-10). Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders (see Fig. 1-16) connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 4-8. 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 through A_3 , adds one to it, and generates the incremented output in S_0 through S_3 . The output carry C_4 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. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

incrementer

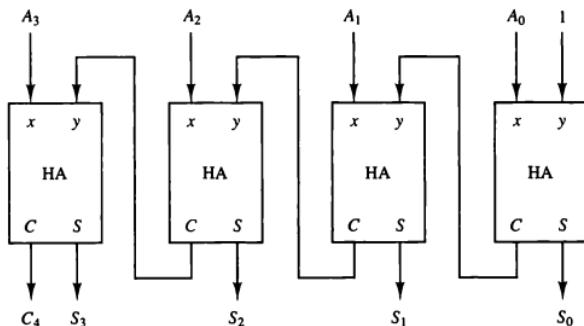


Figure 4-8 4-bit binary incrementer.

Arithmetic Circuit

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. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S_1 and S_0 . The input carry C_{in} 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. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol $+$ in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4-4.

input carry

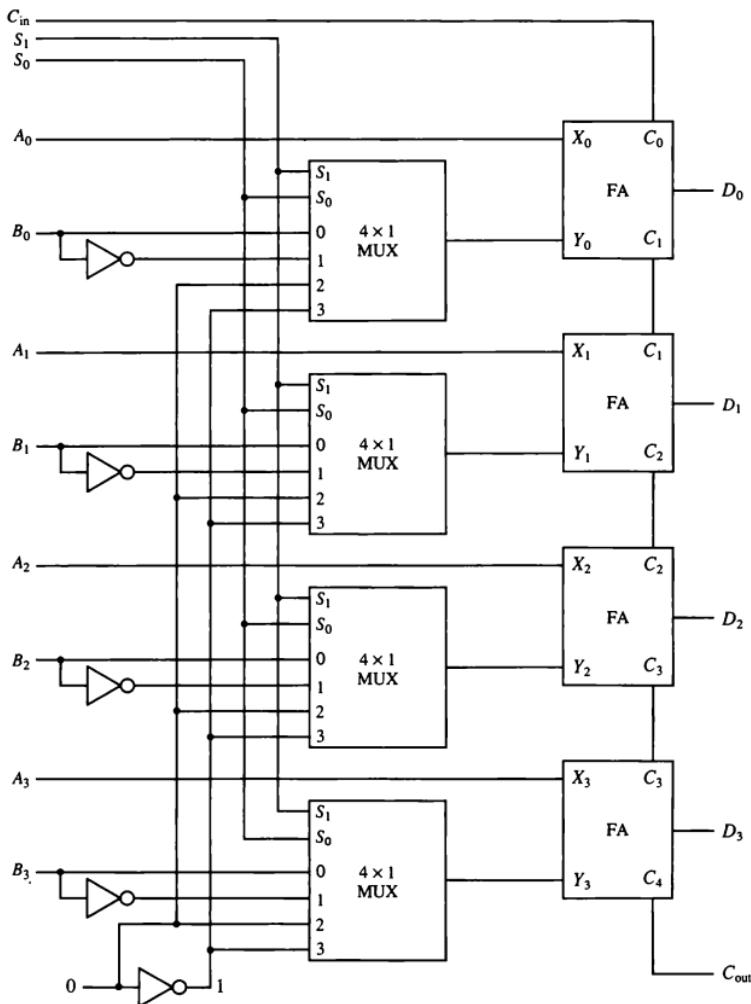


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

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

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.

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

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.

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 $F = A + 2^k$'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.

4-5 Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement

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

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable $P = 1$. As a numerical example, assume that each register has four bits. Let the content of $R1$ be 1010 and the content of $R2$ be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

$$\begin{array}{ll} 1010 & \text{Content of } R1 \\ \underline{1100} & \text{Content of } R2 \\ 0110 & \text{Content of } R1 \text{ after } P = 1 \end{array}$$

The content of $R1$, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in $R2$ and previous values of $R1$. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol \vee will be used to denote an OR microoperation and the symbol \wedge to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol $+$, when used to symbolize an arithmetic plus, from a logic OR operation. Although the $+$ symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol $+$ occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

$$P + Q: R1 \leftarrow R2 + R3, \quad R4 \leftarrow R5 \vee R6$$

the $+$ between P and Q is an OR operation between two binary variables of a control function. The $+$ between $R2$ and $R3$ specifies an add microoperation. The OR microoperation is designated by the symbol \vee between registers $R5$ and $R6$.

List of Logic Microoperations

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 shown in Table 4-5. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the

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

two variables x and y . Note that the functions are determined from the 16 binary combinations that can be assigned to F .

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B . It is important to realize that the Boolean functions listed in the first column of Table 4-6 represent a relationship between two binary variables x and y . The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B . Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

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

Hardware Implementation

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

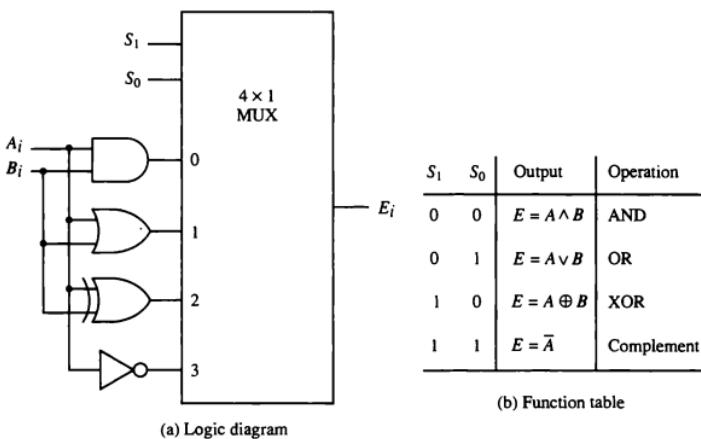
logic circuit

Figure 4-10 shows one stage of 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 through 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 S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i . For a logic circuit with n bits, the diagram must be repeated n times for $i = 0, 1, 2, \dots, n - 1$. The selection variables are applied to all stages. The function table in Fig. 4-10(b) lists the logic microoperations obtained for each combination of the selection variables.

Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a 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. The following examples show how the bits of one register (designated by A) are manipulated

Figure 4-10 One stage of logic circuit.



by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B .

selective-set

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B . It does not affect bit positions that have 0's in B . The following numerical example clarifies this operation:

1010	A before
<u>1100</u>	B (logic operand)
1110	A after

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A . Therefore, the OR microoperation can be used to selectively set bits of a register.

selective-complement

The *selective-complement* operation complements bits in A where there are corresponding 1's in B . It does not affect bit positions that have 0's in B . For example:

1010	A before
<u>1100</u>	B (logic operand)
0110	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

selective-clear

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

1010	A before
<u>1100</u>	B (logic operand)
0010	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB' . The corresponding logic microoperation is

$$A \leftarrow A \wedge \bar{B}$$

The *mask* operation is similar to the selective-clear operation except that the bits of *A* are cleared only where there are corresponding 0's in *B*. The mask operation is an AND micro operation as seen from the following numerical example:

$$\begin{array}{rcl} 1010 & \text{A before} \\ 1100 & \text{B (logic operand)} \\ \hline 1000 & \text{A after masking} \end{array}$$

The two rightmost bits of *A* are cleared because the corresponding bits of *B* are 0's. The two leftmost bits are left unchanged because the corresponding bits of *B* are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The *insert* 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:

$$\begin{array}{rcl} 0110\ 1010 & \text{A before} \\ 0000\ 1111 & \text{B (mask)} \\ \hline 0000\ 1010 & \text{A after masking} \end{array}$$

and then insert the new value:

$$\begin{array}{rcl} 0000\ 1010 & \text{A before} \\ 1001\ 0000 & \text{B (insert)} \\ \hline 1001\ 1010 & \text{A after insertion} \end{array}$$

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares 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 by the following example:

$$\begin{array}{rcl} 1010 & \text{A} \\ 1010 & \text{B} \\ \hline 0000 & A \leftarrow A \oplus B \end{array}$$

When *A* and *B* are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

4-6 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

logical shift

A *logical shift* is one that transfers 0 through the serial input. We will adopt the symbols *shl* and *shr* for logical shift-left and shift-right microoperations. For example:

$$R1 \leftarrow \text{shl } R1$$

$$R2 \leftarrow \text{shr } R2$$

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 register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

circular shift

The *circular shift* (also known as a *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. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 4-7.

TABLE 4-7 Shift Microoperations

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

arithmetic shift

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 sign bit unchanged because the sign of the number remains the same

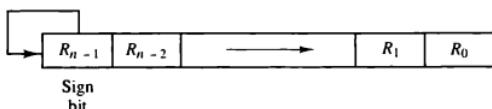


Figure 4-11 Arithmetic shift right.

when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, 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. Figure 4-11 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} , and so on for the other bits in the register. The bit in R_0 is lost.

The arithmetic shift-left inserts a 0 into R_0 , and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced 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 an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2} . An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

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

If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

Hardware Implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load (see Fig. 2-9). Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A_0 through A_3 , and four data outputs, H_0 through H_3 . There are two serial inputs, one for shift left

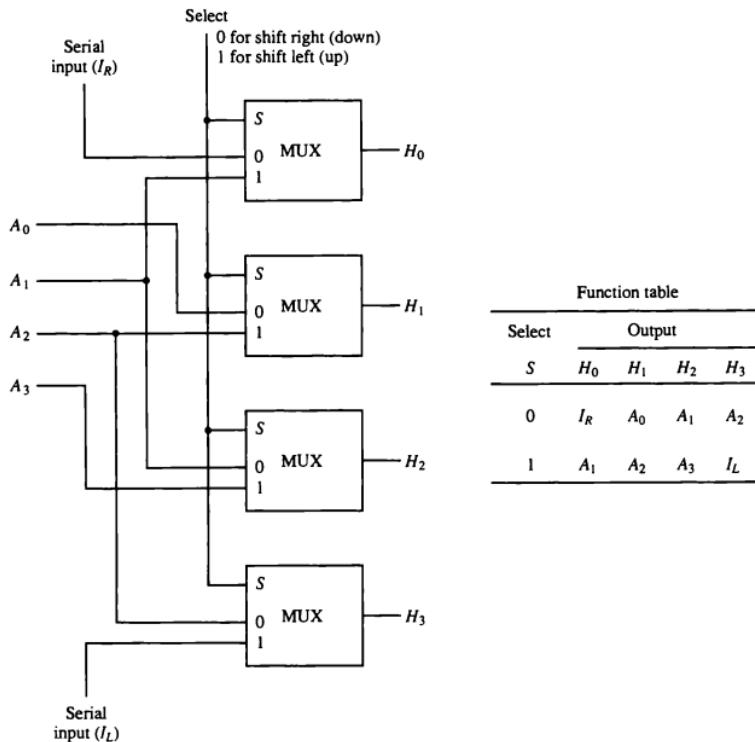


Figure 4-12 4-bit combinational circuit shifter.

(I_L) and the other for shift right (I_R). When the selection input $S = 0$, the input data are shifted right (down in the diagram). When $S = 1$, the input data are shifted left (up in the diagram). The function table in Fig. 4-12 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

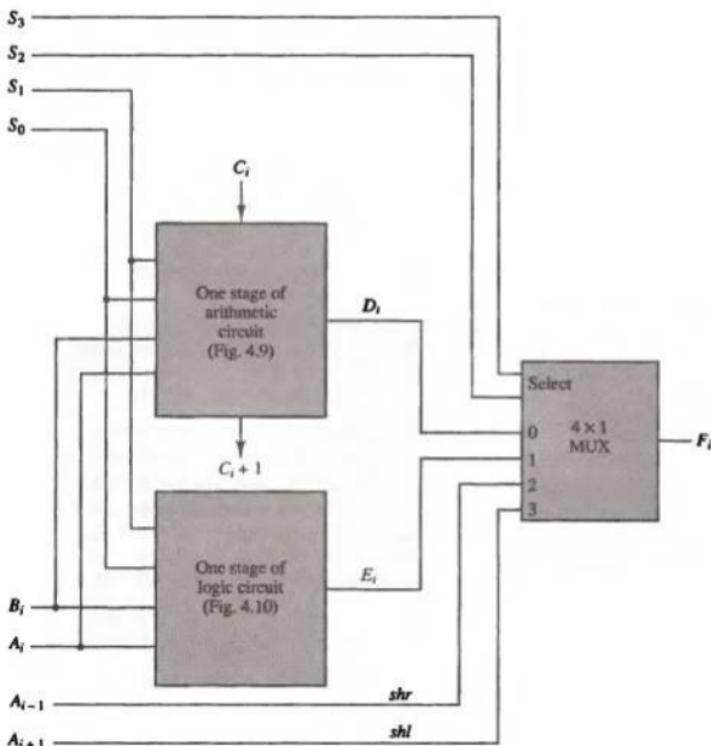
4-7 Arithmetic Logic Shift Unit

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To

perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4-13. The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic

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



units. A particular microoperation is selected with inputs S_1 and S_0 . A 4×1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i . The data in the multiplexer are selected with inputs S_3 and S_2 . The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig. 4-13 must be repeated n times for an n -bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in} , which provides a selection variable for the arithmetic operations.

The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0 , and C_{in} . The input carry C_{in} is used for selecting an arithmetic operation only.

Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations (see Table 4-4) and are selected with $S_3S_2 = 00$. The next four are logic operations (see Fig. 4-10) 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 \times 's. 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					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \bar{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \bar{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	\times	$F = A \wedge B$	AND
0	1	0	1	\times	$F = A \vee B$	OR
0	1	1	0	\times	$F = A \oplus B$	XOR
0	1	1	1	\times	$F = \bar{A}$	Complement A
1	0	\times	\times	\times	$F = \text{shr } A$	Shift right A into F
1	1	\times	\times	\times	$F = \text{shl } A$	Shift left A into F

CHAPTER NINE

Pipeline and Vector Processing

IN THIS CHAPTER

- 9-1 Parallel Processing
- 9-2 Pipelining
- 9-3 Arithmetic Pipeline
- 9-4 Instruction Pipeline
- 9-5 RISC Pipeline
- 9-6 Vector Processing
- 9-7 Array Processors

9-1 Parallel Processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing, and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time,

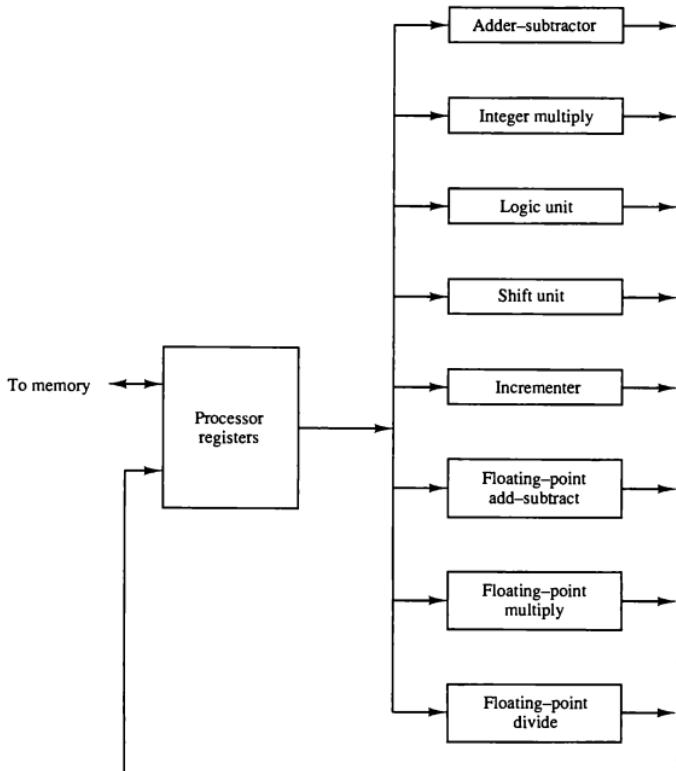
throughput

while registers with parallel load operate with all the bits of the word simultaneously. Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

multiple functional units

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruc-

Figure 9-1 Processor with multiple functional units.



tion associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an *instruction stream*. The operations performed on the data in the processor constitutes a *data stream*. Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction stream, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit. It emphasizes the be-

SIMD

MIMD

havioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors discussed in Sec. 9-7, and MIMD multiprocessors presented in Chap. 13.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

9-2 Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

The pipeline organization will be demonstrated by means of a simple

example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. R_1 through R_5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into R_1 and

Figure 9-2 Example of pipeline processing.

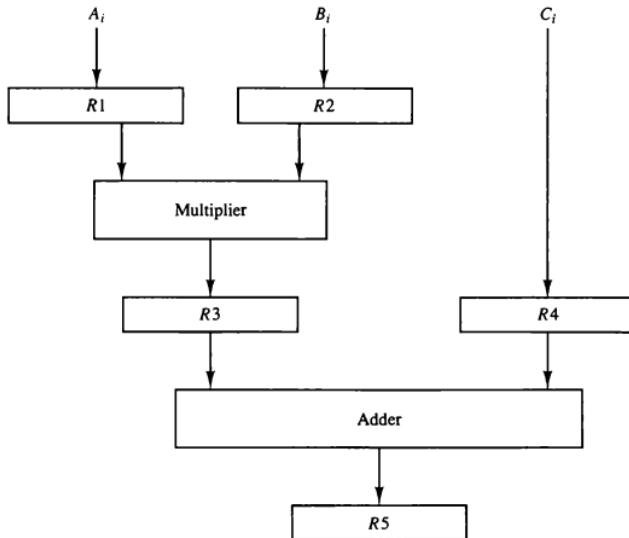


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

R2. The second clock pulse transfers the product of R1 and R2 into R3 and C₁ into R4. The same clock pulse transfers A₂ and B₂ into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A₃ and B₃ into R1 and R2, transfers the product of R1 and R2 into R3, transfers C₂ into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

General Considerations

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a *task* as the total operation performed going through all the segments in the pipeline.

task

space-time diagram

The behavior of a pipeline can be illustrated with a *space-time diagram*. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives the

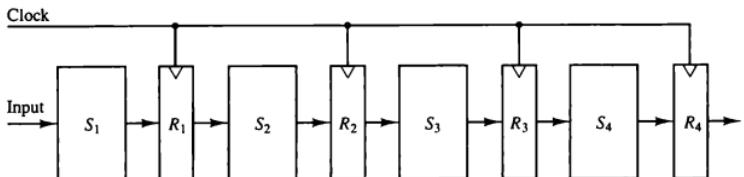


Figure 9-3 Four-segment pipeline.

segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1 , while segment 1 is busy with task T_2 . Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a k -segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining $n - 1$ tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete n tasks using a k -segment pipeline requires $k + (n - 1)$ clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that performs the same operation and takes a time equal to t_n to complete each task. The total time required for n tasks is nt_n . The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4 Space-time diagram for pipeline.

	1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n . Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speedup that a pipeline can provide is k , where k is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to $t_p = 20$ ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a nonpipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to $8000/2060 = 3.88$. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes $60/20 = 3$.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k -segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always

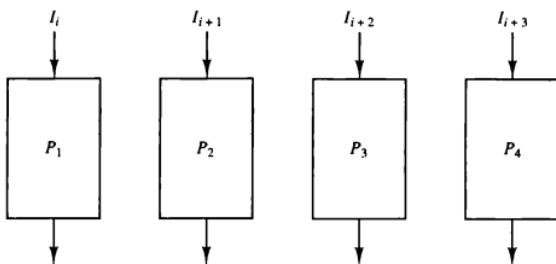


Figure 9-5 Multiple functional units in parallel.

correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into suboperations for execution in the pipeline segments. An *instruction pipeline* operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

9-3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

A and *B* are two fractions that represent the mantissas and *a* and *b* are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled *R* are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $3 - 2 = 1$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of *Y* to the right to obtain

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

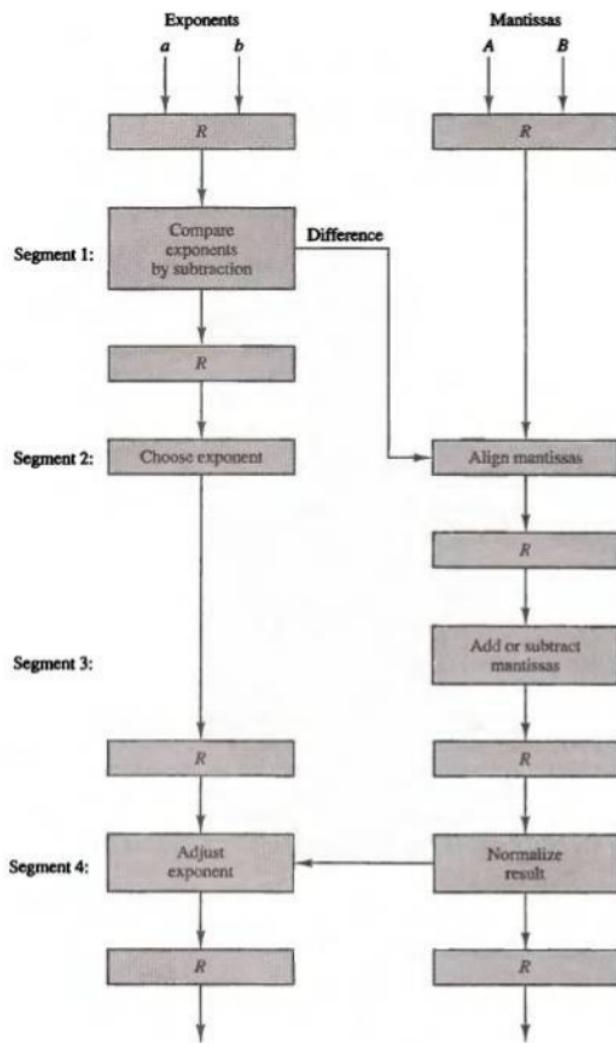


Figure 9-6 Pipeline for floating-point addition and subtraction.

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent nonpipeline floating-point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of $320/110 = 2.9$ over the nonpipelined adder.

9-4 Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

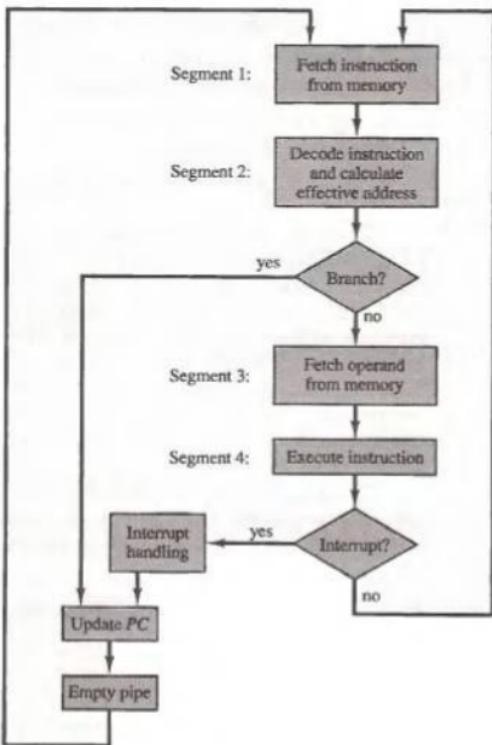


Figure 9-7 Four-segment CPU pipeline.

Figure 9-8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence

Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX								
	2		FI	DA	FO	EX							
(Branch)	3			FI	DA	FO	EX						
	4			FI	-	-	FI	DA	FO	EX			
	5				-	-	-	FI	DA	FO	EX		
	6							FI	DA	FO	EX		
	7								FI	DA	FO	EX	

Figure 9-8 Timing of instruction pipeline.

of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

- 1. *Resource conflicts* caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
- 2. *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- 3. *Branch difficulties* arise from branch and other instructions that change the value of PC.

Data Dependency

A difficulty that may cause a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when

an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

hardware interlocks

The most straightforward method is to insert *hardware interlocks*. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

operand forwarding

Another technique called *operand forwarding* uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as *delayed load*. An example of delayed load is presented in the next section.

delayed load

Handling of Branch Instructions

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.

CHAPTER ELEVEN

Input-Output Organization

IN THIS CHAPTER

- 11-1 Peripheral Devices
- 11-2 Input-Output Interface
- 11-3 Asynchronous Data Transfer
- 11-4 Modes of Transfer
- 11-5 Priority Interrupt
- 11-6 Direct Memory Access
- 11-7 Input-Output Processor
- 11-8 Serial Communication

11-1 Peripheral Devices

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. A computer serves no useful purpose without the ability to receive information from an outside source and to transmit results in a meaningful form.

The most familiar means of entering information into a computer is through a typewriter-like keyboard that allows a person to enter alphanumeric information directly. Every time a key is depressed, the terminal sends a binary coded character to the computer. The fastest possible speed for entering information this way depends on the person's typing speed. On the other hand, the central processing unit is an extremely fast device capable of performing operations at very high speed. When input information is transferred to the processor via a slow keyboard, the processor will be idle most of the time while waiting for the information to arrive. To use a computer efficiently, a

large amount of programs and data must be prepared in advance and transmitted into a storage medium such as magnetic tapes or disks. The information in the disk is then transferred into computer memory at a rapid rate. Results of programs are also transferred into a high-speed storage, such as disks, from which they can be transferred later into a printer to provide a printed output of results.

Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the total computer system. Input or output devices attached to the computer are also called *peripherals*. Among the most common peripherals are keyboards, display units, and printers. Peripherals that provide auxiliary storage for the system are magnetic disks and tapes. Peripherals are electromechanical and electromagnetic devices of some complexity. Only a very brief discussion of their function will be given here without going into detail of their internal construction.

monitor and keyboard Video monitors are the most commonly used peripherals. They consist of a keyboard as the input device and a display unit as the output device. There are different types of video monitors, but the most popular use a cathode ray tube (CRT). The CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically. To produce a pattern on the screen, a grid inside the CRT receives a variable voltage that causes the beam to hit the screen and make it glow at selected spots. Horizontal and vertical signals deflect the beam and make it sweep across the tube, causing the visual pattern to appear on the screen. A characteristic feature of display devices is a cursor that marks the position in the screen where the next character will be inserted. The cursor can be moved to any position in the screen, to a single character, the beginning of a word, or to any line. Edit keys add or delete information based on the cursor position. The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.

printer Printers provide a permanent record on paper of computer output data or text. There are three basic types of character printers: daisywheel, dot matrix, and laser printers. The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon. The dot matrix printer contains a set of dots along the printing mechanism. For example, a 5×7 dot matrix printer that prints 80 characters per line has seven horizontal lines, each consisting of $5 \times 80 = 400$ dots. Each dot can be printed or not, depending on the specific characters that are printed on the line. The laser printer uses a rotating photographic drum

that is used to imprint the character images. The pattern is then transferred onto paper in the same manner as a copying machine.

magnetic tape Magnetic tapes are used mostly for storing files of data: for example, a company's payroll record. Access is sequential and consists of records that can be accessed one after another as the tape moves along a stationary read-write mechanism. It is one of the cheapest and slowest methods for storage and has the advantage that tapes can be removed when not in use. Magnetic disks have high-speed rotational surfaces coated with magnetic material. Access is achieved by moving a read-write mechanism to a track in the magnetized surface. Disks are used mostly for bulk storage of programs and data. Tapes and disks are discussed further in Sec. 12-1 in conjunction with their role as auxiliary memory.

magnetic disk Other input and output devices encountered in computer systems are digital incremental plotters, optical and magnetic character readers, analog-to-digital converters, and various data acquisition equipment. Not all input comes from people, and not all output is intended for people. Computers are used to control various processes in real time, such as machine tooling, assembly line procedures, and chemical and industrial processes. For such applications, a method must be provided for sensing status conditions in the process and sending control signals to the process being controlled.

ASCII The input-output organization of a computer is a function of the size of the computer and the devices connected to it. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral. Certain techniques common to most peripherals are presented in this chapter.

ASCII Alphanumeric Characters

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer. The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in Table 11-1. The seven bits of the code are designated by b_1 through b_7 , with b_7 being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z, the 26 lowercase letters, the 10 numerals 0 through 9, and 32 special printable characters such as %, *, and \$.

The 34 control characters are designated in the ASCII table with abbrevi-

TABLE 11-1 American Standard Code for Information Interchange (ASCII)

$b_4 b_3 b_2 b_1$	$b_7 b_6 b_5$							
000	001	010	011	100	101	110	111	
0000	NUL	DLE	SP	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	,	7	G	W	g	w
1000	BS	CAN	{	8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	-
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

Control characters

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

ated names. They are listed again below the table with their functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include

the familiar typewriter controls, such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions like paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are useful during the transmission of text between remote terminals. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message when transmitted through a communication medium.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font. When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded character.

11-2 Input–Output Interface

Input–output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called *interface* units because they interface between the processor bus and the peripheral device.

In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

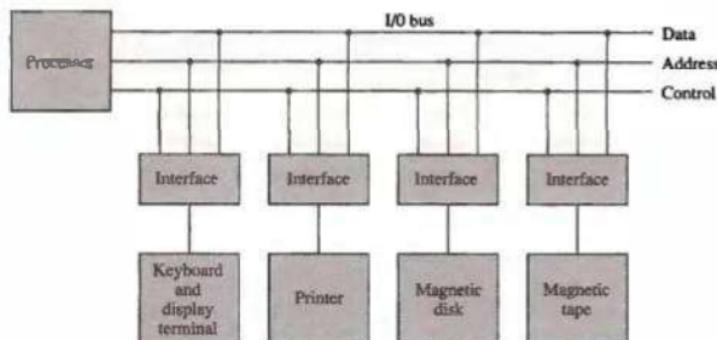
I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 11-1. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface

Figure 11-1 Connection of I/O bus to input-output devices.



I/O command

selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

control command

A *control command* is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

status

A *status command* is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

output data

A *data output command* causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

input data

The *data input command* is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

IOP

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel. In Sec. 11-7 we discuss the function of the IOP in more detail.

Isolated versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The *I/O read* and *I/O write* control lines are enabled during an I/O transfer. The *memory read* and *memory write* control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the *isolated I/O method* for assigning addresses in a common bus.

isolated I/O

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

memory-mapped

The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as *memory-mapped I/O*. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduces the memory address range available.

In a memory-mapped I/O organization there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.

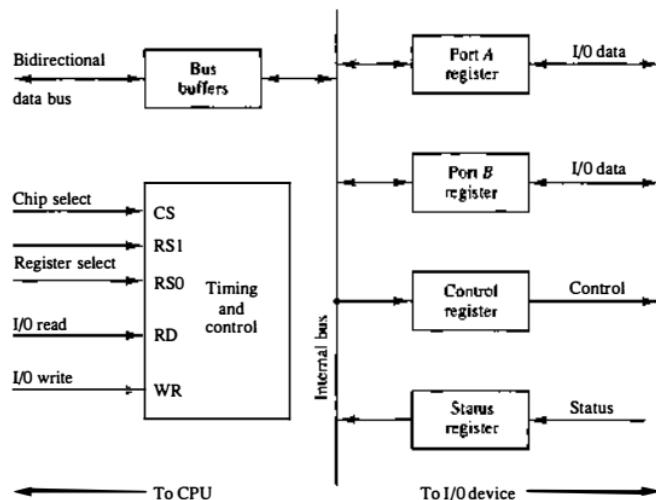
Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input–output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O all instructions that refer to memory are also available for I/O.

Example of I/O Interface

An example of an I/O interface unit is shown in block diagram form in Fig. 11-2. It consists of two data registers called *ports*, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The I/O data to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular interface register with which the CPU communicates.

The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. For example, port A may be defined as an input port and port B as an output port. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in



CS	RS1	RS0	Register selected
0	x	x	None: data bus in high-impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure 11-2 Example of I/O interface unit.

the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port A has received a new data item from the I/O device. Another bit in the status register may indicate that a parity error has occurred during the transfer.

The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the address bus. These two inputs

select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

11-3 Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a *strobe* pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as *handshaking*.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

Strobe Control

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure 11-3(a) shows a source-initiated transfer.

strobe

handshaking

timing diagram

~~the movement of data through the registers. Whenever the F_i bit of the control register is set ($F_i = 1$) and the F_{i+1} bit is reset ($F'_{i+1} = 1$), a clock is generated causing register $R(I + 1)$ to accept the data from register R_I . The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. This ripple-through operation stops when the data reach a register R_I with the next flip-flop F_{i+1} being set to 1, or at the last register R_4 . An overall master clear is used to initialize all control register flip-flops to 0.~~

Data are inserted into the buffer provided that the *input ready* signal is enabled. This occurs when the first control flip-flop F_1 is reset, indicating that register R_1 is empty. Data are loaded from the input lines by enabling the clock in R_1 through the *insert* control line. The same clock sets F_1 , which disables the *input ready* control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that R_2 is empty. The data in R_1 are transferred into R_2 and F_1 is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word. If the FIFO is full, F_1 remains set and the *input ready* line stays in the 0 state. Note that the two control lines *input ready* and *insert* constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The *output ready* control line is enabled when the last control flip-flop F_4 is set, indicating that there are valid data in the output register R_4 . The output data from R_4 are accepted by a destination unit, which then enables the *delete* control signal. This resets F_4 , causing *output ready* to disable, indicating that the data on the output are no longer valid. Only after the *delete* signal goes back to 0 can the data from R_3 move into R_4 . If the FIFO is empty, there will be no data in R_3 and F_4 will remain in the reset state. Note that the two control lines *output ready* and *delete* constitute a source-initiated pair of handshake lines.

11-4 Modes of Transfer

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

programmed I/O

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory. DMA transfer is discussed in more detail in Sec. 11-6.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in Sec. 11-7.

Example of Programmed I/O

In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

interrupt**DMA****IOP**

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 11-10. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig. 11-5.

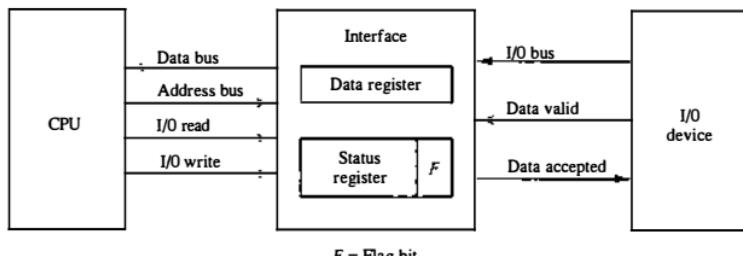
A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. 11-11. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. A program that

Figure 11-10 Data transfer from I/O device to CPU.



F = Flag bit

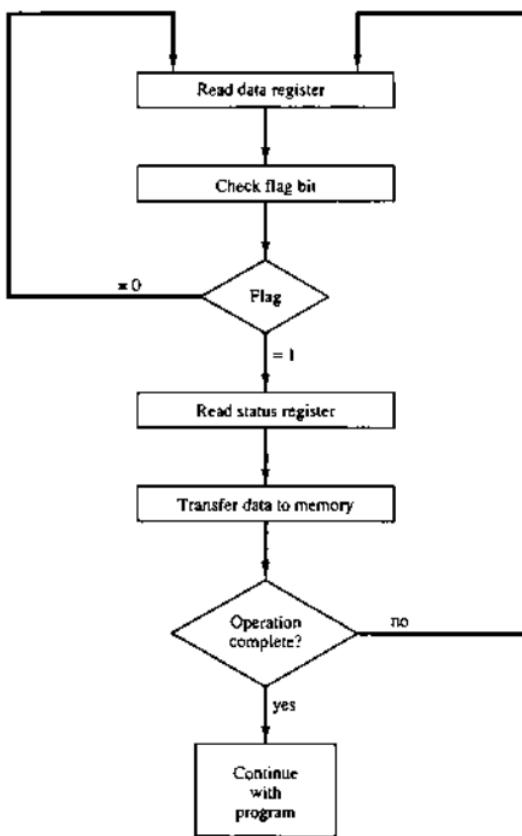


Figure 11-11 Flowchart for CPU program to input data.

stores input characters in a memory buffer using the instructions defined in Chap. 6 is listed in Table 6-21.

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 μ s. Assume that the input device transfers its

data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 μ s. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

Interrupt-Initiated I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called *vectored interrupt* and the other, *nonvectored interrupt*. In a nonvectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the *interrupt vector*. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored. A system with vectored interrupt is demonstrated in Sec. 11-5.

Software Considerations

The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers. In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.

vectored interrupt

I/O routines

Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

11-5 Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer. As discussed in Sec. 8-7, some processors also push the current PSW (program status word) onto the stack and load a new PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of I/O interrupts.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and

priority interrupt

polling

source.

instructions that service the interrupt

11-6 Direct Memory Access (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly

would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

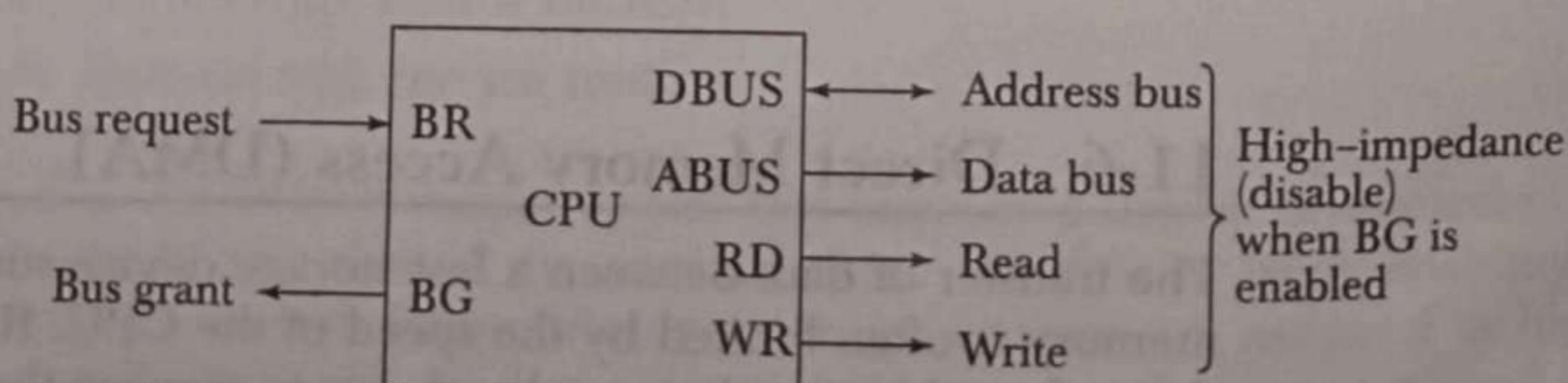
The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 11-16 shows two control signals in the CPU that facilitate the DMA transfer. The *bus request (BR)* input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance (see Sec. 4-3). The CPU activates the *bus grant (BG)* output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA *burst transfer*, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called *cycle stealing* allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines

Figure 11-16 CPU bus signals for DMA transfer.

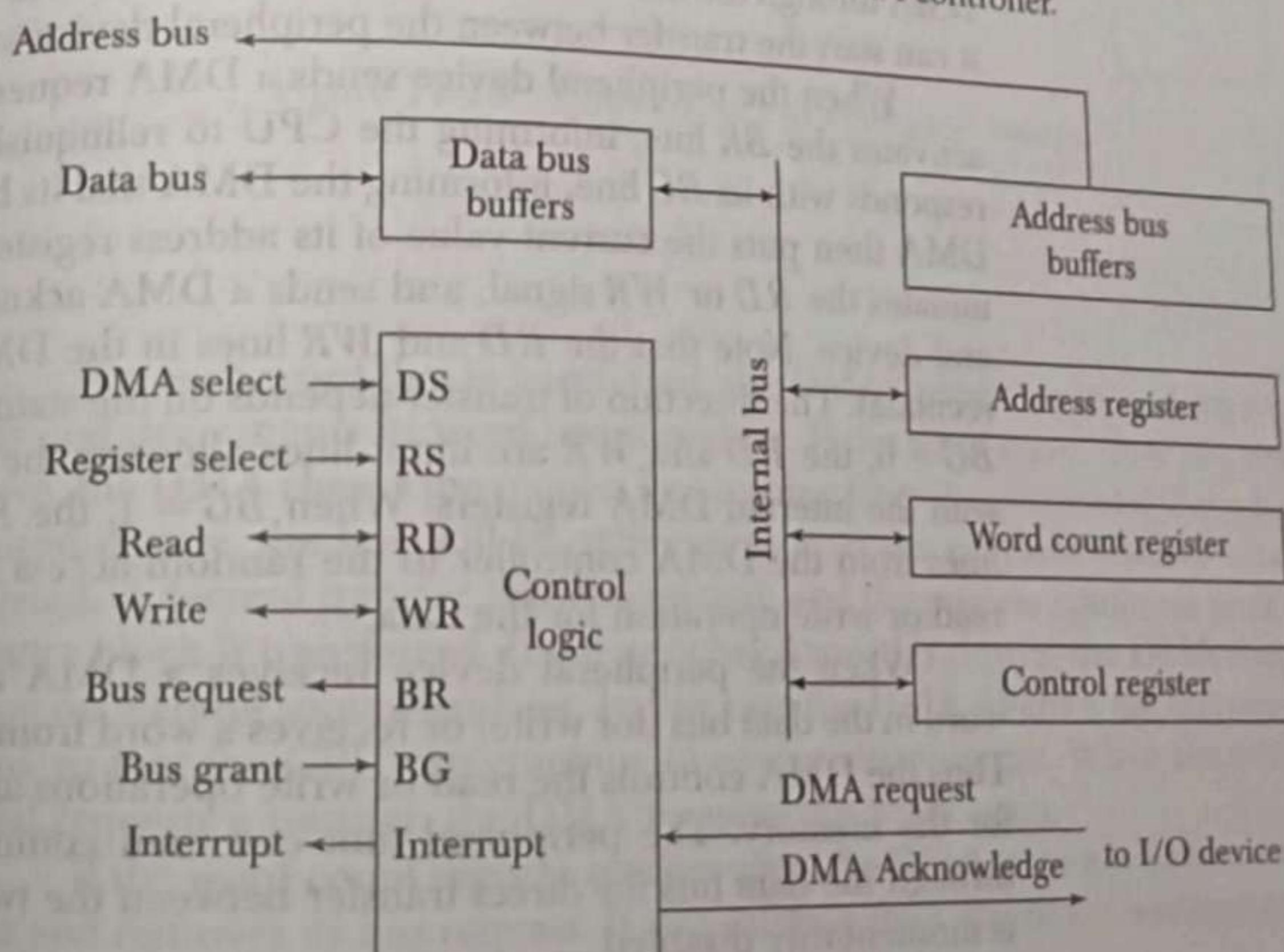


are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 11-17 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the *DS* (DMA select) and *RS* (register select) inputs. The *RD* (read) and *WR* (write) inputs are bidirectional. When the *BG* (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When *BG* = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the *RD* or *WR* control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

Figure 11-17 Block diagram of DMA controller.

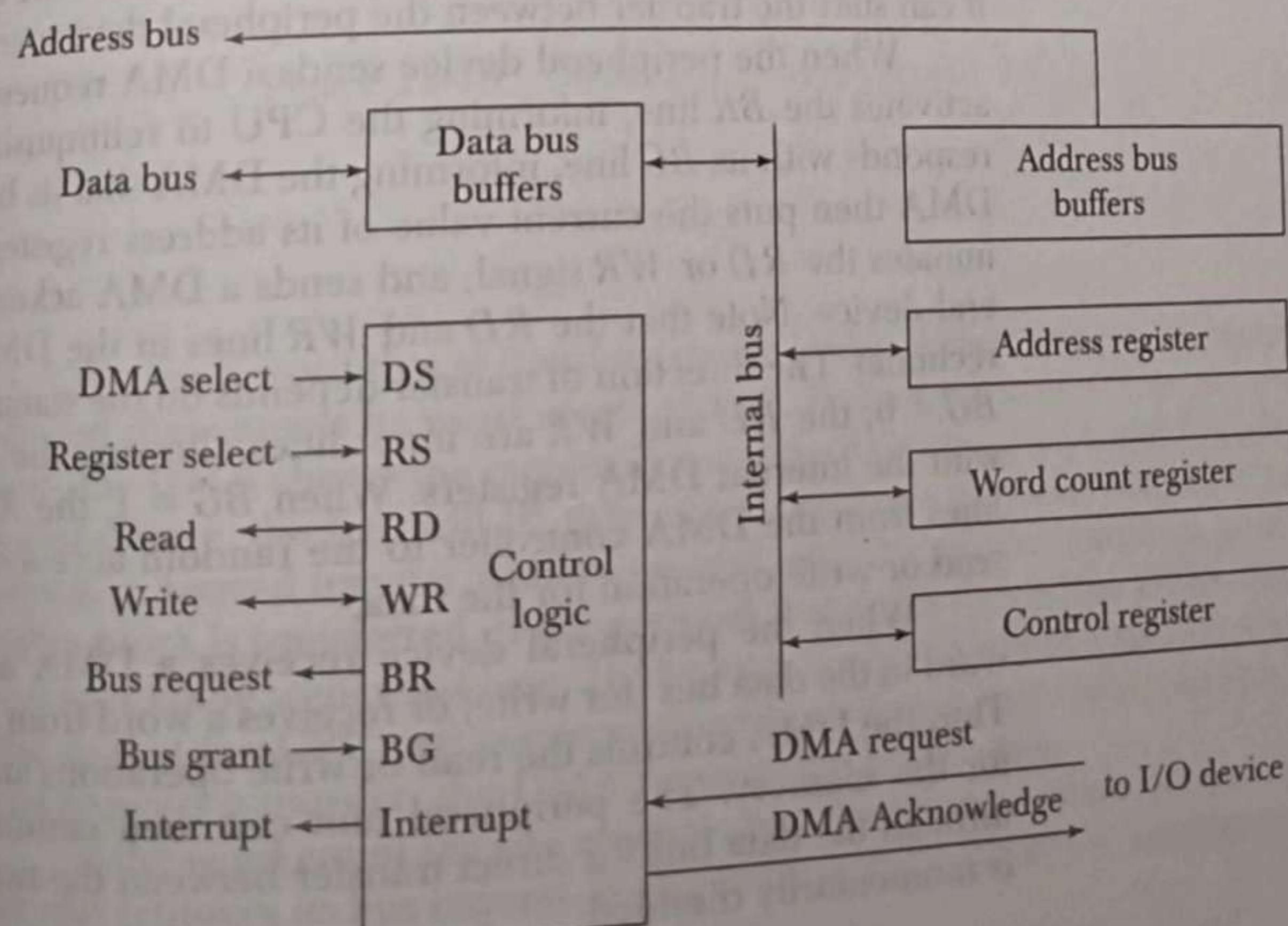


are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 11-17 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the *DS* (DMA select) and *RS* (register select) inputs. The *RD* (read) and *WR* (write) inputs are bidirectional. When the *BG* (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When *BG* = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the *RD* or *WR* control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

Figure 11-17 Block diagram of DMA controller.



The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Fig. 11-18. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the *DS* and *RS* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device. Note that the *RD* and *WR* lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the *BG* line. When *BG* = 0, the *RD* and *WR* are input lines allowing the CPU to communicate with the internal DMA registers. When *BG* = 1, the *RD* and *WR* are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

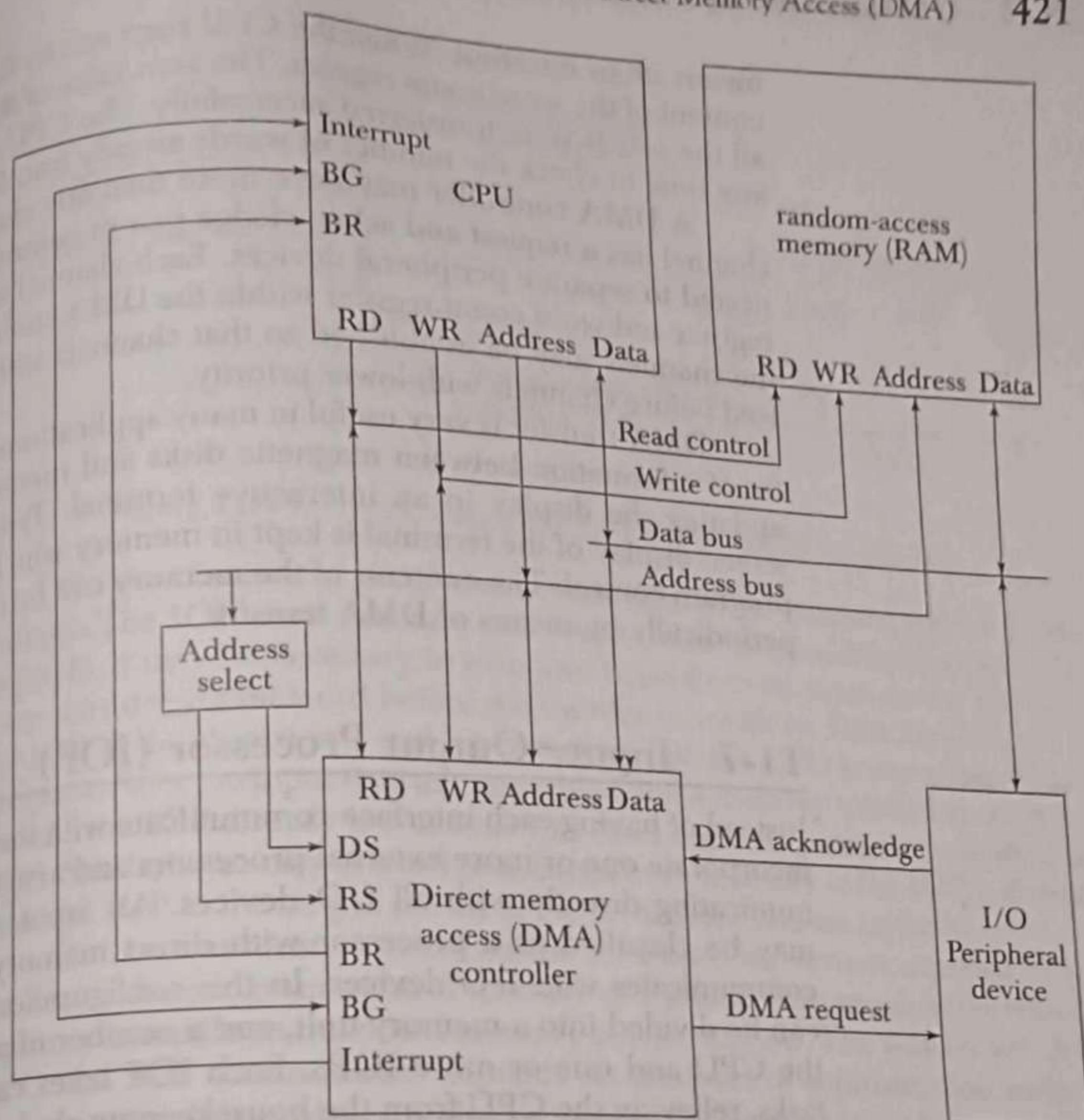


Figure 11-18 DMA transfer in a computer system.

For each word that is transferred, the DMA increments its address register and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by

CHAPTER ELEVEN Input-Output Organization

means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than one channel. In this case, each channel has a request and acknowledge pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

CHAPTER TWELVE

Memory Organization

IN THIS CHAPTER

- 12.1 Memory Hierarchy
- 12.2 Main Memory
- 12.3 Auxiliary Memory
- 12.4 Associative Memory
- 12.5 Cache Memory
- 12.6 Virtual Memory
- 12.7 Memory Management Hardware

12-1 Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the *main memory*. Devices that provide backup storage are called *auxiliary memory*. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All

auxiliary memory

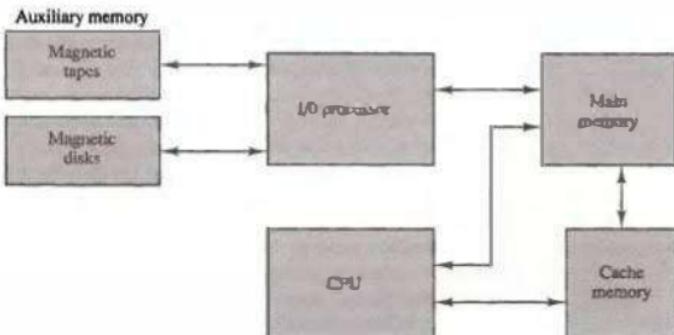
other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. Figure 12-1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

cache memory

A special very-high-speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.

Figure 12-1 Memory hierarchy in a computer system.



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. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100 ns, while main memory access time may be 700 ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called *multiprogramming*, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

With multiprogramming the need arises for running partial programs, for varying the amount of main memory in use by a given program, and for moving programs around the memory hierarchy. Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the

multiprogramming

program is to be executed, it is transferred to main memory to be executed by the CPU. Thus one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the *memory management system*. The hardware for a memory management system is presented in Sec. 12-7.

12-2 Main Memory

random-access memory (RAM) The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, *static* and *dynamic*. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles.

read-only memory (ROM) Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

bootstrap loader Among other things, the ROM portion of main memory is needed for storing an initial program called a *bootstrap loader*. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the

computer startup

first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024×8 memory constructed with 128×8 RAM chips and 512×8 ROM chips.

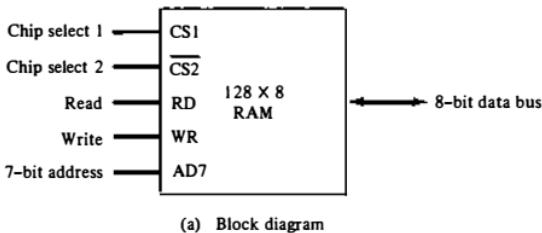
RAM and ROM Chips

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation, or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is shown in Fig. 12-2. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit

bidirectional bus

Figure 12-2 Typical RAM chip.



(a) Block diagram

CSI	CS2	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedance

(b) Function table

address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations of read or write.

The function table listed in Fig. 12-2(b) specifies the operation of the RAM chip. The unit is in operation only when $CS_1 = 1$ and $\overline{CS}_2 = 0$. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When $CS_1 = 1$ and $\overline{CS}_2 = 0$, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 12-3. For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be $CS_1 = 1$ and $\overline{CS}_2 = 0$ for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a *memory address map*, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

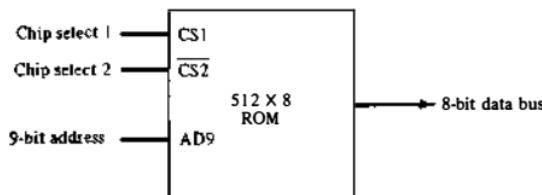


Figure 12-3 Typical ROM chip.

to be used are specified in Figs. 12-2 and 12-3. The memory address map for this configuration is shown in Table 12-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. 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: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained from the information under the address bus assignment. The address bus lines are

TABLE 12-1 Memory Address Map for Microprocomputer

subdivided into groups of four bits each so that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

Memory Connection to CPU

RAM and ROM chips are connected to a CPU through 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 its chip select inputs. The connection of memory chips to the CPU is shown in Fig. 12-4. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 12-1. 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 2×4 decoder whose outputs go to the CS1 inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first 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 RD control line for the ROM chip to be enabled only during a 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, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

12-3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have a knowledge of magnetics, electronics, and electromechanical systems. Al-

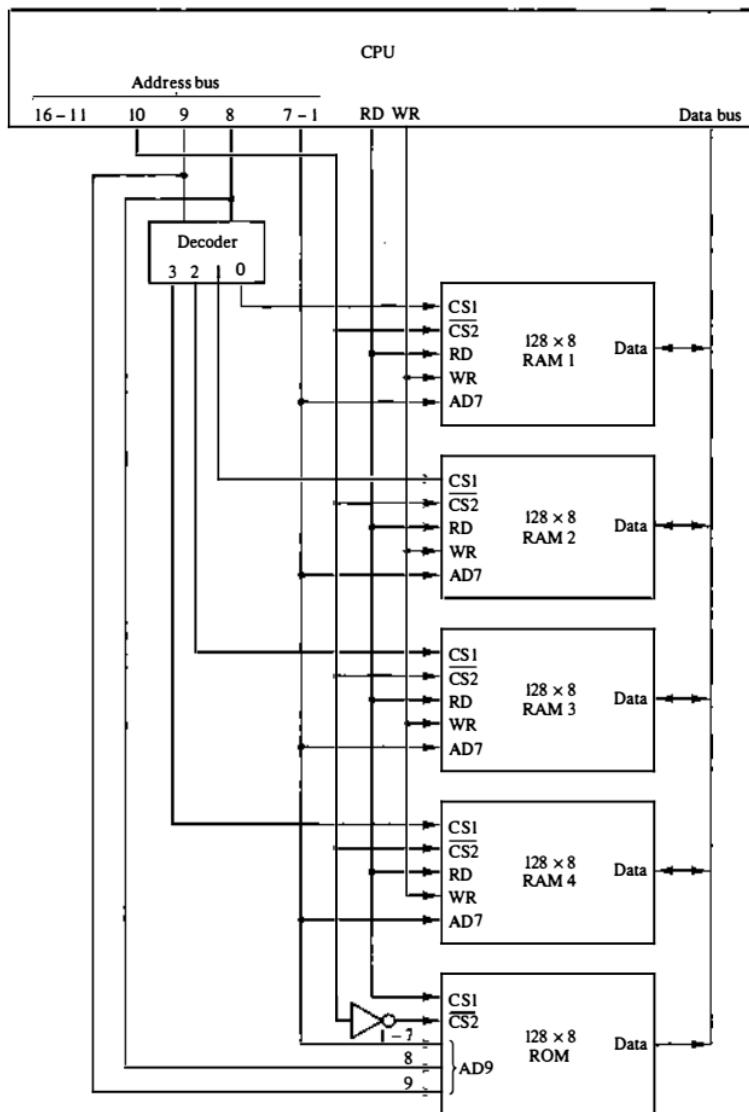


Figure 12-4 Memory connection to the CPU.

though the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a *seek* time required to position the read-write head to a location and a *transfer* time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. 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 drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. The recording surface rotates at uniform speed and is not started or stopped during access operations. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a *write head*. Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a *read head*. The amount of surface available for recording in a disk is greater than in a drum of equal physical size. Therefore, more information can be stored on a disk than on a drum of comparable size. For this reason, disks have replaced drums in more recent computers.

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 stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started for access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector. The subdivision of one disk surface into tracks and sectors is shown in Fig. 12-5.

Some units use a single read/write head for each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

Permanent timing tracks are used in disks to synchronize the bits and

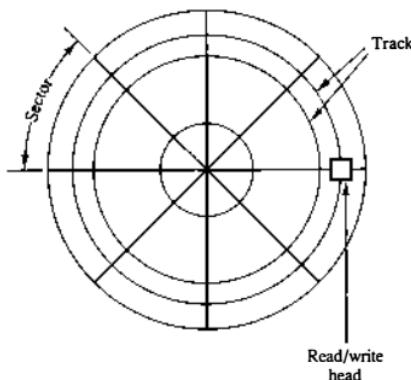


Figure 12-5 Magnetic disk.

recognize the sectors. 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 in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called *hard disks*. A disk drive with removable disks is called a *floppy disk*. The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

Magnetic Tape

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is 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 form a character together 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. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit 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 of fixed or variable length.

12-4 Associative Memory

Many data-processing 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 established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an *associative memory* or *content addressable memory* (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. 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 word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.

Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on

**content addressable
memory**

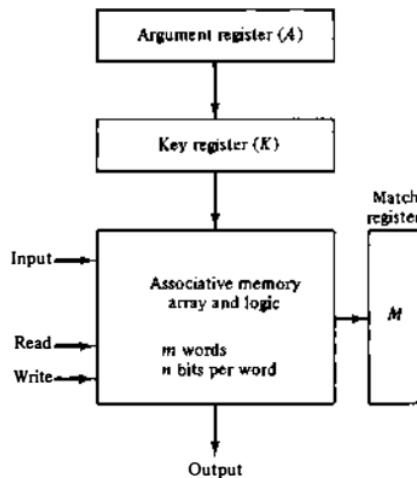
an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications 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. It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each have n bits, one 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 the fact that their 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 field or key in the argument word. The entire argument is compared with each 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

Figure 12-6 Block diagram of associative memory.



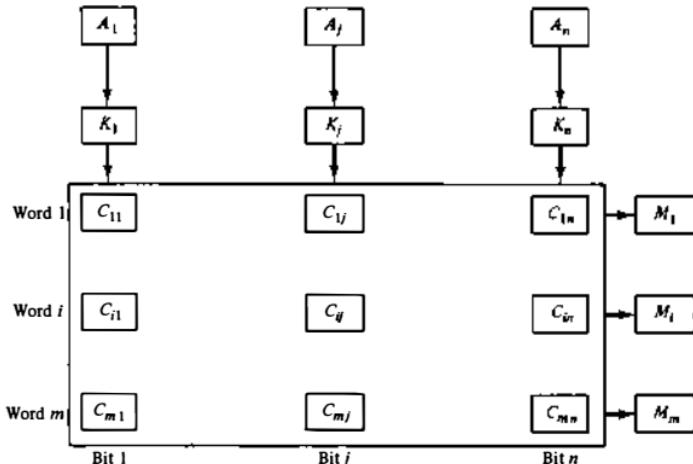
specifies how the reference to memory is made. To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in Fig. 12-7. 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 column j of the array provided that $K_j = 1$. This is done for all columns $j = 1, 2, \dots, n$. If a match occurs between all the unmasked bits of the argument and the bits in word i , the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure 12-7 Associative memory of m word, n cells per word.



The internal organization of a typical 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 bit is transferred into the storage cell during a write operation. The bit stored is read out during a read 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 .

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 $A_j = F_{ij}$ for $j = 1, 2, \dots, n$. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$$x_j = A_j F_{ij} + A'_j F'_{ij}$$

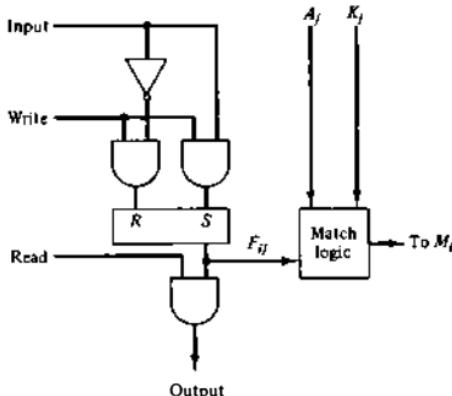
where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \cdots x_n$$

and constitutes the AND operation of all pairs of matched bits in a word.

Figure 12-8 One cell of associative memory.



We now include the key bit K_j in the comparison logic. The requirement is that if $K_j = 0$, the corresponding bits of A_j and F_{ij} need no comparison. Only when $K_j = 1$ must they be compared. This requirement is achieved by ORing each term with 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 $K_j = 1$, we have $K'_j = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K'_j = 1$ and $x_j + 1 = 1$. A term $(x_j + K'_j)$ will be in the 1 state if its pair of bits is not compared. This is necessary because each term is 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 $K_j = 1$.

The match logic for 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 $K_j = 0$. If $K_j = 1$, the term will be either 0 or 1 depending on the value of x_j . A match will occur and M_i will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of x_j , 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 \prod is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word $i = 1, 2, 3, \dots, m$.

The circuit for matching one word is shown in Fig. 12-9. Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j 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 a common AND gate to generate the match signal for M_i . M_i will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

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 M_i bit is a 1.

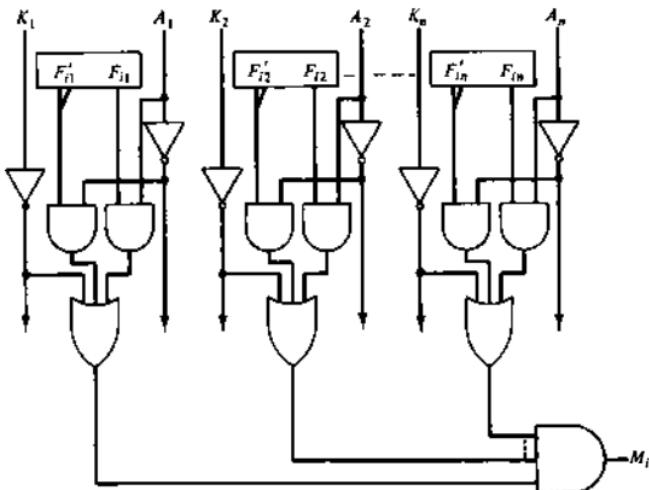


Figure 12-9 Match logic for one word of associative memory.

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having a zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

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 $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a *tag register*, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location. Moreover, the words that have a tag bit of 0 must be masked (together with the K_j bits) with the argument word so that only active words are compared.

12-5 Cache Memory

Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of *locality of reference*. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions are fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively infrequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a *cache memory*. It is placed between the CPU and main memory as illustrated in Fig. 12-1. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the aver-

locality of reference

age 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 requests will be found in the fast cache memory because of the locality of reference property of 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 fast 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. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. 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 frequently measured in terms of a quantity called *hit ratio*. When the CPU refers to memory and finds the word in cache, it is said to produce a *hit*. If the word is not found in cache, it is in main memory and it counts as a *miss*. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a *mapping* process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

- 1. Associative mapping
- 2. Direct mapping
- 3. Set-associative mapping

To help in the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in Fig. 12-10. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is

hit ratio

mapping

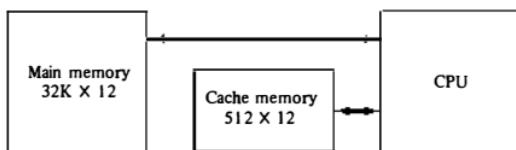


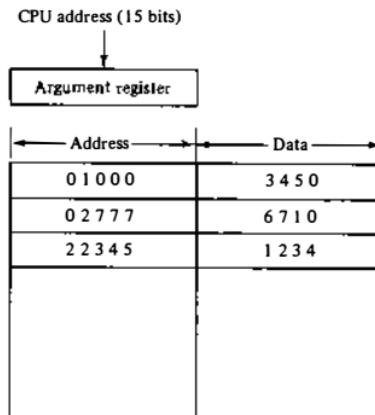
Figure 12-10 Example of cache memory.

a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

Associative Mapping

The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in Fig. 12-11. The associative memory stores both the address and content (data) of 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

Figure 12-11 Associative mapping cache (all numbers in octal).



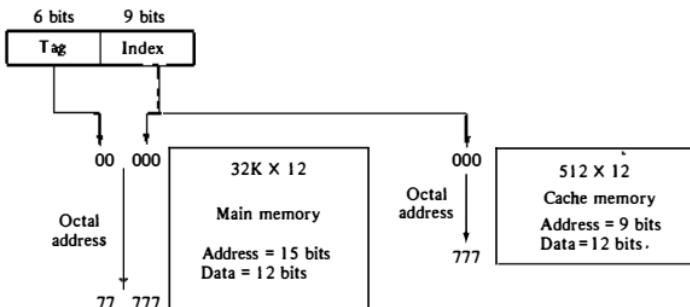
address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address–data pair is then transferred to the associative cache memory. If the cache is full, an address–data pair must be displaced to make room for a pair that 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 cells 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 the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. 12-12. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the *index* field and the remaining six bits form the *tag* field. The figure shows that main memory needs an address that includes both the tag and the index bits. The number of bits in the index field 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 the n -bit address to access the main memory and the k -bit index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. 12-13(b). Each word in 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. When the CPU generates a memory request, the index field is used for the address to access the cache. The

Figure 12-12 Addressing relationships between main and cache memories.



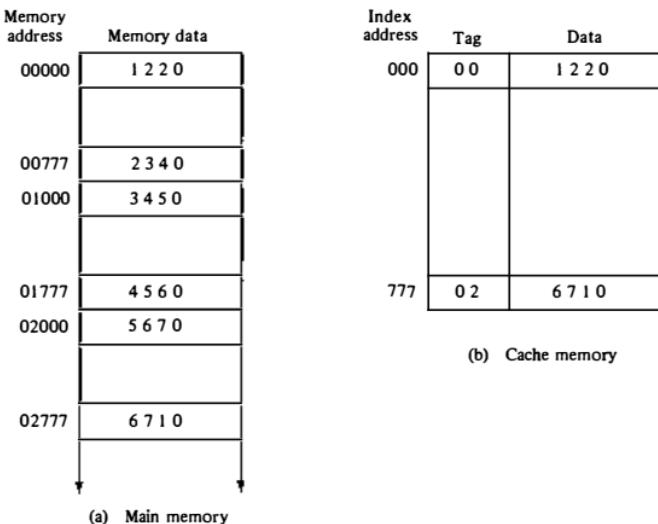


Figure 12-13 Direct mapping cache organization.

tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example.)

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. 12-13. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that 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 direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig. 12-14.

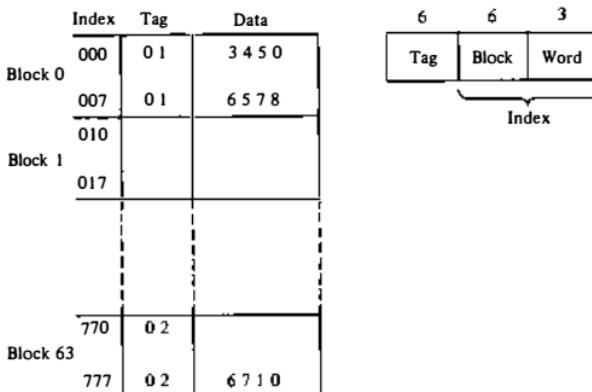


Figure 12-14 Direct mapping cache with block size of 8 words.

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 $64 \times 8 = 512$. The block number is specified with 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 words 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 most likely improve with a larger block size because of the sequential nature of computer programs.

Set-Associative Mapping

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that 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. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the 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 512×36 . 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.

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

Figure 12-15 Two-way set-associative mapping cache.

The octal numbers listed in Fig. 12-15 are with reference to the main memory contents illustrated in Fig. 12-13(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. 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 associative memory search: thus 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 a 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, first-in, first-out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

replacement algorithms

Writing into Cache

An important aspect of cache organization is concerned with memory write requests. 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 that the system can proceed.

write-through

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

write-back

The second procedure is called the *write-back* method. In this method only the 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 it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

valid bit

Cache Initialization

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some nonvalid data. It is customary to include with each word in cache a *valid bit* to indicate whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

12-6 Virtual Memory

In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU. *Virtual memory* is a concept used in some large computer systems that permit the user to construct programs as though a large

memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give 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 addresses 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

An address used by a programmer will be called a *virtual address*, and the set of such addresses the *address space*. An address in main memory is called a *location* or *physical address*. The set of such locations is called the *memory space*. Thus 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 computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main-memory capacity of 32K words ($K = 1024$). 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 $2^{20} = 1024K$ words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M , we then have for this example $N = 1024K$ and $M = 32K$.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in Fig. 12-16. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember that for

address space
memory space

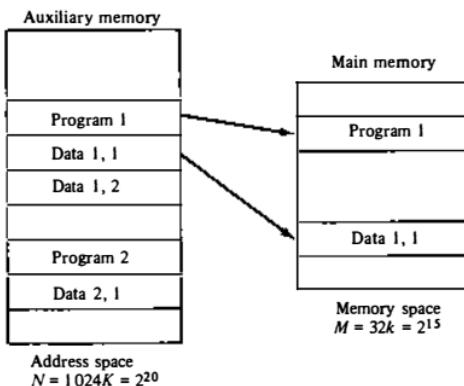
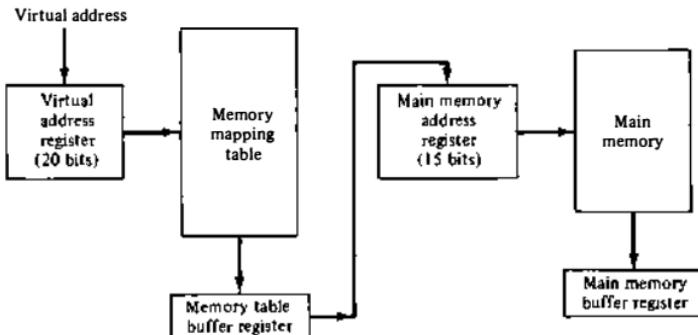


Figure 12-16 Relation between address and memory space in a virtual memory system.

efficient transfers, auxiliary storage moves an entire record to the main memory.) A table is then needed, as shown in Fig. 12-17, to map a virtual address of 20 bits to a physical address of 15 bits. 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 as shown in Fig. 12-17 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table

Figure 12-17 Memory table for mapping a virtual address.



takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

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 down 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 considered to be 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 to 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 page consists of $2^{10} = 1024$ words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within 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.

The organization of the memory mapping table in a paged system is shown in Fig. 12-19. The memory-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 that 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 in the presence 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

pages and blocks

page frame

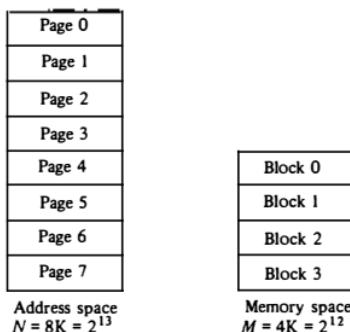
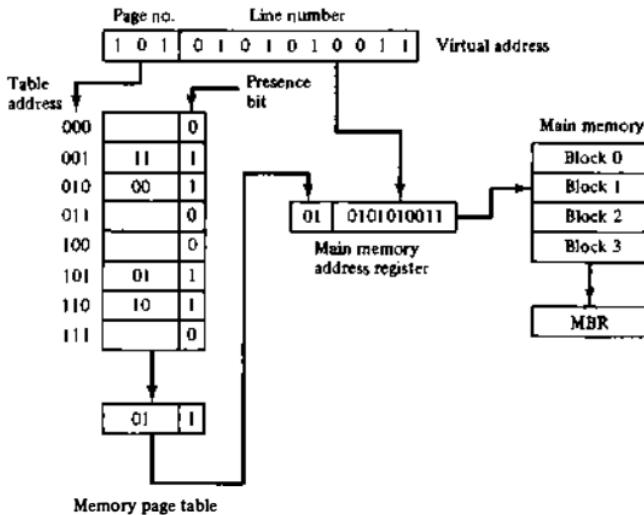


Figure 12-18 Address space and memory space split into groups of 1K words.

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

Figure 12-19 Memory table in a paged system.



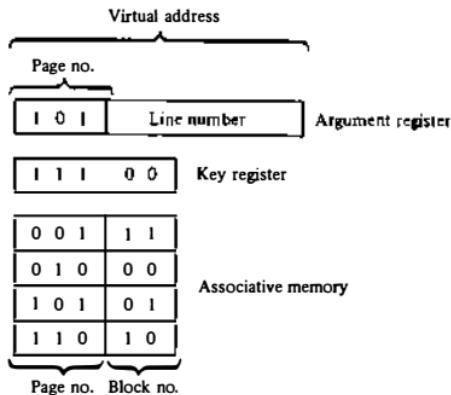
transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is 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 observe 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. In general, a system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

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 an associative memory with each word in memory containing a page number together with its corresponding

Figure 12-20 An associative memory page table.



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.

Consider again the case of eight pages and four blocks as in the example of Fig. 12-19. We replace the random access memory-page table 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 field for storing the page number. The last two bits constitute a field 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 field 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 a 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 ought 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, and (3) where the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual 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 is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called *page fault*. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred 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 replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the *first-in*,

page fault

FIFO

first-out (FIFO) and the *least recently used* (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantage that under certain circumstances pages are removed and loaded from memory too frequently.

LRU

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called *aging registers*, as their count indicates their age, that is, how long ago their associated pages have been referenced.

12-7 Memory Management Hardware

In a multiprogramming environment where many programs reside in memory it becomes necessary to move programs and data around the memory, to vary the amount of memory in use by a given program, and to prevent a program from changing other programs. The demands on computer memory brought about by multiprogramming have created the need for a memory management system. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers. Here we are concerned with the hardware unit associated with the memory management system.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses
2. A provision for sharing common programs stored in memory by different users
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions

The dynamic storage relocation hardware is a mapping process similar to the paging system described in Sec. 12-6. The fixed page size used in the virtual