1.2 DATA REPRESENTATION

* **Data types.**

- Data are numbers and other binary coded information that are operated on to achieve required computational results.
- Data types found in the registers of digital computers may be classified as
  1) numbers used in arithmetic computations.
  2) letters of alphabet used in data processing
  3) other discrete symbols used for specific purpose
- All types of data are represented in binary-coded form because registers are made of flip-flops and flip-flops are 2-stage devices that can store only 0's and 1's.

* **Number Systems**

- A number system of base or radix "r" is a system that uses distinct symbols for "r" digits. And each number is represented by a string of digit symbols.

Eg:- Decimal number system → r = 10 and
symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary number system → r = 2 and
symbols are 0, 1

octal number system → r = 8 and
symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8

Hexadecimal number system → r = 16 and
symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- To determine the quantity that the number represents,
  it is necessary to multiply each digit by an integer power of "r" and then form the sum of all weighted digits.

Eg:- $(724.5)_{10}$

$= 7\times10^2 + 2\times10^1 + 4\times10^0 + 5\times10^{-1}$

$= 700 + 20 + 4 + 0.5 = 724.5$

$(101101)_2$

$= 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 32 + 0 + 8 + 4 + 0 + 1 = 45$.

→ Conversion to decimal system

A number of radix "r" can be converted to the familiar decimal system by forming the sum of weighted digits.

Eg:- $(736.4)_8$

$= 7\times8^2 + 3\times8^1 + 6\times8^0 + 4\times8^{-1}$

$= 7\times64 + 3\times8 + 6\times1 + 4\times0.125$

$= (478.5)_{10}$

$(F3)_{16}$

$= F\times16^1 + 3\times16^0 = 15\times16 + 3\times1$

$= (243)_{10}$

Note: In hexadecimal, A, B, C, D, E, F are equal to 10, 11, 12, 13, 14, 15 respectively.

→ Conversion from decimal to radix "r"

First seperate the 2 numbers into integer and fraction parts.

1) The conversion of decimal integers into radix "r" representation is done by successive divisions by "r" and accumulating remainders.
2) The conversion of decimal fractions into a base "r" representation is accomplished by successive ~~mult~~ multiplication by "r" and accumulating integers.

Eg:- $(41.6875)_{10} = (?)_2$

Integer part

41.

		Remainder
2	41	
2	20	1
2	10	0
2	5	0
2	2	1
2	1	0
	0	1

Fraction part

0.6875

	Integer
$0.6875 \times 2 = 1.3750$	1
$0.375 \times 2 = 0.75$	0
$0.75 \times 2 = 1.5$	1
$0.5 \times 2 = 1.0$	1

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

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

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

→ Advantage with octal & hexadecimal numbers

The conversion from and to binary, octal and hexadecimal numbers play an important role in digital computers.

Radix of octal number system = $8 = 2^3$

⇒ each digit corresponds to 3 digits in binary.

Radix of hexadecimal number system = $16 = 2^4$

⇒ each digit corresponds to 4 digits in binary.

Binary to octal conversion is easily accomplished by partitioning the binary number into groups of 3 bits each. Then assign corresponding octal digit.

Eg:- 1011 111 101 100 011 — data in a 16-bit register.

1 3 7 5 4 3

Binary to hexadecimal conversion is easily accomplished by partitioning the binary number into groups of 4 bits each. Then assign corresponding ~~octal~~ digit. In hexadecimal representation

Eg:- 1010 1111 01100011 — data in a 16-bit register.

A F 6 3

→ comparing the binary-coded form of octal & hexadecimal numbers

Hexadecimal — $(63)_{16} = (01100011)_2 = (99)_{10}$

octal — $(143)_8 = (001100011)_2 = (99)_{10}$

Removing leading 0's, 99 in binary representation, binary-coded octal and binary coded hexadecimal are same.

⇒ Data stored in a register though consists of a string's of 0's and 1's can be interpreted as holding an octal number in binary-coded form or as holding a hexadecimal number in a binary-coded form.

⇒ Specifying the contents of a register in hexadecimal or octal equivalent is more convenient than binary because they require less no. of digits i.e no. of digits is reduced to 1/3rd in octal representation and 1/4th in hexadecimal representation.

→ Decimal representation

The number number system is the most natural system for a computer, but people are accustomed to the decimal number system. one way to solve this problem is:

Convert all the i/p decimal nos. into binary numbers. Let the computer perform all arithmetic operations on binary numbers and then convert binary results back to decimal for the human user to understand. However, it is also possible to perform arithmetic operations directly on decimal nos. if they are stored in binary-coded form in registers.

A binary code is a group of "n" bits that assume upto 2^n distinct combinations of 1's and 0's with each combination representing 1 element of the set that is being coded.

Eg:- Set of 4 elements can be represented with 2 bits : 00, 01, 10, 11

A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple of power of 2.

Eg:- 1010 1111 0110 0011 — Data in a 16 bit register
A F 6 3

- Comparing the binary-coded form of octal & hexadecimal numbers

Hexadecimal — $(63)_{16} = (01100011)_2 = (99)_{10}$

octal — $(143)_8 = (001100011)_2 = (99)_{10}$

Removing leading zeros, 99 in binary representation, binary-coded octal & binary coded hexadecimal are same

⇒ Data stored in a register though exists as a string 0's and 1's, can be interpreted as holding an octal number in binary-coded form or as holding a hexadecimal number in a binary coded form.

⇒ Specifying the contents of a register in hexadecimal or octal equivalent is more convenient than binary because they require less no. of digits i.e., no. of digits is reduced to 1/3rd in octal representation and 1/4th in hexadecimal representation.

- Decimal Representation

- The binary number system is the most natural system for a computer, but people are accustomed to the decimal number system.
- one way to solve this problem is

Convert all the i/p decimal nos. into binary numbers. Let the computer perform all arithmetic operations on binary numbers and then convert binary results back to decimal for the human user to understand.

However, it is also possible to perform arithmetic operations directly on decimal nos. if they are stored in binary-coded form in the registers.

- A binary code is a group of "n" bits that assume upto 2^n distinct combinations of 1's and 0's with each combination representing 1 element of the set that is being coded.

Eg:- set of 4 elements can be represented with 2 bits : 00, 01, 10, 11

A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple of power of 2.

Eg:- 10 decimal digits : Require 4 bits but we use only 10 out of 16 combinations of 1's and 0's.

A bit assignment most commonly used for decimal digits is a straight binary assignment as shown below:

0 - 0000	4 - 0100	8 - 1000
1 - 0001	5 - 0101	9 - 1001
2 - 0010	6 - 0110	1010 to 1111 are
3 - 0011	7 - 0111	unassigned

This is called binary coded decimal. (BCD).

Eg:- $(10)_{10} = (0001\ 0000)_{BCD} = (A)_{16}$

$(15)_{10} = (0001\ 0101)_{BCD} = (F)_{16}$

$(20)_{10} = (0010\ 0000)_{BCD} = (14)_{16}$

Alphanumeric Representation

Many applications of digital computers require handling of data that consists not only numbers but also letters of alphabet and some special characters.

Alphanumeric character set

= 10 decimal digits + 26 letters of alphabet + special characters like $, +, =, etc.

= 32 to 64 elements (if only lowercase are included)
⇒ 6 bits are used for representation

= 64 to 128 elements (if uppercase are also included)
⇒ 7 bits are used for representation

The standard alphanumeric binary code is ASCII (American Standard Code for Information Interchange), which is 7 bit to code 128 elements.

character	ASCII code	character	ASCII code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
⋮		2	011 0010
O	100 1111	⋮	⋮
P	101 0000	9	011 1001
Q	101 0001	space	010 0000
⋮		.	010 1110
Z	101 1010	(	010 1000
—	010 1101	+	010 1011
/	010 1111	$	010 0100
)	010 1100	*	010 1010
=	010 1101	)	010 1001

(Pb) Convert the following into decimal:

a) 101110

b) 1110101

c) 110110100

Sol: a) 101110

$5\ 4\ 3\ 2\ 1\ 0$

$1\ 0\ 1\ 1\ 1\ 0$

$= 2^5 + 2^3 + 2^2 + 2^1$

$= 32 + 8 + 4 + 2$

$= 46$

$\Rightarrow (101110)_2 = (46)_{10}$

b) 1110101

$6\ 5\ 4\ 3\ 2\ 1\ 0$

$1\ 1\ 1\ 0\ 1\ 0\ 1$

$= 2^6 + 2^5 + 2^4 + 2^2 + 2^0$

$= 64 + 32 + 16 + 4 + 1$

$= 117$

$\Rightarrow (1110101)_2 = (117)_{10}$

c) 110110100

$8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0$

$1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0$

$= 2^8 + 2^7 + 2^5 + 2^4 + 2^2 = 256 + 128 + 32 + 16 + 4$

$= 436$

$\Rightarrow (110110100)_2 = (436)_{10}$

(Pb) Convert the following numbers with the indicated bases to decimal :

a) $(12121)_3$

b) $(4310)_5$

c) $(50)_7$

d) $(198)_{12}$

Sol: a) $(12121)_3$

$$\overset{4}{1}\ \overset{3}{2}\ \overset{2}{1}\ \overset{1}{2}\ \overset{0}{1}$$

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

$$= 81 + 54 + 9 + 6 + 1$$

$$= (151)_{10}$$

$$\Rightarrow (12121)_3 = (151)_{10}$$

b) $(4310)_5$

$$\overset{3}{4}\ \overset{2}{3}\ \overset{1}{1}\ \overset{0}{0}$$

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

$$= 500 + 75 + 5$$

$$= 580$$

$$\Rightarrow (4310)_5 = (580)_{10}$$

c) $(50)_7$

$$\overset{1}{5}\ \overset{0}{0}$$

$$= 5\times 7^1 + 0\times 7^0$$

$$= 35$$

$$\Rightarrow (50)_7 = (35)_{10}$$

d) $(198)_{12}$

$$\overset{2}{1}\ \overset{1}{9}\ \overset{0}{8}$$

$$= 1\times 12^2 + 9\times 12^1 + 8\times 12^0$$

$$= 144 + 108 + 8$$

$$= 260$$

$$\Rightarrow (198)_{12} = (260)_{10}$$

(Pb) Convert the following decimal numbers to binary.

a) 1231

b) 673

c) 1998

d) 175

Sol: a) 1231

2 | 1231	
2 | 615	1
2 | 307	1
2 | 153	1
2 | 76	1
2 | 38	0
2 | 19	0
2 | 9	1
2 | 4	1
2 | 2	0
2 | 1	0
0	1

$\Rightarrow (1231)_{10} = (10011001111)_2$

b) 673

2 | 673	
2 | 336	1
2 | 168	1
2 | 84	0
2 | 42	0
2 | 21	0
2 | 10	1
2 | 5	0
2 | 2	1
2 | 1	0
0	1

$\Rightarrow (673)_{10} = (1010100011)_2$

c) 1998

2 | 1998	
2 | 999	0
2 | 499	1
2 | 249	1
2 | 124	1
2 | 62	0
2 | 31	0
2 | 15	1
2 | 7	1
2 | 3	1
2 | 1	1
0	1

$\Rightarrow (1998)_{10} = (11111001110)_2$

d) 175

2 | 175	
2 | 87	1
2 | 43	1
2 | 21	1
2 | 10	1
2 | 5	0
2 | 2	1
2 | 1	0
0	1

$\Rightarrow (175)_{10} = (10101111)_2$

(Pb) Convert the following decimal numbers to the bases indicated

a) 7562 to octal

b) 1938 to hexadecimal

Sol: a) 7562 to octal

8 | 7562		
8 | 945	2	
8 | 118	1	
8 | 14	6	
8 | 1	6	↑
0	1	

$\Rightarrow (7562)_{10} = (16612)_8$

b) 1938 to hexadecimal

16 | 1938		
16 | 121	2	
16 | 7	9	↑
0	7	

$\Rightarrow (1938)_{10} = (792)_{16}$

(Pb) Convert the following hexadecimal number F3A7C2 to binary and octal.

Sol: $(F3A7C2)_{16} = (1111\ 0011\ 1010\ 0111\ 1100\ 0010)_2$

$= (7\ \ 4\ \ 7\ \ 2\ \ 3\ \ 7\ \ 0\ \ 2)_8$

(Pb) Show the value of all bits of a 12 bit register that holds the number equivalent to decimal 215 in (a) binary (b) binary coded decimal (c) binary coded octal (d) binary coded hexadecimal.

Sol: $(215)_{10} = (11010111)_2$

2 | 215		
2 | 107	1	
2 | 53	1	
2 | 26	1	
2 | 13	0	
2 | 6	1	
2 | 3	0	
2 | 1	1	↑
0	1	

a) In a 12 bit register

$(0000\ 1101\ 0111)_2$

b) In binary coded octal

$(000\ 011\ 010\ 111)_2 = (0327)_8$

c) In binary coded hexadecimal

$(0000\ 1101\ 0111)_2 = (0D7)_{16}$

d) binary-coded decimal.

(215) in BCD = $(0010\ 0001\ 0101)_2$

(Pb) show the bit configuration of a 24-bit register when its content represents the decimal equivalent of 295. a) in binary b) in BCD c) in ASCII using 8 bits even parity.

Sol: $(295)_{10} = (?)_2$

2 | 295		
2 | 147	1	
2 | 73	1	
2 | 36	1	
2 | 18	0	
2 | 9	0	
2 | 4	1	
2 | 2	0	
2 | 1	0	
0	1	↑

$(295)_{10} = (100100111)_2$

a) in 24-bit register

$(0000\ 0000\ 0000\ 0001\ 0010\ 0111)_2$

b) in BCD

$(0000\ 0000\ 0000\ 0010\ 1001\ 0101)_{BCD}$

c) in ASCII using 8 bit with even parity

$(10110010\ 00111001\ 00110101)_{ASCII}$

* Complements

- complements are used in digital computers for simplifying subtraction operation and for logical operations.
- There are 2 types of complements for each base "r" system:
  1) (r-1)'s complement
     eg:- 1's complement in binary system
          9's complement in decimal system.
  2) (r)'s complement
     eg:- 2's complement in binary system
          10's complement in decimal system

→ (r-1)'s Complement

Let the number be "N" in base "r" having "n" digits

Then (r-1)'s complement = $(r^n - 1) - N$

eg:- IN decimal no. $r = 10 \Rightarrow r - 1 = 9$

$\Rightarrow 10^n - 1 = 99\ldots9$ (n 9's) i.e for $n=4$, $10^4 - 1 = 9999$

⇒ 9's complement is obtained by subtracting each digit from "9".

Eg:- 9's complement of 549700

= 999999 − 549700

= 450299

For binary number system, $r = 2 \Rightarrow (r-1) = 1$

1's complement of N = $(2^n - 1) - N$

= 11...1 − N

Eg:- $n = 4 \Rightarrow 2^4 - 1 = 10000 - 1 = 1111$

⇒ 1's complement is obtained by subtracting each digit from 1

Eg:- 1's complement of 1101

= 1111 − 1101

= 0010

Note:- Subtractiong each digit from 1 cause bit to change from 1 to 0 and 0 to 1 ⇒ 1's complement is obtained by complementing each bit.

Eg:- 1's complement of 11010011 = 00101100

(r-1)'s complement of octal & hexadecimal are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

→ r's complement

- The r's complement of an n-digit no. in base "r" is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$.
- r's complement = (r-1)'s complement + 1

i.e, $r^n - N = [(r^n - 1) - N] + 1$

Eg:- 10's complement of decimal no. 2389 = 9's complement + 1

= 7610 + 1 = 7611

2's complement of binary no. 101100 = 1's complement + 1

= 010011 + 1 = 010100

Since 10^n is a number represented by a 1 followed by n 0's, The $10^n - N$ i.e., 10's complement of N, can be formed also by leaving all significant 0's unchanged, subtracting all higher significant digits from 9 and subtracting The 1st non-zero least significant digit from 10.

Eg:- 246700 → 10's complement = (9-2) (9-4) (9-6) (10-7) 0 0

= 753300

Illy, 2's complement can be formed by leaving all least significant 0's and The first 1 unchanged (because 2-1 = 1), and Then replacing 1's by 0's and 0's by 1's (because subtracting from 1 complements the binary digit) in all other significant bits.

Eg:- 1101100 → 2's complement = 0010100

↓ complement each bit. ↳ unchanged

(Pb) obtain The 1's and 2's complement of The following 8-bit binary nos.

a) 10101110 c) 1000 0000 e) 0000 0000.

b) 1000 0001 d) 0000 0001

Sol:

	1's complement	2's complement
a) 1010 1110	0101 0001	0101 0010
b) 1000 0001	0111 1110	0111 1111
c) 1000 0000	0111 1111	1000 0000
d) 0000 0001	1111 1110	1111 1111
e) 0000 0000	1111 1111	0000 0000.

Rule: 1's complement ⇒ complement each bit

2's complement ⇒ Leave The least significant 0's and The first least significant non-zero bit unchanged, & complement all higher order bits.

(Pb) obtain the 9's complement of the following 8-digit decimal nos.

a) 12349876 — 87650123

b) 00980100 — 99019899

c) 9000 9951 — 0999 0048

d) 0000 0000 — 9999 9999

Rule: Subtract each digit from 9.

(Pb) obtain the 10's complement of the following 6 digit decimal no.

a) 123900 — 876100

b) 090657 — 909343

c) 100000 — 900000

d) 000000 — 000000

Rule: ① leave significant zero's unchanged

② least significant non-zero no. is subtracted from 10

③ All higher order digits are subtracted from 9.

→ Subtraction of unsigned nos.

The subtraction of 2 n-digit unsigned nos. $M-N$ $(N \neq 0)$ in base "r" can be done as follows:

1. Add the minuend "M" to the r's complement of subtrahend "N"

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

2. If $M \geq N$, the sum will produce a carry r^n which is discarded and what is left is the result $M-N$.

3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N-M)$, which is the r's complement of $(N-M)$. To obtain the answer in a familiar form, take the r's complement of sum and place -ve sign in front.

Eg:- 72582 − 13250 (here $M > N$)

72582 = M

+ 86750 = 10's complement of N

(1) 59282

↓ discard carry

Result = 59282

13250 − 72582 ($M < N$)

13250 = M

+ 27468 = 10's complement of N

40718

↓ No carry

Result = −59282

Eg:- 1010100 - 1000011 (X>Y)

```
  1010100   = X
+ 0111101   = 2's comp of Y
(1) 0010001
```

↓ discard carry

Result = 0010001

1000011 - 1010100 (X<Y)

```
  1000011
+ 0101100
  1101111
```

↓ No carry

Result = -(0010001)

Rule: when subtracting two unsigned nos., if carry occurs = +ve result
if no carry = -ve result.

(Pb) Perform the subtraction with the following unsigned decimal numbers taking the 10's complement of the subtrahend.

a) 5250 - 1321
b) 1753 - 8640
c) 20 - 100
d) 1200 - 250

Sol: a) 5250 - 1321

```
  5250
+ 8679
(1) 3929
```

↓ carry discarded

Result = 3929

b) 1753 - 8640

```
  1753
+ 1360
  3113
```

↓ No carry

Result = -6887

c) 20 - 100

```
  020
+ 900
  920
```

↓ No carry

Result = -80

d) 1200 - 250

```
  1200
+ 9750
(1) 0950
```

↓ discard carry

Result = 950

(Pb) Performs the subtraction of the following binary nos. taking the 2's complement of the subtrahend.

a) 11010 - 10000
b) 11010 - 1101
c) 100 - 110000
d) 1010100 - 1010100

Sol: a) 11010 - 10000

```
  11010
+ 10000
(1) 01010
```

↓ discard carry

Result = 01010

b) 11010 - 1101

```
  11010
+ 10011
(1) 01101
```

↓ discard carry

Result = 01101

c) 100 - 110000

```
  000100
+ 010000
  010100
```

↓ discard carry

Result = -101100

d) 1010100 - 1010100

```
  1010100
+ 0101100
(1) 0000000
```

↓ discard carry

Result = 0000000

* Fixed-point representation

- Positive numbers, including zero, can be represented as unsigned numbers. To represent negative numbers, in ordinary arithmetic,

  positive number is indicated by "+" sign

  negative number is indicated by "−" sign

  In computer because of hardware implementation limitations, everything should be represented using 1's and 0's, including sign bit.

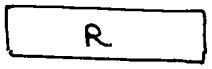
  ⇒ leftmost bit is used to represent sign

  1 ⇒ +ve
  0 ⇒ −ve
  } signed nos.

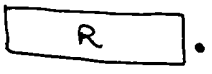
- In addition to sign, a number may have a binary (or decimal) point. Position of binary point is needed to represent fractions, integers or mixed-integer-fraction numbers.
- Two ways to represent binary point position in a register are
  1) Fixed-point representation
  2) Floating-point representation
- Fixed-point representation assumes that the binary point is always fixed at one position.

  The two positions most widely used are

  1) binary point in the extreme left of register
     ⇒ no. is a fraction

  2) binary point in the extreme right of register
     ⇒ no. is an integer

  In either case, binary point is not actually present, but its presence is assumed from the fact that number stored in the register is treated as integer or fraction.

- Integer representation
  - If number is positive ⇒ sign = 0

    Magnitude = positive binary number.

    Eg:- +14 = 0 0001110
    s mag.

- If number is negative ⇒ sign = 1

Magnitude is represented in 3 ways

1. Signed Magnitude representation

   Eg:- −14 = 1 000 1110

2. Signed 1's complement representation

   Eg:- −14 = 1 111 0001

3. Signed 2's complement representation

   Eg:- −14 = 1 111 0010

+14 −14

0 0001110 → 1 0001110 — complement only sign bit

→ 1 1110001 — complement all bits

→ 1 1110010 — 2's complement including sign bit.

1) Signed - Magnitude Representation

- used in ordinary arithmetic
- but awkward when implemented in computer arithmetic

⇒ Signed - complement representation is used in computer arithmetic

2) Signed 1's complement representation

- Imposes difficulties because it has two representations for zero.

  +0 = 0 0000000

  −0 = 1 1111111

  ⇒ not used for arithmetic operations.

- Used in logical operations since change from 1 to 0 and 0 to 1 is equal to logical complement operation.

3) Signed 2's complement representation

- r's complement of an n-digit number $N = r^n - N$
- 2's complement of zero = 0 000 0000

  ⇒ +0 and −0 has same representation in 2's complement representation, hence widely used in computer arithmetic.

```
+6  →  0 0
       0 000 0110
+13 →  0 000 1101
+19    0 0 001 0011
       ↓   ↳ +ve result
   No carry.
```

Result = 0001 0011
= +19

```
-6  →  1 1
       1 111 1010
-13 →  1 111 0011
-19    1 1 110 1101
       ↓   ↳ -ve result
   carry
```

Result = 1110 1101
= − (2's complement 1110 1101)
= −(0001 0011) = −19

Rule:- when addition is performed between two operands of same sign Then if result has no carry ⇒ +ve result
if result has carry ⇒ −ve result (result is in signed 2's complement representation with carry into and out of sign bit being equal.

```
+6  →  0 0
       0 000 0110
-13 →  1 111 0011
-7     0 1 111 1001
       ↓   ↳ -ve result
   No carry.
```

Result = 1 111 1001
= − (2's comp of 1111 1001)
= −7

```
-6  →    1 111 1010
+13 →    0 000 1101
+7     1 0 000 0111
       ↓   ↳ +ve result
   carry.
```

Result = 0 000 0111
= +7.

Rule:- when addition is performed between 2 operands of different sign Then if result has no carry ⇒ −ve result
if result has carry ⇒ +ve result.

(Pb) Perform the arithmetic operations (+42) + (−13) and (−42) − (−13) in binary using signed 2's complement representation for −ve nos.

a) (+42) + (−13)

+42 → 0 101010

−13 → 1 110011

+29 (1) (0) 011101

(1): carry; (0): +ve

Result = 0011101

= +29

b) (−42) − (−13)

−42 → 1 010110

+13 → 0 001101

−29 (0) (1) 100011

(0): No carry; (1): −ve

Result = 1100011

= −29.

→ overflow

- when two nos. of n digits are added & the sum occupies n+1 digits, we say that an overflow occurred.
- If nos. are in unsigned ~~nos~~ representation

  A carry out of most significant digit indicates overflow.
- If nos. are in signed representation

  Sign bit is part of the representation & end carry does not define overflow. If carry into sign bit and out of sign bit are different, an overflow condition is produced. This occurs only if 2 +ve nos. or 2 −ve nos. are added.

Eg:-

+70	= 0 (0 1) 1000110	−70 = 1 (1 0) 0111010
+80	= 0 1010000	−80 = 1 0110000
+150	(1) 0010110 → −ve result	−150 → 1 (0) 1101010 → +ve result
> $2^7 - 1$	⇒ overflow occurred	⇒ overflow occurred
> 127	⇒ Result is corrong	⇒ Result is corrong
		< −128

If the two carriers (carry into sign bit & carry out of sign bit) are applied to an XOR gate, an overflow will be detected when o/p of XOR gate is equal to 1.

In 8086, it is indicated by OF (Overflow) flag.

* Floating point representation

The floating point representation of a number has 2 parts:

1) a signed, fixed-point number called Mantissa.

2) designates the position of binary point called exponent.

Eg:- +6132.789

fraction	exponent	
+0.6132789	+04	$\Rightarrow 0.6132789 \times 10^{+4}$

$\Rightarrow$ floating point representation is of the form:

$$m \times r^e$$

where m = fractional mantissa

r = radix of no. system

e = exponent.

Eg:- +1001.11

fraction	exponent	
+0.100111	000100	$\Rightarrow +(0.100111) \times 2^{+4}$
↙ sign bit (+ve)	↓ 8 bit	↓ 6 bit

The floating point no. is said to be normalized if the most significant digit of mantissa is nonzero.

Eg:- 0.00367 — not normalized

0.367 — normalized.

** Normalized nos. provide the maximum possible precision for the floating-point nos. A zero cannot be normalized because it doesnot have a nonzero digit. It is usually represented by all 0's in mantissa & exponent.

(Pb) Represent the number $(+46.5)_{10}$ as a floating-point binary no. with 24 bits. The normalized mantissa has 16 bits and exponent has 8 bits.

$(+46.5)_{10}$

2 | 46		
2 | 23	0	
2 | 11	1	
2 | 5	1	
2 | 2	1	
2 | 1	0	↑
0	1	

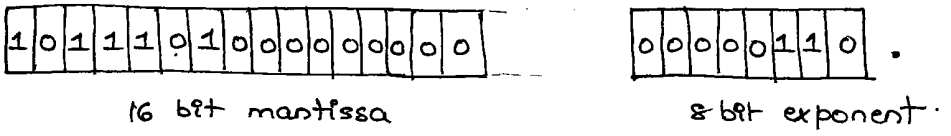
$0.5 \times 2 = 1.0 \quad 1$

$\therefore (0.5)_{10} = (0.1)_2$

$\Rightarrow (46.5)_{10} = (101110.1)_2$

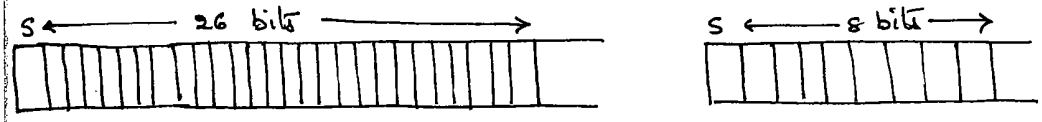
$= 0.1011101 \times 2^6$

$(46)_{10} = (101110)_2$



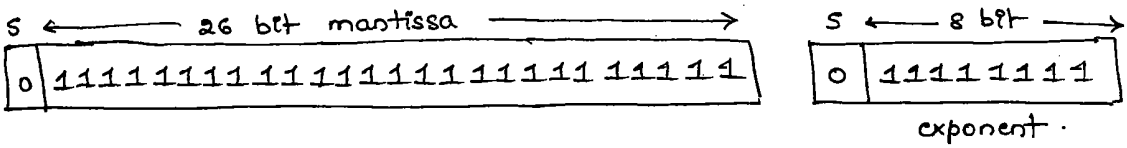
(Pb) A 36-bit floating-point binary number has 8 bits plus sign for the exponent and 26 bits plus sign for the mantissa. The mantissa is a normalized fraction. what are the smallest & largest +ve quantities that can be represented, excluding zero?

Sol: Because zero cannot be normalized, it is not included.



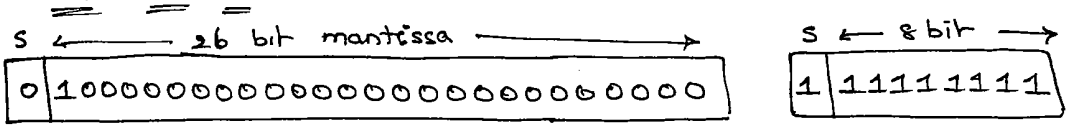
Largest +ve no

s=0 and mantissa = 26, 1's and exponent = 8, 1's with s=0



$\Rightarrow 0.111\cdots1 \times 2^{+255}$ = largest +ve no.

Smallest +ve no.



$\Rightarrow 0.100\cdots0 \times 2^{-255}$ = smallest +ve no.

UNIT – 1

(PART -3)

REGISTER TRANSFER AND MICRO-OPERATIONS

- ✓ Register Transfer Language
- ✓ Register Transfer
- ✓ Bus And Memory Transfers
- ✓ Arithmetic Micro-Operations
- ✓ Logic Micro-Operations
- ✓ Shift Micro-Operations
- ✓ Arithmetic Logic Shift Unit

1.3 REGISTER TRANSFER AND MICRO-OPERATIONS

* Basic Definitions

→ Digital System

- Digital system is a collection of digital hardware modules
- Digital system design invariably uses a modular approach
- Modules are constructed using digital components like registers, decoders, arithmetic elements and control unit.
- Various modules are connected via
  - Datapaths :- routes on which information is moved
  - Control paths:- routes on which control signals are moved
- Digital systems are best defined by the registers they contain and the operations that are performed on the data stored in them

→ Micro-operations

- Micro-operations are elementary operations performed on the data stored in 1 or more registers

  Eg:- shift, count, clear, load
- The result of the micro-operation can replace the binary information of a register or may be transferred to another register

* Register Transfer Language (RTL)

- The internal hardware organization of a digital computer is best defined by
  1) set of registers it contains & their functions
  2) Sequence of micro-operations performed on the binary information stored in registers.
  3) Control that initiates the sequence of micro-operations

- It is possible to specify the sequence of micro-operations in a computer by explaing every operation in words, but this procedure usually involves a lengthy descriptive explanation
- Use of symbols instead of narrative explanation provides an organized and concise manner for listing the micro-operations sequences in registers and control functions that initiate them.

A symbolic notation used to describe the micro-operation transfers among registers is called "Register Transfer Language" (RTL).

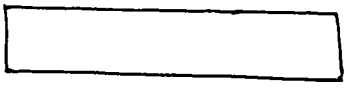
* Register Transfer

→ Registers.

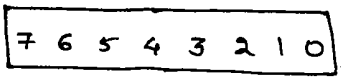
- Computer registers are denoted by upper letters and are optinally followed by digits and letters.

Eg:- MAR (Memory Address Register)
PC (Program Counter)
IR (Instruction Register)
R_1 (Processor Register 1)

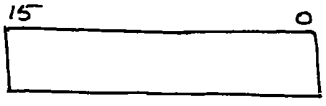
- An n-bit register requires "n" flip-flops, each F/F holding 1 bit data. Each F/F in an n-bit register is numbered in a sequence from 0 to n-1, starting from 0 in the rightmost position and increases towards left.



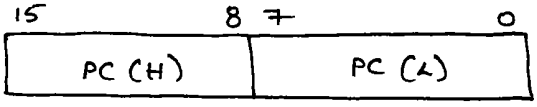
a) Register R_1



b) Showing individual bits



c) Numbering of bits

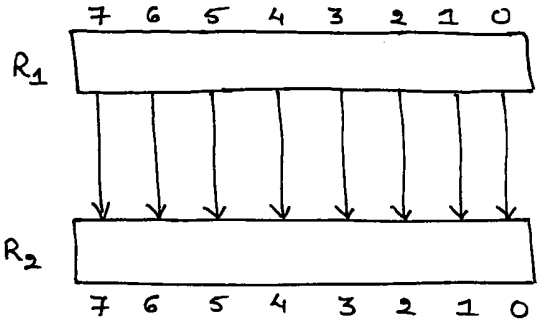


d) devided into 2 parts

- common way to represent a register is by a rectangular box with the name inside it (fig (a))
- The numbering of bits is shown / marked on the top of the register (fig (c))
- PC a 16-bit register is also accessed as 2, 8-bit registers.
  Lower byte = PC (L) or PC (0-7)
  Higher byte = PC (H) or PC (8-15)

→ Register Transfer

- Information transfer from one register to another is designated in symbolic form by means of a "replacement operator" (←)
  Eg:- $R_2 \leftarrow R_1$ denotes a transfer of data from register R_1 into register R_2 and also indicates that contents of R_2 are replaced by contents of R_1. Here, contents of R_1 are unaltered.
- A statement that specifies a register transfer implies that circuits are available from the outputs of source register to inputs of destination register and the destination register has "parallel load capability."



⇒ parallel load capability.

All bits of R_1 are transferred to R_2 at the same time.

→ Control function

- If we want the transfer to occur only under predetermined control condition, it can be shown by an if-Then statement.
  Eg:- if (P = 1) Then $R_2 \leftarrow R_1$
  where P = control signal generated by a control unit.

- we can seperate the transfer control variables from register transfer operation by specifying a control function
- A control function is a boolean variable that is equal to 0 or 1. It is included in register transfer statement as

$$P : R_2 \leftarrow R_1$$

here, control condition is terminated by a colon (:)

=> transfer operation will be executed by hardware only if $P = 1$

(Pb) Represent the following conditional control statement by two register transfer statements with control functions.

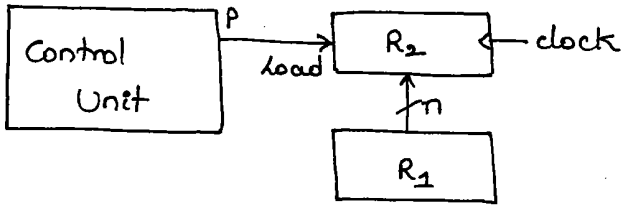
If $(P=1)$ Then $(R_1 \leftarrow R_2)$ else if $(Q=1)$ Then $(R_1 \leftarrow R_3)$

Sol: $P : R_1 \leftarrow R_2$ (if $P=1$)

$\bar{P}Q : R_1 \leftarrow R_3$ (if $P=0$ and $Q=1$)

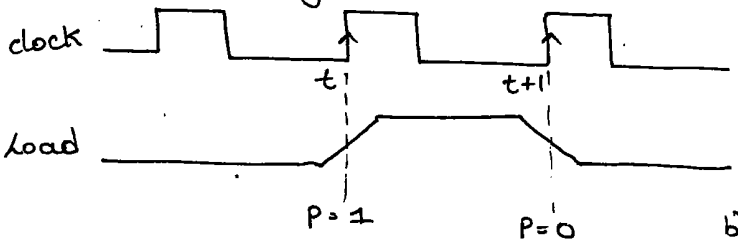
→ Hardware Implementation of RTL statement

- Every statement written in RTL implies a hardware construction for implementing the statement.
- Eg:- Block diagram for $P : R_2 \leftarrow R_1$ is as shown below



a) Block diagram

- "n" o/p's of register R_1 are connected to "n" i/p's of register R_2.
- n is used to indicate no. of bits in a register
- R_2 has a load i/p activated by control variable "P"
- Assume "P" is synchronized with the clock signal applied to R_2.



b) Timing diagram

- "P" is activated by the control unit at the rising edge of a clock pulse at "t". The next transition of clock at "t+1" finds load i/p active and data i/p's of R_1 are loaded into R_2 in parallel.
- "P" may go back to "0" at "t+1", otherwise transfer will occur with every clock pulse transition when P remains active.
- clock is not included in RTL statement assuming that all transfers occur during a clock edge transition only.

→ Basic symbols for register transfers

Symbol	Description	Examples
Letters	Denotes a register	MAR, R_2, R_1
( )	Denotes part of a register	$R_2(0-7)$, $R_2(L)$
←	Denotes transfer of Information	$R_2 \leftarrow R_1$
,	Seperates 2 μ-operations	$R_2 \leftarrow R_1$, $R_1 \leftarrow R_2$

Consider the statement

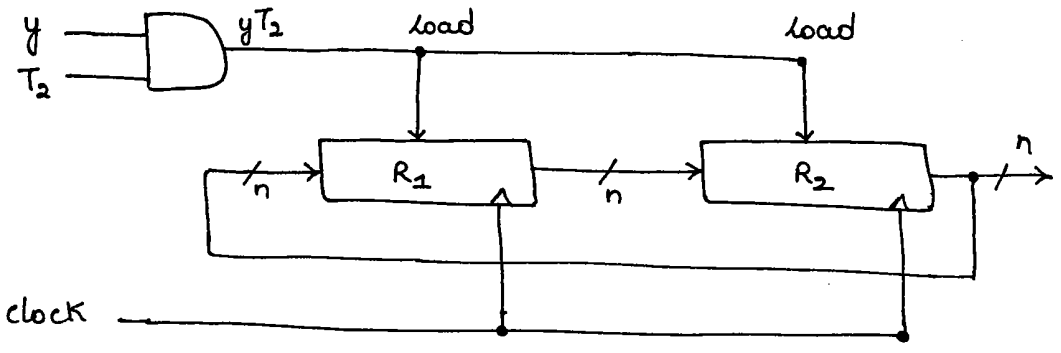
$$T: R_2 \leftarrow R_1 \ , \ R_1 \leftarrow R_2$$

denotes operation that exchanges contents of 2 registers during one common clock pulse provided $T = 1$.

(Pb) show the block diagram of the hardware that implements the following register transfer statement

$$yT_2 : R_2 \leftarrow R_1 \ , \ R_1 \leftarrow R_2.$$

Sol:



(Pb) The outputs are 4 registers R_0, R_1, R_2 and R_3 are connected Through 4-to-1 multiplexers to The i/p's of a 5th register R_5. Each register is 8 bits long. The required transfers are dictated by 4 timing variables T_0 Through T_3 as follows:

$T_0 : R_5 \leftarrow R_0$

$T_1 : R_5 \leftarrow R_1$

$T_2 : R_5 \leftarrow R_2$

$T_3 : R_5 \leftarrow R_3$

The timing variables are mutually exclusive, which means only one variable is equal to 1 at any given time, while The other 3 are equal to 0.

Draw The block diagram showing The hardware implementation of register transfers. Include The connections necessary from The 4 timing variables to The selection i/p's of The multiplexers and to The load i/p of register R_5.

Sol:

T_0	T_1	T_2	T_3	S_1	S_0		Load R_5
0	0	0	0	X	X		0
1	0	0	0	0	0	(R_0)	1
0	1	0	0	0	1	(R_1)	1
0	0	1	0	1	0	(R_2)	1
0	0	0	1	1	1	(R_3)	1

S_1 and S_0 are select i/p's of 4 to 1 multiplexer

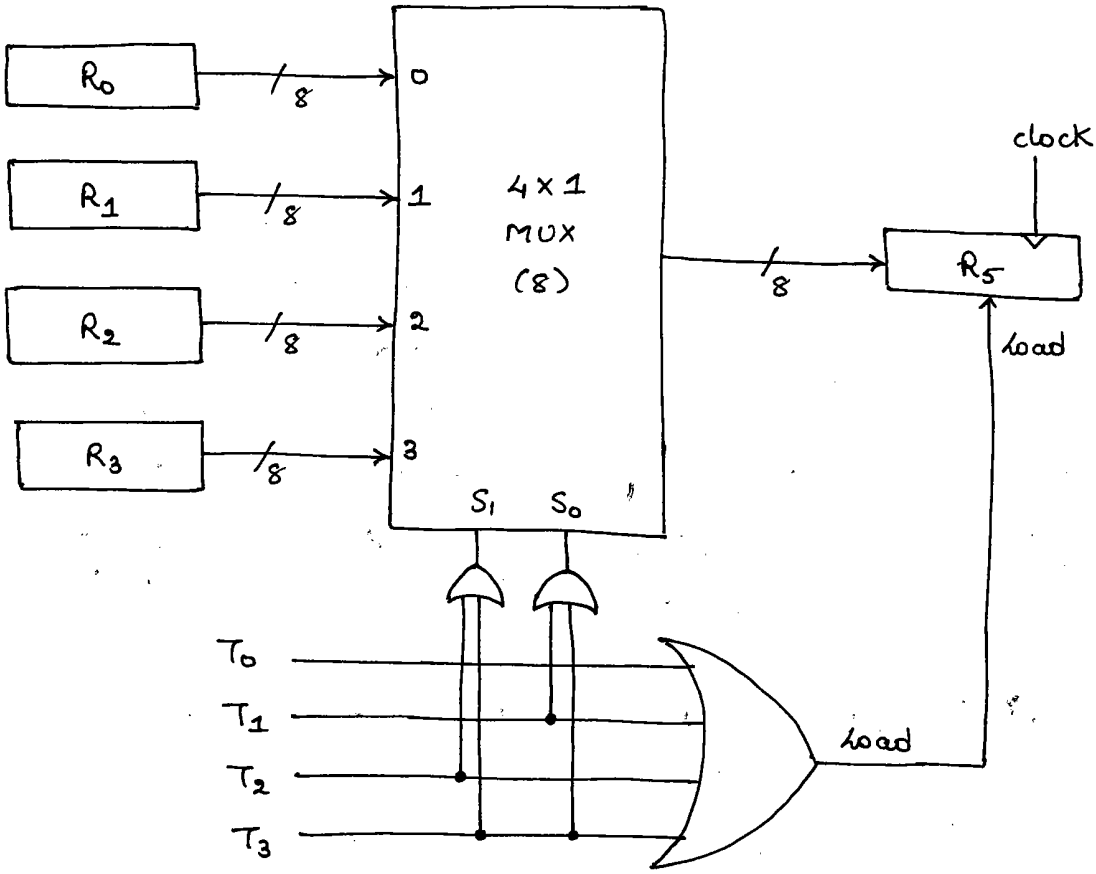
$\Rightarrow S_0 = T_1 + T_3$ (OR operation)

$S_1 = T_2 + T_3$ (OR operation)

Load $R_5 = T_0 + T_1 + T_2 + T_3$ (OR operation)

Consider each register is of 8 bits

$\Rightarrow$ we require 8, 4X1 multiplexers to provide parallel load capability.



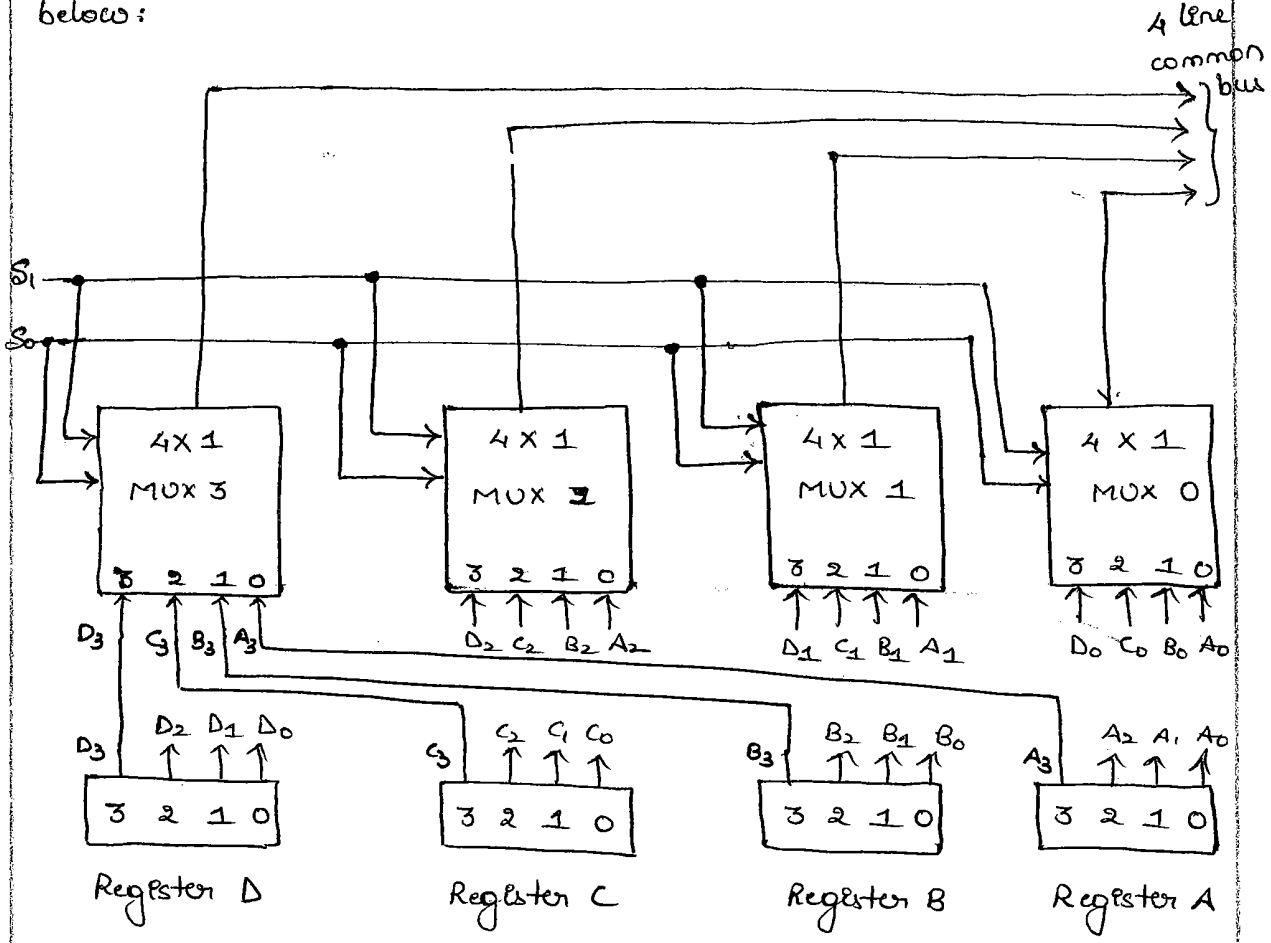
* Bus and Memory transfers

→ Bus transfer

- A more efficient way of transferring information between registers in a multiple-register configuration is a common bus system.
- A bus structure consists of a set of common lines, each of which carriers 1 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.
- Different ways of constructing a common bus are
  1) Using Multiplexers
  2) Using Tri-state buffers.

1) Using Multiplexers

A bus system for 4 registers using multiplexers is shown below:



Each register has 4 bits numbered 0 to 3

No. of registers = 4 ⇒ Size of mux connecting the register bits to common bus line

= 4 x 1 mux

Since, each register has 4 bits

⇒ we require 4 lines in a common bus to transfer the 4 bits in parallel.

⇒ we require 4 muxes.

and 1 I/p is selected using 2 select lines as size of the mux is 4x1.

⇒ Bus consists of 4, 4x1 multiplexers each having four data inputs 0 Through 3, and 2 select lines S_1 and S_0.

Eg:- Mux 0 multiplexes the four 0^{th} bits of all registers i.e A_0, B_0, C_0, D_0

⇒ Bits in the same significant position in each register are connected to the data i/p's of 1 mux to form 1 line of bus.

Bus selection

The selection inputs S_1 and S_0 are connected to the selection inputs of all 4 multiplexers. Select lines choose the 4 bits of 1 register and transfer them into the four-line common bus.

Eg:- when $S_1S_0 = 00$ ⇒ 0^{th} data i/p's of all 4 muxes i.e A_0, A_1, A_2, A_3 are selected and applied to the o/p's that form the bus. This causes the bus lines to receive the contents of register A since the o/p's of this register are connected to 0^{th} data i/p's of multiplexers.

S_1	S_0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

A bus system with "k" registers of "n" bits each, produce an n-line common bus

⇒ No. of multiplexers required = n

= no. of bits in each register

Size of each mux = $K \times 1$

= no. of registers $\times$ 1

Bus transfer

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines

to the i/p's of all destination registers and activating the load control of particular destination register selected.

The symbolic notation for a bus transfer, when the bus is included in the statement is

$$BUS \leftarrow C, \quad R_1 \leftarrow BUS \quad \equiv \quad R_1 \leftarrow C$$

contents of C are placed on bus and contents of bus are loaded into R_1 by activating its load control i/p.

If bus is known to exist in the system, one can directly write the statement as shown above.

(Pb) A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers?

a) How many selection inputs are there in each multiplexer?

b) what size of multiplexers are needed?

c) How many multiplexers are there in the bus?

Sol: a) Each multiplexer has to select 1 out of 16 registers

$\Rightarrow$ each multiplexer requires 4 selection i/p's $(2^4 = 16)$

b) Size of each mux = no. of registers $\times$ 1

$= 16 \times 1$

c) No. of muxes in the bus = no. of bits in each register

$= 32$

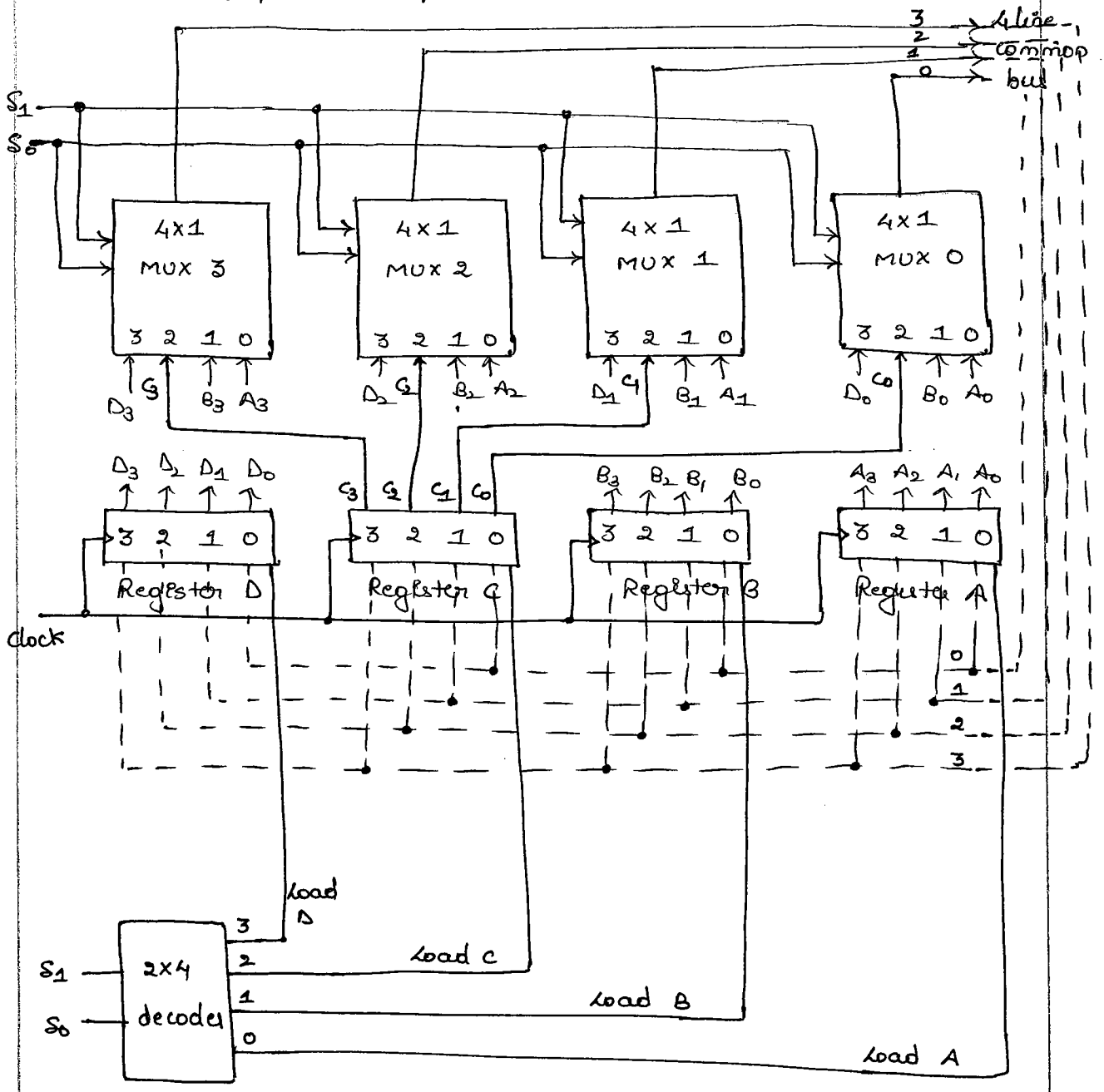
(Pb) what has to be done to the bus system designed using muxes to be able to transfer information from any register to any other register? Specially, show the connections that must be included to provide a path from the outputs of register C to the i/p's of register A?

Sol: To transfer information from any register to any other register in the bus system using muxes:

1. Connect the 4-line common-bus to all register i/p's.
2. Provide a load i/p in each register
3. Provide a clock i/p for each register.

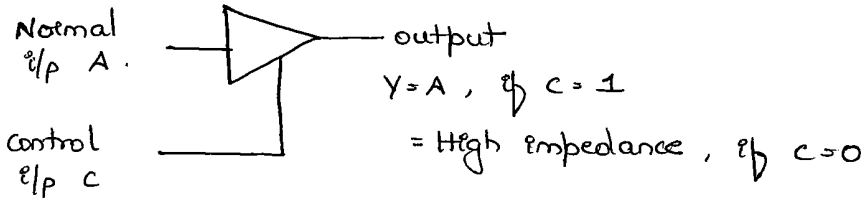
To transfer from register C to register A

1. Apply $S_1 S_0 = 10$ (to select C for the bus)
2. Enable load i/p of A
3. Apply a clock pulse.



2) Using Tristate Buffers

- A tristate gate is a digital circuit that exhibits 3 states.
  - 2 states – Signals equivalent to logic 1 and logic 0 in a conventional gate
  - 3rd state – 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. Symbol of tri state buffer is

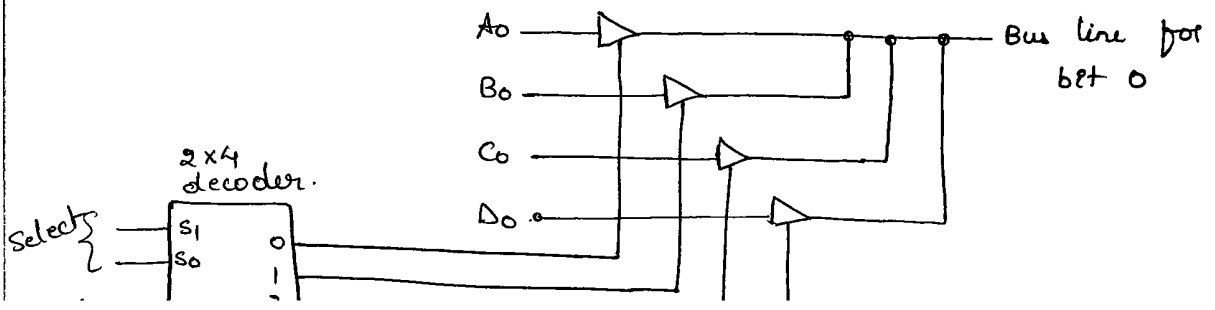


when control i/p C = 1,
- ⇒ acts as a normal gate / buffer
- ⇒ Y = A

when control i/p C = 0,
- ⇒ o/p is disabled and gate goes into high-impedance state, regardless of i/p A.

Tri state buffer may perform any logic such as AND, NAND, etc. However, tri-state gate is most commonly used in the design of a bus system.

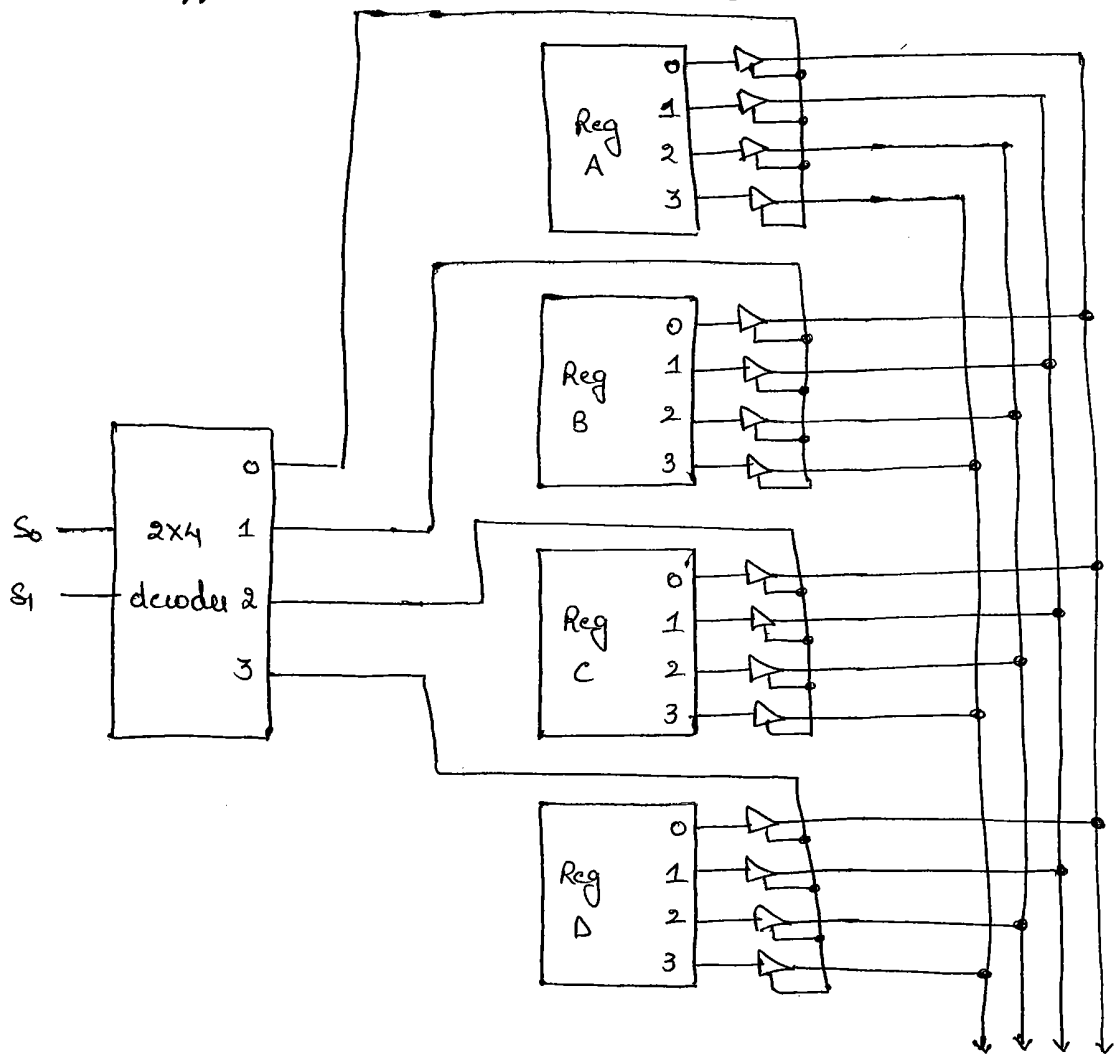
- The high-impedance state of a 3-state gate provides a special feature not available in other gates. Because of this feature, a large no. of 3-state gates o/p's can be connected with wires to form a common bus line without endangering loading effect.
- The construction of bus with 3-state buffers is shown below:



- o/p's of 4 buffers are connected to form a single bus line.
- (~~This~~ This is not possible with conventional gates without 3rd state)
- control i/p's to buffer determine which of the 4 normal i/p's will communicate with the bus line.
- No more than 1 buffer is active at any given time
  ⇒ we use a decoder to ensure it.
  when E=0 ⇒ bus line is in high-impedance state
  when E=1 ⇒ 1 of the 4 buffers will be active depending on the binary select i/p's S_1 and S_0.

To construct a common bus for 4 registers of n bits each using tri state buffers, we need "n" circuits with 4 buffers in each

(Pb) Draw the diagram of a bus system for 4 registers that uses 3-state buffers and a decoder instead of multiplexers.



→ Memory Transfer

- The transfer of information from memory word to outside environment is called Read operation
- A transfer of new information to be stored into memory is called a write operation.
- Memory word is symbolized by "M".
- A particular memory word among the available is selected by memory address during the transfer.
- Memory address is enclosed in square brackets "[ ]" followed by "M".
- consider a memory unit receives the address from a register, called Address register (AR). The data is transferred to another register called Data register (DR).

Read : $DR \leftarrow M[AR]$ ⇒ Read operation

- The write operation transfers the content of a data register to a memory word "M" selected by the address. Assume that i/p data are in register R_1 and address in AR.

write : $M[AR] \leftarrow R_1$ ⇒ write operation.

(Pb) The following transfer statements specify a memory. Explain the the memory operation in each case.

a) $R_2 \leftarrow M[AR]$

b) $M[AR] \leftarrow R_3$

c) $R_5 \leftarrow M[R_5]$

Sol: a) $R_2 \leftarrow M[AR]$

Read memory word specified by the address in AR into R_2

b) $M[AR] \leftarrow R_3$

write contents of R_3 into memory word specified by address in AR.

c) $R_5 \leftarrow M[R_5]$

Read word specified by the address in R_5 and transfer content to R_5. It destroys the previous value of R_5.

* Arithmetic μ-operations

A micro-operation is an elementary operation performed with the data stored in the registers.

The micro-operations most often encountered in digital computer are classified into 4 categories:

1) Register-transfer μ-operations

It is transfer of binary information from 1 register to another. It doesnot change the information content when the binary information moves from source register to the destination register. The other 3 types of μ-operations change the information content during the transfer.

2) Arithmetic μ-operations

Performs arithmetic operations on numbers stored in registers

3) Logical μ-operations

Performs bit manipulation operations on non-neumeric data stored in registers.

4) shift μ-operations

Performs shift operations on data stored in registers.

→ Basic Arithmetic operations are

- Addition
- Subtraction
- Increment
- Decrement
- shift (explained in conjunction with shift μ-operations)

a) Add μ-operations

$$R_3 \leftarrow R_1 + R_2$$

It denotes that contents of register R_1 are added to contents of register R_2 and the sum is transferred to register R_3

To implement statement with hardware we require 3 registers and a digital component that performs the addition operation.

b) Subtract μ-operations

Instead of using minus operator, we can specify subtraction by the following statement.

$$R_3 \leftarrow R_1 + \overline{R_2} + 1$$

here $\overline{R_2}$ = 1's complement of R_2

$\overline{R_2} + 1$ = 2's complement of R_2

Adding contents of R_1 with 2's complement of R_2 is equal to $R_1 - R_2$.

c) Increment & Decrement μ-operation

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

$R_2 \leftarrow R_2 + 1$ = Increment

$R_1 \leftarrow R_1 - 1$ = Decrement

A list of arithmetic μ-operations are tabularized below:

Symbolic designation	Description
1) $R_3 \leftarrow R_1 + R_2$	contents of R_1 plus R_2 transferred to R_3
2) $R_3 \leftarrow R_1 - R_2$	contents of R_1 minus R_2 transferred to R_3
3) $R_2 \leftarrow \overline{R_2}$	complements contents of R_2 (1's complement)

4) $R_2 \leftarrow \overline{R_2} + 1$ — 2's complement of R_2

5) $R_3 \leftarrow R_1 + \overline{R_2} + 1$ — R_1 plus 2's complement of R_2 (subtraction)

6) $R_1 \leftarrow R_1 + 1$ — Increment contents of R_1 by 1

7) $R_2 \leftarrow R_2 - 1$ — Decrement contents of R_2 by 1

Multiply and divide operations are valid arithmetic operations but not included in the basic set of μ-operations

In most computers, multiplication operation is implemented with an add & shift operation and division is implemented with a sequence of subtract and shift μ-operations.

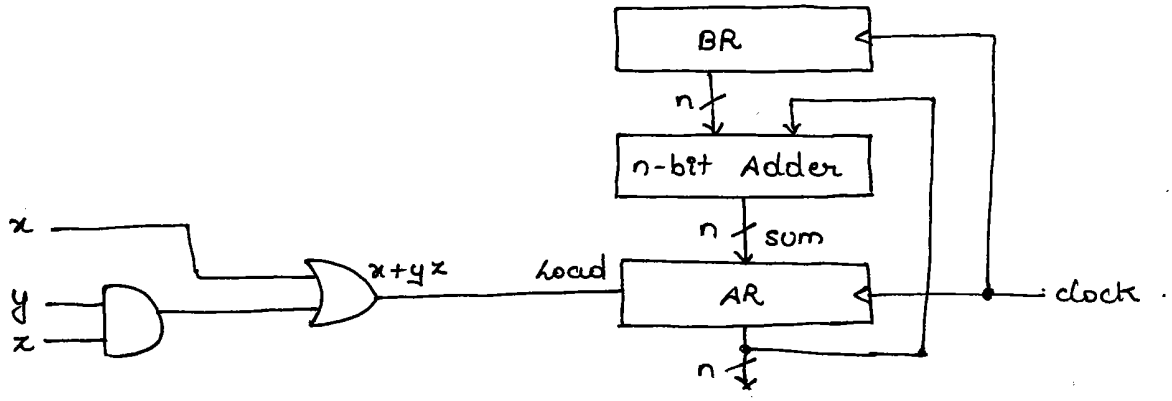
(Pb) Draw the block diagram for the hardware that implements the following statement

$$x + yz : AR \leftarrow AR + BR$$

where AR and BR are 2, n-bit registers and x, y, z are control variables. Include the logic gates for control functions (Remember that the symbol and designates OR operation in control of boolean function but it represents an arithmetic plus in μ-operation)

Sol:

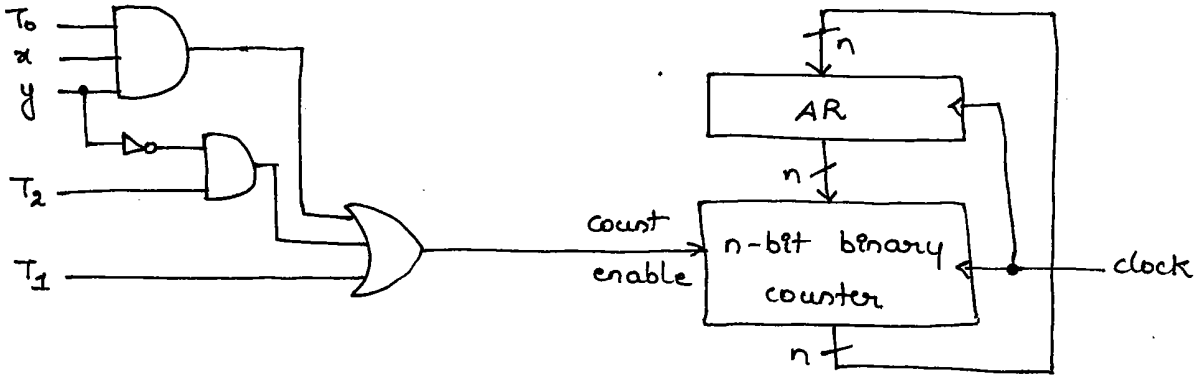
$$\underbrace{x + yz}_{\text{OR operation}} : AR \leftarrow \underbrace{AR + BR}_{\text{addition}}$$



(Pb) Show the hardware that implements the following statement. Include the logic gates for the control input (function) and a block diagram for the binary counter with a count enable input.

$$xyT_0 + T_1 + y'T_2 : AR \leftarrow AR + 1$$

Sol:



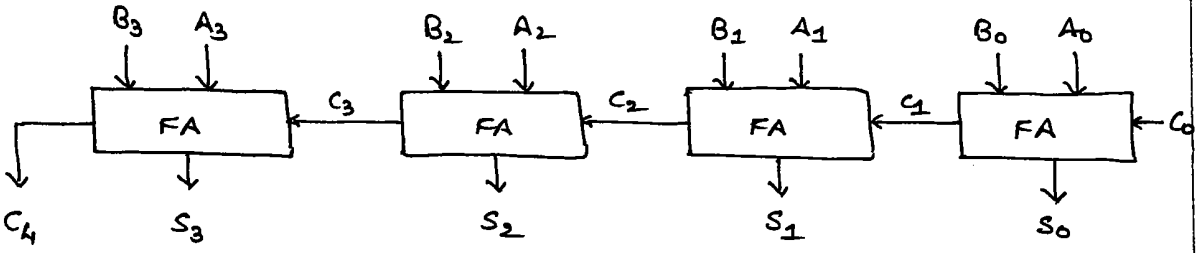
n-bit binary counter is enabled by the control function $xyT_0 + T_1 + y'T_2$. n-bit binary counter is loaded with parallel data from AR register and o/p from binary counter is loaded into AR after incremented by 1, i.e after 1 clock pulse.

→ Binary Adder

To implement the add micro-operation with hardware, we need registers to hold the data and a digital component that performs the arithmetic addition.

Digital circuit that performs arithmetic sum of 2 bits and a previous carry is a full adder. The digital circuit that performs arithmetic sum of 2 binary numbers of any length is called a binary adder.

A binary adder circuit for 4-bit numbers is shown below



The binary adder is constructed with full adder circuits connected in cascade, with the o/p carry from one full adder connected to i/p carry of next full adder.

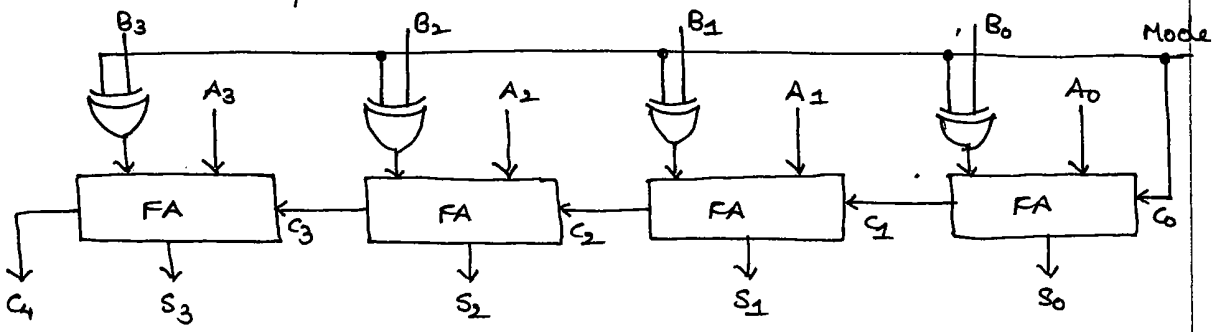
C_0 is i/p carry to binary adder and C_4 is the o/p carry. The S o/p's of full adders generate the required sum bits.

A n-bit binary adder requires n-full adders. The n-bits for the A i/p's come from one register say R_1 and the n-bits for the B i/p's come from another register say R_2. The final sum can be transferred to a 3rd register or to any one of the source registers replacing the previous contents.

→ Binary - Adder - Subtractor

The addition and subtraction operations can be combined into 1 common circuit by including an X-OR gate with each full adder

A 4-bit binary adder-subtractor circuit is shown below



The mode bit "M" controls the operation

when M=0, the circuit is an adder and

when M=1, the circuit becomes a subtractor

Each XOR gate receives i/p M and 1 of the i/p's of B.

when M=0, we have $0 \oplus B = B$ ⇒ full adder receives A + B

⇒ A+B ⇒ Addition

when M=1, we have $1 \oplus B = \overline{B}$ ⇒ full adder receives A, $\overline{B}$

and i/p carry of 1 ⇒ $A+\overline{B}+1$

⇒ Subtraction.

(Pb) The adder-subtractor circuit has the following values of I/p mode M and data inputs A and B. In each case, determine the values of O/p's S_3, S_2, S_1, S_0 and C_4.

	M	A	B
a)	0	0111	0110
b)	0	1000	1001
c)	1	1100	1000
d)	1	0101	1010
e)	1	0000	0001

Sol:

	M	A	B	C_4	$S_3 S_2 S_1 S_0$	
a)	0	0111 +	0110	0	1101	(A=7, B=6, M=0 ⇒ A+B = 7+6 = 13) 1101
b)	0	1000 +	1001	1	0001	(A=8, B=9, M=0 ⇒ A+B = 8+9 = 17) 10001.
c)	1	1100 −	1000	1	0100	(A=12, B=8, M=1 ⇒ A−B = 12−8 = 4 0100 discard carry).
d)	1	0101 −	1010	0	1011	(A=5, B=10, M=1 ⇒ A−B = 5−10 = −5 Result in 2's complement)
e)	1	0000 −	0001	0	1111	(A=0, B=1, M=1 ⇒ A−B = 0−1 = −1 Result in 2's complement)

for subtraction take 2's complement of B and add it to A.

if carry = 1 ⇒ discard carry & result is +ve

if carry = 0 ⇒ result is −ve and is represented in its 2's complement

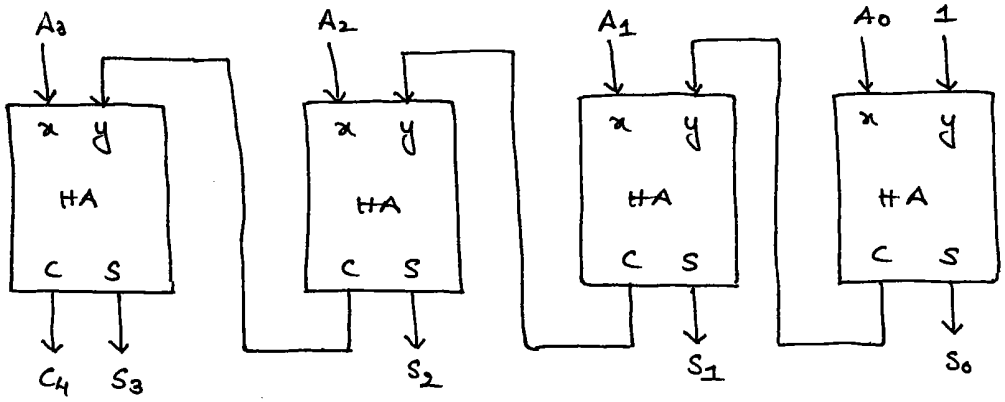
→ Binary Incrementer

- Binary Incrementer μ-operation adds 1 to a number in a register

Eg:- 0110 → 0111

- Binary Incrementer μ-operation can be easily implemented by a binary counter. Each time the clock is enabled, the clock pulse transition increments the content of register by 1.
- when increment operation must be done independent of a particular register coe use a combinational circuit.

This is accomplished by means of half-adder connected in cascade.



one of the i/p's to the least significant HA is connected to logic 1 and the other i/p is connected to the least significant bit of the number to be incremented.

o/p carry from one half-adder is connected to one of the i/p's of the next higher-order half-adder.

The circuit receives the 4 bits from A_0 through A_3, adds one to it and generates o/p in S_0 through S_3. o/p carry C_4 will be 1 only after incrementing binary 1111. This also causes o/p's S_0 to S_3 to goto "0".

An n-bit binary incrementer requires "n" half-adders. The least significant bit must have 1 i/p connected to logic 1. other i/p's receive the no. to be incremented & carry from previous stage.

(Pb) Design a 4-bit combinational circuit decrementer using 4 full adder circuits.

Sol: Decrementer = A − 1

= A + 2's complement of 1

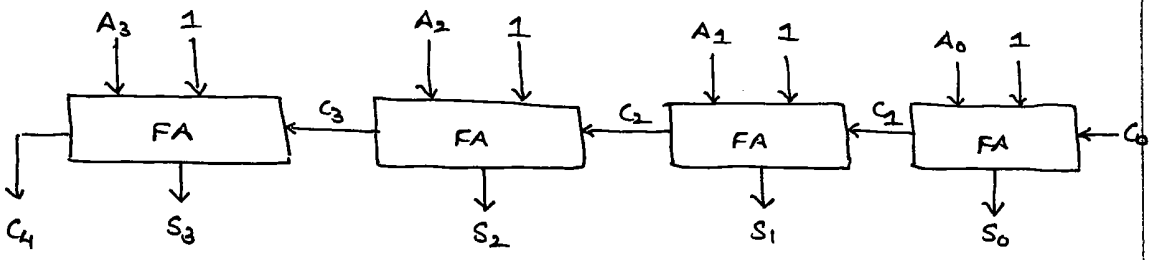
Since A is a 4 bit no., 1 is represented as 0001

⇒ 2's complement of 0001 = 1111

⇒ Decrement A ⇒ A + 1111

= $A_3 A_2 A_1 A_0$ + 1111

⇒ A binary adder with one number as 1111 will act as decrementer as shown below



→ Arithmetic Circuit

All arithmetic micro-operations can be implemented in one composite 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.

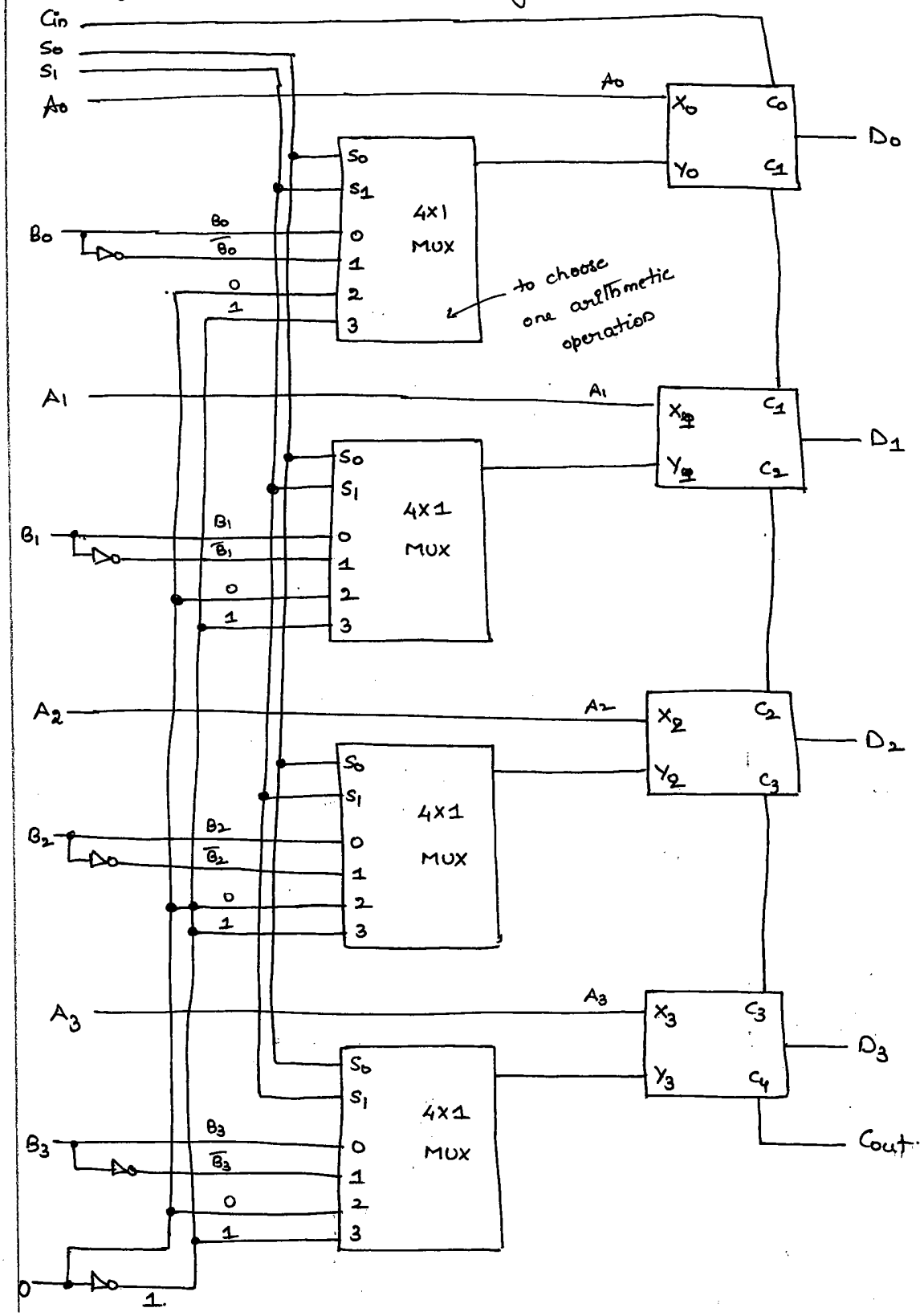
The diagram for a 4 bit arithmetic circuit is shown below:

- here i/p's are A ($A_3 A_2 A_1 A_0$) and B ($B_3 B_2 B_1 B_0$) − 4 bits each
- multiplexers are used for choosing different operations
- 4 i/p's of A are directly connected to one i/p X_i of full adder
- Each of the 4 i/p's of B are connected to data i/p's of muxes
- Data i/p's of muxes also receive the complement of B i/p's
- other two i/p's of muxes are logic 0 and logic 1.
- select lines S_1 and S_0 are used to control the multiplexers.
- The i/p carry C_{in} goes to carry i/p of the least significant position full adder.

- other carriers are connected from 1 stage to the next stage.
- o/p binary adder is given by

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

- By controlling the values of Y with select line S_0 and S_1 and making $C_{in} = 0$ or 1, we can generate 8 arithmetic μ-operations



Select			Input	output	Micro operation
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	
0	0	0	B	$D = A + B + 0$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B} + 0$ $= A - B - 1$.	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0000	$D = A$	Transfer A
1	0	1	0000	$D = A + 1$	Increment A
1	1	0	1111	$D = A - 1$	Decrement A
1	1	1	1111	$D = A - 1 + 1$ $= A$	Transfer A

$1 = C_{in}$

let A = 1101
Y = 0000
1110
$= A + 1$

$C_{in} = 0$
let A = 1101
Y = 1111
1100
$= A - 1$.

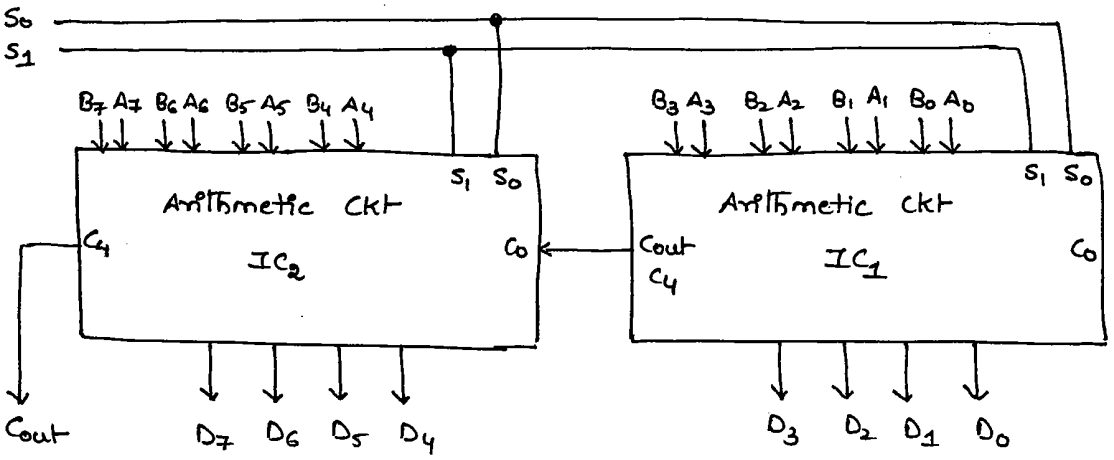
$1 = C_{in}$
let A = 1101
Y = 1111
1101 = A.

$(1)_{10} = (0001)_2$

$\Rightarrow$ 2's complement $= (1111)_2$

(Pb) Assume That The 4-bit arithmetic circuit is enclosed in one IC package. Show The connections among 2 such IC's to form an 8 bit arithmetic circuit.

Sol:



0000 and 1111 are default i/p's to both The IC's.

(Pb) Design an arithmetic circuit with one selection variable S and two n-bit data inputs A and B. The circuit generates the following 4 arithmetic operations in conjunction with i/p carry C_{in}. Draw the logic diagram for the 1st two stages.

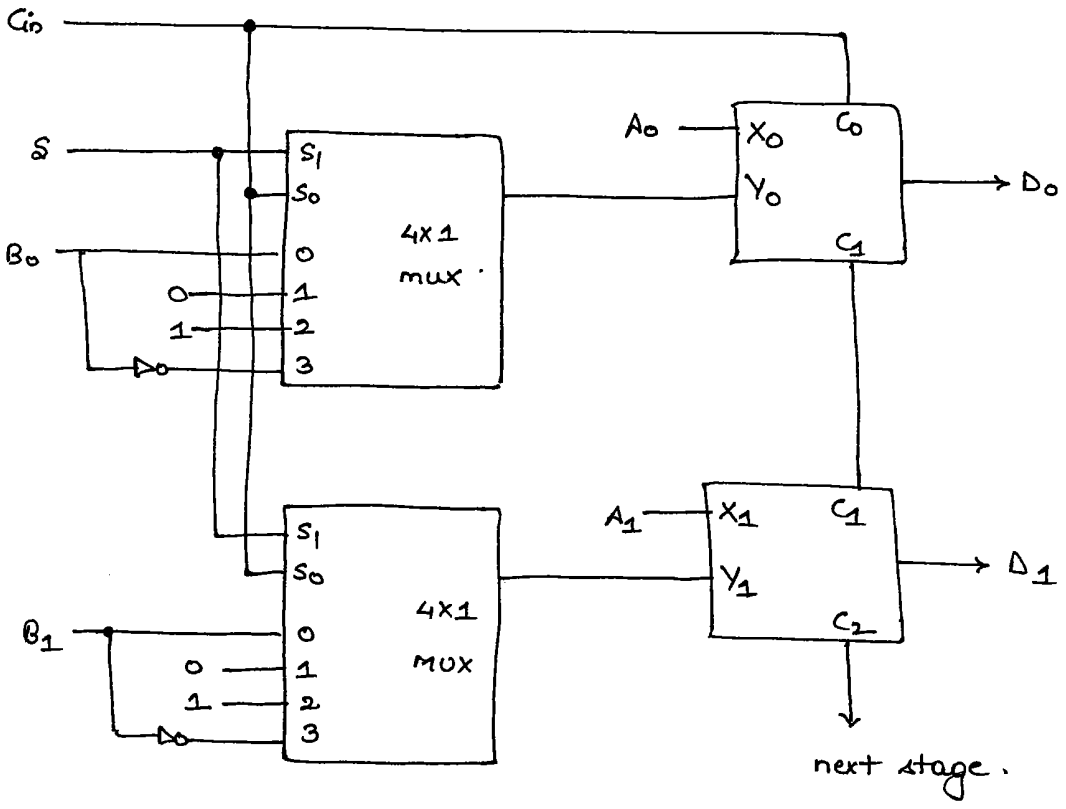
S	$C_{in}=0$	$C_{in}=1$
0	D = A+B (add)	D = A+1 (increment)
1	D = A-1 (decrement)	D = A+$\bar{B}$+1 (subtract)

Sol:

S	C_{in}	D	X	Y	C_{in}	S_1	S_0
0	0	A+B	A	B	0	0	0
0	1	A+1	A	00..0	1	0	1
1	0	A-1	A	11..1	0	1	0
1	1	A+$\bar{B}$+1.	A	$\bar{B}$	1	1	1

$S_1 = S$

& $S_0 = C_{in}$



* **Logic Micro-operations**

Logic μ-operations specify binary operations for strings of bits stored in registers, considering each bit of the register seperately

Eg:- XOR μ-operation is symbolized by the statement

$$P: R_1 \leftarrow R_1 \oplus R_2$$

Let $R_1 = 1010$, $R_2 = 1100$ Then

$$R_1 \oplus R_2 \Rightarrow \begin{array}{r} 1010 \\ \underline{1100} \\ 0110 \end{array} = R_1 \text{ if } P=1.$$

μ-operations are seldom used in scientific computations, but they are very useful in bit manipulation of binary data and for making logical decisions

Special symbols will be adopted by logical μ-operations OR, AND and complement to distinguish

1) Them from the corresponding symbols used to express boolean control functions
2) The symbol "+", when used to symbolize arithmetic plus, from a logic OR operation.

Symbols used are

$\wedge$ → AND μ-operation

$\vee$ → OR μ-operation

(bar) $\overline{\ }$ → complement μ-operation

Eg:- $P + Q : R_1 \leftarrow R_2 + R_3 , R_4 \leftarrow R_5 \vee R_6$

↓	↓	↓
OR operation blw 2 binary/ boolean variables of a control func.	Arithmetic plus i.e., add μ-operation	OR μ-operation blw $R_5 + R_6$ (bit wise)

Reason ②

Reason ①

→ List of logic μ-operations

There are 16 logic operations that can be performed with 2 binary variables. They can be determined from the following truth table obtained with 2 binary variables x and y.

A	B															
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

F_0: $F \leftarrow 0$ (clear), $F = 0$

F_1: $F \leftarrow A \wedge B$ (AND), $F = xy$

F_2: $F \leftarrow A \wedge B'$, $F = xy'$

F_3: $F \leftarrow A$ (transfer), $F = x$

F_4: $F \leftarrow \bar{A} \wedge B$, $F = x'y$

F_5: $F \leftarrow B$ (transfer), $F = y$

F_6: $F \leftarrow A \oplus B$ (XOR), $F = x \oplus y$

F_7: $F \leftarrow A \vee B$ (OR), $F = x + y$

F_8: $F \leftarrow \overline{A \vee B}$ (NOR), $F = (x + y)'$

F_9: $F \leftarrow \overline{A \oplus B}$ (XNOR), $F = (x \oplus y)'$

F_{10}: $F \leftarrow \bar{B}$, $F = \bar{y}$

F_{11}: $F \leftarrow A \vee \bar{B}$, $F = x + \bar{y}$

F_{12}: $F \leftarrow \bar{A}$, $F = \bar{x}$

F_{13}: $F \leftarrow \bar{A} \vee B$, $F = \bar{x} + y$

F_{14}: $F \leftarrow \overline{A \wedge B}$ (NAND), $F = (xy)'$

F_{15}: $F \leftarrow 1$ (set to 1's), $F = 1$.

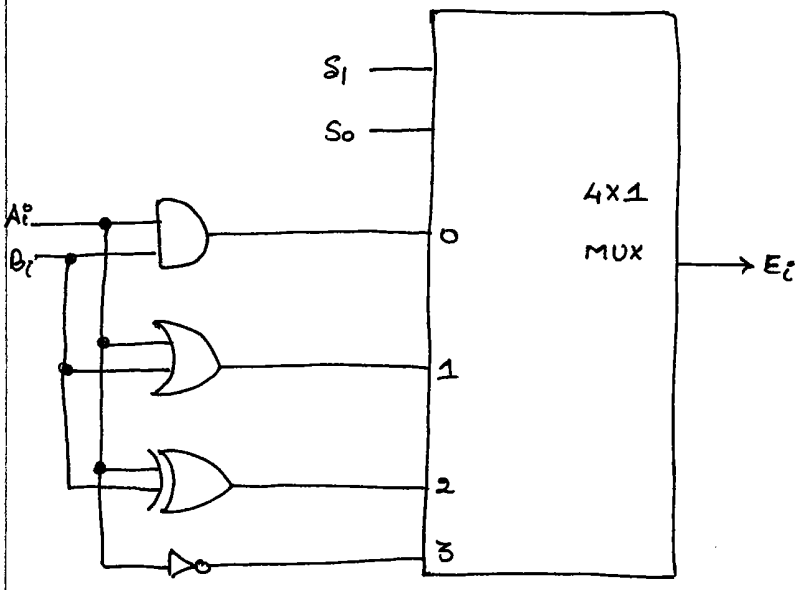
→ Hardware Implementation

The hardware implementation of logic μ-operations 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 functions only 4 are most commonly used - AND, OR, XOR and complement, from which others can be derived.

Figure below shows one stage of a circuit that generates the four basic logic μ-operations

1) AND
2) OR
3) XOR
4) complement



This logic diagram shows one typical stage with subscript "i"

For a logic circuit with "n" bits, the diagram must be repeated "n" times for $i = 0, 1, 2, \ldots n-1$.

Select lines are applied to all the stages.

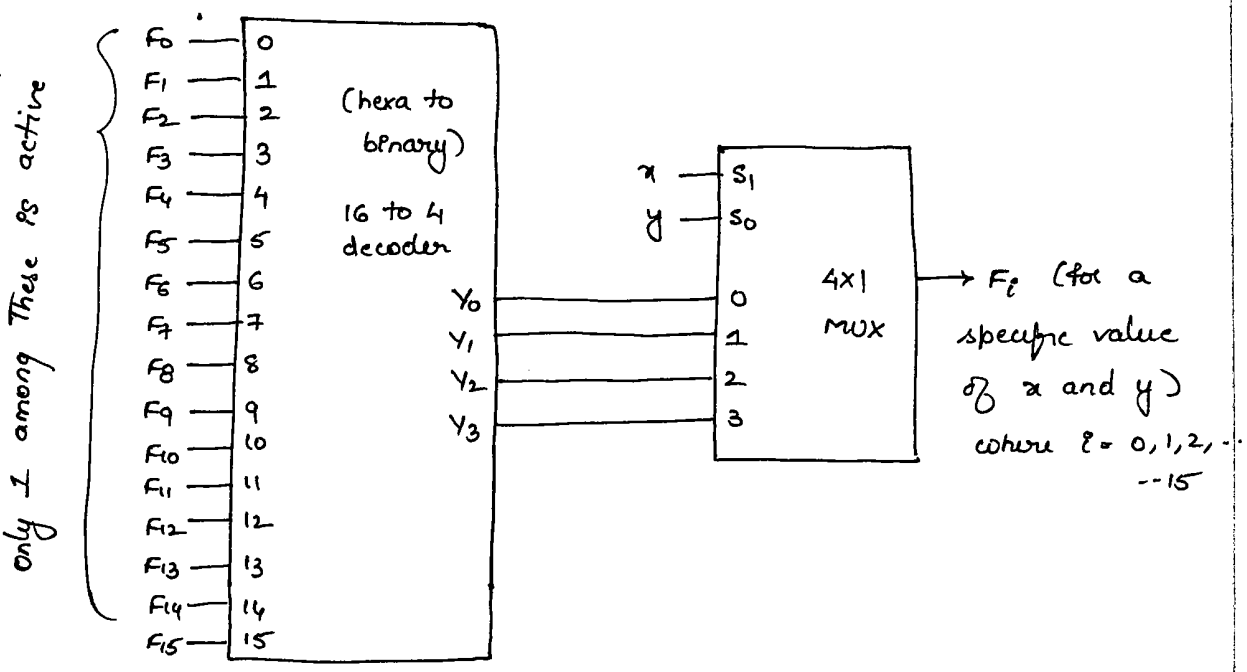
S_1	S_0	output	operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

- consists of 4 gates and a multiplexer.
- each of the four logic operations is generated through a gate that performs required logic
- o/p's of gates are connected to data i/p's of mux.
- S_1 and S_0 choose 1 data i/p and direct to o/p.

(P6) Derive a combinational circuit that selects and generates any of the 16 logic functions.

Sol:

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



→ Some applications of logic μ-operations

Logic μ-operations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to

1) change - bit values
2) delete group of bits
3) insert new bits values into a register

The following example shows how the bits of one register (designated by A) are manipulated by logic μ-operations as a function of bits of another register (designated by another register B)

In typical application,

register A = processor register
register B = logical operand extracted from memory

a) Selective Set

Sets to 1 the bits of register A, where there are corresponding 1's in register B i.e., if bit is 1 in B, o/p should be 1 and if B bit is 0, o/p should be A.

Eg:- 1010 A before
1100 B (logic operand)
1110 A after

⇒ This is OR μ-operation

B	A	o/p
0	0	0
0	1	1
1	0	1
1	1	1.

if B = 1, o/p = 1
if B = 0, o/p = A

Selective Set ⇒ OR μ-operation

b) Selective complement

complements bits of register A, where there are corresponding 1's in register B, i.e., if B bit is 1, o/p should be $\overline{A}$ and if B bit is 0 then o/p should be A.

Eg:- 1010 A before
1100 B (logic operand)
0110 A after

⇒ This is XOR μoperation

B	A	o/p
0	0	0
0	1	1
1	0	1
1	1	0

if B = 0, o/p = A
if B = 1, o/p = $\overline{A}$

Selective complement ⇒ XOR μ-operation

c) Selective clear

clears to 0 the bits of register A, where there are corresponding 1's in register B i.e if B bit is 1, o/p should be 0 and if B bit is 0, then o/p should be A.

Eg:- 1010 A before
1100 B (logic operand)
0010 A after

⇒ This is

$A \leftarrow A \wedge \overline{B}$

B	A	o/p
0	0	0
0	1	1
1	0	0
1	1	0

if B = 0, o/p = A
if B = 1, o/p = 0

d) Mask operation

Same as selective clear except for bits of A are cleared if corresponding bits of B are 0's.

Eg:- 1010 A before
1100 B (logic operand)
1000 A after masking

B	A	o/p
0	0	0
0	1	0
1	0	0
1	1	1

⇒ This is AND μ-operation

If B = 0, Then o/p = 0
If B = 1, Then o/p = A

Mask operation ⇒ AND μ-operation

Note: Mask operation is more convinent to use than selective clear operation because most computers provide an AND instruction and a few provide an instruction that execute selective clear.

e) 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.

Eg:- A = 0110 1010

Now replace the left most 4 bits by the value 1001

Step 1: Mask the 4 unwanted bits by ANDing

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

Step 2: Insert new value by ORing

0000 1010 A before
1001 0000 B (Insert)
1001 1010 A after insertion

Insert operation ⇒ OR μ-operation

f) clear operation

compares words in A and B and produces an all 0's result if the two nos. are equal. This is achieved using XOR operation.

Eg:- 1010 A before
1010 B
0000 A after

All 0's result is used to check whether the 2 nos. were equal

$A \leftarrow A \oplus B$

clear operation ⇒ XOR μ-operation

* Note: Logical operand B is selected based on the manipulating bit positions

Eg:- if we want to manipulate only bit positions 3 and 2
Logical operand = 1100

if we want to manipulate 2, 3 bit positions
Logical operand = 0110

⇒ operand B has 1 for bit positions which have to be manipulated. and 0 for bit positions which have to be left unaltered.

Pb. Register A holds the 8-bit binary 11011001. Determine the B operand and the logic μ-operation to be performed in order to change the value in A to:

a) 01101101 b) 11111101

Sol: a) A before = 1 1 0 1 1 0 0 1

A after = 0 1 1 0 1 1 0 1
7 6 5 4 3 2 1 0

at bit positions 7, 5, 4, 2, the bits are complemented

⇒ Selective complement operation

⇒ XOR μ-operation with logical operand B. 7 6 5 4 3 2 1 0
1 0 1 1 0 1 0 0

⇒ 1 1 0 1 1 0 0 1 A before
XOR ⊕ 1 0 1 1 0 1 0 0 B (logical operand)
0 1 1 0 1 1 0 1 — A after

b) A before = 7 6 5 4 3 2 1 0
1 1 0 1 1 0 0 1

A after = 1 1 1 1 1 1 0 1

at bit positions 7, 6, 5, 4, 3, 2, The bits are set.

⇒ selective set operation

⇒ OR μ-operation with logical operand B = 11111100

⇒
$$\begin{array}{ll} & 11011001 \quad - \text{A before} \\ \text{OR} & \underline{11111100} \quad - \text{B (logical operand)} \\ (\vee) & \underline{11111101} \quad - \text{A after} \end{array}$$

* Shift μ-operations

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

The contents of the register should be shifted left or right. At the same time, the bits are shifted, the 1st flip-flop receives its binary information from the serial input. The information transferred through the serial i/p determines the type of shift.

There are 3 types of shifts

a) Logical shift

Shifts 1 bit to right or left that transfers 0 through serial i/p.

Eg:- $R_1 \leftarrow$ shl R_1 (shift left) $R_1 = 11010$

$R_2 \leftarrow$ shr R_2 (shift left) $R_2 = 11010$

shl R_1 = 1010(0)←

shr R_2 = →(0)1101

Note:- Register symbol should be same on both sides of the arrow because source & destination are same.

b) Circular shift

Circulates the bits of register around the 2 ends without loss of information. This is accomplished by connecting the serial o/p of shift register to serial i/p.

Eg:- $R_1 \leftarrow$ cil R_1 (Circular left shift)

$R_2 \leftarrow$ cir R_2 (Circular right shift)

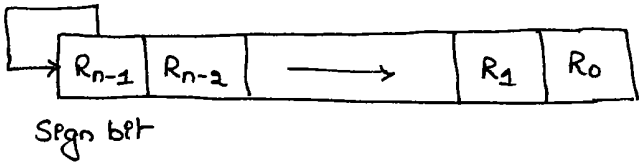
$R_1 = 11010 \Rightarrow$ cil $R_1 = 10101$

$R_2 = 11010 \Rightarrow$ cir $R_2 = 01101$

c) Arithmetic shift

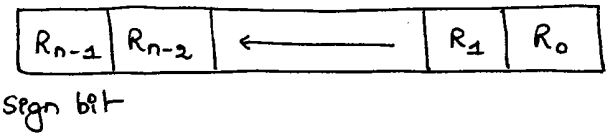
Arithmetic shift is a micro-operation that shifts a signed binary number to left or right.

Arithmetic right shift divides the no. by 2.



$R_1 \leftarrow$ ashr R_1

Arithmetic left shift multiplies the no. by 2



$R_2 \leftarrow$ ashl R_2

Note:- During arithmetic left shift sign bit changes if multiplication by 2 causes an overflow

Eg:- $R_1 =$ 0100 (+4) $\Rightarrow$ ashl $R_1 =$ 1000 (-ve)

Note:- Sign should not change because multiplication & division by 2 doesnot change sign.

Select	output				
S	H_0	H_1	H_2	H_3	
0	I_R	A_0	A_1	A_2	→ shift right (down)
1	A_1	A_2	A_3	I_L	→ shift left (up)

function table.

Note: A shifter with n-data i/p's and o/p's require n multiplexers. The two serial i/p's can be controlled by another multiplexer to provide the 3 possible types of shifts.

(Pb) In combinational shifter circuit, what is the o/p value of H i/p i/p $A = 1001$, $S = 1$, $I_R = 1$ and $I_L = 0$?

Sol: $S = 1 \Rightarrow$ shift left.

o/p = H_0 H_1 H_2 H_3

A_1 A_2 A_3 I_L

0 0 1 0

$A_0 A_1 A_2 A_3$

$A = 1001$

shl A = 0010.

(Pb) An 8-bit register contains the binary value 10011100. what is the register value after an arithmetic shift right? starting from the initial number 10011100, determine the register value after an arithmetic shift left, and state whether there is an overflow?

Sol: ashr 10011100 = 11001110

ashl 10011100 = 00111000 (here sign bit changed from 1 to 0 ⇒ overflow occurs

(Pb) starting from an initial value of R = 11011101, determine register value after an logical shift ~~left~~ left, and followed by a circular shift right, followed by a logical shift right and a circular shift left?

→ Hardware Implementation

- A possible choice for shift would be a bidirectional shift register with parallel load.

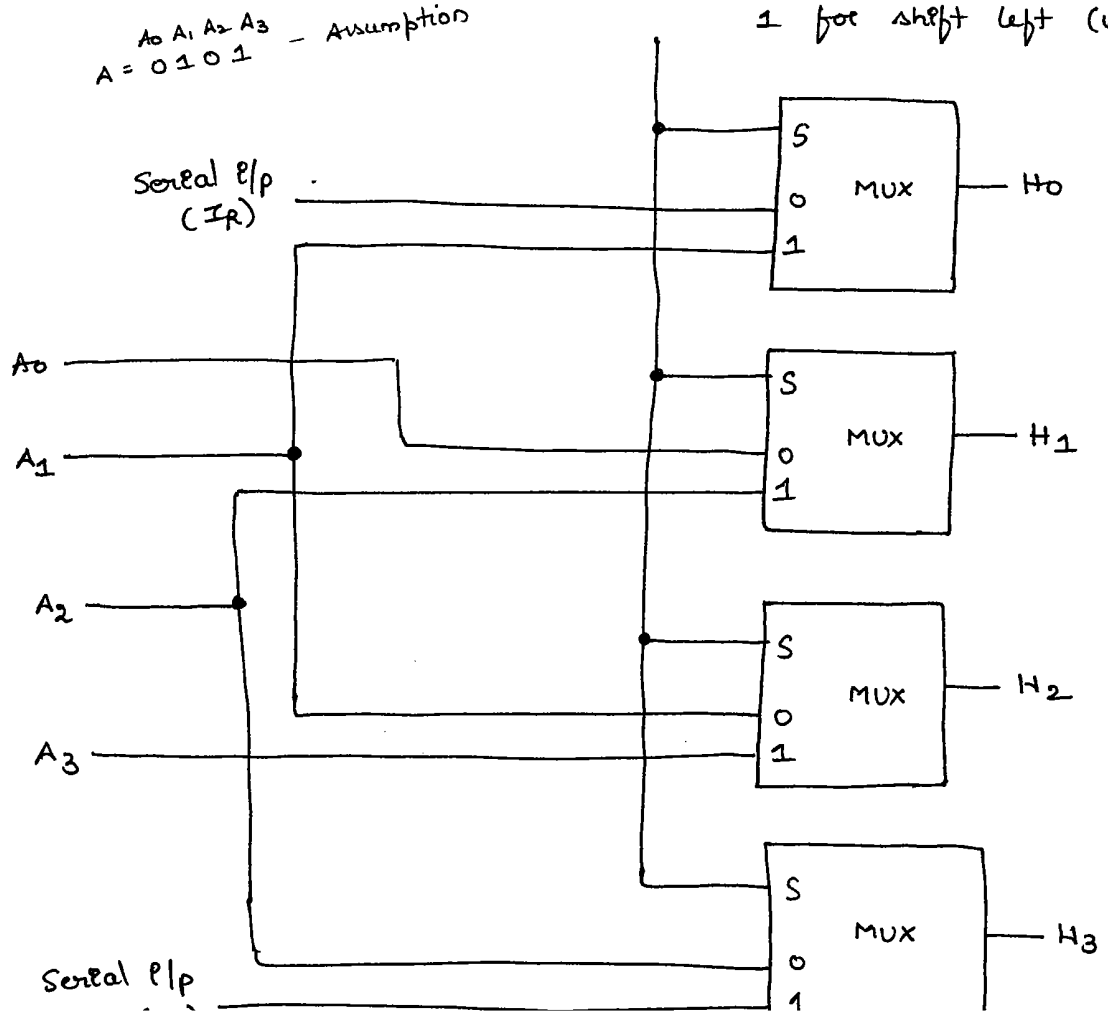
Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, another pulse is needed to initiate the shift.

- In a processor unit it is more efficient to implement the shift operation with a combinational circuit.

In this way, content of a register is first placed on a common bus whose o/p is connected to the combinational shifter and the shifted no. is then loaded back into the register. This requires only 1 clock pulse for loading the shifted value into the register.

- A combinational circuit shifter can be constructed with multiplexers as shown below:

Select 0 for shift right (down)
1 for shift left (up)



Sol R = 11011101

shl R = 10111010 ← 0

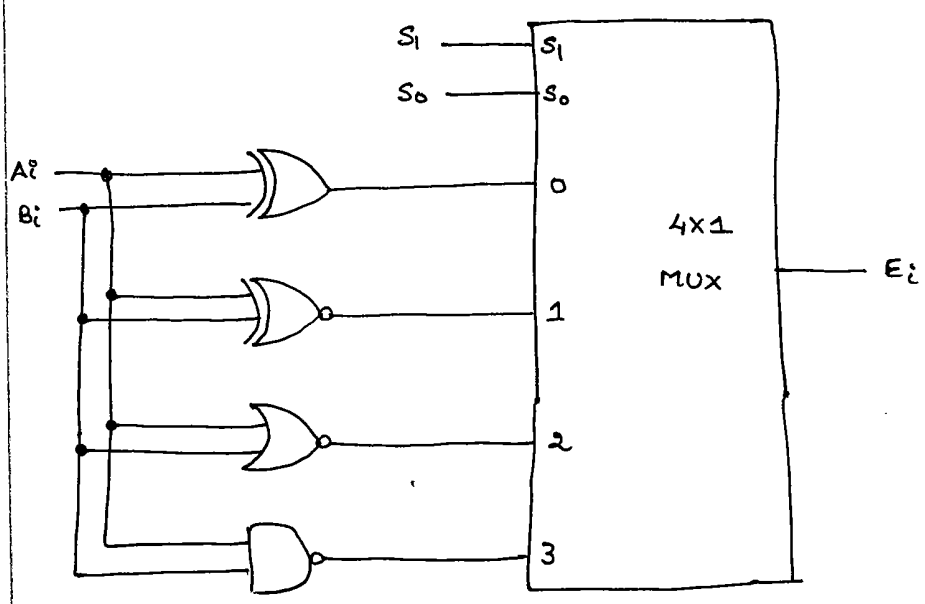
clr R = 01011101

shr R = 00101110

cil R = 01011100

(Pb) Design a circuit that performs the four logic operations of XOR, XNOR, NOR and NAND. Use 2 selection variables. Show the logic diagram for 1 stage.

Sol:



* Arithmetic logic shift Unit

→ Instead of having individual registers performing the μ-operations directly, computer systems employ a no. of storage registers connected to a common operational unit called arithmetic logic shift unit.

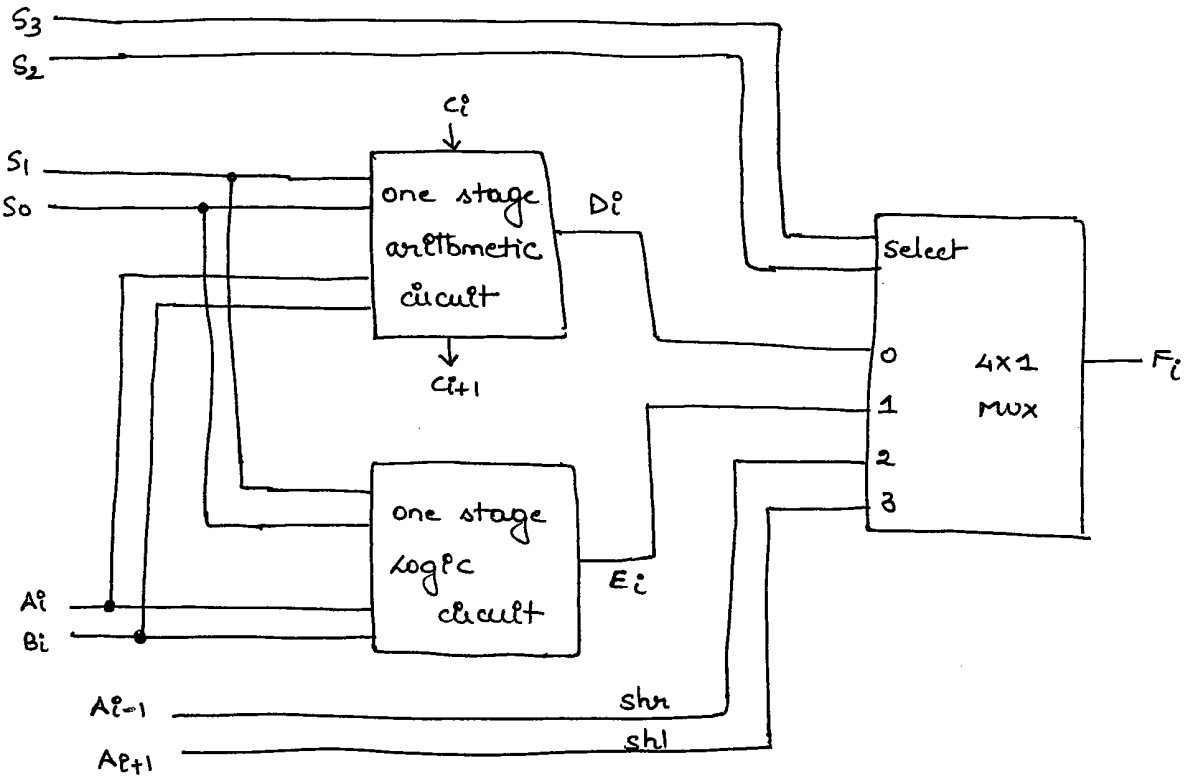
→ To perform a μ-operation, the contents of specified registers are placed in the i/p's of a common ALU. The ALU performs an operation and the result of this operation is then transferred to a destination register.

→ 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 1 clock pulse period.

→ Shift operations are often performed in a seperate unit, but sometimes shift unit is made part of the overall ALU.

→ Arithmetic, logic & shift circuits can be combined into 1 ALU with common selection variables.

one stage of an ALU is shown below

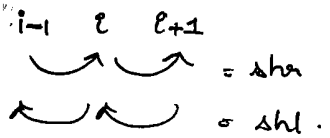


If $S_3S_2 = 00$ ⇒ arithmetic operation

If $S_3S_2 = 01$ ⇒ logical operation

If $S_3S_2 = 10$ ⇒ shr (shift right)

If $S_3S_2 = 11$ ⇒ shl (shift left)



function table for ALU

	operation select						
	S_3	S_2	S_1	S_0	C_{in}	operation	
Arithmetic μ-operation	0	0	0	0	0	$F = A + B$	Add
			0	0	1	$F = A + B + 1$	Add with carry
			0	1	0	$F = A + \overline{B}$	subtract with borrow
			0	1	1	$F = A + \overline{B} + 1$	subtract
			1	0	0	$F = A$	transfer A
			1	0	1	$F = A + 1$	Increment A
			1	1	0	$F = A - 1$	Decrement A
			1	1	1	$F = A$	transfer A
Logic μ-operation	0	1	0	0	X	$F = A \wedge B$	AND
			0	1	X	$F = A \vee B$	OR
			1	0	X	$F = A \oplus B$	XOR
			1	1	X	$F = \overline{A}$	complement A
shift μ-operation	1	0	x	x	x	F = shr A	shift right A into F
	1	1	x	x	x	F = shl A	shift left A into F.

Other Binary codes:

A **binary code** is a group of n bits that assume up to 2" distinct combinations of 1's and 0's with each combination representing one element of the set that is being coded.

- For example, a set of 4 elements can be coded by a 2-bit code with each element assignedone of the following bit combinations; 00, 01, 10, or 11.
- A set of 8 elements requires a 3-bit code,
- A set of 16 elements requires a 4-bit code, and so on.

A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2, one of such type of code is called ***binary-coded decimal*** *and is commonly referred to by its abbreviation* ***BCD****.*

The below table shows the BCD numbers

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

Alphanumeric Representation: Many applications of digital computers require the handling of data that consist not only of numbers, but also letters of the alphabet and certain special characters. An ***alphanumeric character*** set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, + , and = .

The standard alphanumeric binary code is the **ASCII (American Standard Code for Information Interchange)**, which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table shown below.

Some of the ASCII codes are shown in the below table.

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

Digital computers also employ other binary codes for special applications. A few additional binary codes encountered in digital computers are:

1.Gray code

2.Other decimal codes

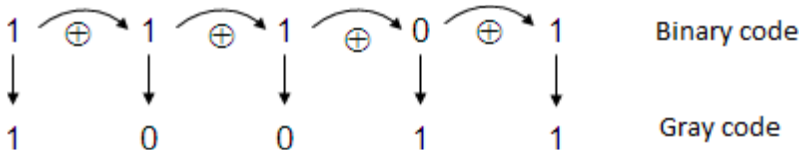
3.Other alphanumeric codes

1.**Gray code:** It is a kind of binary number system in which every successive pair of numbersdiffers in only one bit.

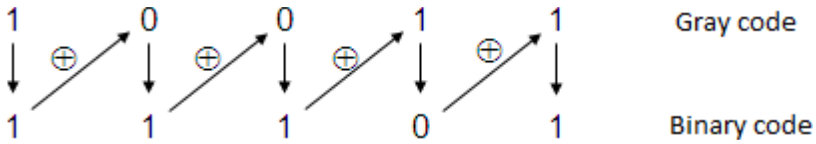
The advantage of the Gray code over straight binary numbers is that the Gray code changes byonly one bit as it sequences from one number to the next.

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

Binary code to Gray code:



Gray code to Binary code:



2.Other decimal codes:

i.Weighted code: The 2421 is an example of a weighted code. In a weighted code, the bitsare multiplied by the weights indicated and the sum of the weighted bits gives the decimaldigit.

EX: The bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of 2 x 1 + 4 x 1 + 2 x 0 + 1 + 1 = 7. The BCD code can be assigned the weights 8421 and for this reason it is sometimes called the 8421 code .

ii.excess-3: It is a decimal code that has been used in older computers . This is an unweightedcode . Its binary code assignment is obtained from the corresponding BCD equivalentbinary number after the addition of binary 3 (0011).

iii.Self-complementing code: The 9' s complement is easily obtained with the 2421 and theexcess-3 codes are self-complementing.

A self-complementing property means that *the 9' s complement of a decimal number, whenrepresented in one of these codes, is easily obtained by changing 1's to 0's and 0's to 1's.*
Four different binary codes for the decimal digit are shown in the below table

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

3.Other alphanumeric codes:

The ASCII code is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity. The code consists of 128 characters.

Another alphanumeric code used in IBM equipment is the **EBCDIC (Extended BCD Interchange Code).** It uses eight bits for each character (and a ninth bit for parity).EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.