DATA REPRESENTATION

ZATIN
COA 2nd Year

1

in digital Computers is stored in memory or processor

stors Contain either data or control info.

control Info: It is a bit or group of bits used to specify sequence of Command signals needed for manipulation of data in other registers.

* Registers are made up of flip-flops. Flip-flops are two state devices that can store only 1's & 0's.

Number System :-

1) - Binary Number System
- Decimal Number System
- Octal & Hexadecimal Number System.

Radix: or Base - (r) :-

r is a system that uses distinct symbols for r digits, from 0 to r-1.

Ex:

1) Decimal no system - radix = 10 -
Symbols are => 0, 1, 2, --- 9.

Ex: $728.79 = 7\times10^{2} + 2\times10^{1} + 8\times10^{0} + 7\times10^{-1} + 9\times10^{-2}$

2) for Binary no system radix = 2
Symbols are => 0 & 1.

3) for octal no system - radix = 8
Symbols are => 0, 1, 2, 3 ... 7.

4) for Hexadecimal no. system - radix = 16
Symbols are => 0, 1, 2, 3, .. 9, A, B, C, D, E, F

5) for r number system → radix = r
Symbols are => 0, 1, 2, --- (r-1).

Conversion :-

1) Any number system r to decimal →

$(\quad)_r \rightarrow (\quad)_{10}$

$\Rightarrow (A_1 A_2 A_3 \cdots A_n)_r = (A_1 \times r^{n-1} + A_2 \times r^{n-2} + \cdots + A_{n-1} \times r^1 + A_n \times r^0)_{10}$

$\Rightarrow (A_1 A_2 A_3 \cdots A_n \cdot B_1 B_2 \cdots B_m) \Rightarrow$

$(A_1 \times r^{n-1} + A_2 \times r^{n-2} + \cdots + A_n \times r^1 + A_n \times r^0 + B_1 \times r^{-1} + B_2 \times r^{-2} + \cdots B_m \times r^{-m})_{10}$

Ex:

(1) $(10110)_2 \rightarrow (\quad)_{10}$

$\Rightarrow 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0$

$= 16 + 0 + 4 + 2 + 0 = 22$

(2) $(127)_8 \rightarrow (\quad)_{10}$

$\Rightarrow 1\times8^2 + 2\times8^1 + 7\times8^0$

$= 64 + 16 + 7 = (87)_{10}$

(3) $(A12.B)_{16} \rightarrow (\quad)_{10}$

$\Rightarrow (10\times16^2 + 1\times16^1 + 2\times16^0 + 11\times16^{-1})_{10}$

(4) $(234)_5 \rightarrow (\quad)_{10}$

$\Rightarrow 2\times5^2 + 3\times5^1 + 4\times5^0$

$= 50 + 15 + 4 = (69)_{10}$

(2) Decimal to any number system $\xrightarrow{r}$

- First seprate the number into two parts - (1) Integer part (2) fractional part.
- Devide Integer part by r, it produce quotient & remainder, again devide quotient by r, it produce again new quotient & remainder. This process is repeated until quotient becomes 0. Take first remainder as last order bit.

- Multiply fraction part by r. we get integer & fraction. fractional part again multiplied by r. we get new integer & fraction.
This process repeated until we get zero fractional part or until no. of digits obtained gives required accuracy.
* first integer computed being the digit to be placed next to decimal point.
- finally two parts are combined.

$(41.205)_{10} \rightarrow (\quad)_4$

(3) Octal to Hexadecimal ⇒

$(\quad)_8 \rightarrow (\quad)_{16}$

$(\quad)_8 \rightarrow (\quad)_{10} \rightarrow (\quad)_{16}$

$(\quad)_8 \rightarrow (\quad)_1$

(4 bits) (4 bits) (4 bits) (4 bits)

Ex:

$(127.6)_8 \rightarrow (\quad)_{16}$

$(0000\ 0\ 1\ 0\ 1\ 1\ 1\ .\ 1\ 1\ 0\ 0)_8 \rightarrow (\ 0\ 5\ 7\ .\ C\ )_{16}$

$\Rightarrow (57.C)_{16}$

(4) Hexadecimal to Octal ⇒

$(\quad)_{16} \rightarrow (\quad)_8$

(3 bits) (3 bits)

(5) Any number system m to any number system n ⟶

$(\quad)_m \rightarrow (\quad)_{10} \rightarrow (\quad)_n$

⇒ BCD (Binary Coded Decimal Numbers) ⇒

Decimal Number	BCD Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
20	0010 0000

Complements ⇒

1) $(r-1)^{'}s$ Complement ⇒

$(r-1)^{'}s$ complement of N is defined as —

$$(r^n - 1) - N.$$

in this a number N in base r having n digits.

Ex:

(1) 9's complement of a number N:-

$(10-1)^{'}s$ = 9's Complement $= (10^n - 1) - N$

$r = 10$

(2) 1's complement of a binary number —

$(2-1)^{'}s$ = 1's Complement $= (2^n - 1) - N$

Ex: 1's complement of binary number 1001

$= (2^4 - 1) - (1001)_2$

$= (1111)_2 - (1001)_2 = (0110)_2$

or $1 \leftrightarrow 0$

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

(3) 4's complement of a number N – $(5^n - 1)_{10} - (N)_5$

(5-1)'s complement = 4's complement

Base of number N must be 5 ⇒

Ex) $(412)_5$, find 4's complement of this number

$(5^3 - 1)_{10} - (412)_5$

$(124)_{10} - (412)_5$

$(124)_{10} - (107)_{10} = (17)_{10}$

$= (032)_5$

Note ⇒

(r-1)'s complement of any number N of base r, subtract each digit from (r-1).

(B) r's complement ⇒

r's complement of an n digit number N in base r is defined as $(r^n - N)$ for $N \neq 0$

If N = 0 then r's complement is 0.

r's complement = (r-1)'s complement + 1

or

r's complement = $r^n - N$.

Ex:

(1) 10's complement of number 6789 is –

(a) first 9's complement of this number = 3210

(b) Now add +1 as – 3210 + 1 = 3211.

(2) 2's complement of binary number 10001 is –

FLOATING POINT REPRESENTATION :→

* Floating point describes a method of representing real numbers in a way that can support a wide range of values.

- Numbers are represented approx. to a fixed no of significant digits & scaled using an exponent.

then representation is –

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

⇒ Floating point refers to the fact that the radix point (decimal point) can float, it can be placed anywhere relative to significant digits of number.

→ fixed point or integer representation :→

Ex: 7 decimal digits with two decimal places –

12345.67, 123.45, 1.23, .12

→ Floating point format provide much wider range of values –

Ex: IEEE 754 decimal 32 format →

* Floating point representation requires more space for storage.

Ex:

$$152853.5047 = 1.52853\,5047 \times 10^5$$

Range of Floating point Numbers :→

Ex: $0.12 \times 0.12 = 0.0144$ can be expressed as –

$$(1.2 \times 10^{-1}) \times (1.2 \times 10^{-1}) = (1.44 \times 10^{-2})$$

→ floating point with 3 digit

With fixed point system with

$$0.120 \times 0.120 = 0.014$$

→ After decimal 3 digits

* range of floating point no's depends on the no. of bits or digits used for representation of the significand & for the exponent.

- for double precision (64-bit) binary floating point number has a coefficient of 53 bits, an exponent of 11 bits & one sign bit.

Range of exponent is $(-1022, 1023)$

& $\log_{10} 2^{1023} = 308$.

So range is $(10^{-308}$ to $10^{308})$.

IEEE 754: FLOATING POINT REPRESENTATION :-

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

ERROR DETECTION CODE :-

Error: Due to external noise bits can be changed from 1→0 or 0→1 through communication medium. This is called error.

- Error detection code only detect the presence of error
- If error occur in frequent way, then erroneous info. is transmitted again.

Parity Bit: Error detection code. It is an extra bit included with binary message to make total number of 1's either odd or even.

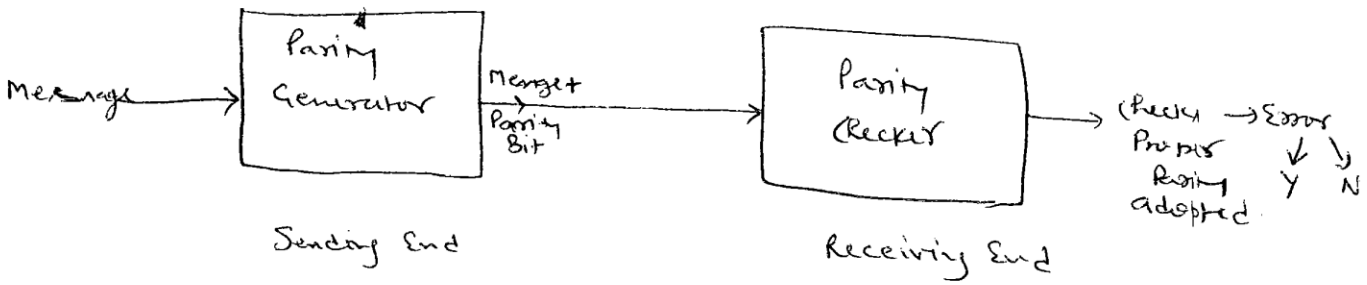
Two types :-

(1) ODD Parity Bit → Total no of 1's in message + Parity Bit should be Odd.

(2) EVEN Parity Bit → Total no of 1's in message + Parity bit should be even.

* EVEN Parity scheme has disadvantage of having a bit combination of all 0's
* ODD Parity scheme has disadvantage of having only one of the message bit equal to 1.

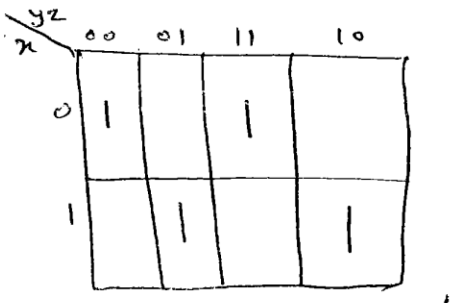
Parity Generator :-



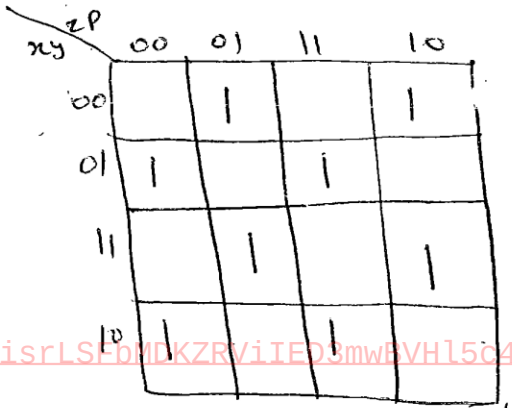
* This method detects presence of one, three or any odd number of errors.
* An even number of errors is not detected.

Error detection with odd parity bit →

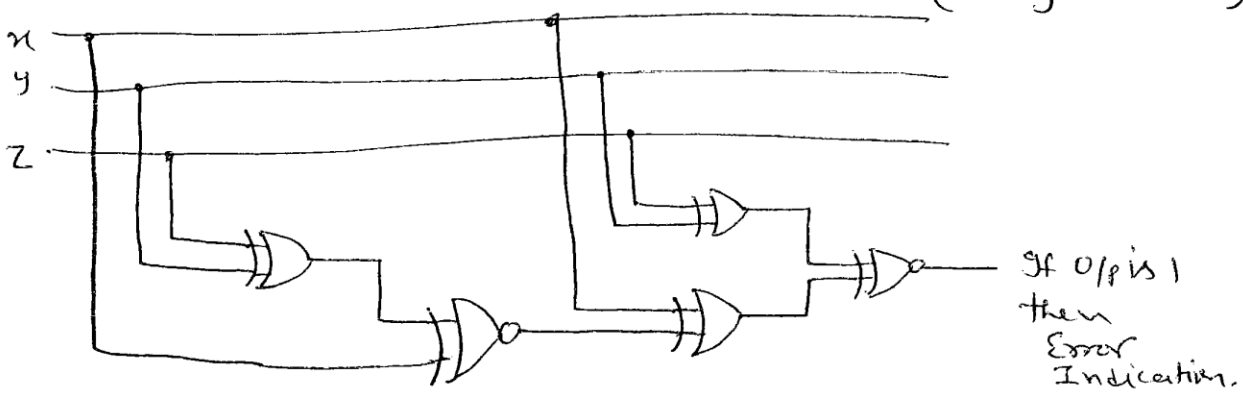
	Message	Odd parity bit	At Receiving End ODD Parity
0	000	1	0
1	001	0	0
2	010	0	0
3	011	1	0
4	100	0	0
5	101	1	0
6	110	1	0
7	111	0	0



$f(x,y,z) = (x \oplus y \oplus z)'$



$(x \oplus y \oplus z \oplus p)'$



Parity Generator

Error detection with even parity bit →

Message	Even Parity Bit
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

Hamming Code :- It can detect upto two & correct upto one bit errors.

* Simple Parity bit can not correct error.

Mathematically -

for each integer $r \geq 2$,

Total Block length $= n \leq 2^r - 1$

& Message length $k \leq 2^r - r - 1$.

r : No of Parity Bit.

Rate of Hamming code $= k/n = \frac{2^r - r - 1}{2^r - 1} = 1 - r/(2^r - 1)$,

which is highest possible for codes with distance 3 & block length $2^r - 1$.

Ex:- for $r = 4$,

Total Block length $= 15$ & Message length $= 11$.

$D_{14}\ D_{13}\ D_{12}\ D_{11}\ D_{10}\ D_9\ D_8\ \underline{D_7}\ D_6\ D_5\ D_4\ \underline{D_3}\ D_2\ \underline{D_1}\ \underline{D_0}$

Bit D_0 is responsible for checking errors for bit positions -
$D_1, D_3, D_5, D_7, D_9, D_{11}, D_{13}$

Bit D_1 is responsible for checking errors for bit positions -
$D_2, D_3, D_6, D_7, D_{10}, D_{11}, D_{14}, D_{15}$

Bit D_3 is responsible for checking errors for bit positions -
$D_4, D_5, D_6, D_7, D_{12}, D_{13}, D_{14}, D_{15}$

Bit D_7 is responsible for checking errors for bit positions -
$D_8, D_9, D_{10}, D_{11}, D_{12}, D_{13}, D_{14}, D_{15}$

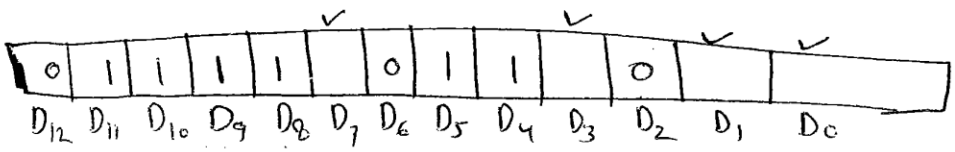
Ex: Generate Hamming code for 01111010

$9 \leq 2^r - r - 1$

$10 \leq 2^r - r$

Here $r = 4$.

Consider odd Parity –



$D_0 \Rightarrow$ $D_1, D_3, D_5, D_7, D_9, D_{11}$ (1, 1, 1) $\Rightarrow$ D_0 should be 0.

$D_1 \Rightarrow$ $D_2, D_3, D_6, D_7, D_{10}, D_{11}, D_{14}, D_{15}$ (0, 0, 1, 1) $\Rightarrow$ D_1 should be 1

$D_3 \Rightarrow$ $D_4, D_5, D_6, D_7, D_{12}, D_{13}, D_{14}, D_{15}$ (1, 1, 0, 0) $\Rightarrow$ D_3 should be 1

$D_7 \Rightarrow$ $D_8, D_9, D_{10}, D_{11}, D_{12}$ (1, 1, 1, 1, 0) $\Rightarrow$ D_7 should be 1

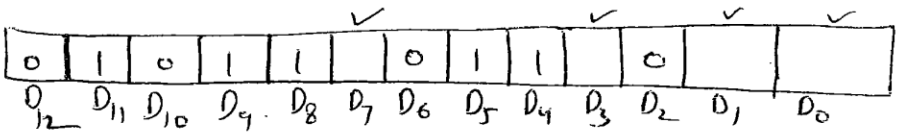
Now Message block –

0 1 1 1 1 1 0 1 1 1 0 1 0

Now error occurs & message becomes →

0 1 0 1 1 1 0 1 1 1 0 1 0

Again find value of Parity bits –



$D_0 \Rightarrow 0$

$D_1 \Rightarrow 0$

$D_3 \Rightarrow 1$

$D_7 \Rightarrow 0$

D_7	D_3	D_1	D_0
0	1	0	0

$$\begin{array}{r} 1110 \\ 0100 \\ \hline 1010 \end{array}$$

D_{10}

Now find X-OR between previous parity bits & newly calculated parity bits.

$$\begin{array}{r} 1110 \\ 0100 \\ \hline 1010 \end{array} \Rightarrow D_{10} \text{ is changed}$$

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

[redacted]

Subtraction →

Arithmetic oprn — Addition
— Subtraction
— Multiplication
— Division.

* Arithmetic processor is a part of processor unit that executes arithmetic operations.

* Negative no. can be represented as —
- first bit is 1 (from left).
- Rest of the no. may be represented in one of 3 possible ways —

(a) Signed Magnitude Representation
(b) Signed -1's complement representation
(c) Signed -2's complement representation.

Ex: ① +14 will be stored in 8 bit Register as —

00001110

② -14 will be stored in 8 bit register in one of 3 ways as —

In sign magnitude — 10001110

In Signed - 1's complement - 11110001

In Signed - 2's complement - 11110010

* For floating point operation most of computer uses signed - magnitude representation for the mantissa.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

Addition & Subtraction with signed-Magnitude Data:-

Magnitude of two numbers → A & B.

* When two equal no. are subtracted, result should be +0 ni-

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

Addition (Subtraction) Algo:

i) When signs of A & B are identical (different), add the two magnitudes & attach the sign of A to the result.

ii) When sign of A & B are different (identical), compare the magnitudes & subtract smaller number from the larger. Choose sign of result to be same as A if A>B or complement of the sign of A if A<B.

iii) If two magnitude are equal, subtract B from A & make the sign of result positive.

iv) Sign can be determined from an exclusive OR gate with As & Bs as inputs.

H/w Implementation:-

- A & B : Two Registers hold magnitude of numbers.
- As & Bs : Two flip flops hold corresponding sign.
- Results of opern may be transferred to third register.
- Saving is achieved if result is transferred into A & As.

A & As : Together form accumulator register.

⇒ H/w :

(1) A parallel adder is needed to perform microoperation A+B.

(2) A comparator circuit is needed to establish if A>B, A=B or A<B.

(3) Two parallel - subtractor circuits are needed to perform microopern A-B & B-A.

(4) Sign relationship can be determined from an X-OR gate with As & Bs as inputs.

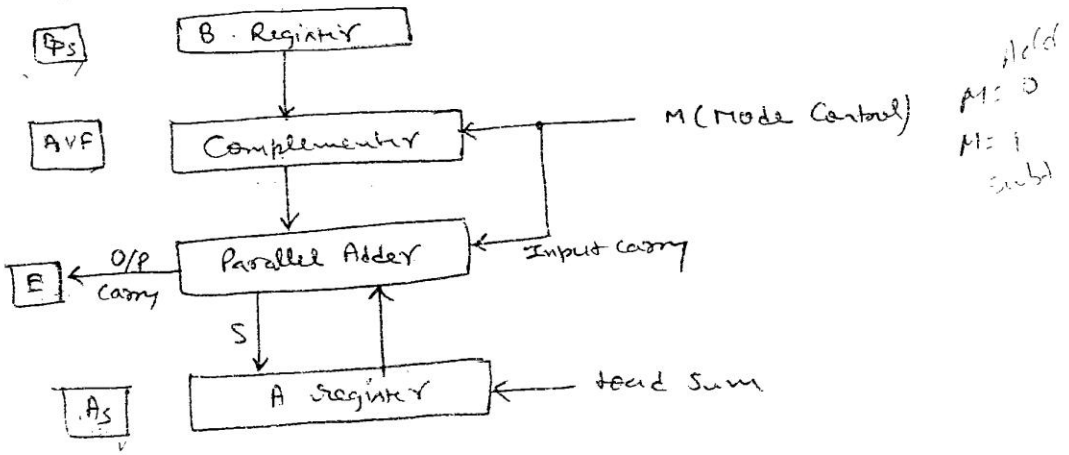
Different Procedure —

(1) Subtraction can be accomplished by means of complement & add.

(2) Result of comparision can be determined from the end carry after the subtraction.

⇒ It requires only an adder & complementer.

H/w —



It consist –

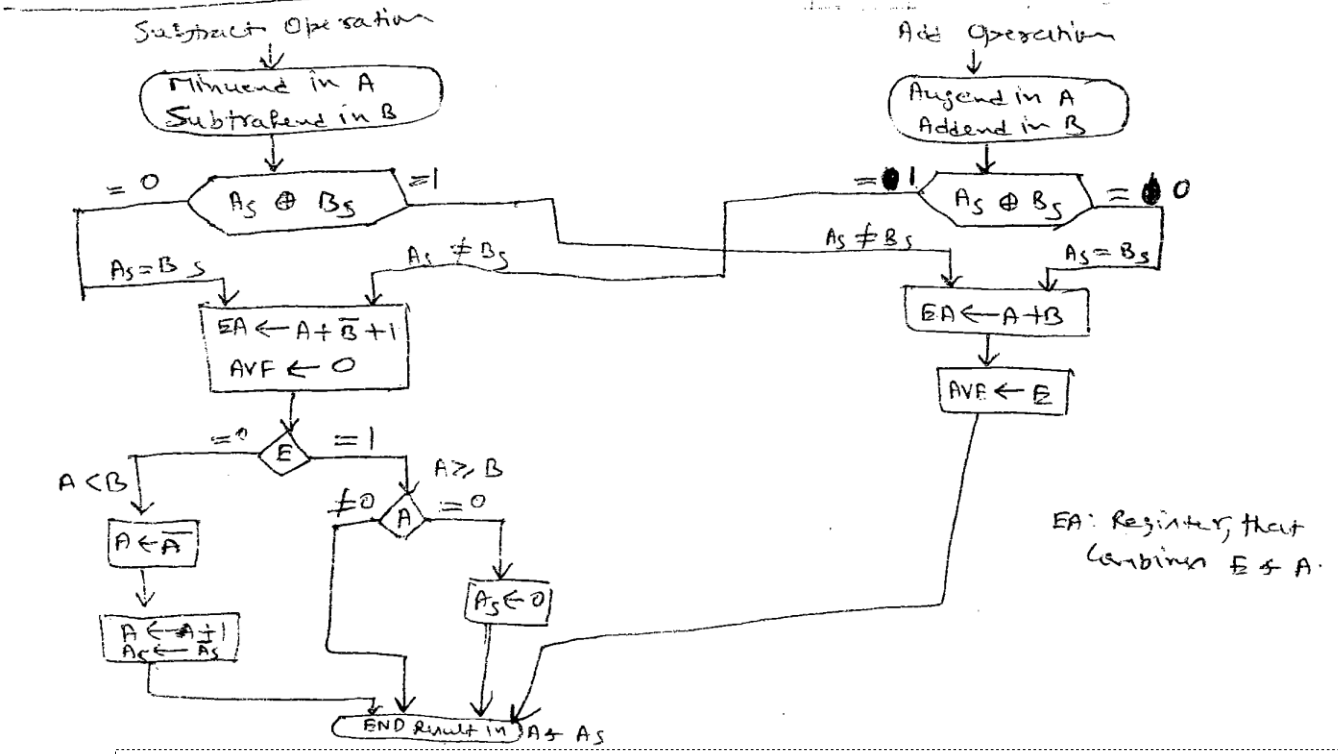
- Register A & B, & Sign flipflop A_S & B_S.
- Subtraction is done by adding A to 2's complement of B.
- O/P carry is transferred to flip flop E. (It determine relative magnitude of two numbers).
- AVF : Add overflow flip flop ∴ It holds the overflow bit when A & B are added.
- Sum(S), O/P of adder is applied to the input of register A.

* Complementer consist of Ex-OR gates.
* Parallel adder consist of full adder circuits.
* When M = 0, O/P of B is transferred to adder, input carry is 0, O/P of adder is equal to sum A+B.
* When M=1, 1's complement of B is applied to adder, input carry is 1 & O/P S = A + $\bar{B}$ + 1. (A + 2's complement of B) i.e (A−B).

M = 0 : for Addition

M = 1 : for Subtraction.

H.W Algorithm (flow chart) ++



Addition & Subtraction with Signed 2's Complement data —

* A division of two no in Signed-2's complement form consists of adding the numbers with sign bits treated the same as other bits of the number.
- A carry out of sign bit position is discarded.

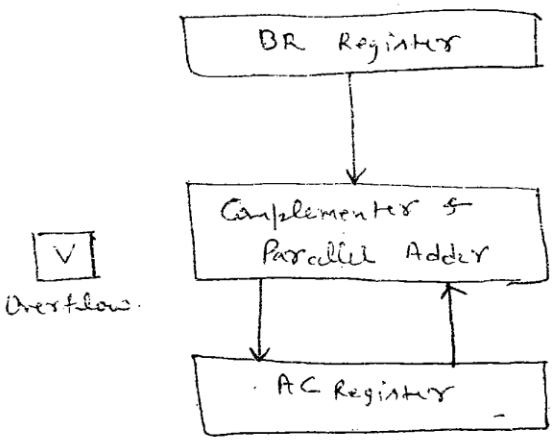
- Subtraction consists of first taking 2's complement of subtrahend then adding it to minuend.

* If two nos of n digits each are added & sum occupies (n+1) digits, we say an overflow occurred.

* An overflow can be detected by inspecting last two carries out of the addition.

* When two carries are applied to an X-OR gate, overflow is detected when O/P of gate is equal to 1.

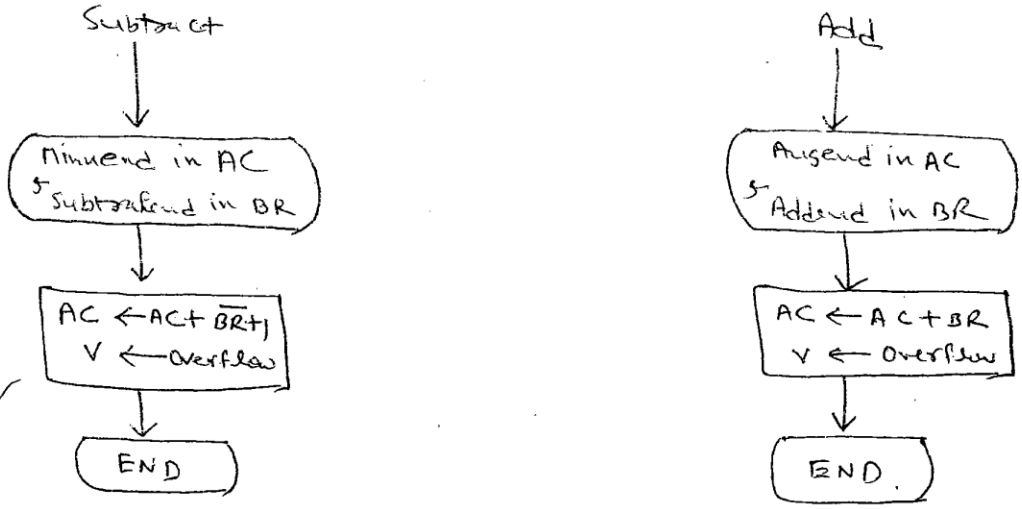
H/w3 —



A Register → AC (Accumulator)
B Register → BR
} leftmost bits in AC & BR represent sign bits of numbers.

→ Overflow flip-flop (V) is set to 1. (If there is an overflow).
→ O/P carry in this case is discarded.

Flow chart –



V : 1 when X-OR of last two carries is 1. Otherwise it is 0.

* An overflow must be checked during this opⁿ bcoz two nos added could have same sign.

* If an overflow occurs, then will be an erroneous result in AC register.

MULTIPLICATION ALGORITHM:-

Ex:

10111	Multiplicand
X 10011	Multiplier
10111	
10111 X	
00000 XX	
00000 XXX	
10111	
110111001	Product.

* Successive Shift & Add operation.

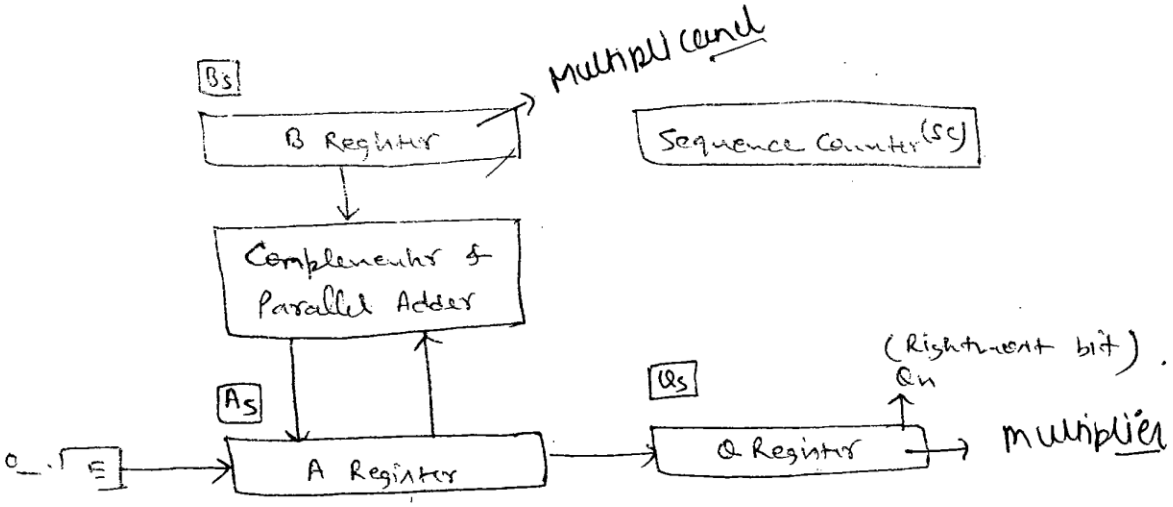
* If Multiplier is 1 then same multiplicand is copied otherwise 0's are copied.

* Sign of product depends on signs of multiplicand & multiplier.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

H/w Implementation :-



Steps :-

(1) Instead of providing registers to store & add simultaneously as many binary numbers as there are bits in multiplier, & it is convenient to provide an adder for the summation of only two binary nos & successively accumulate partial products in register.

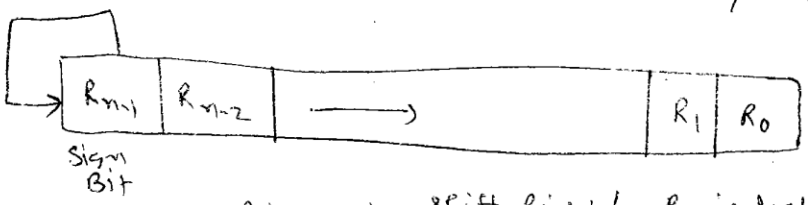
(2) Instead of shifting multiplicand to the left, partial product is shifted to the right, which results in leaving partial product & multiplicand in the required relative positions.

(3) When corresponding bit of multiplier is 0, then it is no need to add all zeros to the partial product.

In H/w —

- Multiplier is stored in Register Q & sign is stored in Q_s.
- Sequence counter (SC) is initially set to a number equal to number of bits in multiplier.
- Counter is decremented by 1 after forming each partial product.
- When content of counter reaches to zero, product is formed & process stops.
- Initially Multiplicand is in Register B & multiplier is in Register Q.

- Arithmetic shift left multiplies a binary number by 2.
- Arithmetic shift Right divides a number by 2.



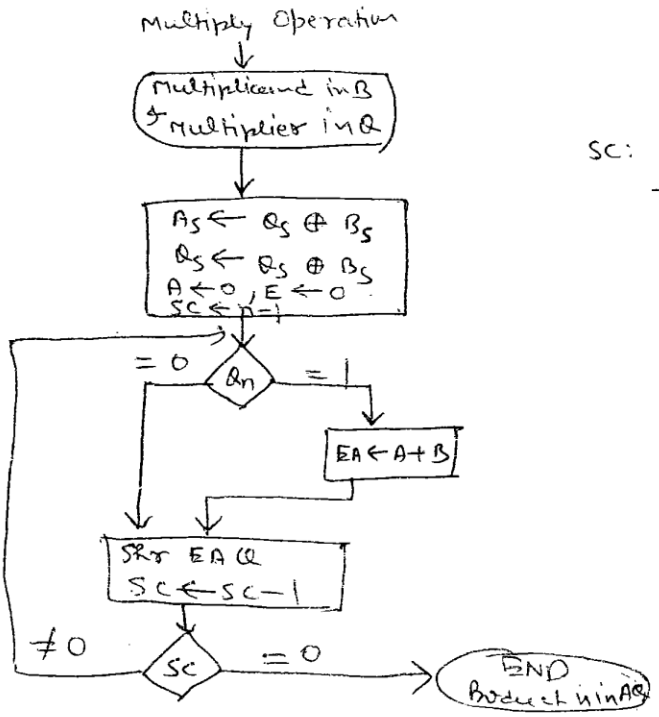
Arithmetic shift Right (R_0 is lost).

* Sum of A & B forms a partial product which is transferred to EA register.
- Both partial product & multiplier are shifted to right.
- This shift will be denoted by statement SHR EAQ to designate the right shift.

* L.S.B. of A is shifted into M.S.B. of Q.
* Bit from E is shifted into most significant position of A, & C is shifted into E.

* Right most bit in Register Q, (designated by Q_n), will hold the bit of multiplier, which must be inspected next, because after the shift, one bit of partial product is shifted into Q, pushing multiplier bits one position to right.

H/w Algorithm



SC: Initially Set to number equal to number of bits of multiplier.

Ex:

$$\begin{array}{r} 1011 \quad B \\ \times\ 101 \quad Q \\ \hline \end{array}$$

$A \leftarrow 0$ $E \leftarrow 0$ $SC \leftarrow 2$

$EA \leftarrow 1010$ $Q: 101$, $Q_n = 1$

SR EAQ: 0101 010 $SC \leftarrow 1$

Now one bit is shifted in Q (partial product)

0010 101 $Q_n = 0$

SR EAQ 0001 010 $Q_n = 1$ $SC \leftarrow 0$

$EA \leftarrow A+B$ 1100 010

SR EAQ 0110 001, $SC \leftarrow 0$

$B: 1011$

$Q: 101$

$B \leftarrow 1011$

$Q \leftarrow 101$

Ex:

$A \leftarrow 0$, $E \leftarrow 0$, $SC \leftarrow 2$

(i) $Q_n = 1 \rightarrow$

$EA \leftarrow 1011$

SR EAQ: 1011 101

0101 110 $SC \leftarrow 2$

$$\begin{array}{r} 1011 \\ 0010 \\ \hline 1101 \end{array}$$

(ii) $Q_n = 0$

SR EAQ: 0010 111 $SC \leftarrow 1$

(iii) $Q_n = 1$

$EA \leftarrow 1101$

SR EAQ: EA 1101, Q 111

0110 111 $SC \leftarrow 0$

$32\ 16\ 8$ $4\ 2\ 1$

Another Example:— Initially multiplier will be in Q.

Multiplicand B = 10111	E	A	Q	SC
	0	00000	10011	101
— Qn = 1, add A+B	0	10111	10011	101
Shr EAQ & SC ← SC−1	0	01011	11001	100
— Qn = 1, add B+B	1	00010	11001	100
Shr EAQ & SC ← SC−1	0	10001	01100	011
— Qn = 0, Shr EAQ & SC ← SC−1	0	01000	10110	010
— Qn = 0, ~~add A+B~~, Shr EAQ & SC ← SC−1	0	00100	01011	001
— Qn = 1, add A+B	0	11011	01011	001
Shr EAQ & SC ← SC−1	0	01101	10101	000
Final result is =>	0	1101	0101	

(B = 16 + 4 + 2 + 1 = 23; Q = 16 + 2 + 1 = 19)

10111
01011
100010

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

BOOTH MULTIPLICATION ALGORITHM

- It gives a procedure for multiplying binary integer in signed 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting.

String of 1's in the multiplier from bit weight 2^K to weight 2^M can be treated as $2^{K+1} - 2^M$.

Ex: Binary no. 01110 (+14) has a string of 1's from 2^3 to 2^1 Here

$K=3, M=1$ then number can be represented as

$2^{K+1} - 2^M = 2^4 - 2^1 = 16 - 2 = 14$.

then multiplication $M \times 14$ M: Multiplicand
14: Multiplier.

$M \times 14 : M \times 2^4 - M \times 2^1$.

* So product can be obtained by shifting multiplicand four times left & subtracting M after shifting left once.

* Booth Algo requires -
  (1) Examination of multiplier bits.
  (2) Shifting of partial product.

Rules →

(1) Multiplicand is subtracted from partial product upon encountering the first least significant 1 in a string of 1's in multiplier.
(2) Multiplicand is added to partial product upon encountering first 0 (provided that there was a previous 1) in a string of 0's in multiplier.
(3) Partial product does not change when multiplier bit is identical to previous multiplier bit.

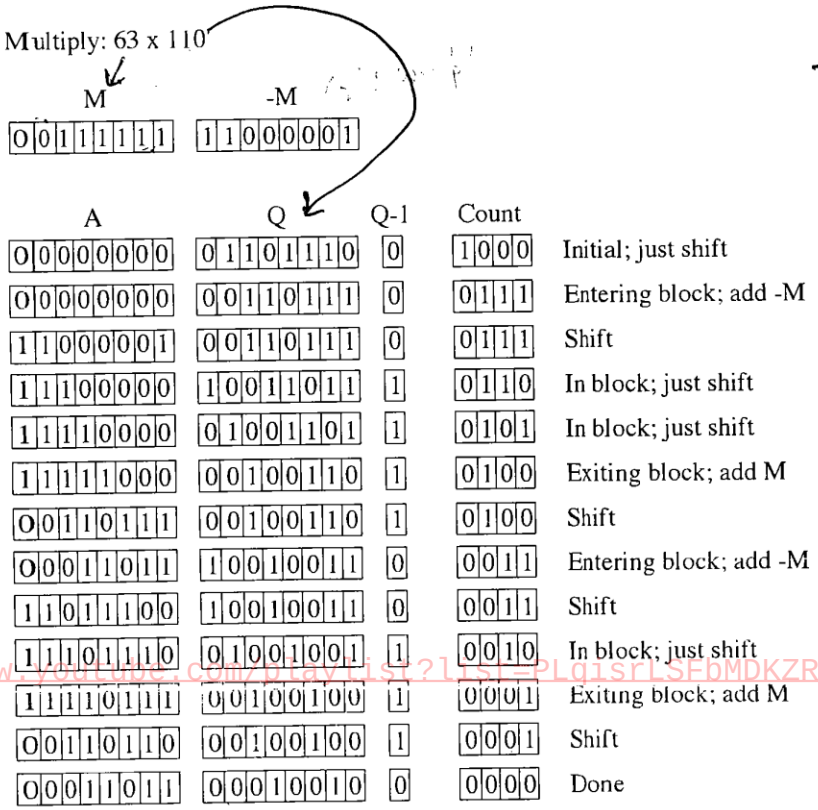


Figure 11.7: Using Booth's algorithm to multiply $63_{10} \times 110_{10}$.

Next, we see that we are about to enter a block of 1s, since bit 0 of Q is 1 and Q-1 is 0. We subtract M from A by adding $-M$, and then shift. The next two steps just shift everything, since we are within the 1s block. However, the following step requires us to add M to A, since we are leaving the block of 1s. We then shift. Immediately, we discover that we are about to enter another block, and so we add $-M$ to A and shift. The next two steps require us to shift, then add M and shift. At this point, the Count register is 0, signaling that we are done. The result is contained in the combined A/Q registers: 0001101100010010, which is 6930_{10}, the product of 63×110.

To see how this works when one of the numbers is negative, Figure 11.8 shows the algorithm computing -63×110.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

Multiply: -63 x 110

M	-M
11000001	00111111

A	Q	Q-1	Count	
00000000	01101110	0	1000	Initial; just shift
00000000	00110111	0	0111	Entering block; add -M
00111111	00110111	0	0111	Shift
00011111	10011011	1	0110	In block; just shift
00001111	11001101	1	0101	In block; just shift
00000111	11100110	1	0100	Exiting block; add M
11000100	11100110	1	0100	Shift
11100100	01100111	0	0011	Entering block; add -M
00100011	01100111	0	0011	Shift
00010001	10111001	1	0010	In block; just shift
00001000	11011001	1	0001	Exiting block; add M
11001001	11011000	1	0001	Shift
11100100	11011100	0	0000	Done

Figure 11.8: Using Booth's algorithm to multiply $-63_{10} \times 110_{10}$.

11.4 Conclusion

11.5 Further Reading

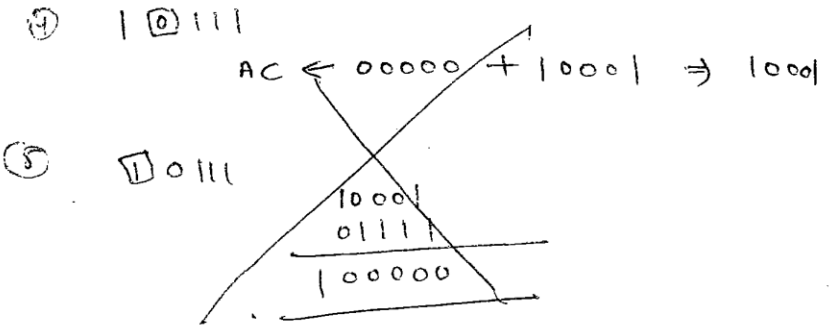
Sources:

11.6 Exercises

1. Give the two's complement representation of the following numbers (use 8 bits):
   (a) 9
   (b) -15
   (c) 68
   (d) -79
   (e) -152

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L



* This algo works for +ve or -ve multipliers in 2's complement representation.

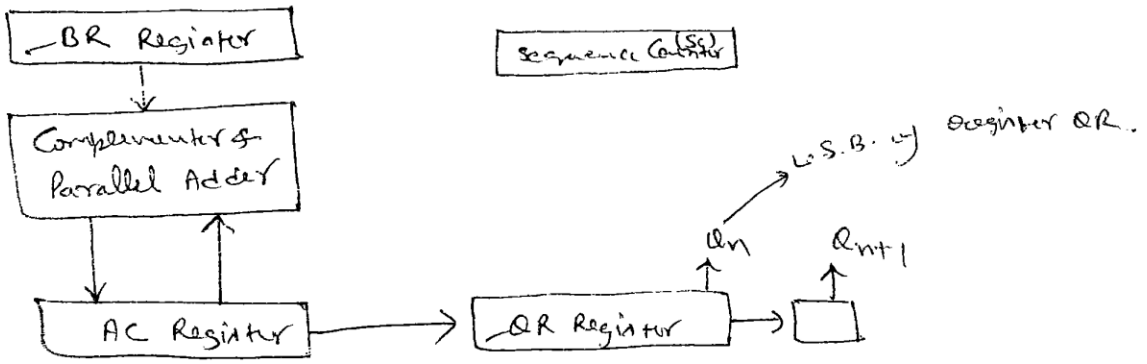
* $-14 \Rightarrow -2^4 + 2^2 - 2^1 = -14$

Last operation is for subtraction.

Acc to rules for last opn subtraction, string of multiplier ends with 1. So Algo works for -ve number also.

How for Booth Algo →

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L



* Sign bits are not separated from rest of registers.

BR: Multiplicand

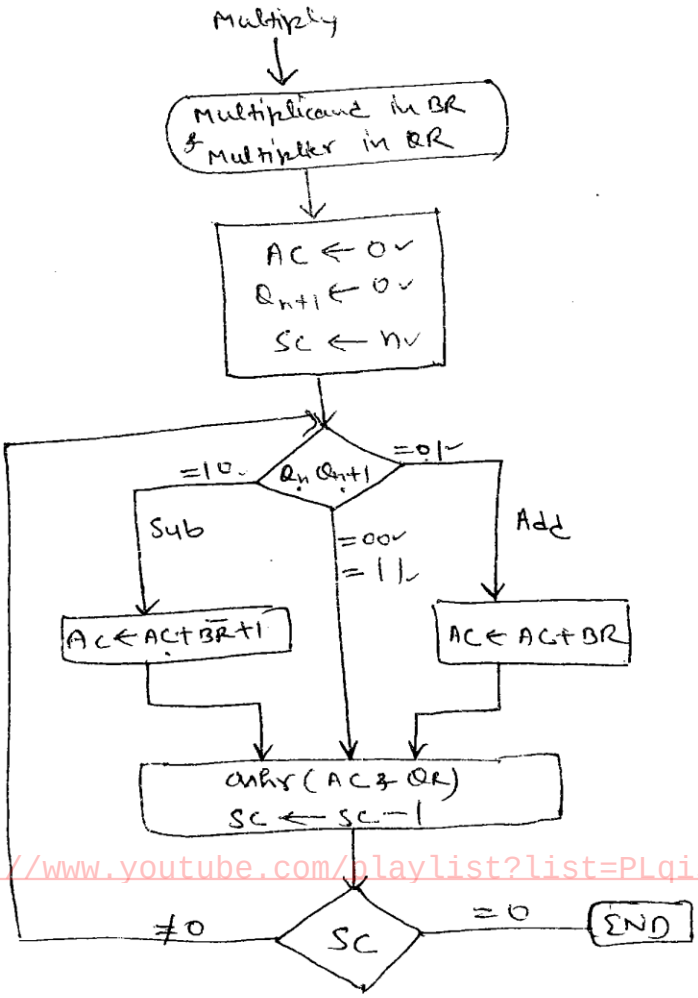
QR: Multiplier

Q_{n+1}: Flipflop: It facilitate double bit inspection of multiplier.

Initially – AC: 0

Q_{n+1}: 0

SC: N



* Overflow can not occur bcoz addition & subtraction of multiplicand follow each other.

ashr (AC & QR):- Arithmetic shift right operation which shifts AC & QR to the right & leaves the sign bit in AC unchanged.

* Final value of Q_{n+1} is original sign bit of multiplier.

Ex: Multiplicand : 10111 (2's comp of −9)
Multiplier : 10011 (2's comp of −13).

Q_n	Q_{n+1}	BR = 10111	AC	QR	Q_{n+1}	SC	
1	0	Subtract BR from AC	00000	10011	0	101	00000 + 01001 = 01001
			01001				
		ashr AC, QR & SC ← SC − 1	00100	11001	1	100	
1	1	ashr & SC ← SC − 1	00010	01100	1	011	
0	1	add BR to AC	11001				11 + 00010 + 10111 = 11001
		ashr & SC ← SC − 1	11100	10110	0	010	

Q_n	Q_{n+1}	BR	AC	QR	Q_{n+1}	SC
0	0	ashr & SC←SC−1	11110	01011	0	001
1	0	Subtract BR from AC	00111			
		ashr & SC←SC−1	00011	10101	1	000

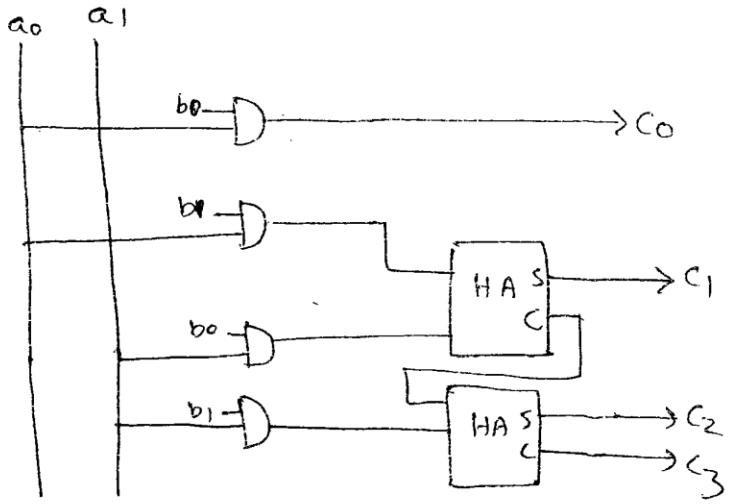
00011101015
64 32 16 8 4 2 1 ⇒ 117

Array Multiplier

* Multiplication can be done with one micro-operation by means of a combinational circuit that forms product bits all at once.
- It requires large no. of gates so it is not economical until development of IC.

Ex:- 2 bit by 2 bit array multiplier –

b_1 b_0
a_1 a_0
a_0b_1 a_0b_0
a_1b_1 a_1b_0
C_3 C_2 C_1 C_0



* If we have more than two bits in partial product for addition, Full adders are used to produce sum.

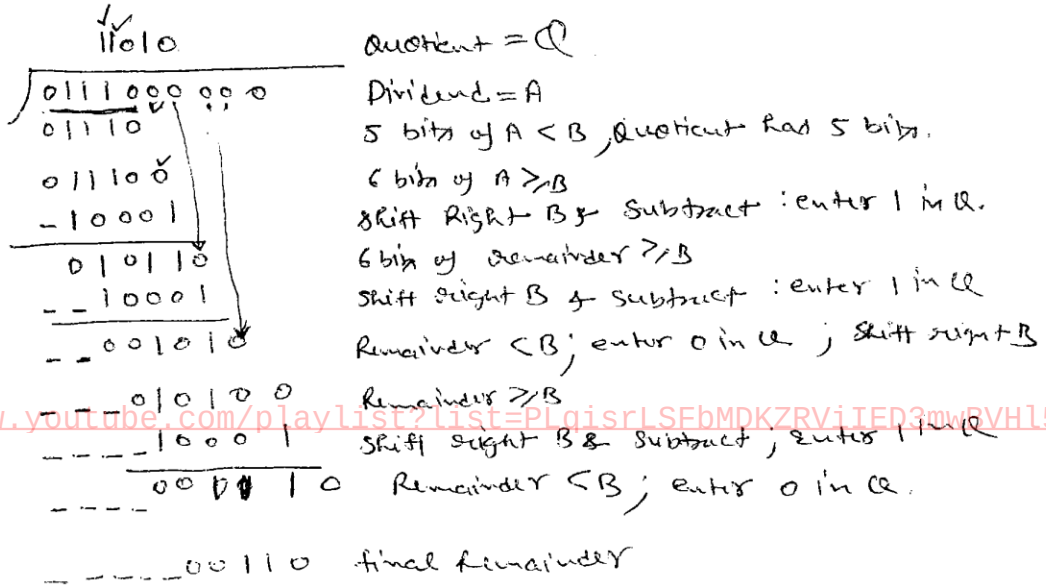
Logic :- A bit of multiplier is ANDed with each bit of multiplicand in as many levels as there are bits in multiplier.

- Binary o/p in each level of AND gates is added in parallel with partial product of previous level to form a new partial product.
- Least level produces product.
- For j multiplier bits & K multiplicand bits we need jXK AND gates & (j-1) K-bit adders to produce a product of j+K bits.

DIVISION ALGORITHMS:-

Ex:

Divisor
B = 10001

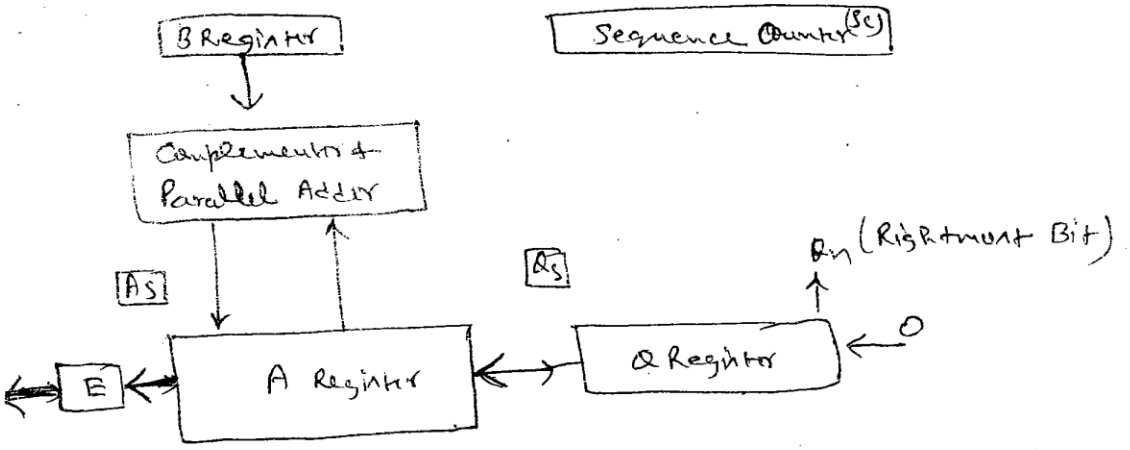


- Partial Remainder / Division could have stopped here to obtain a quotient of 1
- If partial remainder ≥ Divisor, quotient bit is equal to 1.
- Divisor is then shifted right & subtracted from partial remainder.
- If partial remainder is less than divisor, quotient bit is 0 & no subtraction is needed.

H/W Implementation:

- Instead of shifting divisor to the right, dividend or partial remainder, is shifted to left.
- Subtraction is achieved by adding A to 2's complement of B.

Bs:



* Register EAQ is now shifted to the left with 0 inserted into Qn & previous value of E lost.

~~Example~~

Divisor B = 10001,

Double length dividend is stored in A & Q.

* Dividend is shifted to the left & divisor is subtracted by adding its 2's complement value.

* E : Contain info abt relative magnitude.

If E = 1, it means A ≥ B.

then quotient bit 1 is inserted into Qn.

& partial remainder is shifted to left.

If E = 0, it means A < B.

then Qn remains 0.

- Value of B is then added to restore partial remainder in A to its previous value.
- Partial remainder is shifted to left & process is repeated again until all five quotient bits are formed.

* finally quotient is in Q & final remainder will be in A.

Ex:

Divisor B = 10001 $\quad \overline{B}+1 = 01111$

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

Neglect E

Remainder in A: 00110

Quotient in Q: 11010

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

DIVIDE OVERFLOW:

* Division operation may result in a quotient with an overflow.
* In given example, quotient will consist of 6 bits if five most significant bits of dividend constitute a number greater than divisor.
* Overflow bit will require one more flip-flop for storing the sixth bit.
- When dividend is twice as long as the divisor, condition for overflow can be stated as — A divide overflow condition occurs if the high order half bits of dividend constitute a number greater than or equal to divisor.
- Division by zero problem must be avoided.
  * This occurs b'coz any dividend will be greater than or equal to a divisor which is equal to zero.

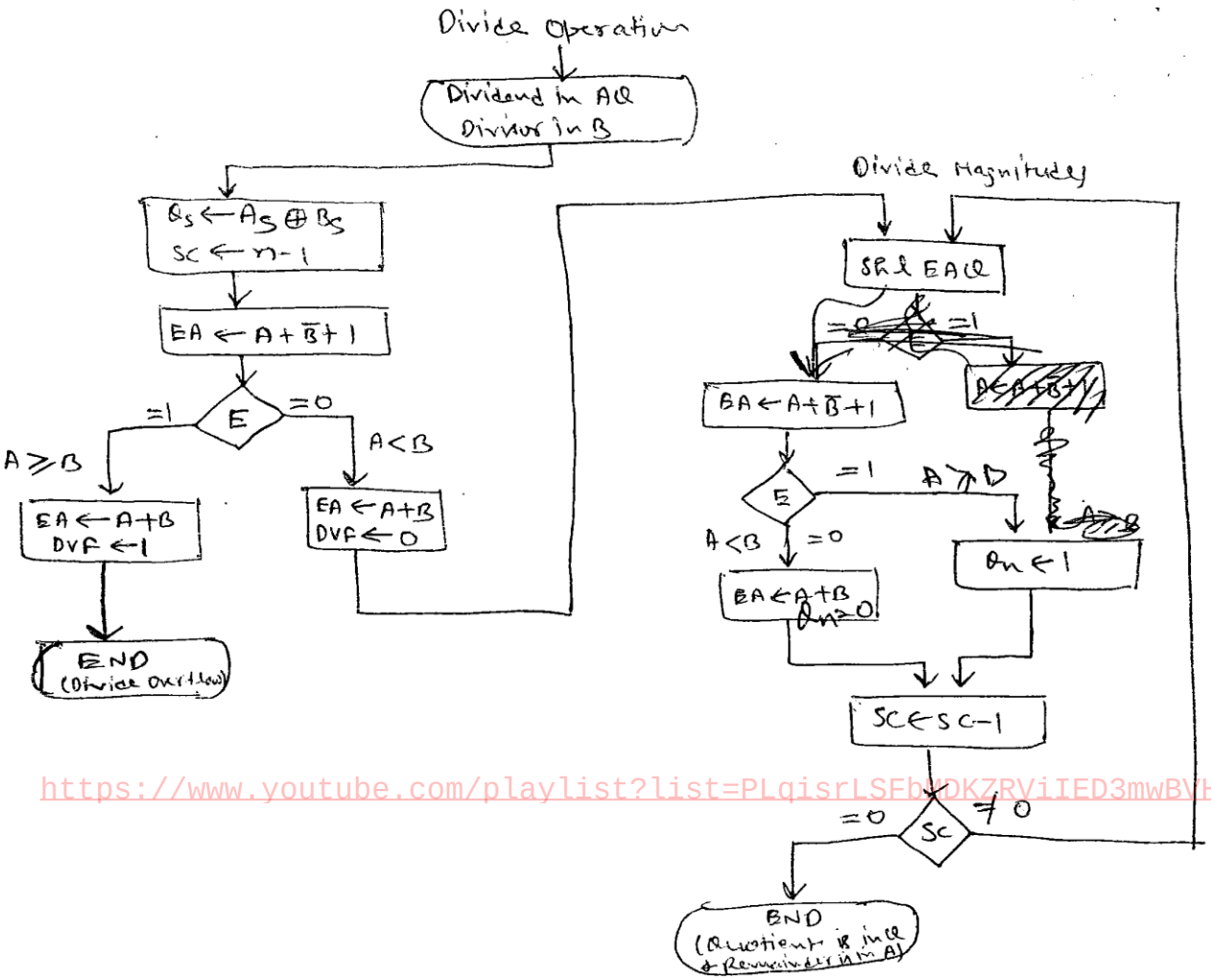
DVF (Divide-Overflow flip-flop): This is set when overflow condition is detected.

Provision for handling divide overflow →

1) In some computers, it is responsibility of programmers to check if DVF is set after each divide instruction.
2) In older computers, occurrence of divide overflow stopped the computer & this condition was referred to as divide stop.
3) To provide an interrupt request when DVF is set. Interrupt causes the computer to suspend the current program & branch to a service routine to take a corrective measure. (To ensure the program & type an error message explaining the reason why program could not be completed).
4) To use floating point data.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

H/w Algorithm



PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

FLOATING POINT ARITHMETIC OPERATIONS

Basic :- Floating point number can be represented as -

$m \times r^e$ m: Mantissa May be a fraction or integer

Ex: $537.25 = 0.53725 \times 10^3 = e$

(M)

* A floating point number is normalized if most significant digit of mantissa is non zero.
* Floating point representation increases range of numbers that can be accommodated in a given register.

Ex: 48-bit words:

Range of fixed point integer - $\pm(2^{47}-1)$ ≈ $\pm 10^{14}$

for floating point representation -

36 Bits Mantissa + 12 Bits for exponent

Range of numbers -

$$\pm (1 - 2^{-35}) \times 2^{2047}$$

Addition & Subtraction: -

- Adding or subtracting two numbers requires first an alignment of radix point since exponent parts must be made equal before adding or subtracting mantissa.

Ex:

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

- Shifting to the left : loss of Most significant digits. → It causes an error.
- Shifting to the right : loss of least significant digits. (Preferable)
  ↳ It only reduce Accuracy

* Usual Alligment procedure - To shift the mantissa that has smaller exponent to the right by a number of places equal to difference b/w exponents.

$$.5372400 \times 10^2$$
$$.0001580 \times 10^2$$

* Overflow: When two normalized mantissas are added, sum may contain an overflow digit.
- An overflow can be corrected by shifting sum once to the right & incrementing exponent.

- When two numbers are subtracted, result may contain most significant zeros as -

$$\begin{array}{r} .56780 \times 10^5 \\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

Underflow: - It occurs in most significant position of mantissa.

Solution: Shift the mantissa to the left & decrement the exponent until a nonzero digit appears in first position.

- A normalized procedure is performed after each operation to ensure that all results are in normalized form.

- Floating point multiplication & division do not require alignment of ~~mantissas~~ exponent.

* Operation performed with mantissa are same as in fixed point numbers. (Same registers & circuits).

* Operation performed with exponents are compare, & increment (for aligning mantissa), add & subtract (for multiplication & division) & decrement (to normalize the result).

* Exponent may be represented as -
- Sign magnitude
- Signed - 2's complement
- Signed - 1's complement.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

https://www.youtube.com/playlist?list=PLqisrLSFbMDKZRViIED3mwBVHl5c4LG-L

Register Configuration :-

- Same registers & adder are used for fixed point arithmetic are used for processing the mantissa.

- There are 3 Registers — BR, AC & QR.

* Mantissa part has same uppercase letter symbols as in fixed point representation.

* Exponent part uses correspondig lowercase letter symbol.

Biased Exponent :

* Bias is a +ve number that is added to each exponent as floating point number is formed, so that internally all exponents are positive.

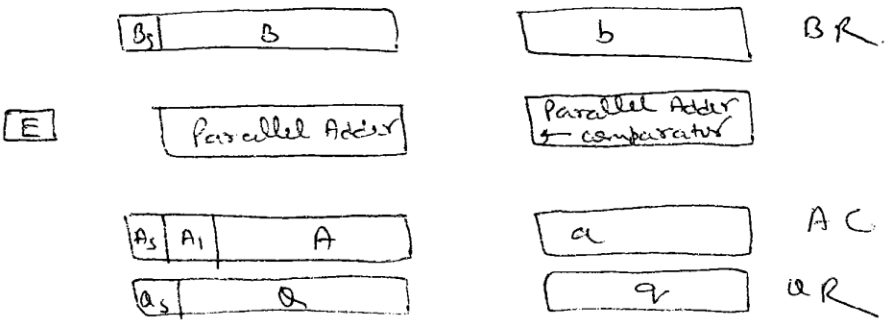
Ex: Exponent has range −50 to 49.

Add bias of 50, then exponent register contains number e+50. e: actual exponent.

then exponents are represented in registers as positive numbers in the range of 00 to 99.

Adv : - They contain only +ve numbers.
- Simpler to compare their relative magnitudes w/o being concerned with their signs.

* So each floating point number has a mantissa in signed magnitude representation & a biased exponent.



AC: Has mantissa, sign in A_S & Magnitude in A.

- Exponent is stored in a register.

A_1: most significant bit of A. (for normalized number, this bit position must be a 1).

BR: Subdivided into B_S, B & b.

QR: subdivided into Q_S, Q & q.

* Parallel adder adds the two mantissas & transfer the sum into A & carry into E.
* A separate parallel adder is used for exponents.
* Floating point numbers are so large that the chance of an exponent overflows is very less & for this reason, exponent overflow will be neglected.

PREPARED BY: MR. PIYUSH GUPTA, MR. ZATIN GUPTA

Arithmetic Logic Unit (ALU) Design

An arithmetic logic unit (ALU) represents the fundamental building block of the central processing unit of a computer.

An ALU is a digital circuit used to perform arithmetic and logic operations.

What Is an ALU?

An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations.

It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers.

A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data, and the ALU stores the result in an output register.

The control unit moves the data between these registers, the ALU, and memory.

How an ALU Works

An ALU performs basic arithmetic and logic operations. Examples of arithmetic operations are addition, subtraction, multiplication, and division. Examples of logic operations are comparisons of values such as NOT, AND, and OR.

All information in a computer is stored and manipulated in the form of binary numbers, i.e. 0 / 1.

Transistor switches are used to manipulate binary numbers since there are only two possible states of a switch: open or closed. An open transistor, through which there is no current, represents a 0. A closed transistor, through which there is a current, represents a 1.

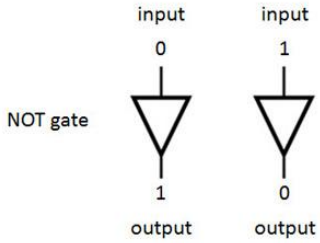
Operations can be accomplished by connecting multiple transistors. One transistor can be used to control a second one - in effect, turning the transistor switch on or off depending on the state of the second transistor.

This is referred to as a gate because the arrangement can be used to allow or stop a current.
The simplest type of operation is a NOT gate.

This uses only a single transistor. It uses a single input and produces a single output, which is always the opposite of the input.

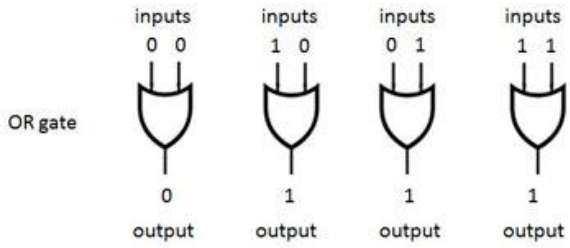
Mr. Zatin Gupta (Assistant Professor)
Department of Computer Science & Engineering,
KIET Group of Institutions, Ghaziabad, UP
Zatin.gupta@kiet.edu, Zatin.gupta2000@gmail.com

This figure shows the logic of the NOT gate:

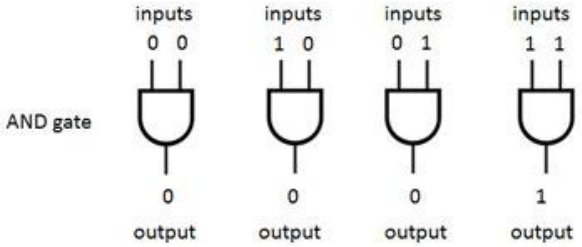


Other gates consist of multiple transistors and use two inputs.

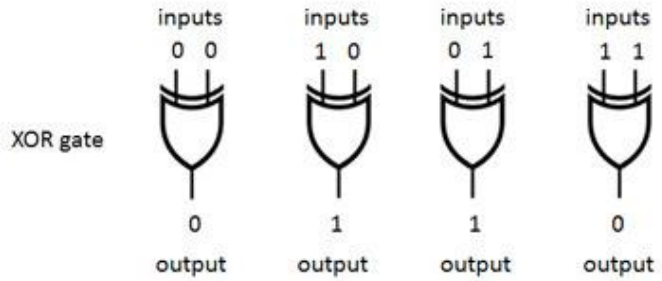
The OR gate results in a 1 if either the first or the second input is a 1. The OR gate only results in a 0 if both inputs are 0. This figure shows the logic of the OR gate:



The AND gate results in a 1 only if both the first and second input are 1s. This figure shows the logic of the AND gate:



The XOR gate, also pronounced X-OR gate, results in a 0 if both the inputs are 0 or if both are 1. Otherwise, the result is a 1. This figure shows the logic of the XOR gate:

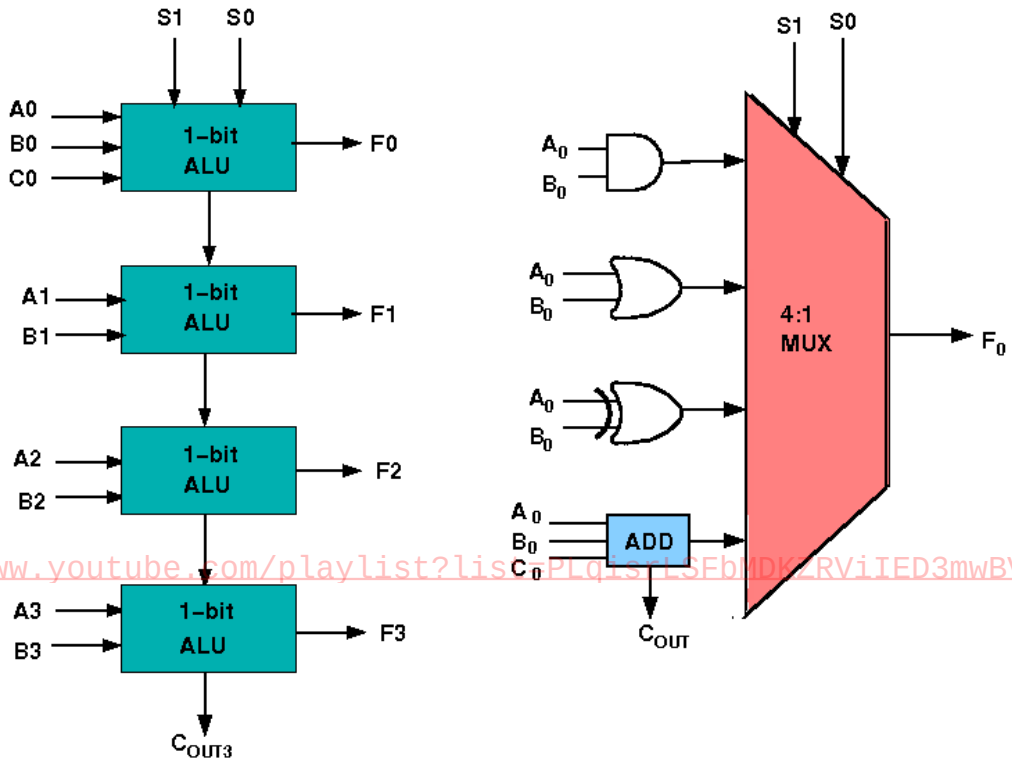


Mr. Zatin Gupta (Assistant Professor)
Department of Computer Science & Engineering,
KIET Group of Institutions, Ghaziabad, UP
Zatin.gupta@kiet.edu, Zatin.gupta2000@gmail.com

Design of ALU:

ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction, division, multiplication and logical operations like and, or, xor, nand, nor etc.

A simple block diagram of a 4-bit ALU for operations and, or, xor and Add is shown here:



The 4-bit ALU block is combined using 4 1-bit ALU blocks

Design Issues:

The circuit functionality of a 1-bit ALU is shown here, depending upon the control signal S_1 and S_0 the circuit operates as follows:

for Control signal **S_1 = 0**, S_0 = 0, the output is **A And B**,

for Control signal **S_1 = 0**, S_0 = 1, the output is **A Or B**,

for Control signal **S_1 = 1**, S_0 = 0, the output is **A Xor B**,

for Control signal **S_1 = 1**, S_0 = 1, the output is **A Add B**.

Mr. Zatin Gupta (Assistant Professor)
Department of Computer Science & Engineering,
KIET Group of Institutions, Ghaziabad, UP
Zatin.gupta@kiet.edu, Zatin.gupta2000@gmail.com