Lab-1: Familiarizing with the syntax, data types and operator of VHDL.

Description:

* Syntax:
  - Statements end with a semicolon (;),
  - Keywords are reserved words, and are written in lower case (e.g. entity, architecture, process)
  - Identifier are names given to signals, entities, architectures, etc. and can consist of letters, number, and underscore.
  - Comments start with "--" and extend to the end of the line.
  - Entities and architectures define the structure and behaviour of the hardware.
  - Signals represent wires in the hardware.

* Data types:

  VHDL provides several data types to model different types of hardware elements. Some common data types include:
  - std_logic: Represents single bits, can take values '0', '1', 'Z' (high impedance), 'W' (weak), 'L' (weak pull-down), 'H' (Weak pull up), or '-' (undefined)
  - std_logic_vector: Represents arrays of bits.
  - integer: Represents integer number.
  - boolean: Represents Boolean values (true/false)
  - real: Represent Real numbers.
  - time: Represent time intervals.

* Operators

↳ VHDL supports various operator for arithmetic, logical and relational operations. Some commonly used operator include:

- Arithmetic operators: +, -, *, /, **
- Relational operators: and, or, not, nand, nor, xor, xnor.
- Shift operators: SLL (Logical Left Shift), SRL (Logical Right Shift), SLA (Arithmetic left Shift), SRA (Arithmetic Right Shift)

LAB 2 : Design of Basic Logic Gates using VHDL.

Aim: To design Logic Gates using VHDL.

Description :

A logic gate performs a logical operation on one or more logic inputs and produces a single logic output. The logic normally performed is Boolean logic and found in digital circuit. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are descripted below with aid of truth tables.

AND Gate

Input		Output
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

NOR Gate

Input		Output
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

OR Gate

Input		Output
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Ex NOR Gate

Input		Output
A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Ex OR Gate

Input		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

NAND Gate

Input		Output
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

NOT Gate

Input	Output
A	F
0	1
1	0

Program:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity all_gates is
Port (a: in STD_LOGIC;
      b: in STD_LOGIC;
      c: out STD_LOGIC;
      c1: out STD_LOGIC;
      c2: out STD_LOGIC;
      c3: out STD_LOGIC;
      c4: out STD_LOGIC;
      c5: out STD_LOGIC;
      c6: out STD_LOGIC;
 );
 end all_gates;
 architecture Behavioral of all_gates is
 begin
  c <= a and b;
  c1 <= a or b;
  c2 <= a nand b;
  c3 <= a nor b;
  c4 <= a xor b;
  c5 <= a xnor b;
  c6 <= not b;
 end Behavioral;
```

Lab 3: Design of Half Adder and Full Adder using VHDL.

Aim: To design of Half Adder and Full Adder using VHDL.

Description:

Half Adder:

$Sum = A'B + AB' = A \oplus B$

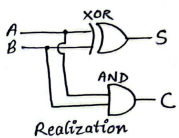
$Carry = AB$

The truth table, schematic representation and XOR//AND realization of a half adder are shown in figure below:-

Input		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth Tables

Schematic



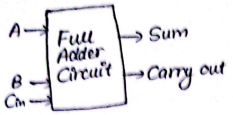
Realization

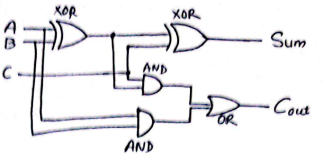
Full Adder:

$Sum = A \oplus B \oplus C_{in}$

$Carry = AB + BC_{in} + C_{in}A$

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





Program:

* Half Adder

```
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
port (a, b: in std_logic;
s, c: out std_logic);
end half_adder;
architecture dataflow_half of half_adder is
begin
s <= a xor b;
c <= a and b;
end dataflow_half;
```

* Full Adder

1) VHDL code for full adder data flow:

```
library ieee;
use ieee.std_logic_1164.all;
entity fa is
Port (a: in std_logic;
      b: in std_logic;
      cin: in std_logic;
      s: out std_logic;
      cout: out std_logic);
end fa;
architecture Behavioral of fa is:
begin
s <= (a xor b) xor cin;
cout <= (a and b) or (b and cin) or (a and cin);
end Behavioral;
```

2) VHDL code for full adder behavioral :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity fa1 is
port (a, b, ci : in std_logic;
s, co : out std_logic
);
end fa1;
architecture Behavioral of fa1 is
begin
process (a, b, ci)
begin
s<= a xor b xor ci;
co <= (a and b) or (b and ci) or (ci and a);
end process;
end Behavioral;
```

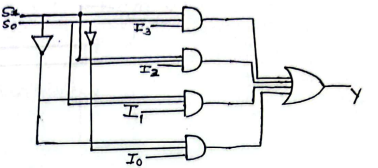
Lab 4: Design of Multiplexer and De-multiplexer using VHDL.

Aim: To design of multiplexer and demultiplexer using VHDL.

Theory:

Multiplexer:

Selection Lines		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

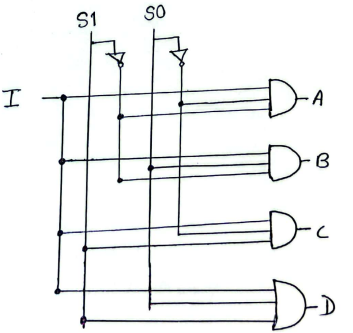


From Truth Table, we can directly write the Boolean function for output, Y as -

$$Y = S_1' S_0' I_0 + S_1' S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$$

Demultiplexer:-

I	S1	S0	ABCD
0	X	X	0000
1	0	0	1000
1	0	1	0100
1	1	0	0010
1	1	1	0001



Program for MUX:

```
library ieee;
use ieee.std_logic_1164.all;
entity Multiplexer is
port (a, b, c, d : in std_logic;
s: in std_logic vector (1 downto 0);
y: out std (logic);
end Multiplexer;
architecture beh of Multiplexer is
begin
process (a, b, c, d, s)
begin
case s is
when "00" => y <= a;
when "01" => y <= b;
when "10" => y <= c;
when "11" => y <= d;
when others => y <= 'U';
end case;
end process;
end beh;
```

Program for DEMUX:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity demultiplexer is
Port (i: in STD_LOGIC;
      s0: in STD_LOGIC;
      s1 : in STD_LOGIC;
      a : out STD_LOGIC;
      b : out STD_LOGIC;
      c : out STD_LOGIC;
      d : out STD_LOGIC;)
end demultiplexer;
```

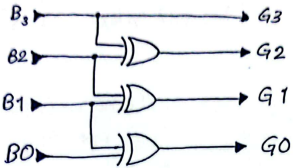
```
architecture Behavioral of demultiplexer is
signal p,q : STD_LOGIC;
begin
 p<= not S0;
 q<= not S1;
 a<= i and p and q;
 b<= i and q and S0;
 c<= i and p and S1;
 d<= i and S1 and S0;
 end Behavioral;
 ..........
 library IEEE;
 use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.STD_LOGIC_ARITH.ALL;
 use IEEE.STD_LOGIC_UNSIGNED.ALL;
 entity DEMUX is
 Port (I: in STD_LOGIC;
       S: in STD_LOGIC_VECTOR (1 downto 0);
       Y: out STD_LOGIC_VECTOR (3 downto 0));
 end DEMUX;
 architecture Behavioral of DEMUX is
 begin
 Process (I,S)
 begin
 if (S<= "00") then
   Y(0) <= I;
 elseif (S<= "01") then
  Y(1)<= I;
  elsif (S<= "10") then
   Y(2) <= I;
   else
    Y(3)<= I;
   end if;
   end process;
   end Behavioral;
```

LAB 5: Design 4-bit binary-to-gray and gray-to-binary code converters using VHDL.

Aim: To design 4-bit binary-to-gray and gray-to-binary code converters using VHDL.

Theory:

1) The 4-bit, binary-to-gray code converter



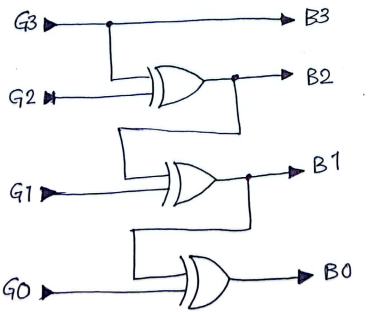
Truth Table:

B3	B2	B1	B0	Binary code	G3	G2	G1	G0	Gray code
0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	2	0	0	1	1	3
0	0	1	1	3	0	0	1	0	2
0	1	0	0	4	0	1	1	0	6
0	1	0	1	5	0	1	1	1	7
0	1	1	0	6	0	1	0	1	5
0	1	1	1	7	0	1	0	0	4
1	0	0	0	8	1	1	0	0	12
1	0	0	1	9	1	1	0	1	13
1	0	1	0	10	1	1	1	1	15
1	0	1	1	11	1	1	1	0	14
1	1	0	0	12	1	0	1	0	10
1	1	0	1	13	1	0	1	1	11
1	1	1	0	14	1	0	0	1	9
1	1	1	1	15	1	0	0	0	8

VHDL program:

```
library ieee;
use ieee.std_logic_1164.all;
entity b2g_code is
port (b: in std_logic_vector(3 downto 0);
      g: out std_logic_vector(3 downto 0));
end b2g_code;
architecture b2g_arch of b2g_code is
begin
g(3) <= b(3);
g(2) <= b(3) xor b(2);
g(1) <= b(2) xor b(1);
g(0) <= b(1) xor b(0);
end b2g_arch;
```

2) The gray-to-binary code converter circuit



VHDL program :

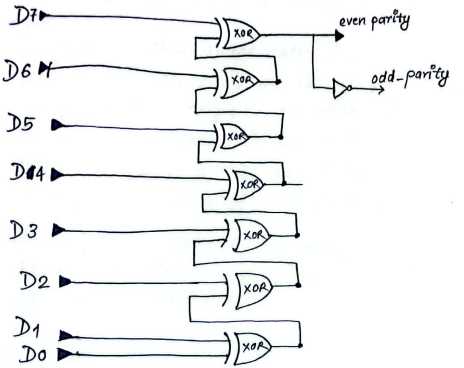
```
library ieee;
use ieee.std_logic_1164.all;
entity g26_code is
port (g : in std_logic_vector (3 downto 0);
      b : out std_logic_vector (3 downto 0));
end g26_code;
architecture g26_arch of g26_code is
begin
   b(3) <= g(3);
   b(2) <= g(3) xor g(2);
   b(1) <= g(3) xor g(2) xor g(1);
   b(0) <= g(3) xor g(2) xor g(1) xor g(0);
end g26_arch;
```

LAB 6 : Design 8-bit parity generator and checker circuits using VHDL.

Aim: To design 8-bit parity generator and checker circuits using VHDL.

Theory:

The 8-bit parity generator



Truth Table:

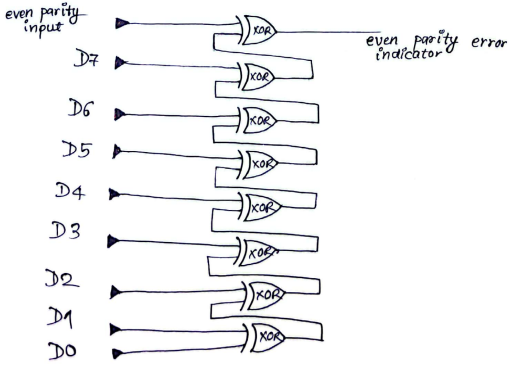
D7	D6	D5	D4	D3	D2	D1	D0	Even parity	odd Parity
1	0	1	1	0	0	1	0	0	1
1	1	0	0	1	0	0	0	1	0
1	1	1	1	1	0	1	1	1	0
1	0	1	1	1	1	1	0	0	1
0	0	1	0	1	0	1	0	1	0
0	1	1	1	0	1	0	1	1	0
0	1	0	1	0	0	1	1	0	1

VHDL Program:

```
library ieee;
use ieee.std_logic_1164.all;
entity Parity Generator is
 port (data: in bit_vector (7 downto 0);
       even_parity, odd_parity : out bit);
end Parity Generator;
architecture parity_generator of ParityGenerator is
signal temp : bit_vector (5 downto 0);
 begin
  temp (0) <= data(0) xor data(1);
  temp (1) <= temp (0) xor data (2);
  temp (2) <= temp (1) xor data (3);
  temp (3) <= temp (2) xor data (4);
  temp (4) <= temp (3) xor data (5);
  temp (5) <= temp (4) xor data (6);

 even_parity <= temp(5) xor data(7);
 odd_parity <= not(temp (5) xor data (7));
 end parity_generator;
```

The 8-bit parity checker



Truth Table

D7	D6	D5	D4	D3	D2	D1	D0	Parity input	Error
1	0	1	1	0	0	1	0	1	1
1	1	0	0	1	0	0	0	1	0
1	0	1	1	1	0	1	1	1	1
1	0	1	1	1	1	1	0	0	0
0	0	1	0	1	0	1	0	1	0
0	1	1	1	0	1	0	1	0	1
0	1	0	1	0	0	1	1	1	1

Note: Here not all of the 256 combinations of the D0-D7 are displayed. Just a few are taken as examples.

VHDL program:

```
library ieee;
use ieee.std_logic_1164.all;
entity Parity Checker is
  port(data : in bit_vector (7 downto 0);
       p: in bit;
       e: out bit);
end Parity Checker;
architecture parity_checker of Parity Checker is
signal temp : bit_vector (6 downto 0);
begin
  temp(0)<= data(0) xor data(1);
  temp(1)<= temp(0) xor data(2);
  temp(2)<= temp(1) xor data(3);
  temp(3)<= temp(2) xor data(4);
  temp(4)<= temp(3) xor data(4);
  temp(5)<= temp(4) xor data(5);
  temp(6) <= temp(5) xor data(7);
  e<= p xor temp(6);
end parity_checker;
```

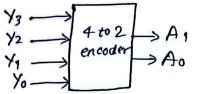
LAB 7: Design Encoder and Decoder using VHDL.

Theory:

Encoder:

An encoder is a combinational circuit that converts binary information in the form of a 2^n input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.

Block diagram:



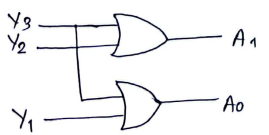
Truth Table

Inputs				Outputs	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Boolean function:

$A_1 = Y_3 + Y_2$, $A_0 = Y_3 + Y_1$

Circuit diagram:



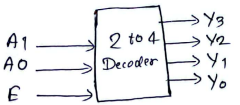
VHDL program for 4 to 2 encoders:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity encod1 is
Port (a: in STD_LOGIC_VECTOR (3 downto 0);
      q: out STD_LOGIC_VECTOR (1 downto 0));
   end encod1;
  architecture Behavioral of encod1 is
begin
 q <= "00" when a="0001" else
 "01" when a="0010" else
 "10" when a="0100" else
 "11";
 end Behavioral;
```

Decoder:

A decoder is a device which does the reverse of an encoder, undoing the encoding so that the original information can be retrieved. The same method used to encode is usually just reversed in order to decode.

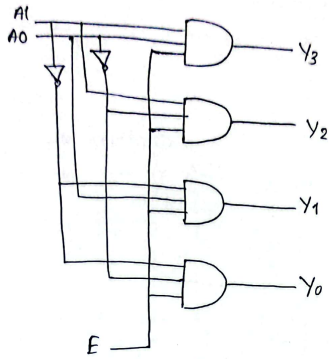
Block Diagram:



Truth Table

Enable	Inputs		Outputs			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Circuit Diagram:



VHDL Program for 2:4 decoder:

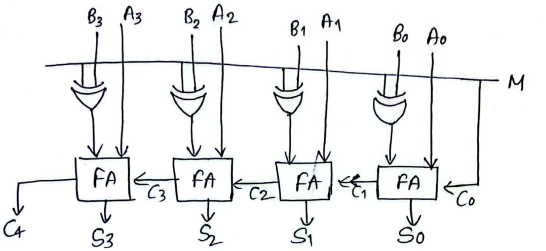
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity decoder is
Port (en: in STD_LOGIC;
S: in STD_LOGIC_VECTOR (1 downto 0);
Z: out STD_LOGIC_VECTOR (3 down to 0));
end decoder;
architecture arch of decoder is
signal p,q : STD_LOGIC;
begin
  p <= not S(1);
  q <= not S(0);
  Z(0)<= p and q ;
  Z(1)<= p and S(0);
  Z(2) <= q and S(1);
  Z(3) <= S(0) and S(1);
end arch;
```

LAB 8: Design 2's Complement Adder - Subtractor using VHDL.

Aim: to design 2's complement adder-subtractor using VHDL.

Description:

4 bit adder subtractor:



VHDL Program:

```
library ieee;
use ieee.std_logic_1164.all;
entity addsub is
 port (M: in std_logic;
       A,B : in std_logic_vector (3 downto 0);
        S : out std_logic_vector (3 downto 0);
       cout, OVERFLOW : out std_logic);
 end addsub;
architecture struct of addsub is
component full-Adder is
 port (X, Y, Cin : in std_logic;
       sum, Cout : out std-logic);

end component;
signal C1, C2, C3, C4: std-logic;
signal TMP: std-logic-vector (3 down to 0);
```

```
begin
  TMP(0) <= M xor B(0);
  TMP(1) <= M xor B(1);
  TMP(2) <= M xor B(2);
  TMP(3) <= M xor B(3);
  FA0: Full_Adder port map (A(0), TMP(0), M, S(0), C1);
  FA1: Full-Adder port map (A(1), TMP(1), C1, S(1), C2);
  FA2: Full-Adder port map (A(2), TMP(2), C2, S(2), C3);
  FA3: Full Adder port map (A(3), TMP(3), C3, S(3), C4);
  OVERFLOW <= C3 XOR C4;
  Cout <= C4;
  end struct;
```

LAB 9: Design of Registers using VHDL (SR flip-flop or JK flip-flop or D flip-flop or T flip-flop).

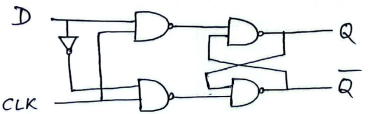
Description:

* D flip flop

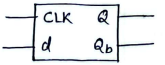
⊙ Truth Table:

Inputs	Outputs	
D	Q	Q_b
0	0	1
1	1	0

⊙ Circuit Diagram:



⊙ RTL Schematic:



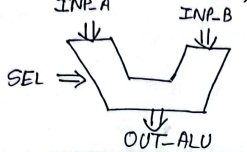
VHDL Program:

```
library ieee;
use ieee.std_logic_1164.all;
entity d_flipflop is
port (d, clk: in std_logic;
    Q: input std_logic := '0';
    Qb: inout std_logic := '1';
end d_flipflop;
architecture behaviour of d_flipflop is
begin
    process (d, clk)
begin
    if (clk = '0' and clk event) then
        Q<=d;
        Qb<= not (d);
end if;
end process;
end behaviour;
```

LAB 10 : Design 4-bit ALU using VHDL.

Description:

ALU's comprise the combinational logic that implements logic operations such as AND, OR, NOT gate and arithmetic operations, such as Adder, Subtractor.



Selection Input	Operation performed
0 0 0	A+B
0 0 1	A-B
0 1 0	A-1
0 1 1	A+1
1 0 0	A and B
1 0 1	A or B
1 1 0	not B
1 1 1	A xor B

VHDL program for ALU (4-bit ALU)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity alu is
  port (inp_a : in signed (3 downto 0);
  inp_b: in signed (3 downto 0);
  sel ; in STD_LOGIC_VECTOR (2 downto 0),
  out_alu : out signed (3 downto 0));
end alu;
architecture Behavioral of alu is
begin
process (inp_a, inp_b, sel)
begin
```

```
case sel is
  When "000" =>
  out_alu <= inp_a + inp_b;
  When "001" =>
  out_alu <= inp_a = inp_b;
  When "010" =>
  out_alu <= inp_a - 1;
  When "011" =>
  out_alu <= inp_a + 1;
  When "100" =>
  out_alu <= inp_a and inp_b;
  When "101" =>
  out alu <= inp_a or inpb;
  When "110" =>
  out_alu <= not inp_a;
  When "111" =>
  out_alu <= inp_a xor inp_b;
  When others =>
  NULL;
  end case;
  end process
end Behavioral;
```

LAB-12: Simulation of 5 stage or 4 stage or 3stage pipelining.

Description:

A 4-stage pipeline is a common architectural design used in processors and digital systems to improve performance by overlapping the execution of multiple instruction or operation.

Stages of pipeline:

1) Fetch stage
2) Decode stage
3) Execution stage
4) Writeback stage

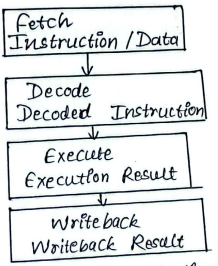


Fig - 4 Stage Pipeline

VHDL Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pip_4stage is
 port(
     clk : in STD_LOGIC;
     reset : in STD_LOGIC;
```

```
input : in STD_LOGIC_VECTOR (7 downto 0);
output: out STD_LOGIC_VECTOR (7 downto 0));
end pip-4 stage;
architecture Behavioral of Pip-4 stage is
type stage is (FETCH, DECODE, EXECUTE, WRITEBACK);
signal current-stage, next stage : stage;
signal fetch_data, decode_data, execute_data;
writeback_data : STD_LOGIC_VECTOR (7 downto 0);

begin
--pipeline stages
process (clk, reset)

begin
  if reset = '1' then
  current_ stage <= FETCH;
elsif rising-edge (clk) then
  current-stage <= next-stage;
end if
 end process;
-- state transitions
  process (current_stage)
 begin
   case current-stage is
  When FETCH =>
    -- FETCH stage: fetch instruction /data
    fetch_data <= input;
    -- Assuming input is instruction / data to fetch
    next-stage <= DECODE;
  When DECODE =>
    --DECODE stage : decode instruction
     Decode_data <= Fetch_data;
    -- Assuming decode logic here
   next-Stage <= EXECUTE;
   When EXECUTE =>
```

LAB-13 : Simulation of Booth addition and subtraction of signed 2's complement data.

Description:

Algorithm for adding and subtracting two binary numbers in signed 2's complement is shown in flowchart below -

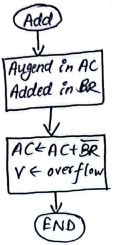


Fig- Flowchart for adding.

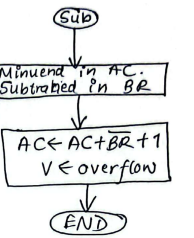


Fig- Flowchart for subtracting

C program:

```
#include<stdio.h>
int signedAddition (int x, int y)
{
  int carry = 0;
  while (y! = 0)
   {
     carry = x&y;
     x = x^y;
     y = carry<<1;
   }
  return x;
}
  int signedSubtraction (int x, int y)
{
   y = signedAddition (~y, 1);
   return signedAddition (x, y);
}
```

```
int boothAddition (int x, int y)
{ int sum = 0;
  int carry = 0;
  while (y!=0)
  { sum = x^y;
    carry = x&y;
    carry = carry << 1;
    x = sum;
    y = carry;
  }
  return sum;
}
int boothSubtraction (int x, int y)
{ y= signedAddition (~y, 1);
  return boothAddition (x, y);
}
int main ()
{
   int x,y;
   printf (" Enter 1st & 2nd operand ~~(x&y)~~:");
   scanf (" %d %d ", &x, &y);
   int additionResult = SignedAddition (x,y) ;
   printf(" Result of Signed Addition : %d \n.", additionResult);

   int subtractionResult = signedSubtraction (x,y);
   printf (" Result of signed subtraction : %d \n", subtractionResult);

   int boothSubResult = boothSubtraction (x,y);
   printf (" Result of Booth Subtraction : %d \n", boothSubResult);

   ~~return 0;~~
   int boothAddResult = booth Addition (x,y);
   printf (" Result of Booth Addition : %d \n", boothAddResult);

   return 0;
}
```

LAB 14: Simulation of Boot multiplication and division algorithm.

Description:

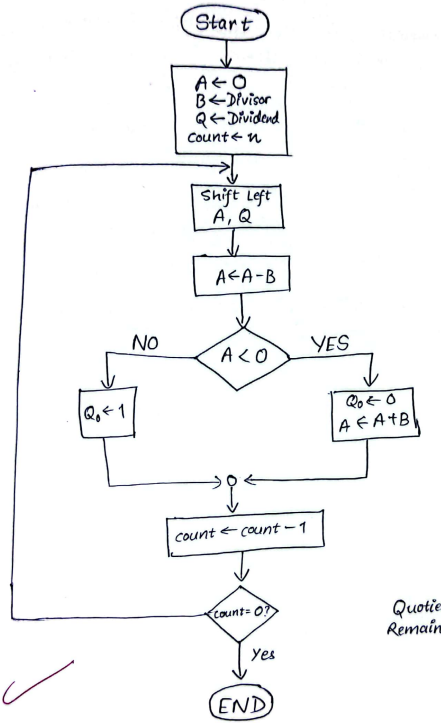


Fig- Flowchart for restoring division operation.

C program for restoring division operation

```
#include<stdio.h>
#include<math.h>

int a=0, b=0, c=0, com[5]={1,0,0,0,0}, s=0;
int anum[5]={0}, bnum[5]={0}, anumcp[5]={0};
int acomp[5]={0}, bcomp[5]={0}, rem[5]={0}, quo[5]={0};
    res[5]={0};

void binary()
{
  a=fabs(a);
  b=fabs(b);
  int r, r2, i, temp;
  for(i=0; i<5; i++)
  {
    r=a%2; a=a/2; r2=b%2; b=b/2;
    anum[i]=r; anumcp[i]=r; bnum[i]=r2;
    if(r2==0)
    {
      bcomp[i]=1;
    }
    if(r==0)
    {
      acomp[i]=1;
    }
  }
  c=0;
  for(i=0; i<5; i++)
  {
    res[i]=com[i]+bcomp[i]+c;
    if(res[i]>=2)
      c=1;
    else
      c=0;
    res[i]=res[i]%2;
  }
  for(i=4; i>=0; i--)
    bcomp[i]=res[i];
}
```

```
void add (int num[])
{ int i, c=0;
for (i=0; i<5; i++)
{ res[i] = rem[i] + num[i]+c;
  if (res[i] >=2)
    c=1;
  else
    c=0;
  res[i]=res[i] %2;
}

for (i=4; i>=0; i--)
{ rem[i] = res[i];
  printf ("%d", rem[i]);
}
  printf ("a:");
  for (i=4; i>=0; i--)
  { printf ("%d", anumCp[i]);
  }
}
  void sh()
  {
    int i;
    for (i=4; i>0; i--)
    { rem[i] = rem[i-1];
    }
    rem[0] = anumcp[4];
    for (i=4; i>0; i--)
    { anumcp[i] = anumcp[i-1]; }
    anumcp[0] = 0;
    printf ("\n SHIFT LEFT ");
    for (i=4; i>=0; i--)
    {printf ("%d", rem[i]); }
     printf (" :");
     for (i=4; i>=0; i--)
     {printf ("%d", anumcp[i]); }
}
```

```
int main()
{
  int i;
  printf ("Restoring Division \n Enter two numbers
          to divide \n Both numbers should be less
          than 16 \n\n");
  do { printf ("Enter the dividend: ");
       scanf ("%d", &a);
       printf ("Enter the divisor: ");
       scanf ("%d", &b);
     } while (a>=16 || b>=16);
printf("\n Expected Quotient: %d \n", a/b);
printf ("Expected Remainder: %d \n", a%b);
if (pow(a,b)<0)
      s=1;
binary ();
printf ("\n\n Unsigned Binary Equivalents are: \n");
printf ("A: ");
for (i=4; i>=0; i--)
 {
  printf ("%d", anum[i]);
 }
printf("\n B: ");
for (i=4; i>=0; i--)
{printf ("%d", bnum[i]); }
printf ("\n B'+1: ");
for (i=4; i>=0; i--)
{ printf ("%d", bcomp[i]); }
 printf ("\n\n-----> ");
shl();
```

```
for (i=0; i<5; i++)
  {
    printf ("\n-->");
    printf ("\n SUB B:");
    add (6comp);
    if (rem[4] == 1)
    {
      printf ("\n --> Restore \n ADD B:");
      anumcp[0]= 0;
      add(num);
    }
    else
      anumcp[0] = 1;
    if (i<4)
      shl ();
  }
  printf ("n ------\n");
  printf("Sign of result : %d \n", s);
  printf(" Remainder : ");
  for (i=4; i>=0; i--)
    printf(" %d", rem[i]);
  printf ("Quotient:");
  for (i=4; i>=0; i--)
  printf ("%d ", anumcp[i]);
  return 0;
}
```