Experiment 1: Data representation

Objective:

1. To illustrate the concept of data representation

Theory:

Data representation refers to the way data is formatted, stored, processed, and transmitted in a computer system. Computers work with electrical signals that can be either "on" or "off," which is typically represented using binary numbers (0s and 1s). The different types of data representation are:

- Binary
- Hexadecimal
- Octal
- 1’s and 2’s complement

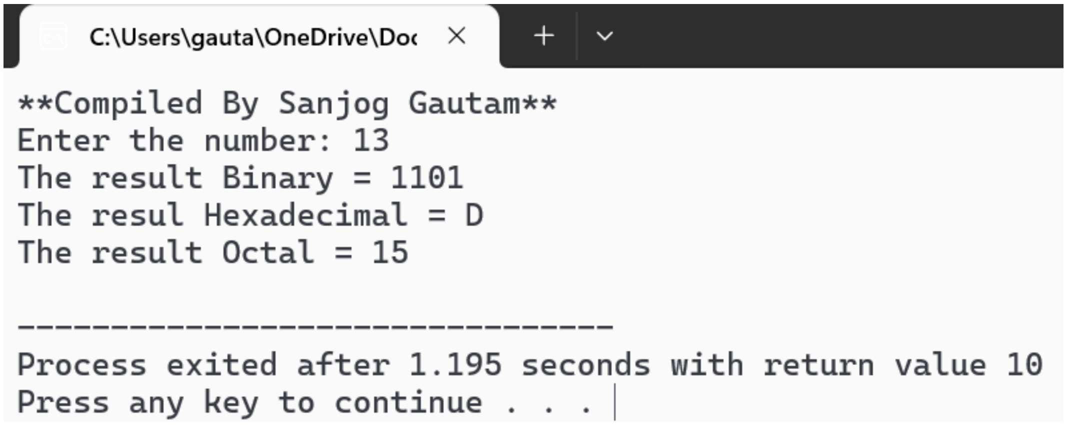
Algorithm:

```
i. Start
ii. Initialize a function to find binary 1’s and 2’s complement, hexadecimal, octal equivalent of number
iii. Input a number to be converted
iv. Find the equivalent numbers
v. Print the binary 1’s and 2’s complement, hexadecimal and octal equivalent numbers
vi. Stop
```

Source code:

```
#include<stdio.h>
int conversion(int num, int base)
{
    int rem;
        if (num == 0)
    {
        return 1;
    }
    else
    {
        rem = num % base;
        conversion(num / base, base);
        if (base == 16 && rem >= 10)
        {
            printf("%c", rem + 55);
        }
        else
        {
            printf("%d", rem);
        }
    }
}
int main()
{
    int num;
      printf("**Compiled By Sanjog Gautam**\n");
      printf("Enter the number: ");
      scanf("%d", &num);
      if(num!=0)
      {
            printf("The result Binary = ");
            conversion(num, 2);
            printf("\n");
            printf("The result Hexadecimal = ");
            conversion(num, 16);
            printf("\n");
            printf("The result Octal = ");
            conversion(num, 8);
            printf("\n");
      }
      else
      {
            printf("The  binary number is=O\n");
            printf("The  hexadecimal number is=O\n");
            printf("The  octal number is=O\n");
      }
}
```

Output:

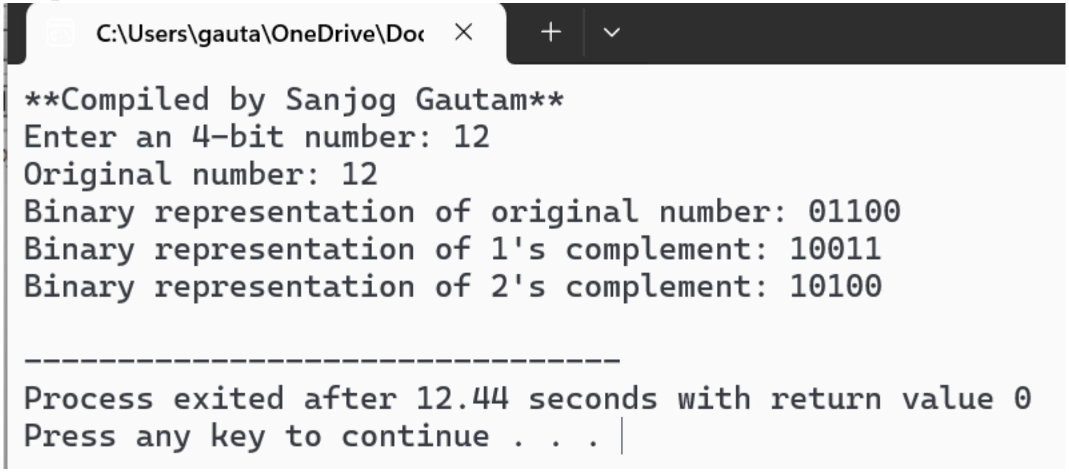


```
**Compiled By Sanjog Gautam**
Enter the number: 13
The result Binary = 1101
The resul Hexadecimal = D
The result Octal = 15

--------------------------------
Process exited after 1.195 seconds with return value 10
Press any key to continue . . .
```

Source code:

```
#include <stdio.h>
int i;
char onesComplement( char num)
{
    return ~num;
}
char twosComplement( char num)
{
    return onesComplement(num) + 1;
}
void printBinary( char num)
{
    for (i = 4; i >= 0; i--)
   {
        printf("%d", (num >> i) & 1);
    }
}

int main()
{
     char num;
    printf("**Compiled by Sanjog Gautam**\n");
    printf("Enter an 4-bit number: ");
    scanf("%d", &num);
    unsigned char onesComp = onesComplement(num);
    unsigned char twosComp = twosComplement(num);

    printf("Original number: %d\n", num);
    printf("Binary representation of original number: ");
    printBinary(num);
    printf("\n");
    printf("Binary representation of 1's complement: ");
    printBinary(onesComp);
    printf("\n");
    printf("Binary representation of 2's complement: ");
    printBinary(twosComp);
    printf("\n");

    return 0;
}
```

Output:



```
**Compiled by Sanjog Gautam**
Enter an 4-bit number: 12
Original number: 12
Binary representation of original number: 01100
Binary representation of 1's complement: 10011
Binary representation of 2's complement: 10100

--------------------------------
Process exited after 12.44 seconds with return value 0
Press any key to continue . . .
```

Experiment 2: Data overflow

Objective:

1. To understand the concept of data overflow

Theory:

Overflow occurs when an arithmetic operation or a memory allocation exceeds the capacity of the data type or storage that is available. There are specific conditions under which overflow happens, depending on the type of operation and the data type being used.

Condition for overflow:

```
((AS==BS) &(AS==RS) || (AS! =BS))
{
NO overflow, display result;
}
```

Algorithm:

i. Start

ii. Observe carry into the sign bit position & carry out the sign bit position

iii. If the two carry aren't equal, overflow should be detected

iv. If the two carry are applied to an X-OR gate overflow will be detected when

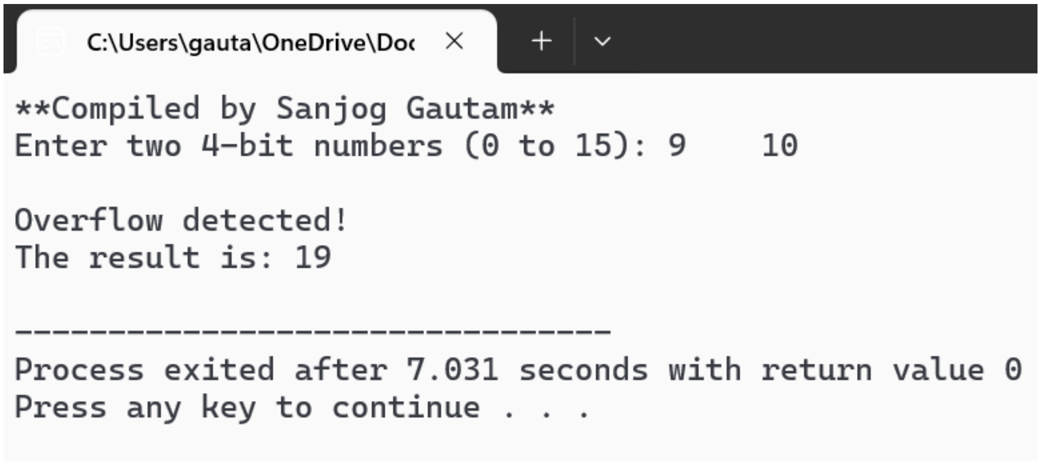
output of gate is 1.

v. Stop

Source code:

```
#include <stdio.h>
int main() {
    int num1, num2, result,sign1, sign2, signResult;;
    printf("**Compiled by Sanjog Gautam**\n")
    printf("Enter two 4-bit numbers (0 to 15): ");
    scanf("%d%d", &num1, &num2);
            result = num1 + num2;
    // Extract the 4th bit (sign bit) of each number and the result
    sign1 = (num1 >> 3) & 1;  // 4th bit of num1
    sign2 = (num2 >> 3) & 1;  // 4th bit of num2
    signResult = (result >> 3) & 1;  // 4th bit of result
    // Check for overflow
    if ((sign1 == sign2 && sign1 != signResult) || (sign1 != sign2 && signResult == 1)) {
        printf("\nOverflow detected!\nThe result is: %d\n", result);
    } else {
        printf("\nNo overflow.\nResult = %d\n", result);
    }
    return 0;
}
```

Output:



```
**Compiled by Sanjog Gautam**
Enter two 4-bit numbers (0 to 15): 9    10

Overflow detected!
The result is: 19

--------------------------------
Process exited after 7.031 seconds with return value 0
Press any key to continue . . .
```

Experiment 3: Introduction to VHDL

Objective:

1. To implement the Basic gates and universal gates using VHDL

Theory:

VHDL stands for (VHSIC Hardware Description Language). It is a hardware description language used to model and simulate digital systems, such as circuits and systems designed using VLSI (Very-Large-Scale Integration) technology. VHDL is used primarily in designing hardware and describing its behavior, structure, and timing. In VHDL, entity is used to describe a hardware module. An entity can be described
using:

i. Entity declaration
ii. Architecture
iii. Configuration
iv. Package declaration
v. Package body

Entity Declaration:

It defines the name, input/output signals and modes of hardware module.

Syntax:

```
entity entity_name is
port declaration;
end entity_name;
```

Architecture:

Can be described using structural, data flow, behavior or mixed type.

Syntax:

```
architecture architecture_name of entity_name is
architecture architecture_declarative part;
begin
statements;
end architecture_name;
logic operation: N
```

MBM
Madan Bhandari Memorial College

Experiment i: Introduction to VHDL

Objective:

1. To implement the AND gate using VHDL

Theory:

An AND gate is a basic digital logic gate that implements logical conjunction—meaning it outputs true (or `1`) only if all its inputs are true (or `1`). If at least one input is false (or `0`), the output will be false (or `0`). The AND gate is one of the fundamental building blocks in digital electronics.

Truth Table for AND Gate:

Input A	Input B	Output (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

Source code: For and gate

Testbench:

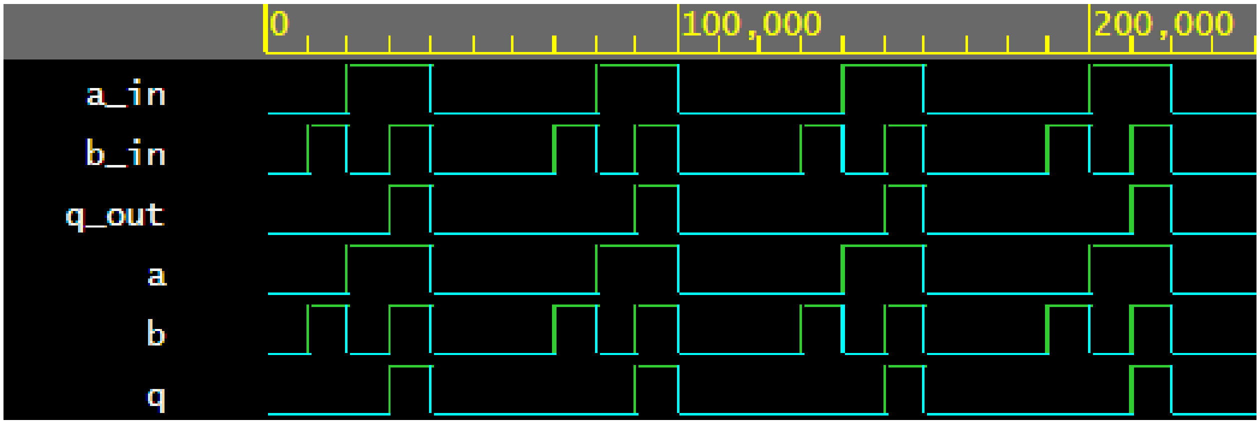
```
-- Testbench for AND gate
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
-- empty
end testbench;

architecture tb of testbench is
-- DUT component
component and_gate is
port(
        a: in std_logic;
        b: in std_logic;
        q: out std_logic);
end component;
signal a_in, b_in, q_out: std_logic;
begin
  -- Connect DUT
  DUT: and_gate port map(a_in, b_in, q_out);
  process
  begin
    a_in <= '0';
    b_in <= '0';
    wait for 10 ns;

    a_in <= '0';
    b_in <= '1';
    wait for 10 ns;

    a_in <= '1';
    b_in <= '0';
    wait for 10 ns;

    a_in <= '1';
    b_in <= '1';
    wait for 10 ns;
```

```
        -- Clear inputs
        a_in <= '0';
        b_in <= '0';
        wait for 20 ns;
            end process;
             end tb;
```

Architecture:

```
-- Simple AND gate design
library IEEE;
use IEEE.std_logic_1164.all;
entity and_gate is
port(
  a: in std_logic;
  b: in std_logic;
  q: out std_logic);
end and_gate;
architecture rtl of and_gate is
begin
  process(a, b) is
  begin
    q <= a and b;
  end process;
end rtl;
```

Output:



MBM
Madan Bhandari Memorial College

Experiment ii: Half adder

Objective:

1. To implement the Half adder using VHDL

Theory:

A half adder is a basic digital circuit used to add two single-bit binary numbers. It has two inputs, typically labeled A and B, and two outputs: the sum (S) and the carry (C).

- **Sum (S)**: This is the result of the bitwise addition of A and B, without considering any carry from a previous operation. It can be found using the XOR (exclusive OR) operation.
- **Carry (C)**: This is the carry-out bit, which is generated when both A and B are 1, meaning there is an overflow into the next higher bit. It can be found using the AND operation.

Truth table:

A	B	Sum(S)	Carry(C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Source code:

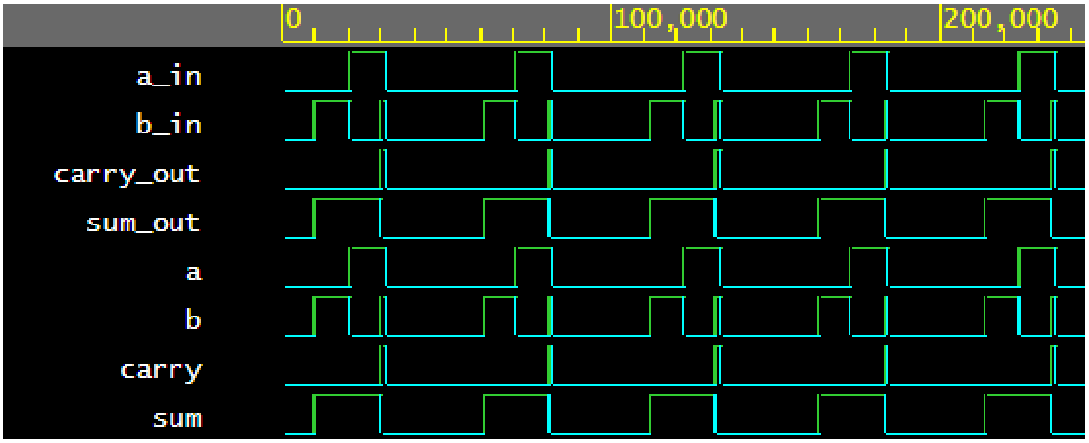
Testbench:

```
-- Testbench for half adder gate
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
-- empty
end testbench;

architecture tb of testbench is
-- DUT component
component halfadder_gate is
port(
  a: in std_logic;
  b: in std_logic;
  sum: out std_logic;
  carry:out std_logic);
end component;
signal a_in, b_in, sum_out, carry_out: std_logic;

begin

  -- Connect DUT
  DUT: halfadder_gate port map(a_in, b_in, sum_out, carry_out);

  process
  begin
    a_in <= '0';
    b_in <= '0';
    wait for 10 ns;

    a_in <= '0';
    b_in <= '1';
    wait for 10 ns;
    a_in <= '1';
    b_in <= '0';
    wait for 10 ns;
    a_in <= '1';
    b_in <= '1';
    wait for 10 ns;
    -- Clear inputs
    a_in <= '0';
    b_in <= '0';
    wait for 20 ns;
  end process;
end tb;
```

Architecture:

```
-- Simple half adder gate design
library IEEE;
use IEEE.std_logic_1164.all;

entity halfadder_gate is
port(
  a: in std_logic;
  b: in std_logic;
  sum: out std_logic;
  carry:out std_logic);
end halfadder_gate;

architecture rtl of halfadder_gate is
begin
  process(a, b) is
  begin
    sum <= a xor b;
    carry <= a and b;
  end process;
end rtl;
```

Output:



Experiment ii: Full adder

Objective:

1. To implement the Full adder using VHDL

Theory: A **Full Adder** is a digital circuit that computes the sum of three binary inputs: two **operands** and a **carry-in** bit. It produces a **sum** and a **carry-out** bit.

The Full Adder is an extension of the **Half Adder**, which only adds two binary bits. In the Full Adder, we also consider a carry from the previous addition, making it more suitable for multi-bit binary addition, such as adding numbers in computer arithmetic. The output of full adder are:

1. **Sum (S)** (The result of adding A, B, and Carry-in)
2. **Carry-out (Cout)** (Carry to the next bit position)

Truth table:

A	**B**	**Cin**	**Sum(S)**	**Carry(Cout)**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

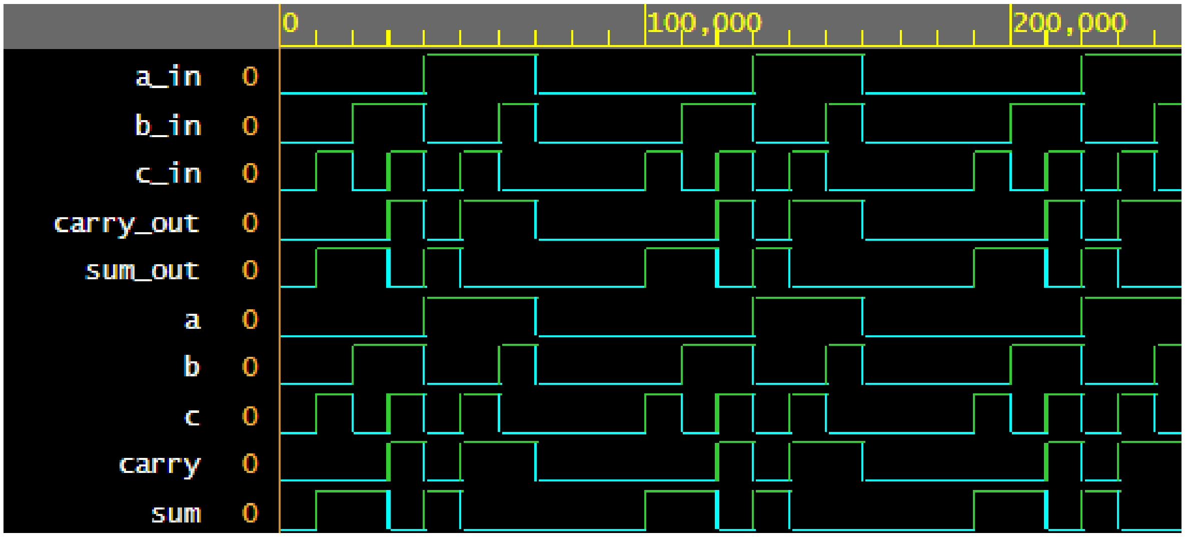
Source code:

```
-- Testbench for fulladder gate
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
-- empty
end testbench;
architecture tb of testbench is
-- DUT component
component fulleladder_gate is
port(
  a: in std_logic_vector(3 downto 0);
  b: in std_logic_vector(3 downto 0);
  c:in std_logic;
  sum: out std_logic_vector(3 downto 0);
  carry:out std_logic);
end component;

signal a_in,b_in:std_logic_vector, c_in:std_logic;
signal sum_out:std_logic_vector(3 downto 0),carry_out:std_logic;

begin
  -- Connect DUT
  DUT: fulladder_gate port map(a_in, b_in, c_in, sum_out, carry_out);
  process
  begin
    a_in <= '0';
    b_in <= '0';
    c_in <= '0';
    wait for 10 ns;

    a_in <= '0';
    b_in <= '0';
    c_in <= '1';
    wait for 10 ns;

    a_in <= '0';
    b_in <= '1';
    c_in <= '0';
    wait for 10 ns;

    a_in <= '0';
    b_in <= '1';
    c_in <= '1';
    wait for 10 ns;
    a_in <= '1';
    b_in <= '0';
    c_in <= '0';
    wait for 10 ns;

    a_in <= '1';
    b_in <= '0';
    c_in <= '1';
    wait for 10 ns;
    a_in <= '1';
    b_in <= '1';
    c_in <= '0';
    wait for 10 ns;
    a_in <= '1';
    b_in <= '1';
    c_in <= '1';
     wait for 10 ns;
    -- Clear inputs
    a_in <= '0';
    b_in <= '0';
             c_in <= '0';
    wait for 20 ns;
  end process;
end tb;
```

Architecture:

```
-- Simple fulladder gate design
library IEEE;
use IEEE.std_logic_1164.all;

entity fulladder_gate is
port(
  a: in std_logic;
  b: in std_logic;
  c: in std_logic;
  sum: out std_logic;
  carry:out std_logic);
end fulladder_gate;

architecture rtl of fulladder_gate is
begin
  process(a, b, c) is
  begin
    sum <= a xor b xor c;
    carry <= (a and b) or (a and c) or (b and c);
  end process;
end rtl;
```

Output:


0
100,000
200,000
a_in 0
b_in 0
c_in 0
carry_out 0
sum_out 0
a 0
b 0
c 0
carry 0
sum 0

Experiment iv: 4-bit parallel adder

Objective:

1. To implement the 4-bit parallel adder using VHDL

Theory:

A 4-bit parallel adder is a digital circuit that adds two 4-bit binary numbers simultaneously, producing a 4-bit sum and a carry-out bit. It is made up of multiple full adders connected in parallel. Each full adder in the circuit adds corresponding bits of the two 4-bit inputs, along with any carry from the previous stage.

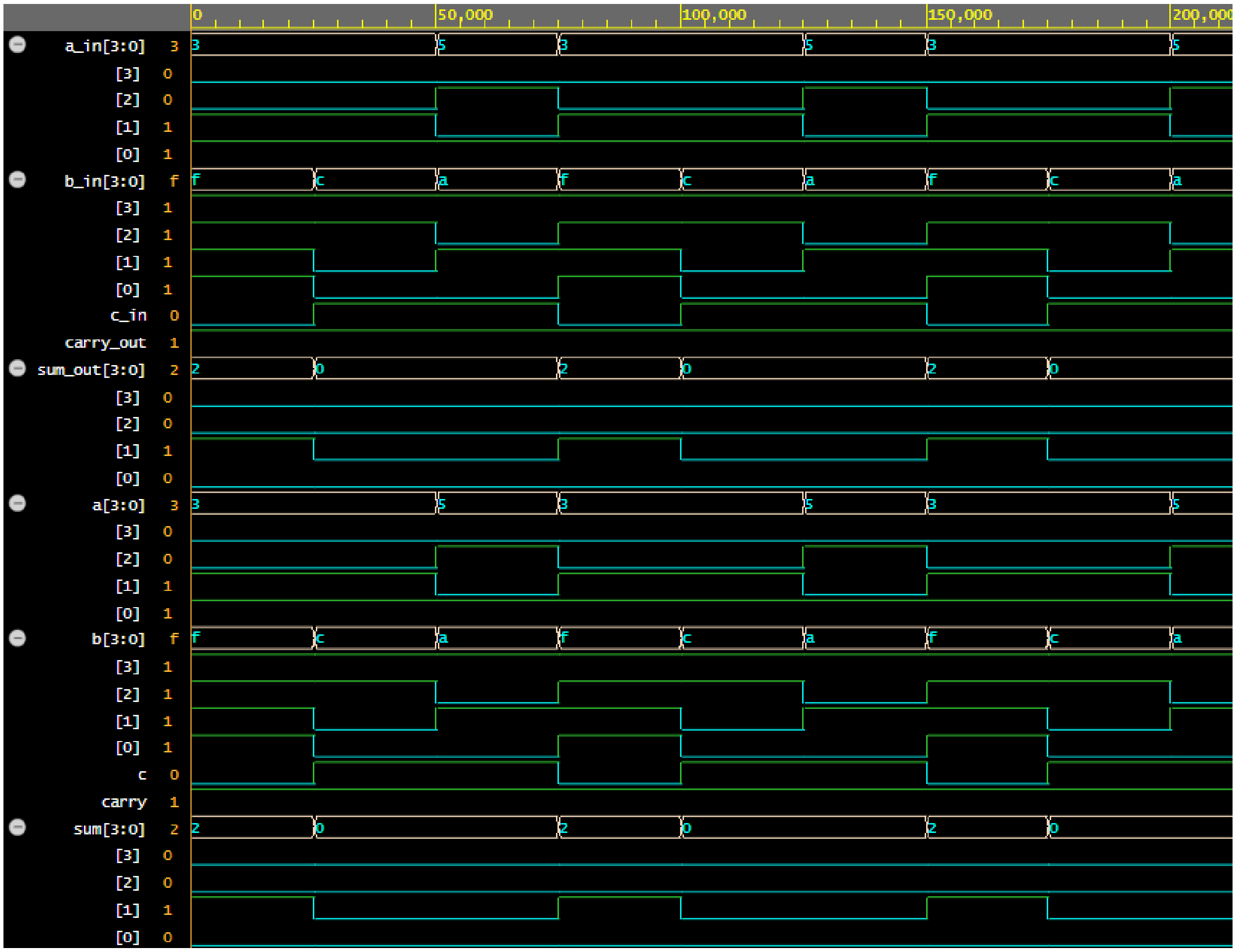
Architecture:

```
-- Simple paralleladder gate design
library IEEE;
use IEEE.std_logic_1164.all;

entity paralleladder_gate is
port(
  a: in std_logic_vector(3 downto 0);
  b: in std_logic_vector(3 downto 0);
  c: in std_logic;
  sum: out std_logic_vector(3 downto 0);
  carry:out std_logic);
end paralleladder_gate;

architecture rtl of paralleladder_gate is
begin
  process(a, b, c) is
  variable temp: std_logic;
  begin
  temp:= c;
  for i in 0 to 3 loop
    sum(i) <= a(i) xor b(i) xor temp;
    temp:=(a(i) and B(i)) or (temp and a(i)) or (temp and b(i));
    end loop;
    carry<=temp;
  end process;
end rtl;
```

Source code:

```
-- Testbench for paralleladder gate
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
-- empty
end testbench;

architecture tb of testbench is

-- DUT component
component paralleladder_gate is
port(
  a: in std_logic_vector(3 downto 0);
  b: in std_logic_vector(3 downto 0);
  c:in std_logic;
  sum: out std_logic_vector(3 downto 0);
  carry:out std_logic);
end component;

signal a_in,b_in:std_logic_vector(3 downto 0):=(others=>'0'); signal c_in:std_logic;
signal sum_out:std_logic_vector(3 downto 0):= (others=>'0');
signal carry_out:std_logic;
begin
  -- Connect DUT
  DUT: paralleladder_gate port map(a_in, b_in, c_in, sum_out, carry_out);
process
  begin
  --test 1
    a_in <="0011";
    b_in <="1111";
    c_in <='0';
    wait for 10 ns;
    --test 2
    a_in <="0011";
    b_in <="1100";
    c_in <='1';
    wait for 10 ns;
    --test 3
    a_in<="0101";
    b_in <="1010";
    c_in <='1';
    wait for 20 ns;
  end process;
  end tb;
```

Output:



Experiment v: Encoder

Objective:

1. To implement encoder using VHDL

Theory:

An encoder is a digital circuit that converts data from one format or code to another, typically from a larger set of inputs to a smaller set of outputs. In the context of binary systems, an encoder is a device that converts an active input line into a binary code corresponding to that input.

For example, in a binary encoder, the number of output bits is fewer than the number of input lines. The encoder "encodes" the input into a binary number representing the position of the active input line.

Architecture:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encoder4_2 is
 Port ( a: in STD_LOGIC_VECTOR (3 downto 0);
 b: out STD_LOGIC_VECTOR (1 downto 0));
end Encoder4_2;
architecture rtl of Encoder4_2 is
begin
 process(a)
 begin
 if (a="0001") then
 b <= "00";
 elsif (a="0010") then
 b <= "01";
 elsif (a="0100") then
 b <= "10";
 elsif (a="1000") then
 b <= "11";
 else
 b <= "ZZ";
 end if;
 end process;
end rtl;
```

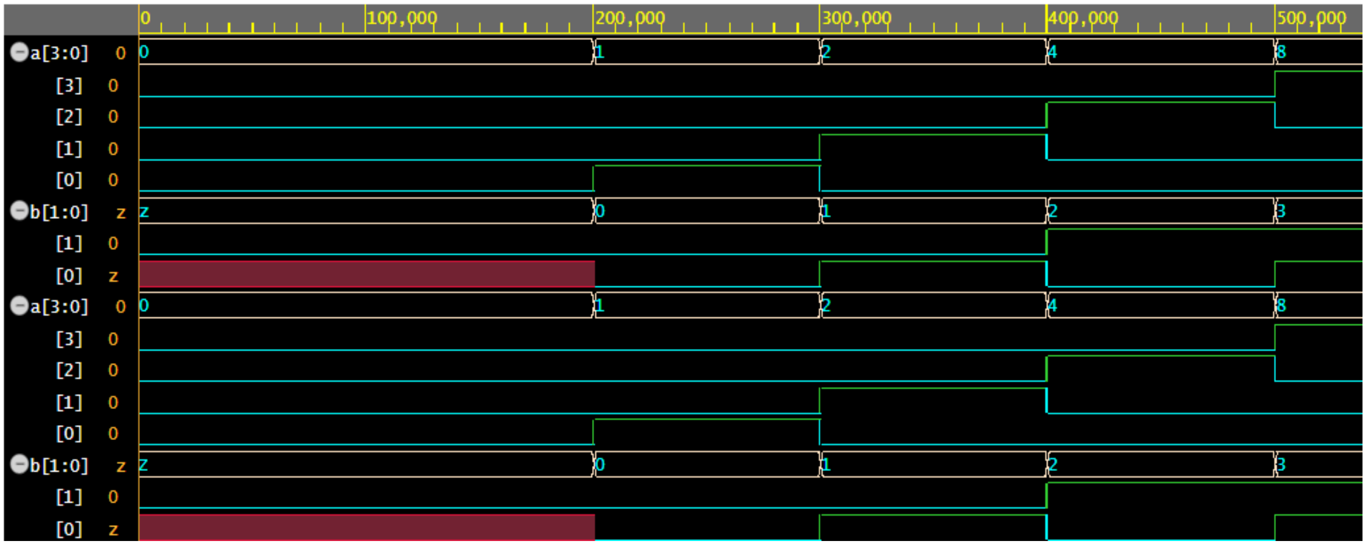
Source code:

 Encoder:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY Encoder_test IS
END Encoder_test;
ARCHITECTURE behavior OF Encoder_test IS
 COMPONENT Encoder4_2
 PORT (a: IN std_logic_vector(3 downto 0);
 b: OUT std_logic_vector(1 downto 0)
 );
 END COMPONENT;
signal a: std_logic_vector(3 downto 0) := (others => '0');
signal b: std_logic_vector(1 downto 0);
BEGIN
uut: Encoder4_2 PORT MAP (a => a,b => b);
process
begin
 wait for 100 ns;
 a <= "0000";
 wait for 100 ns;
 a <= "0001";
 wait for 100 ns;
a <= "0010";
 wait for 100 ns;
 a <= "0100";
 wait for 100 ns;
 a <= "1000";
 wait;
 end process;
```

```
end;
```

Output:


0
100,000
200,000
300,000
400,000
500,000
a[3:0] 0
0 1 2 4 8
[3] 0
[2] 0
[1] 0
[0] 0
b[1:0] z
z 0 1 2 3
[1] 0
[0] z
a[3:0] 0
0 1 2 4 8
[3] 0
[2] 0
[1] 0
[0] 0
b[1:0] z
z 0 1 2 3
[1] 0
[0] z

Experiment vi: ALU

Objective:

1. To implement ALU using VHDL

Theory:

The Arithmetic and Logical unite is the fundamental component in a computing system like a computer. It is basically the actual data processing element within the central processing unit (CPU) in a computing system. It performs all the arithmetic and logical operations and forms the backbone of modern computer technology.

Architecture:

```
--design for alu
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity ALU is
 generic ( constant N: natural:=1); -- number of shifted or rotated bits
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );
end ALU;
architecture Behavioral of ALU is
signal ALU_Result: std_logic_vector (7 downto 0);
signal tmp: std_logic_vector (8 downto 0);
begin
 process(A,B,ALU_Sel)
 begin
 case(ALU_Sel) is
 when "0000" => -- Addition
 ALU_Result <= A + B ;
 when "0001" => -- Subtraction
 ALU_Result <= A - B ;
 when "0010" => -- Multiplication
 ALU_Result<=std_logic_vector(to_unsigned((to_integer(unsigned(A)) *
to_integer(unsigned(B))),8)) ;
 when "0011" => -- Division
 ALU_Result <= std_logic_vector(to_unsigned(to_integer(unsigned(A)) /
to_integer(unsigned(B)),8)) ;
```

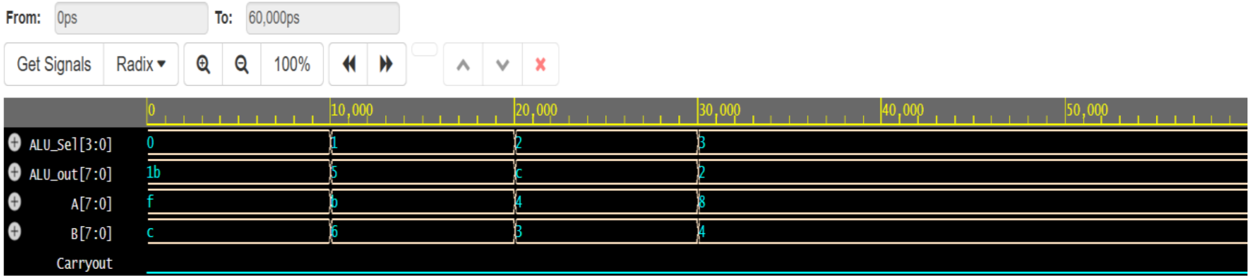
```
 when "0100" => -- Logical shift left
 ALU_Result <= std_logic_vector(unsigned(A) sll N);
 when "0101" => -- Logical shift right
 ALU_Result <= std_logic_vector(unsigned(A) srl N);
 when "0110" => -- Rotate left
 ALU_Result <= std_logic_vector(unsigned(A) rol N);
 when "0111" => -- Rotate right
 ALU_Result <= std_logic_vector(unsigned(A) ror N);
 when "1000" => -- Logical and
 ALU_Result <= A and B;
 when "1001" => -- Logical or
 ALU_Result <= A or B;
 when "1010" => -- Logical xor
 ALU_Result <= A xor B;
 when "1011" => -- Logical nor
 ALU_Result <= A nor B;
 when "1100" => -- Logical nand
 ALU_Result <= A nand B;
 when "1101" => -- Logical xnor
 ALU_Result <= A xnor B;
 when "1110" => -- Greater comparison
 if(A>B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;when "1111" => -- Equal comparison
 if(A=B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;
 when others => ALU_Result <= A + B ;
 end case;
end process;
ALU_Out <= ALU_Result; -- ALU out
tmp <= ('0' & A) + ('0' & B);
Carryout <= tmp(8); -- Carryout flag
end Behavioral;
```

Source code:

```
 -- Testbench for alu adder
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
  -- empty
end testbench;
```

```
architecture tb of testbench is

  -- DUT component
  component ALU is
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );

  end component;

  signal A, B: std_logic_vector(7 downto 0);
  signal ALU_Sel: std_logic_vector(3 downto 0);
  signal ALU_out: std_logic_vector(7 downto 0);
  signal Carryout: std_logic;

begin

  -- Connect DUT
  DUT: ALU
    port map (
      A => A,
      B => B,
      ALU_Sel => ALU_Sel,
      ALU_out => ALU_out,
      Carryout => Carryout
    );

  -- Test process
  process
  begin
    -- Test cases
    A <= "00001111";
    B <= "00001100";
    ALU_Sel<= "0000";
    wait for 10 ns;  -- Expected output: S_out = "0000", Cout_out = '0'


    A <= "00001011";
    B <= "00000110";
    ALU_Sel <= "0001";
    wait for 10 ns;  -- Expected output: S_out = "0010", Cout_out = '0'

    A <= "00000100";
    B <= "00000011";
    ALU_Sel <= "0010";
    wait for 10 ns;
     A <= "00001000";
    B <= "00000100";
    ALU_Sel <= "0011";
    wait for 10 ns;

    -- End of simulation
    wait for 20 ns;
    assert false report "End of simulation" severity failure;
  end process;

end tb;
```

Output:


From: 0ps To: 60,000ps

Conclusion:

In conclusion, designing an ALU in VHDL enables efficient implementation of arithmetic and logic operations in digital systems. VHDL's flexibility and modularity allow for easy testing, simulation, and refinement, ensuring reliable performance. It is a powerful tool for creating high-performance, cost-effective hardware designs.

Experiment vii: 3-Segment Pipeline

Objective:

1. To implement 3-Segment Pipeline using VHDL

Theory:

The Arithmetic and Logical unite is the fundamental component in a computing system like a computer. It is basically the actual data processing element within the central processing unit (CPU) in a computing system. It performs all the arithmetic and logical operations and forms the backbone of modern computer technology.

Architecture:

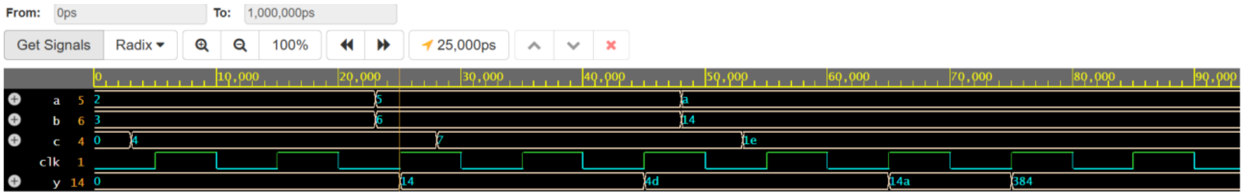
```
--design for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline is
   Port (
      a : in integer;
      b : in integer;
      c : in integer;
      clk : in STD_LOGIC;
      y : out integer
   );
end pipeline;

architecture Behavioral of pipeline is
   signal r1, r2, r3, r4, r5 : integer := 0;
begin
   y <= r5;

   process(clk)
   begin
      if rising_edge(clk) then
         -- Pipeline stage 1
         r1 <= a;
         r2 <= b;

         -- Pipeline stage 2 (registered)
         r4 <= r1 + r2;
         r3 <= c;

         -- Pipeline stage 3 (registered)
         r5 <= r4 * r3;
      end if;
```

```
    end process;
end Behavioral;
```

Source code:

```
--testbench for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline_tb is
end pipeline_tb;

architecture Behavioral of pipeline_tb is
    component pipeline
        Port (
            a : in integer;
            b : in integer;
            c : in integer;
            clk : in STD_LOGIC;
            y : out integer
        );
    end component;

    signal a, b, c : integer := 0;
    signal y : integer;
    signal clk : STD_LOGIC := '0';

    constant clk_period : time := 10 ns;

begin
    uut: pipeline port map (
        a => a,
        b => b,
        c => c,
        clk => clk,
        y => y
    );

    -- Clock generation
    clk_process: process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin

        -- Test case 1
        a <= 2;
        b <= 3;
        wait for 3 ns;
        c <= 4;
        wait for clk_period*2;

        -- Test case 2
        a <= 5;
        b <= 6;
        wait for 5 ns;
        c <= 7;
        wait for clk_period*2;

        -- Test case 3
        a <= 10;
        b <= 20;
        wait for 5 ns;
        c <= 30;
        wait for clk_period*2;

        -- End simulation
        wait;
    end process;
end Behavioral;
```

Output:



From: 0ps To: 1,000,000ps

Conclusion:

In conclusion, implementing a 3-segment pipeline in VHDL enhances system performance by allowing parallel processing of data in stages. This approach increases throughput and reduces latency by efficiently organizing tasks across different pipeline segments. VHDL provides the necessary tools to model, simulate, and optimize the pipeline, ensuring smooth operation and scalability in complex digital systems.

Experiment 4: Booth's Multiplication Algorithm

Objective:

1. To implement Booth's Multiplication Algorithm.

Theory:

Hardware Implementation

It needs same hardware as that of addition and subtraction of signed-magnitude. In addition, it needs two more registers Q and SC.

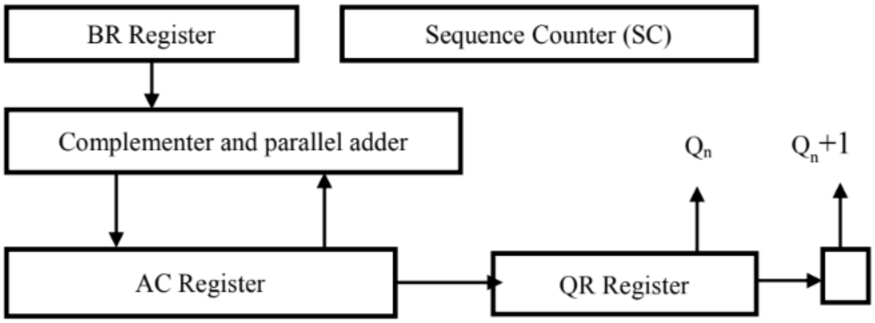


Fig: hardware for booths algorithm

Algorithm:

1. Initially, BR←Multiplicand and QR←Multiplier.
2. AC←0, Q_{n+1}←0 and SC←n (no of bits in multiplier).
3. Inspect Q_n, Qn+1,
   i. If (Q_nQ_{n+1}=10), first 1 in a string of 1' s has been encountered, subtraction required. i.e., **AC←AC+$\overline{BR}$+1**
   ii. If (Q_nQ_{n+1}=01), first 0 in a string of 0' s has been encountered, addition required. **i.e., AC←AC+BR**
   iii. If (Q_nQ_{n+1}=00 or 11(equal)), do nothing
4. Shift right the partial product and the multiplier (including bit Q_{n+1}). This Arithmetic Shift Right (ashr) operation shifts AC and QR to the right and leaves the sign bit in AC unchanged.
5. SC←SC-1. If (SC=0) stop the process else continue and go to step 3.
6. Result in AC and QR

Flowchart:

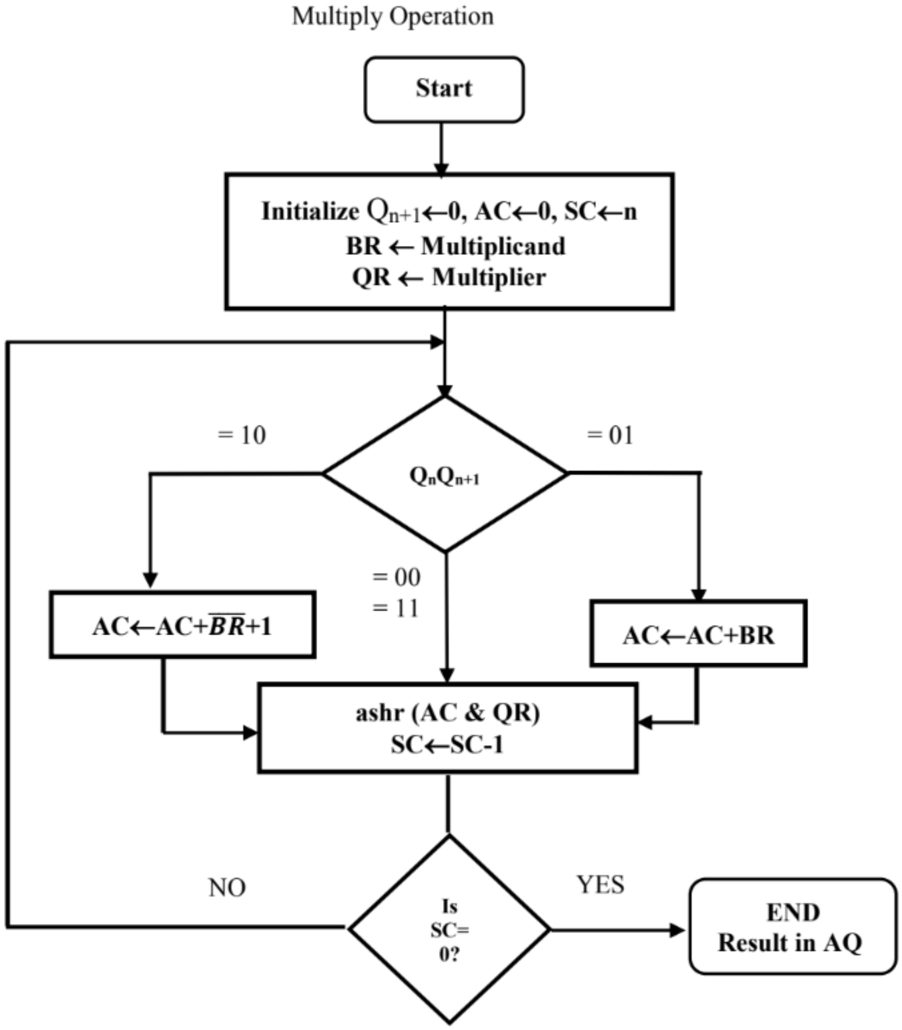
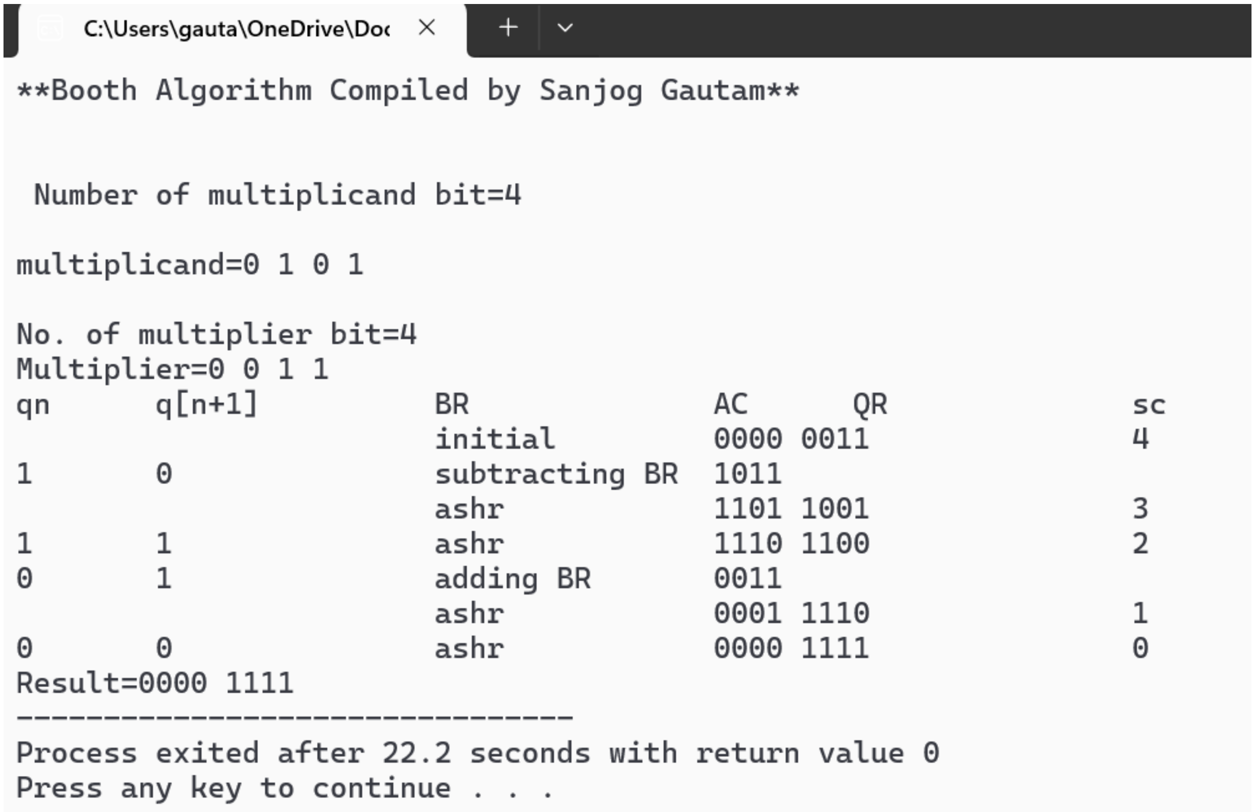


Fig: Booths Algorithm for multiplication of Signed-2's Complement numbers

```
#include<iostream>
using namespace std;
void add(int a[], int x[], int q);
void complement(int a[], int n) {
  int i;
  int x[8] = { NULL };
  x[0] = 1;
  for (i = 0; i < n; i++) {
    a[i] = (a[i] + 1) % 2;
  }
  add(a, x, n);
}

void add(int ac[], int x[], int q) {
  int i, c = 0;
  for (i = 0; i < q; i++) {
    ac[i] = ac[i] + x[i] + c;
    if (ac[i] > 1) {
      ac[i] = ac[i] % 2;
      c = 1;
    }else
    c = 0;
  }
}

void ashr(int ac[], int qr[], int &qn, int q) {
  int temp, i;
  temp = ac[0];
  qn = qr[0];
  cout << "\t\tashr\t\t";
  for (i = 0; i < q - 1; i++) {
    ac[i] = ac[i + 1];
    qr[i] = qr[i + 1];
  }
  qr[q - 1] = temp;
}

void display(int ac[], int qr[], int qrn) {
  int i;
  for (i = qrn - 1; i >= 0; i--)
    cout << ac[i];
  cout << " ";
  for (i = qrn - 1; i >= 0; i--)
    cout << qr[i];
}

int main(int argc, char **argv) {
  int mt[10], br[10], qr[10], sc, ac[10] = { 0 };
  int brn, qrn, i, qn, temp;
  cout<<"**Booth Algorithm Compiled by Sanjog Gautam**\n"<<endl;
  cout << "\n Number of multiplicand bit=";
  cin >> brn;
  cout << "\nmultiplicand=";

  for (i = brn - 1; i >= 0; i--)
      cin >> br[i]; //multiplicand
    for (i = brn - 1; i >= 0; i--)
      mt[i] = br[i];
    complement(mt, brn);
    cout << "\nNo. of multiplier bit=";
    cin >> qrn;
    sc = qrn;
    cout << "Multiplier=";

    for (i = qrn - 1; i >= 0; i--)
      cin >> qr[i];
    qn = 0;
    temp = 0;
    cout << "qn\tq[n+1]\t\tBR\t\tAC\tQR\t\tsc\n";
    cout << "\t\t\tinitial\t\t";

    display(ac, qr, qrn);
    cout << "\t\t" << sc << "\n";

    while (sc != 0) {
      cout << qr[0] << "\t" << qn;
      if ((qn + qr[0]) == 1) {
        if (temp == 0) {
          add(ac, mt, qrn);
          cout << "\t\tsubtracting BR\t";
          for (i = qrn - 1; i >= 0; i--)
            cout << ac[i];
          temp = 1;
        }
        else if (temp == 1) {
          add(ac, br, qrn);
          cout << "\t\tadding BR\t";
          for (i = qrn - 1; i >= 0; i--)
            cout << ac[i];
          temp = 0;
        }
        cout << "\n\t";
        ashr(ac, qr, qn, qrn);
      }
      else if (qn - qr[0] == 0)
        ashr(ac, qr, qn, qrn);
      display(ac, qr, qrn);
      cout << "\t";
      sc--;
      cout << "\t" << sc << "\n";
    }

  cout << "Result=";
  display(ac, qr, qrn);}
```

Output:



```
**Booth Algorithm Compiled by Sanjog Gautam**


 Number of multiplicand bit=4

multiplicand=0 1 0 1

No. of multiplier bit=4
Multiplier=0 0 1 1
qn      q[n+1]          BR              AC      QR              sc
                        initial         0000 0011               4
1       0               subtracting BR  1011
                        ashr            1101 1001               3
1       1               ashr            1110 1100               2
0       1               adding BR       0011
                        ashr            0001 1110               1
0       0               ashr            0000 1111               0
Result=0000 1111
--------------------------------
Process exited after 22.2 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

Booth’s Algorithm is an efficient way to multiply signed binary numbers using shifts and additions. It handles both positive and negative inputs with ease and mimics how real hardware performs multiplication

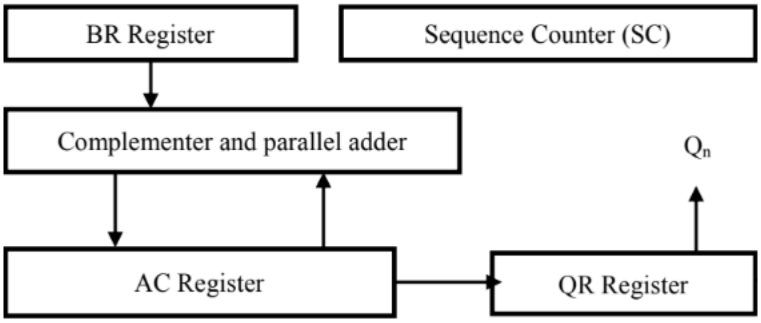
Experiment 5: Restoring Division Algorithm

Objective:

1. To implement Restoring Division Algorithm.

Theory:

Hardware Implementation



Algorithm:

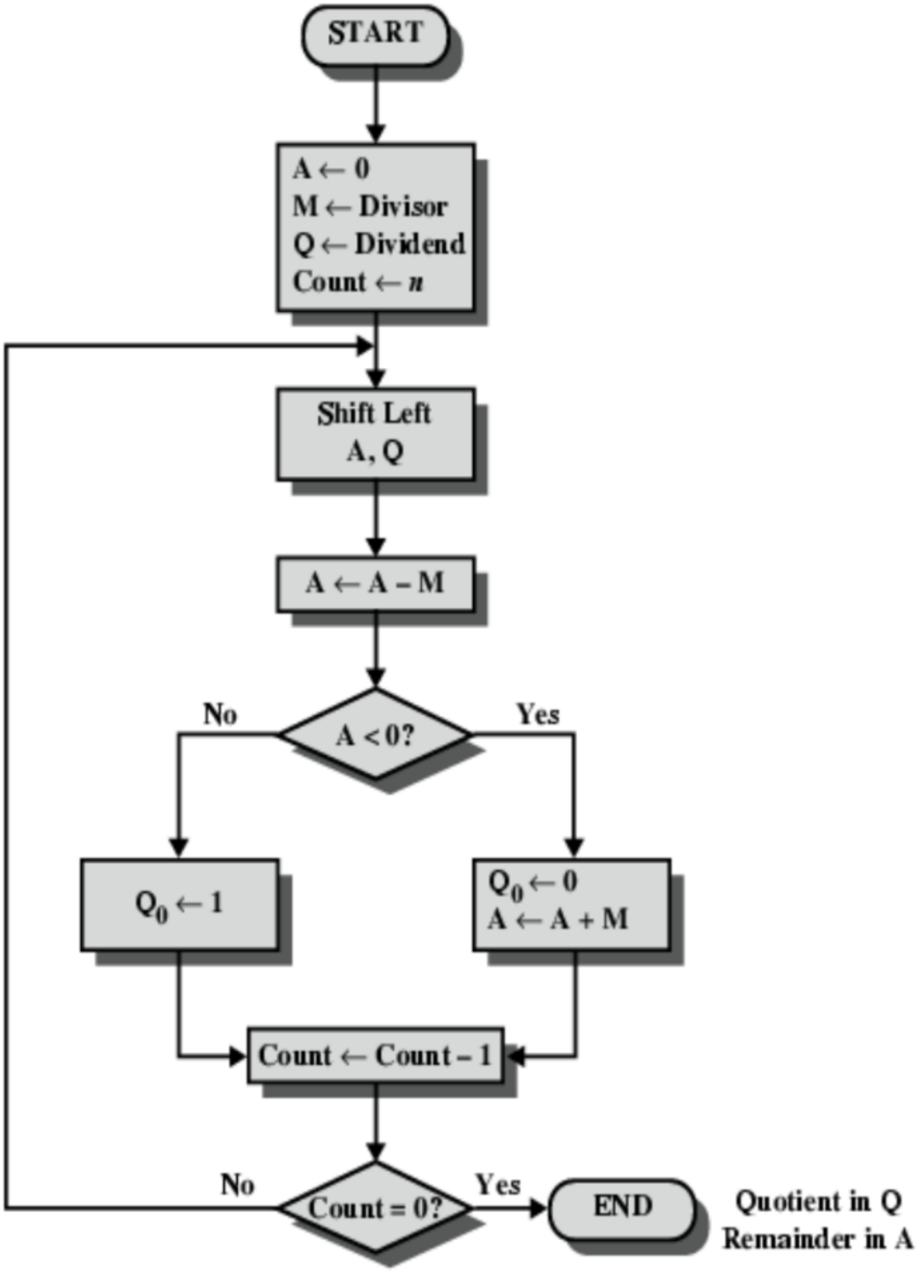
Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and counter to n where n is the number of bits in the dividend.

Step 2: Shift A, Q left one binary position.

Step 3: Subtract M from A placing answer back in A. If sign of A is 1, set Q to zero and add M back to A (restore A). If sign of A is 0, set Q to 1.

Step 4: Decrease counter; if counter > 0, repeat process from step 2 else stop the process. The final remainder will be in A and quotient will be in Q.

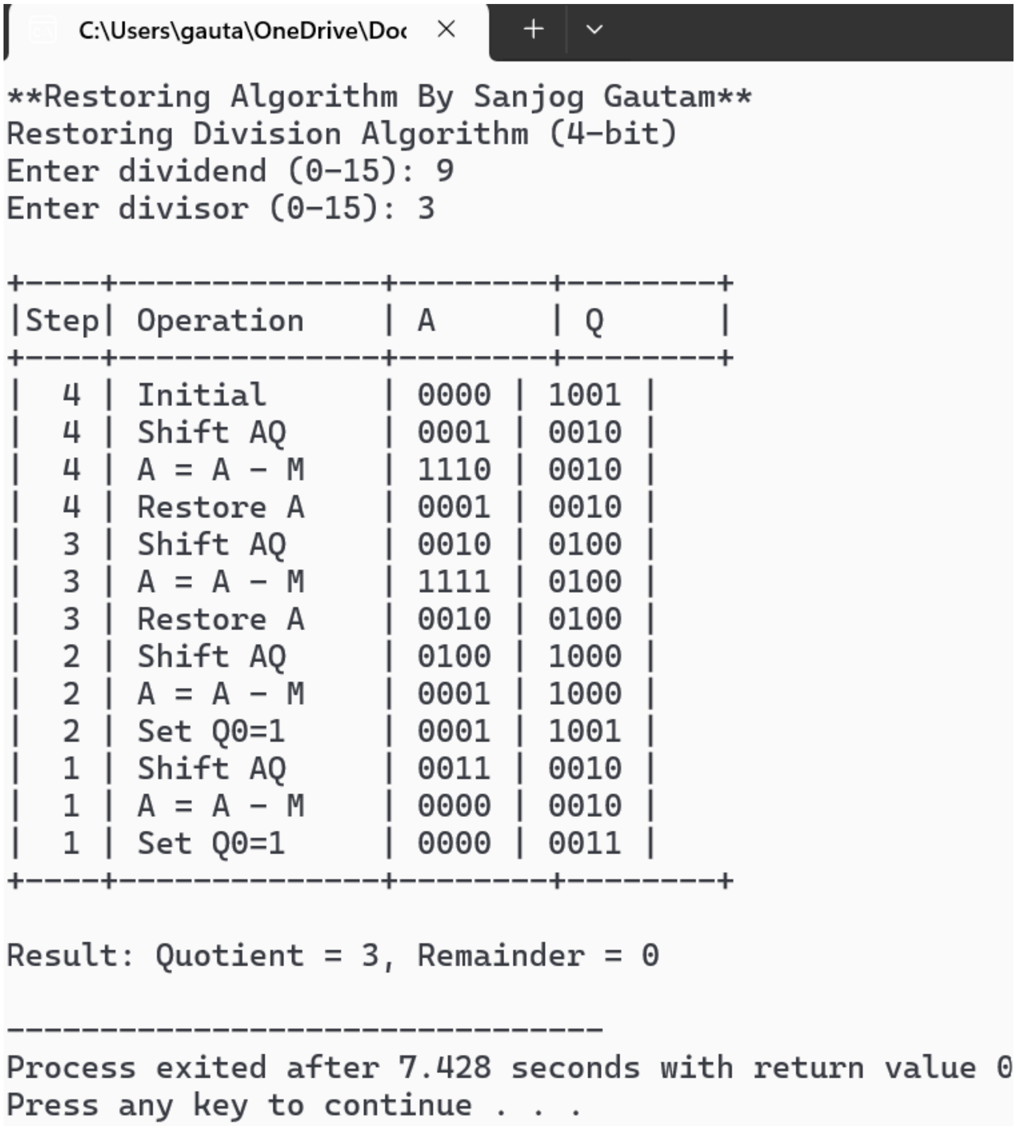
Flowchart:


START
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left
A, Q
A ← A – M
No
A < 0?
Yes
Q0 ← 1
Q0 ← 0
A ← A + M
Count ← Count – 1
No
Count = 0?
Yes
END
Quotient in Q
Remainder in A

Source code:

```
#include <stdio.h>
void print_binary(int num, int bits) {
    int i;
    for (i = bits - 1; i >= 0; i--) {
        printf("%d", (num >> i) & 1);
    }
}
void print_step(int step, const char* op, int A, int Q, int bits) {
    printf("| %2d | %-12s | ", step, op);
    print_binary(A, bits);
    printf(" | ");
    print_binary(Q, bits);
    printf(" |\n");
}
int main() {
    int dividend, divisor;
    int bits = 4;
    int A, Q, M;
    int step;
    printf("**Restoring Algorithm By Sanjog Gautam**\n");
    printf("Restoring Division Algorithm (4-bit)\n");
    printf("Enter dividend (0-15): ");
    scanf("%d", &dividend);
    printf("Enter divisor (0-15): ");
    scanf("%d", &divisor);
    A = 0;
    Q = dividend;
    M = divisor;
    step = bits;
    printf("\n+----+--------------+--------+--------+\n");
    printf("|Step| Operation    | A      | Q      |\n");
    printf("+----+--------------+--------+--------+\n");
    print_step(step, "Initial", A, Q, bits);

    for (step = bits; step > 0; step--) {
        // Shift AQ left
        A = (A << 1) | ((Q >> (bits-1)) & 1);
        Q <<= 1;
        print_step(step, "Shift AQ", A, Q, bits);

        // Subtract M from A
        A -= M;
        print_step(step, "A = A - M", A, Q, bits);

        if (A < 0) {
            A += M; // Restore
            print_step(step, "Restore A", A, Q, bits);
        } else {
            Q |= 1; // Set LSB
            print_step(step, "Set Q0=1", A, Q, bits);
        }
    }

    printf("+----+--------------+--------+--------+\n");
    printf("\nResult: Quotient = %d, Remainder = %d\n", Q & 0xF, A & 0xF); // Mask to 4 bits
    return 0;
}
```

Output:



```
**Restoring Algorithm By Sanjog Gautam**
Restoring Division Algorithm (4-bit)
Enter dividend (0-15): 9
Enter divisor (0-15): 3

+----+--------------+--------+--------+
|Step| Operation    | A      | Q      |
+----+--------------+--------+--------+
|  4 | Initial      | 0000 | 1001 |
|  4 | Shift AQ     | 0001 | 0010 |
|  4 | A = A - M    | 1110 | 0010 |
|  4 | Restore A    | 0001 | 0010 |
|  3 | Shift AQ     | 0010 | 0100 |
|  3 | A = A - M    | 1111 | 0100 |
|  3 | Restore A    | 0010 | 0100 |
|  2 | Shift AQ     | 0100 | 1000 |
|  2 | A = A - M    | 0001 | 1000 |
|  2 | Set Q0=1     | 0001 | 1001 |
|  1 | Shift AQ     | 0011 | 0010 |
|  1 | A = A - M    | 0000 | 0010 |
|  1 | Set Q0=1     | 0000 | 0011 |
+----+--------------+--------+--------+

Result: Quotient = 3, Remainder = 0

--------------------------------
Process exited after 7.428 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

The Restoring Division Algorithm provides a systematic method to divide binary numbers using shifting and subtraction. It is simple to implement in C and effectively handles unsigned binary division, giving accurate quotient and remainder.