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

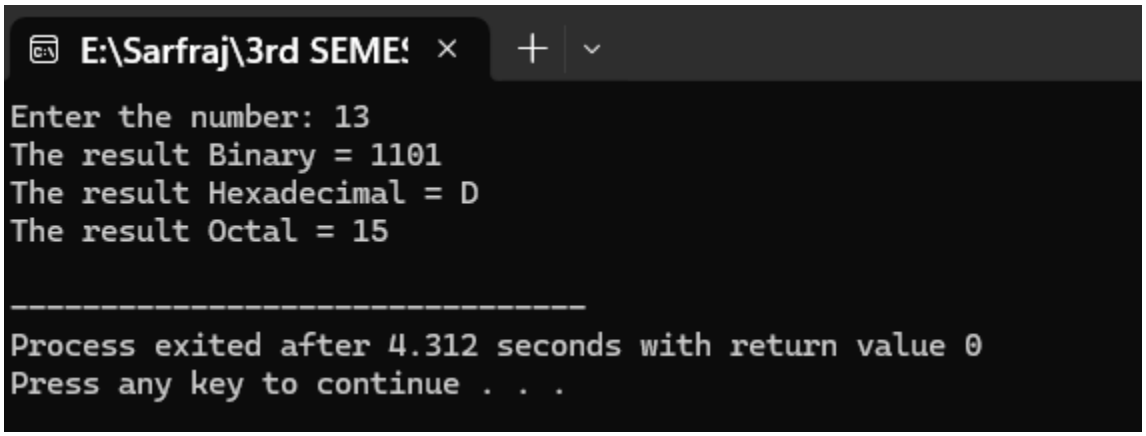
numbers

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("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");
      }
}
```

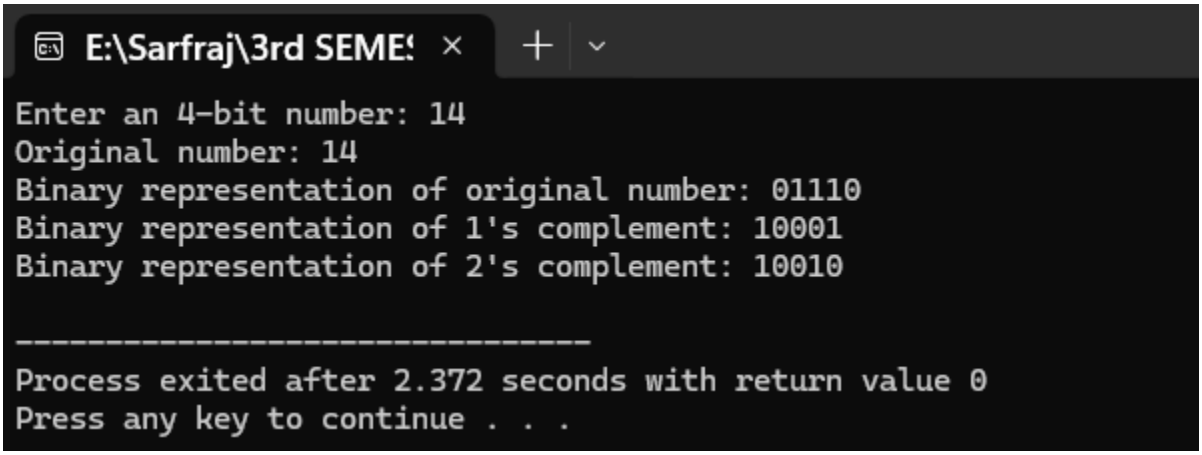
Output:


E:\Sarfraj\3rd SEME
Enter the number: 13
The result Binary = 1101
The result Hexadecimal = D
The result Octal = 15
Process exited after 4.312 seconds with return value 0
Press any key to continue . . .

Source code:

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

int main()
{
     char num;
    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:


E:\Sarfraj\3rd SEME
Enter an 4-bit number: 14
Original number: 14
Binary representation of original number: 01110
Binary representation of 1's complement: 10001
Binary representation of 2's complement: 10010

Process exited after 2.372 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

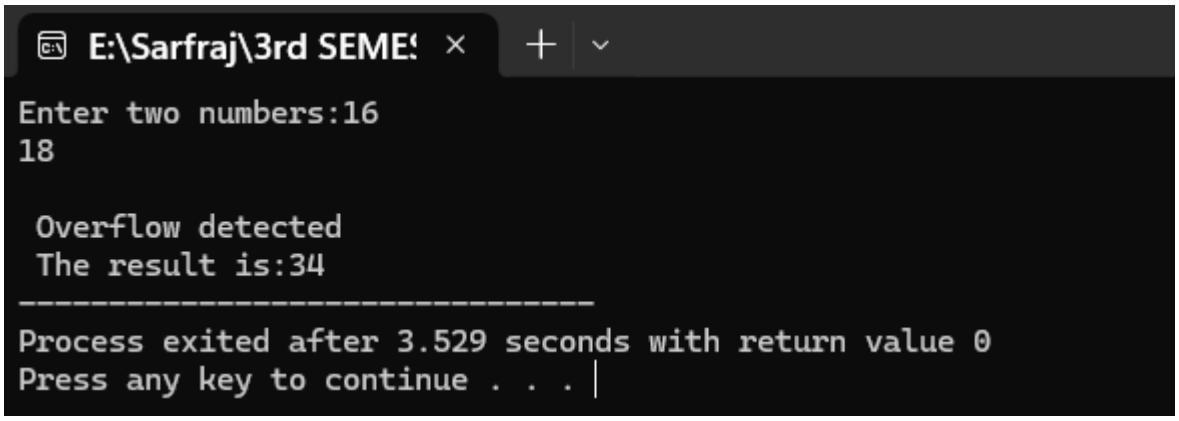
output of gate is 1.

v. Stop

Source code:

```
#include <stdio.h>
int main()
{
      int num1,num2,r,as,bs,rs;
      printf("Enter two numbers:");
      scanf("%d%d", &num1,&num2);
      r=num1+num2;
      as=(num1>>4)&1;
      bs=(num2>>4)&1;
      rs=(r>>4)&1;
      if(((as==bs)&&(as==rs))||(as!=bs))
      {
            printf("\No Overflow \n Result=%d",r);
      }
      else
      {
            printf("\n Overflow detected\n The result is:%d",r);
      }
}
```

Output:


E:\Sarfraj\3rd SEME
Enter two numbers:16
18
Overflow detected
The result is:34
Process exited after 3.529 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
```

Experiment i: Introduction to VHDL

Objective:

1. To implement the and gates 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:

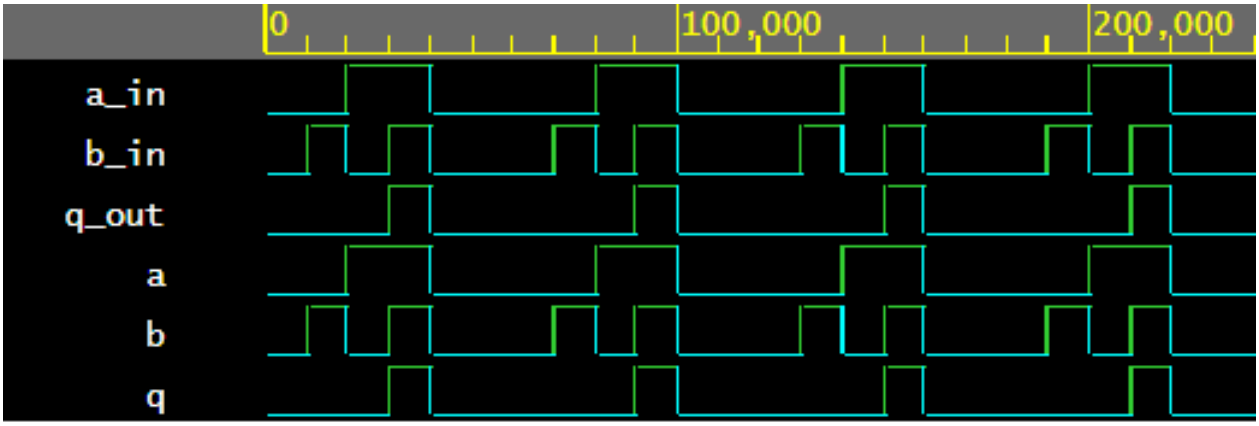
```
-- Testbench for AND gate
library IEEE;
use IEEE.std_logic_1164.all;

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:



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:

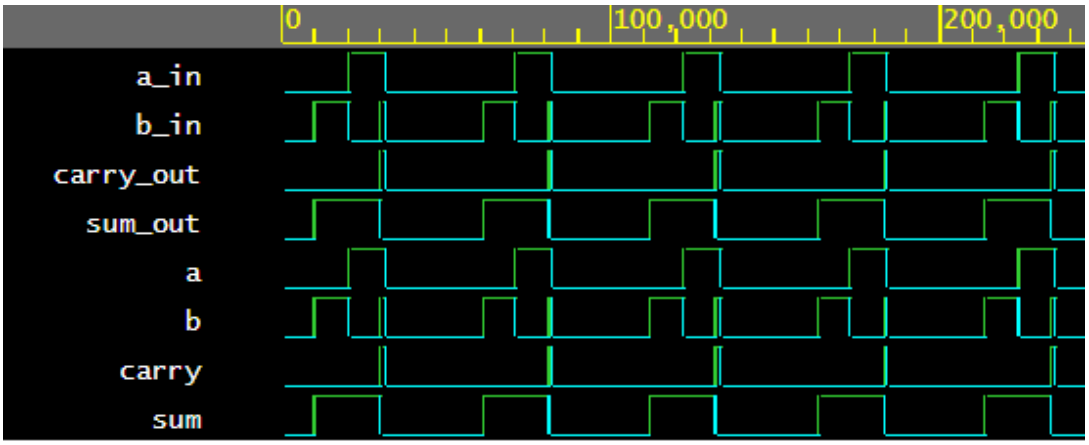
Testbench:

```
-- Testbench for halfadder 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 halfadder 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 iii: 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:

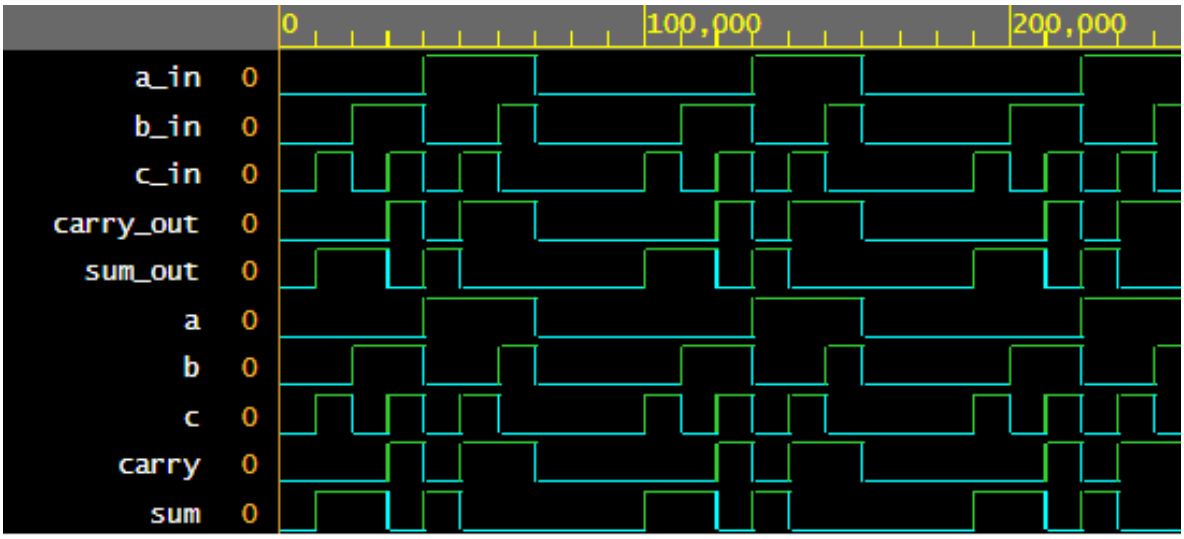
```
-- Testbench for fulladder gate
library IEEE;
use IEEE.std_logic_1164.all;

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.

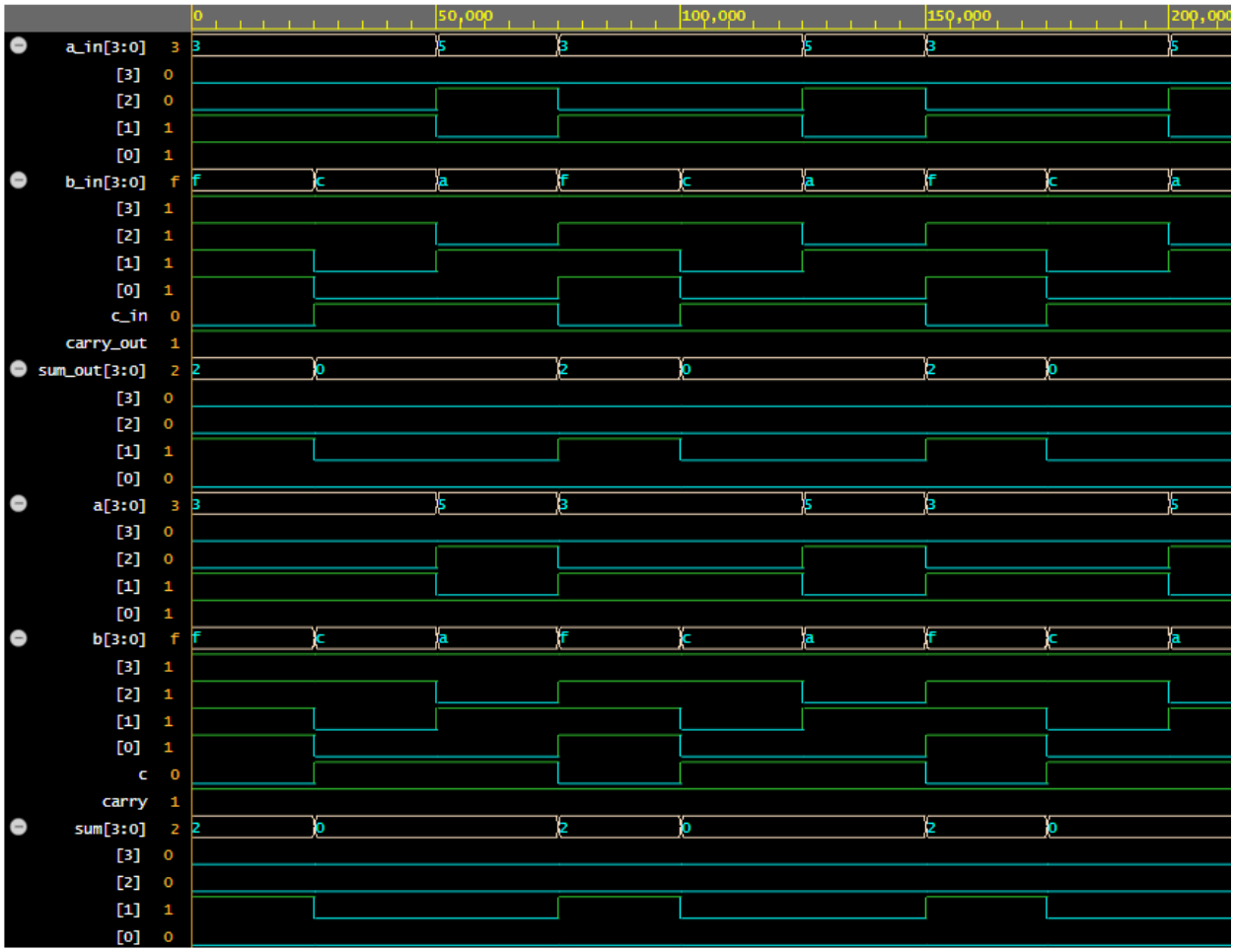
Architecture:

```
-- 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:


0
50,000
100,000
150,000
200,000
a_in[3:0] 3
[3] 0
[2] 0
[1] 1
[0] 1
b_in[3:0] f
[3] 1
[2] 1
[1] 1
[0] 1
c_in 0
carry_out 1
sum_out[3:0] 2
[3] 0
[2] 0
[1] 1
[0] 0
a[3:0] 3
[3] 0
[2] 0
[1] 1
[0] 1
b[3:0] f
[3] 1
[2] 1
[1] 1
[0] 1
c 0
carry 1
sum[3:0] 2
[3] 0
[2] 0
[1] 1
[0] 0

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:

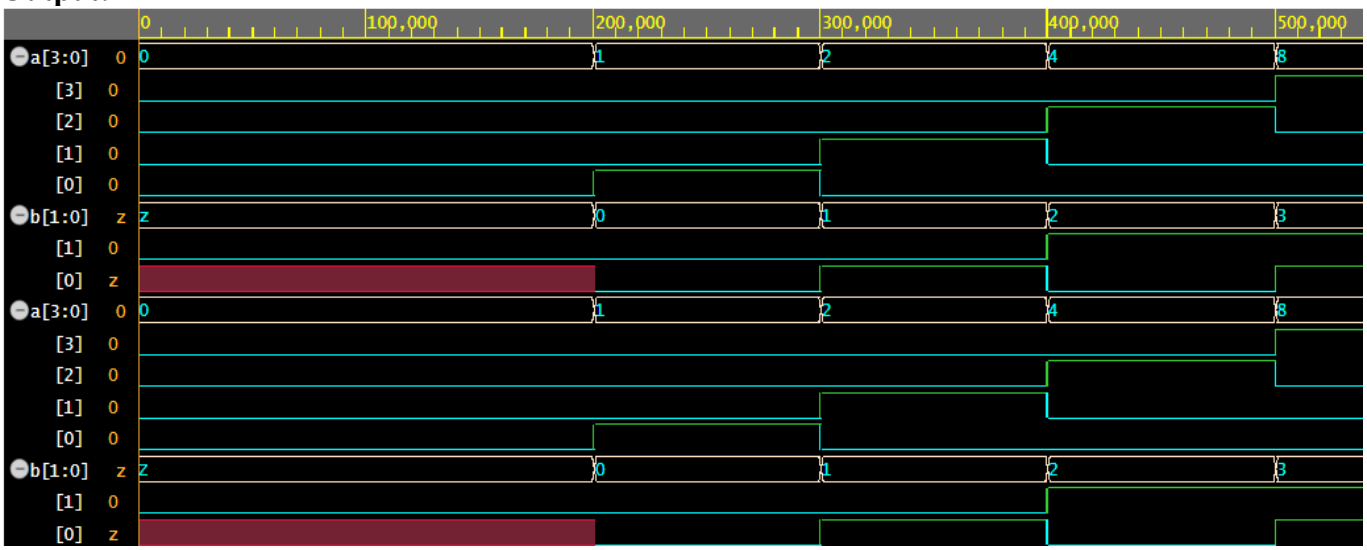
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encoder4_2 is
 Port ( a: in STD_LOGIC_VECTOR (3 downto 0);
 b: out STD_LOGIC_VECTOR (1 downto 0));
end Encoder4_2;
architecture rtl of Encoder4_2 is
begin
 process(a)
 begin
 if (a="0001") then
 b <= "00";
 elsif (a="0010") then
 b <= "01";
 elsif (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
[3] 0
[2] 0
[1] 0
[0] 0
b[1:0] z
[1] 0
[0] z
a[3:0] 0
[3] 0
[2] 0
[1] 0
[0] 0
b[1:0] z
[1] 0
[0] z