EE529 Embedded Systems

Lab Assignment 1

Student Name : Ayan Garg

Roll Number : B23484

School of Computing and Electrical Engineering (SCEE)
Indian Institute Of Technology Mandi
January 26, 2026

Contents

1 Objective

1.1 Question 1

The objective of this experiment is to understand and implement a control-driven digital switching system. This includes designing a 2×2 switch that routes input data to output ports based on a multi-bit control signal, interpreting functional truth tables, and realizing the design using VHDL.

1.2 Question 2

The objective of this experiment is to design and implement a configurable arithmetic circuit capable of performing multiple operations—addition, subtraction, increment, and decrement—using a control signal. The experiment emphasizes hierarchical digital design by starting from a half-adder dataflow model and building complex arithmetic units using structural modeling. Additionally, it aims to compare design approaches using multiple adders versus a single adder in terms of modularity and hardware utilization.

1.3 Question 3

The objective of this experiment is to design an 8-bit signed saturation adder that emulates the saturation behavior of an analog amplifier. The goal is to detect arithmetic overflow conditions and limit the output to the maximum or minimum representable values accordingly. This experiment helps in understanding overflow handling, signed arithmetic, and practical digital signal processing requirements through VHDL implementation.

2 Theory

2.1 Question 1

A digital switch is a combinational logic circuit used to route input data signals to different output ports based on control signals. Such switches are fundamental components in digital communication systems, embedded processors, and on-chip interconnects, where flexible and efficient data routing is required.

A 2×2 switch consists of two input data ports, $x(0)$ and $x(1)$, two output ports, $y(0)$ and $y(1)$, and a 2-bit control signal ctrl$[1:0]$. The switch operates as a purely combinational circuit; therefore, the outputs depend only on the current values of the inputs and the control signal, with no memory elements involved.

The operation of the 2×2 switch is controlled by a 2-bit control signal ctrl$[1:0]$. Table 1 summarizes the different switching modes and the corresponding output behavior.

Table 1: Modes of Operation of the 2×2 Switch

Control Signal (ctrl)	**Mode of Operation**	$y(0)$	$y(1)$
00	Pass Mode	$x(0)$	$x(1)$
01	Cross Mode	$x(1)$	$x(0)$
10	Broadcast $x(0)$	$x(0)$	$x(0)$
11	Broadcast $x(1)$	$x(1)$	$x(1)$

2.2 Question 2

An arithmetic circuit is a fundamental combinational building block used in digital systems to perform basic numerical operations. In this experiment, a configurable 16-bit arithmetic circuit is designed to perform four operations—addition, subtraction, increment, and decrement—on unsigned operands. The specific operation is selected using a 2-bit control signal, making the circuit suitable for use in simple arithmetic logic units (ALUs) and embedded datapaths.

The supported operations are summarized as follows:

- Addition: $a + b$
- Subtraction: $a - b$
- Increment: $a + 1$
- Decrement: $a - 1$

Both input operands a and b are treated as 16-bit unsigned numbers, and all arithmetic is performed using two's-complement binary representation. Since the design is unsigned, overflow and underflow result in wrap-around behavior, with the carry-out signal indicating overflow (for addition) or borrow (for subtraction).

Hierarchical Design Approach

The arithmetic circuit is constructed hierarchically to emphasize modular and reusable digital design principles. At the lowest level, a half adder is implemented using basic logic gates to generate sum and carry outputs. Two half adders are then combined to form a 1-bit full adder, which serves as the fundamental arithmetic unit.

Sixteen such full adders are cascaded in a ripple-carry configuration to realize a 16-bit full adder. This adder forms the core computational block used throughout the design for implementing higher-level arithmetic functions.

Design Using Multiple Arithmetic Blocks (Q2a)

In the first approach, separate hardware blocks are used for each arithmetic operation. Two 16-bit adders are employed to perform addition and subtraction, while dedicated incrementor and decrementor circuits are used for $a+1$ and $a-1$, respectively. Subtraction is implemented using two's-complement arithmetic by inverting the operand b and adding a carry-in of one.

The outputs of these arithmetic units are connected to a 4:1 multiplexer controlled by the 2-bit control signal. This design approach is highly modular and easy to understand, but it incurs higher hardware cost due to the presence of multiple arithmetic units.

Optimized Design Using a Single Adder (Q2b)

In the second approach, the arithmetic circuit is optimized to use only a single 16-bit adder. This is achieved by observing that all required operations can be expressed in the general form:

$$Y = A + B_{\text{sel}} + C_{\text{in}}$$

By appropriately selecting the second adder input B_{sel} and the carry-in C_{in} using control logic, the same adder can perform addition, subtraction, increment, and decrement operations. For example, subtraction is performed by feeding the one's complement of b to the adder and asserting the carry-in, while decrement is achieved by adding a constant value of all ones.

This single-adder design significantly reduces hardware utilization while maintaining functional correctness. Such an approach closely resembles practical ALU designs used in processors, where control logic is used to maximize hardware reuse and efficiency.

2.3 Question 3

Analog amplifiers exhibit inherent output limiting due to finite supply rails. When the amplified signal exceeds the allowable voltage range, the output saturates at either the positive supply (+Vcc) or the negative supply (Vcc). This saturation behavior prevents unbounded output growth and ensures stable operation.

In digital signal processing (DSP) systems employing fixed-point arithmetic, an analogous limitation arises due to finite word length. Conventional two's-complement addition results in wrap-around overflow, where arithmetic overflow causes the result to cycle through the representable range. Such behavior is undesirable in DSP applications, as it introduces large numerical errors and signal distortion.

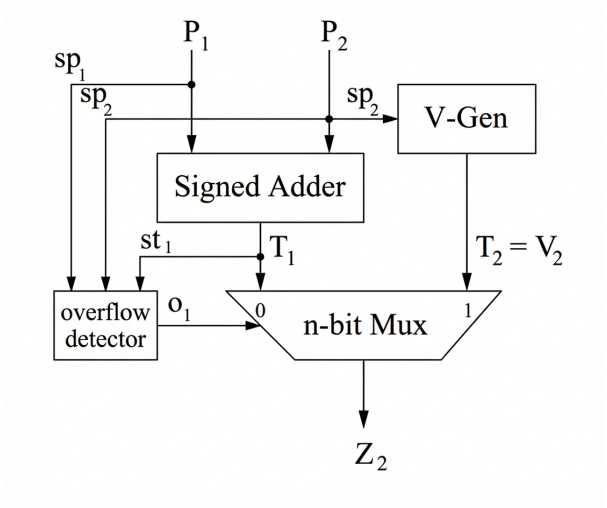

Figure 1: Proposed Circuit Architecture for Signed Saturation Adder

Figure 1 illustrates the proposed 2-input signed saturation adder architecture. The input operands P_1 and P_2 are first processed by a non-saturating signed adder, which computes the temporary sum T_1 using extended precision. The sign bits of the operands (s_{p_1}, s_{p_2}) and the sign of the temporary sum (s_{t_1}) are applied to the overflow detector, which determines whether signed overflow has occurred. In parallel, the saturation value generator (V-Gen) produces the saturation constant $T_2 = V_2$, corresponding to the maximum positive or maximum negative representable value depending on the operand sign. An n-bit multiplexer selects between the normal arithmetic result T_1 and the saturation value T_2 based on the overflow control signal o_1, thereby producing the final saturated output Z_2.

2.3.1 Overflow Detection

For two's-complement signed addition, overflow occurs only when both operands have the same sign and the sign of the result differs. The overflow condition is given by

$$o_1 = (s_{p_1} = s_{p_2}) \wedge (s_{t_1} \neq s_{p_1}) \tag{1}$$

The equivalent Boolean expression is

$$o_1 = (s_{p_1} \cdot s_{p_2} \cdot \overline{s_{t_1}}) + (\overline{s_{p_1}} \cdot \overline{s_{p_2}} \cdot s_{t_1}) \tag{2}$$

Table 2: Signed Overflow Condition for Two's-Complement Addition

s_{p_1}	s_{p_2}	s_{t_1}	o_1
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0
0	1	X	0
1	0	X	0

This overflow detection scheme ensures correct saturation behavior and eliminates wrap-around errors, making the digital adder behavior analogous to that of an analog amplifier operating within supply limits.

3 VHDL Implementation

3.1 Question 1

3.1.1 RTL / Behavioral VHDL Description

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

entity lab1 is
    Port (
        x1  : in  std_logic;
        x2  : in  std_logic;
        sel : in  std_logic_vector(1 downto 0);
        y1  : out std_logic;
        y2  : out std_logic
    );
end lab1;

architecture Behavioral of lab1 is
begin
    process(x1, x2, sel)
    begin
        case sel is
            when "00" =>
                y1 <= x1;
                y2 <= x2;

            when "01" =>
                y1 <= x2;
                y2 <= x1;

            when "10" =>
                y1 <= x1;
                y2 <= x1;

             when others =>
                y1 <= x2;
                y2 <= x2;
        end case;
    end process;
end Behavioral;
```

Listing 1: VHDL Code for 2×2 Switch

3.1.2 Testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_lab1 is
end tb_lab1;

architecture Behavioral of tb_lab1 is

    component lab1
        Port (
            x1  : in  std_logic;
            x2  : in  std_logic;
            sel : in  std_logic_vector(1 downto 0);
            y1  : out std_logic;
            y2  : out std_logic
        );
    end component;

    signal x1  : std_logic := '0';
    signal x2  : std_logic := '0';
    signal sel : std_logic_vector(1 downto 0) := "00";
    signal y1  : std_logic;
    signal y2  : std_logic;

begin

    uut: lab1
        port map (
            x1  => x1,
            x2  => x2,
            sel => sel,
            y1  => y1,
            y2  => y2
        );

    stim_proc: process
    begin

        x1 <= '0'; x2 <= '1'; sel <= "00";
        wait for 10 ns;

        sel <= "01";
        wait for 10 ns;

        sel <= "10";
        wait for 10 ns;

        sel <= "11";
        wait for 10 ns;

        wait;
    end process;

end Behavioral;
```

Listing 2: VHDL Test Bench for 2×2 Switch

3.2 Question 2a

3.2.1 RTL / Behavioral VHDL Description

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity HA is
    Port (
        A     : in  STD_LOGIC;
        B     : in  STD_LOGIC;
        SUM   : out STD_LOGIC;
        CARRY : out STD_LOGIC
    );
end HA;

architecture Behavioral of HA is
begin
    SUM   <= A xor B;
    CARRY <= A and B;
end Behavioral;
```

Listing 3: VHDL Code for Half Adder

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FA is
    Port (
```

```
        A    : in  STD_LOGIC;
        B    : in  STD_LOGIC;
        Cin  : in  STD_LOGIC;
        SUM  : out STD_LOGIC;
        Cout : out STD_LOGIC
    );
end FA;

architecture Structural of FA is
    component HA
        Port (
            A     : in  STD_LOGIC;
            B     : in  STD_LOGIC;
            SUM   : out STD_LOGIC;
            CARRY : out STD_LOGIC
        );
    end component;

    signal S1, C1, C2 : STD_LOGIC;

begin

    HA1: HA
        port map (
            A     => A,
            B     => B,
            SUM   => S1,
            CARRY => C1
        );

    HA2: HA
        port map (
            A     => S1,
            B     => Cin,
            SUM   => SUM,
            CARRY => C2
        );

    Cout <= C1 or C2;

end Structural;
```

Listing 4: VHDL Code for 1 bit Full Adder

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

entity FA_16bit is
    Port (
        A    : in  STD_LOGIC_VECTOR(15 downto 0);
        B    : in  STD_LOGIC_VECTOR(15 downto 0);
        Cin  : in  STD_LOGIC;
        SUM  : out STD_LOGIC_VECTOR(15 downto 0);
        Cout : out STD_LOGIC
    );
end FA_16bit;

architecture Structural of FA_16bit is

    component FA
        Port (
            A    : in  STD_LOGIC;
            B    : in  STD_LOGIC;
            Cin  : in  STD_LOGIC;
            SUM  : out STD_LOGIC;
            Cout : out STD_LOGIC
        );
    end component;

    signal C : STD_LOGIC_VECTOR(16 downto 0);

begin
    C(0) <= Cin;
    GEN_FA : for i in 0 to 15 generate
        FA_i : FA
            port map (
                A    => A(i),
                B    => B(i),
                Cin  => C(i),
                SUM  => SUM(i),
                Cout => C(i+1)
            );
    end generate;
    Cout <= C(16);
end Structural;
```

Listing 5: VHDL Code for 16 bit Full Adder

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

entity Incrementor_16bit is
    Port (
        A    : in  STD_LOGIC_VECTOR(15 downto 0);
        Y    : out STD_LOGIC_VECTOR(15 downto 0);
```

```
        Cout : out STD_LOGIC
    );
end Incrementor_16bit;

architecture Structural of Incrementor_16bit is

    component FA_16bit
        Port (
            A    : in  STD_LOGIC_VECTOR(15 downto 0);
            B    : in  STD_LOGIC_VECTOR(15 downto 0);
            Cin  : in  STD_LOGIC;
            SUM  : out STD_LOGIC_VECTOR(15 downto 0);
            Cout : out STD_LOGIC
        );
    end component;

    constant ZERO : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');

begin

    FA_INC : FA_16bit
        port map (
            A    => A,
            B    => ZERO,
            Cin  => '1',
            SUM  => Y,
            Cout => Cout
        );

end Structural;
```

Listing 6: VHDL Code for 16 bit Incrementor

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

entity top_2a is
    Port (
        a    : in  STD_LOGIC_VECTOR(15 downto 0);
        b    : in  STD_LOGIC_VECTOR(15 downto 0);
        ctrl : in  STD_LOGIC_VECTOR(1 downto 0);
        y    : out STD_LOGIC_VECTOR(15 downto 0);
        cout : out STD_LOGIC
    );
end top_2a;

architecture Structural of top_2a is

    component FA_16bit
        Port (
            A    : in  STD_LOGIC_VECTOR(15 downto 0);
            B    : in  STD_LOGIC_VECTOR(15 downto 0);
            Cin  : in  STD_LOGIC;
            SUM  : out STD_LOGIC_VECTOR(15 downto 0);
            Cout : out STD_LOGIC
        );
    end component;

    component Incrementor_16bit
        Port (
            A    : in  STD_LOGIC_VECTOR(15 downto 0);
            Y    : out STD_LOGIC_VECTOR(15 downto 0);
            Cout : out STD_LOGIC
        );
    end component;

    component Decrementor_16bit
        Port (
            A    : in  STD_LOGIC_VECTOR(15 downto 0);
            Y    : out STD_LOGIC_VECTOR(15 downto 0);
            Bout : out STD_LOGIC
        );
    end component;

    -- Internal signals
    signal add_ab, sub_ab, inc_a, dec_a : STD_LOGIC_VECTOR(15 downto 0);
    signal c_add, c_sub, c_inc, c_dec   : STD_LOGIC;
    signal b_inv                        : STD_LOGIC_VECTOR(15 downto 0);

begin

    b_inv <= not b;

    ADD1 : FA_16bit
        port map (
            A    => a,
            B    => b,
            Cin  => '0',
            SUM  => add_ab,
            Cout => c_add
        );

    ADD2 : FA_16bit
        port map (
            A    => a,
            B    => b_inv,
            Cin  => '1',
```

```
            SUM  => sub_ab,
            Cout => c_sub
        );

    INC1 : Incrementor_16bit
        port map (
            A    => a,
            Y    => inc_a,
            Cout => c_inc
        );

    DEC1 : Decrementor_16bit
        port map (
            A    => a,
            Y    => dec_a,
            Bout => c_dec
        );

    process(ctrl, add_ab, sub_ab, inc_a, dec_a,
            c_add, c_sub, c_inc, c_dec)
    begin
        case ctrl is
            when "00" =>
                y    <= add_ab;
                cout <= c_add;

            when "01" =>
                y    <= sub_ab;
                cout <= c_sub;

            when "10" =>
                y    <= inc_a;
                cout <= c_inc;

            when others =>  -- "11"
                y    <= dec_a;
                cout <= c_dec;
        end case;
    end process;

end Structural;
```

Listing 7: VHDL Code for Top Module

3.2.2 Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library std;
use std.env.all;

entity top_2a_tb is
end top_2a_tb;

architecture Behavioral of top_2a_tb is

    signal a    : STD_LOGIC_VECTOR(15 downto 0);
    signal b    : STD_LOGIC_VECTOR(15 downto 0);
    signal ctrl : STD_LOGIC_VECTOR(1 downto 0);
    signal y    : STD_LOGIC_VECTOR(15 downto 0);
    signal cout : STD_LOGIC;

    component top_2a
        Port (
            a    : in  STD_LOGIC_VECTOR(15 downto 0);
            b    : in  STD_LOGIC_VECTOR(15 downto 0);
            ctrl : in  STD_LOGIC_VECTOR(1 downto 0);
            y    : out STD_LOGIC_VECTOR(15 downto 0);
            cout : out STD_LOGIC
        );
    end component;

begin

    DUT : top_2a
        port map (
            a    => a,
            b    => b,
            ctrl => ctrl,
            y    => y,
            cout => cout
        );

    -- Stimulus process
    stim_proc : process
    begin

        a    <= x"000A";
        b    <= x"0005";
        ctrl <= "00";
        wait for 10 ns;

        a    <= x"000A";
```

```
        b    <= x"0005";
        ctrl <= "01";
        wait for 10 ns;

        a    <= x"000F";
        b    <= x"0000";
        ctrl <= "10";
        wait for 10 ns;

        a    <= x"0001";
        ctrl <= "11";
        wait for 10 ns;

        stop;
    end process;

end Behavioral;
```

Listing 8: VHDL Test Bench for Q2a

3.3 Question 2b

3.3.1 RTL / Behavioral VHDL Description

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

entity top_2b is
    Port (
        a    : in  STD_LOGIC_VECTOR(15 downto 0);
        b    : in  STD_LOGIC_VECTOR(15 downto 0);
        ctrl : in  STD_LOGIC_VECTOR(1 downto 0);
        y    : out STD_LOGIC_VECTOR(15 downto 0);
        cout : out STD_LOGIC
    );
end top_2b;

architecture Structural of top_2b is

    component FA_16bit
        Port (
            A    : in  STD_LOGIC_VECTOR(15 downto 0);
            B    : in  STD_LOGIC_VECTOR(15 downto 0);
            Cin  : in  STD_LOGIC;
            SUM  : out STD_LOGIC_VECTOR(15 downto 0);
            Cout : out STD_LOGIC
        );
    end component;

    signal B_sel : STD_LOGIC_VECTOR(15 downto 0);
    signal Cin   : STD_LOGIC;

begin

    process(ctrl, a, b)
    begin
        case ctrl is
            when "00" =>
                B_sel <= b;
                Cin   <= '0';

            when "01" =>
                B_sel <= not b;
                Cin   <= '1';

            when "10" =>
                B_sel <= (others => '0');
                Cin   <= '1';

            when others =>
                B_sel <= (others => '1');
                Cin   <= '0';
        end case;
    end process;

    ADDER : FA_16bit
        port map (
            A    => a,
            B    => B_sel,
            Cin  => Cin,
            SUM  => y,
            Cout => cout
        );

end Structural;
```

Listing 9: VHDL Code for Top Module

3.3.2 Testbench

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library std;
use std.env.all;

entity top_2b_tb is
end top_2b_tb;

architecture Behavioral of top_2b_tb is
    signal a    : STD_LOGIC_VECTOR(15 downto 0);
    signal b    : STD_LOGIC_VECTOR(15 downto 0);
    signal ctrl : STD_LOGIC_VECTOR(1 downto 0);
    signal y    : STD_LOGIC_VECTOR(15 downto 0);
    signal cout : STD_LOGIC;
    component top_2b
        Port (
            a    : in  STD_LOGIC_VECTOR(15 downto 0);
            b    : in  STD_LOGIC_VECTOR(15 downto 0);
            ctrl : in  STD_LOGIC_VECTOR(1 downto 0);
            y    : out STD_LOGIC_VECTOR(15 downto 0);
            cout : out STD_LOGIC
        );
    end component;

begin

    DUT : top_2b
        port map (
            a    => a,
            b    => b,
            ctrl => ctrl,
            y    => y,
            cout => cout
        );

    stim_proc : process
    begin

        a    <= x"000A";
        b    <= x"0005";
        ctrl <= "00";
        wait for 10 ns;

        a    <= x"000A";
        b    <= x"0005";
        ctrl <= "01";
        wait for 10 ns;

        a    <= x"000F";
        b    <= x"0000";
        ctrl <= "10";
        wait for 10 ns;

        a    <= x"0001";
        ctrl <= "11";
        wait for 10 ns;

        stop;

    end process;
end Behavioral;
```

Listing 10: VHDL Test Bench for Testbench

3.4 Question 3

3.4.1 RTL / Behavioral VHDL Description

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity signed_adder is
    generic (
        N : integer := 8
    );
    port (
        a   : in  signed(N-1 downto 0);
        b   : in  signed(N-1 downto 0);
        sum : out signed(N downto 0)
    );
end signed_adder;

architecture Behavioral of signed_adder is
begin
    sum <= resize(a, N+1) + resize(b, N+1);
end Behavioral;
```

Listing 11: VHDL Code for Signed Adder

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity overflow_detect is
    generic (
        N : integer := 8
    );
    port (
        a       : in  signed(N-1 downto 0);
        b       : in  signed(N-1 downto 0);
        sum : in  signed(N downto 0);
        ovf     : out std_logic
    );
end overflow_detect;

architecture Behavioral of overflow_detect is
begin

    ovf <= '1' when
           (a(N-1) = b(N-1)) and (sum(N-1) /= a(N-1))
           else '0';

end Behavioral;
```

Listing 12: VHDL Code for Overflow Detector

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mux is
    generic (
        N : integer := 8
    );
    port (
        d0  : in  signed(N-1 downto 0);
        d1  : in  signed(N-1 downto 0);
        sel : in  std_logic;
        y   : out signed(N-1 downto 0)
    );
end mux;

architecture Behavioral of mux is
begin
    y <= d0 when sel = '0' else d1;
end Behavioral;
```

Listing 13: VHDL Code for 8-bit 2x1 Mux

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top is
    generic (
        N : integer := 8
    );
    port (
        a   : in  signed(N-1 downto 0);
        b   : in  signed(N-1 downto 0);
        y   : out signed(N-1 downto 0)
    );
end top;

architecture Structural of top is
    signal sum  : signed(N downto 0);
    signal ovf      : std_logic;
    signal sat_val  : signed(N-1 downto 0);
    signal sum_norm : signed(N-1 downto 0);

begin
    adder_inst : entity work.signed_adder
        generic map ( N => N )
        port map (
            a   => a,
            b   => b,
            sum => sum
        );

    sum_norm <= sum(N-1 downto 0);
    ovf_inst : entity work.overflow_detect
        generic map ( N => N )
        port map (
            a       => a,
            b       => b,
            sum => sum,
            ovf     => ovf
        );
```

```
40
41      sat_val <= to_signed( 2**(N-1)-1, N ) when a(N-1) = '0' else
42                 to_signed(-2**(N-1),   N );
43
44      mux_inst : entity work.mux
45          generic map ( N => N )
46          port map (
47              d0  => sum_norm,
48              d1  => sat_val,
49              sel => ovf,
50              y   => y
51          );
52
53  end Structural;
```

Listing 14: VHDL Code for Top Module

3.4.2 Testbench

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   entity tb_top is
6   end tb_top;
7
8   architecture Behavioral of tb_top is
9
10      constant N : integer := 8;
11
12      signal a   : signed(N-1 downto 0);
13      signal b   : signed(N-1 downto 0);
14      signal y   : signed(N-1 downto 0);
15
16  begin
17
18      dut : entity work.top
19          generic map ( N => N )
20          port map (
21              a => a,
22              b => b,
23              y => y
24          );
25
26      stim_proc : process
27      begin
28
29      a <= to_signed(0, N);
30      b <= to_signed(0, N);
31      wait for 1 ns;
32
33          a <= to_signed(10, N);
34          b <= to_signed(20, N);
35          wait for 10 ns;
36          assert y = to_signed(30, N)
37              report "ERROR: 10 + 20 failed"
38              severity error;
39
40          a <= to_signed(127, N);
41          b <= to_signed(1, N);
42          wait for 10 ns;
43          assert y = to_signed(127, N)
44              report "ERROR: Positive saturation failed"
45              severity error;
46
47          a <= to_signed(-128, N);
48          b <= to_signed(-1, N);
49          wait for 10 ns;
50          assert y = to_signed(-128, N)
51              report "ERROR: Negative saturation failed"
52              severity error;
53
54          a <= to_signed(-50, N);
55          b <= to_signed(-20, N);
56          wait for 10 ns;
57          assert y = to_signed(-70, N)
58              report "ERROR: -50 + -20 failed"
59              severity error;
60
61          a <= to_signed(50, N);
62          b <= to_signed(-20, N);
63          wait for 10 ns;
64          assert y = to_signed(30, N)
65              report "ERROR: 50 + -20 failed"
66              severity error;
67
68          report "All saturation adder test cases passed!" severity note;
69          wait;
70      end process;
71  end Behavioral;
```

Listing 15: VHDL Test Bench for Signed Saturation Adder

4 Results

4.1 Question 1

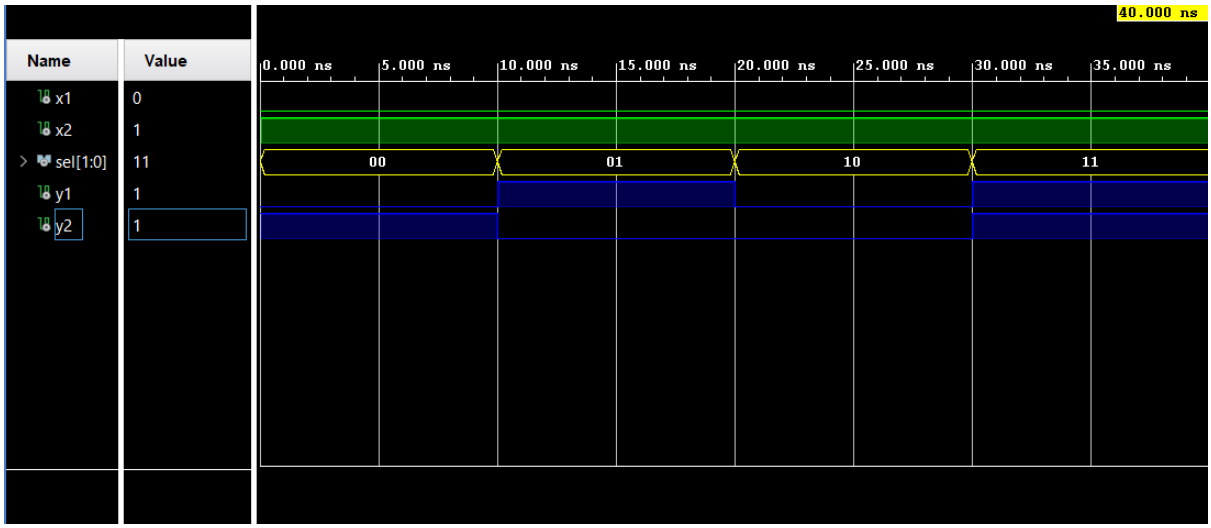

Figure 2: Simulation Result with Expected Output

4.2 Question 2a

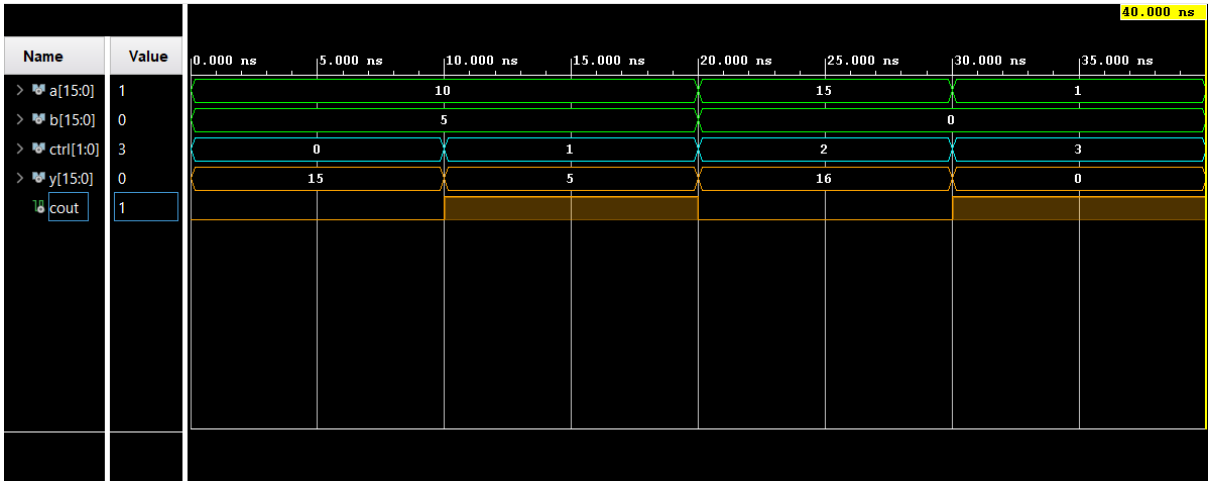

Figure 3: Simulation Result with Expected Output

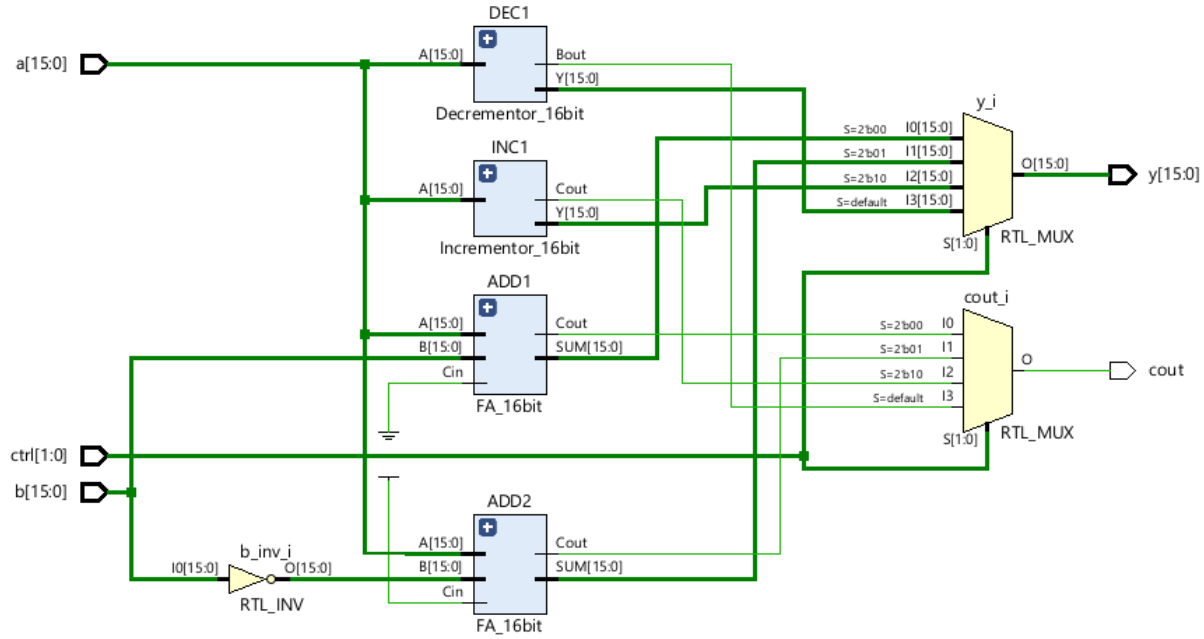

Figure 4: Schematic of Implemented Circuit

4.3 Question 2b

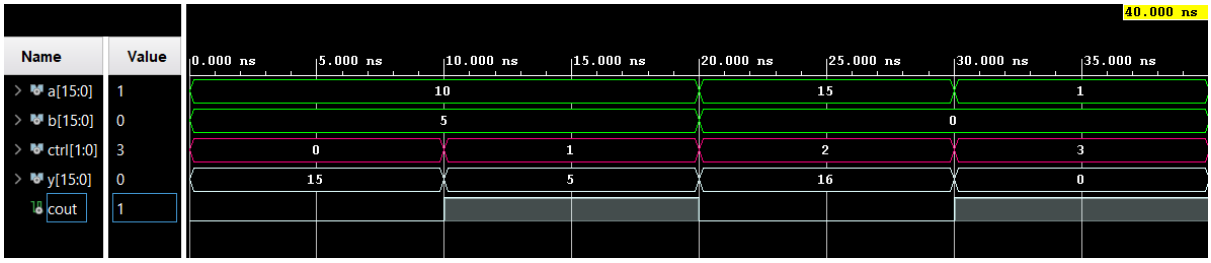

Figure 5: Simulation Result with Expected Output

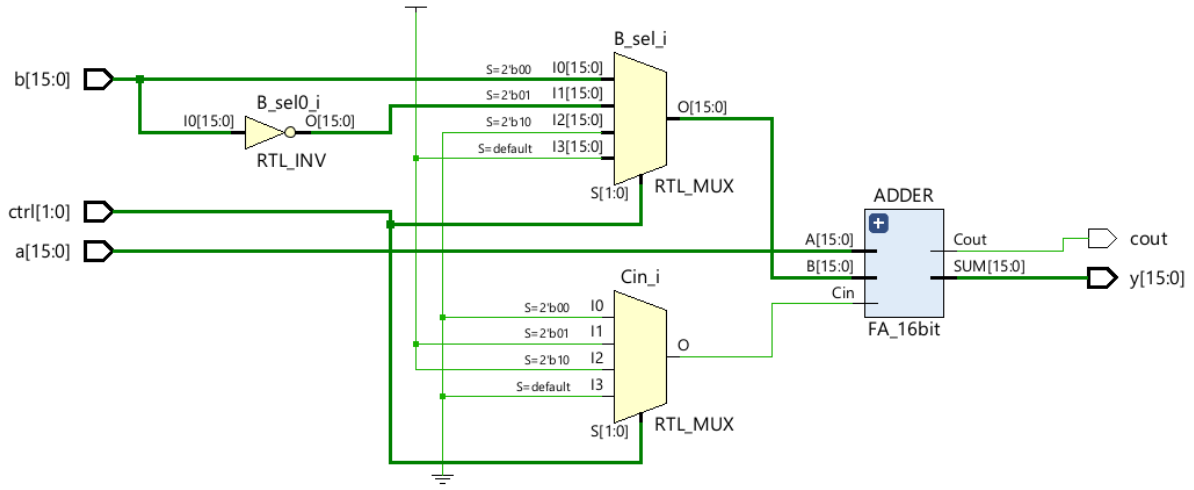

Figure 6: Schematic of Implemented Circuit with only 1 Adder

4.4 Question 3

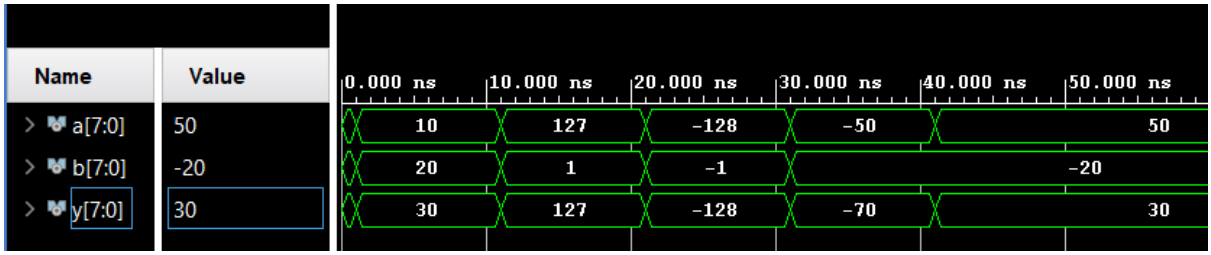

Figure 7: Simulation Result with Expected Output

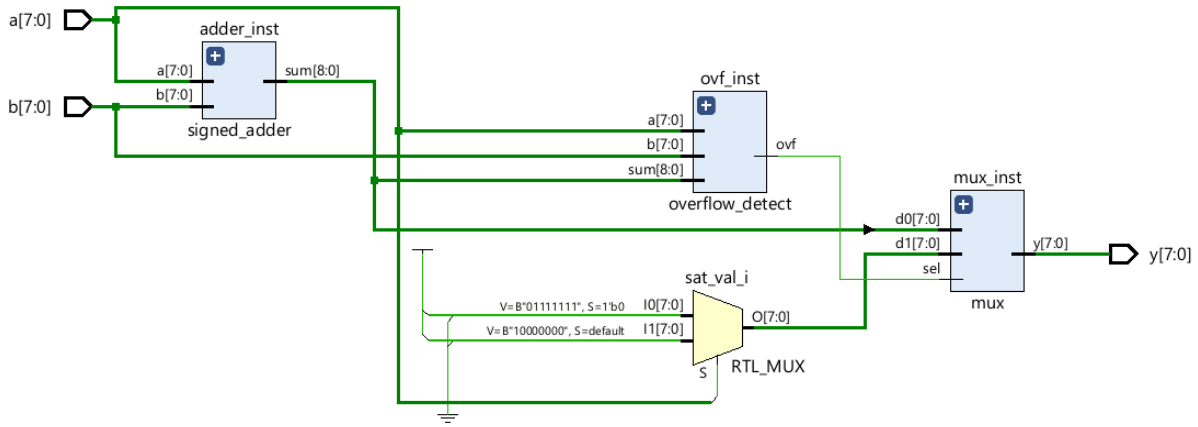

Figure 8: Schematic of Implemented Signed Saturation Adder

5 Conclusion

5.1 Question 1

In this experiment, a 2×2 digital switching circuit was successfully designed and implemented using VHDL. The switch operates purely as a combinational circuit, routing input data signals to output ports based on a 2-bit control signal. The design demonstrates how control signals can be used to dynamically alter data paths in digital systems.

The functional behavior of the switch exactly matches the expected truth table, as verified through simulation. The achieved switching functionality is summarized in Table 3.

Table 3: Truth Table Achieved for 2×2 Digital Switch

Control Signal (ctrl)	$y(0)$	$y(1)$
00	$x(0)$	$x(1)$
01	$x(1)$	$x(0)$
10	$x(0)$	$x(0)$
11	$x(1)$	$x(1)$

This experiment reinforces the concept of control-driven data routing and highlights the importance of multiplexing logic in embedded and digital communication systems.

5.2 Question 2

This experiment focused on the design and implementation of a configurable 16-bit unsigned arithmetic circuit capable of performing addition, subtraction, increment, and decrement operations. Two different architectural approaches were explored to understand the trade-offs between modularity and hardware efficiency.

In the first approach (Question 2a), separate arithmetic blocks—two adders, an incrementor, and a decrementor—were used to implement each operation independently. The desired output was selected using a multiplexer controlled by a 2-bit control signal. This design offers clarity and modularity but results in higher hardware utilization.

In the second approach (Question 2b), the arithmetic circuit was optimized to use only a single 16-bit adder. By carefully selecting the second operand and carry-in value based on the control signal, all four arithmetic operations were realized using the same adder. This approach significantly reduces hardware resources and closely resembles real-world ALU implementations.

The functional correctness of both designs was validated through simulation. The operations achieved for different control inputs are summarized in Table 4.

Table 4: Truth Table Achieved for Arithmetic Circuit (Unsigned)

Control Signal (ctrl)	Operation	Output Expression
00	Addition	$y = a + b$
01	Subtraction	$y = a - b$
10	Increment	$y = a + 1$
11	Decrement	$y = a - 1$

This experiment demonstrates how control logic can be leveraged to maximize hardware reuse without compromising functionality, an essential concept in efficient digital system design.

5.3 Question 3

The final experiment involved the design of an 8-bit signed saturation adder to emulate the output limiting behavior of an analog amplifier. Unlike conventional two's-complement adders that exhibit wrap-around on overflow, the saturation adder clamps the output to the maximum or minimum representable value when overflow occurs.

The design incorporates overflow detection logic based on operand and result sign bits, along with a multiplexer that selects either the normal sum or a saturation constant. Simulation results confirm that the circuit correctly detects both positive and negative overflow conditions and produces the expected saturated output.

The operational behavior of the signed saturation adder is summarized in Table 5.

Table 5: Truth Table Achieved for Signed Saturation Adder

Operand Signs	Overflow	Output Behavior	Result
Same sign	No	Normal addition	$a + b$
Positive overflow	Yes	Saturate to max	$+2^{N-1} - 1$
Negative overflow	Yes	Saturate to min	-2^{N-1}
Different signs	No	Normal addition	$a + b$

This experiment highlights the importance of overflow handling in fixed-point digital signal processing systems and demonstrates how saturation arithmetic improves numerical stability compared to wrap-around behavior.

6 References

References

[1] M. J. Schulte, P. I. Balzola, J. Ruan, and J. Glossner, "Parallel saturating multioperand adders," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, USA, Nov. 2000, pp. 172–179.