EE529 Embedded Systems

Group 1

Lab Assignment 4

Student Name : Ayan Garg and Om Maheshwari

Roll Number : B23484 and B23089

School of Computing and Electrical Engineering (SCEE)
Indian Institute Of Technology Mandi
February 21, 2026

Contents

1 Question 1

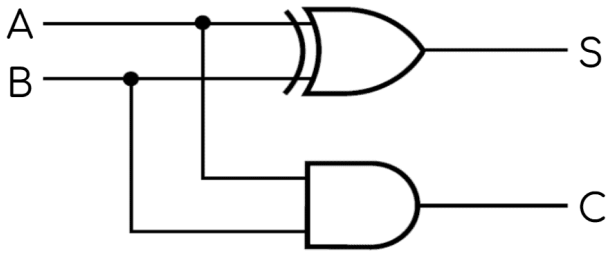


Figure 1: Schematic of Half Adder

1.1 VHDL Implementation of Half Adder

```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 dataflow of HA is
begin

    SUM   <= A xor B;
    CARRY <= A and B;

end dataflow;
```

Listing 1: VHDL Code for Half Adder

```
###Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { A }]; #IO_L19N_T3_VREF_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { B }]; #IO_L24P_T3_34 Sch=sw[1]

##LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { SUM }]; #IO_L23P_T3_35 Sch=led[0]
set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { CARRY }]; #IO_L23N_T3_35 Sch=led[1]
```

Listing 2: Constraint File for Half Adder

1.2 Results

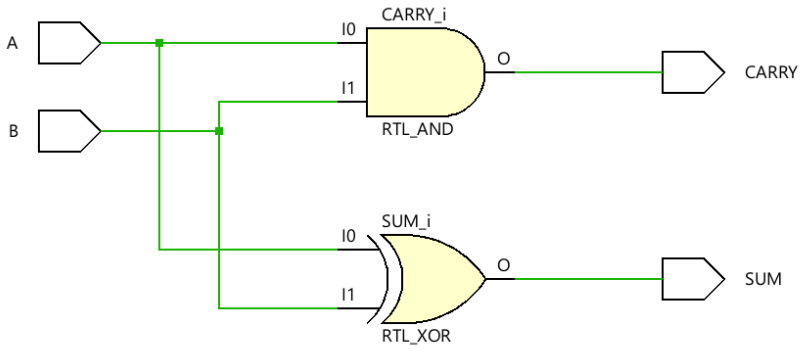


Figure 2: Realized Schematic for Half Adder

2 Question 2

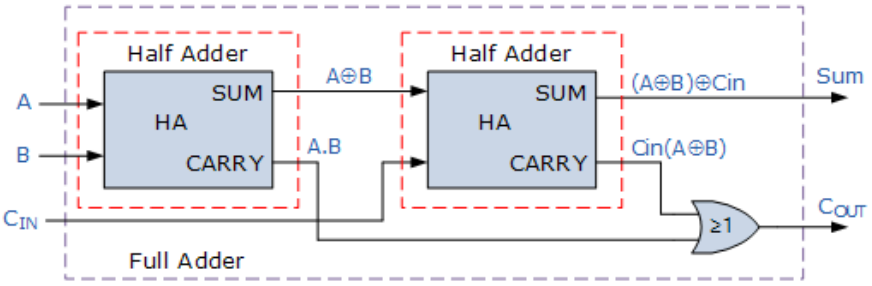


Figure 3: Schematic of Full Adder using Half Adders

2.1 VHDL Implementation of Full 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 3: VHDL Code for Full Adder

2.2 Results

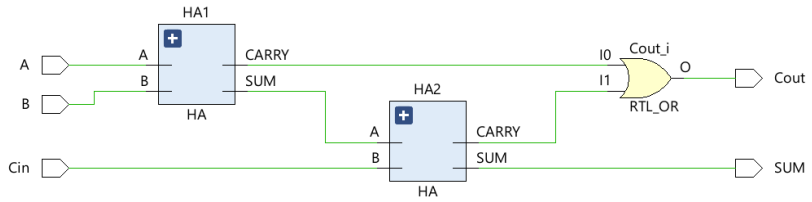


Figure 4: Realized Schematic for Full Adder

3 Question 3

3.1 4-bit Ripple Carry Adder (RCA)

3.1.1 Architecture

Four full adders are cascaded so that the carry-out of stage i feeds the carry-in of stage $i{+}1$, as shown in Figure 5.

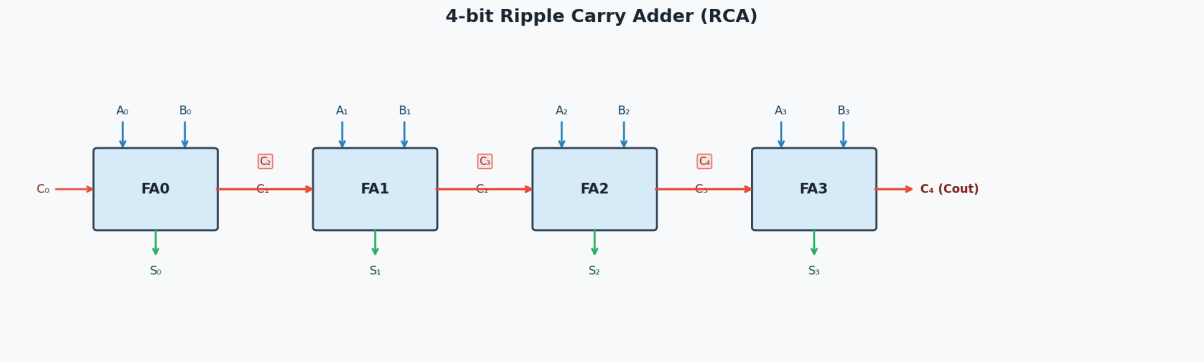


Figure 5: Block diagram of the 4-bit Ripple Carry Adder.

3.1.2 Carry Propagation

The n-bit RCA critical path traverses all n full adders serially:

$$t_{RCA} = n \cdot t_{FA}$$

For $n = 4$ and a typical full-adder gate delay of t_{FA}, the total delay is $4\,t_{FA}$. Each additional bit adds one full-adder delay, making the RCA *linearly slow* for large n.

3.2 Technology Schematic Observations

Examining the synthesised schematic in the CAD tool reveals:

- The carry chain is a long path of XOR and AND–OR gates connected end-to-end, clearly visible as a horizontal daisy-chain.
- The synthesis tool may automatically infer a carry-chain primitive (e.g. FPGA `CARRY4`) to speed up the ripple, but the logical structure remains serial.

- Gate count is minimal ($\approx$ 2 XOR + 3 AND/OR per bit), making the RCA very *area-efficient*.

3.3 4-bit Carry Look-Ahead Adder (CLA)

3.3.1 Architecture

Instead of waiting for each carry to ripple, the CLA pre-computes all carries in parallel using the generate and propagate signals:

$$G_i = A_i \cdot B_i \qquad P_i = A_i \oplus B_i \tag{1}$$

$$C_1 = G_0 + P_0\, C_0 \tag{2}$$

$$C_2 = G_1 + P_1\, G_0 + P_1 P_0\, C_0 \tag{3}$$

$$C_3 = G_2 + P_2\, G_1 + P_2 P_1\, G_0 + P_2 P_1 P_0\, C_0 \tag{4}$$

$$C_4 = G_3 + P_3\, G_2 + P_3 P_2\, G_1 + P_3 P_2 P_1\, G_0 + P_3 P_2 P_1 P_0\, C_0 \tag{5}$$

All carries C_1–C_4 are available simultaneously after a *fixed* two-level AND-OR delay.

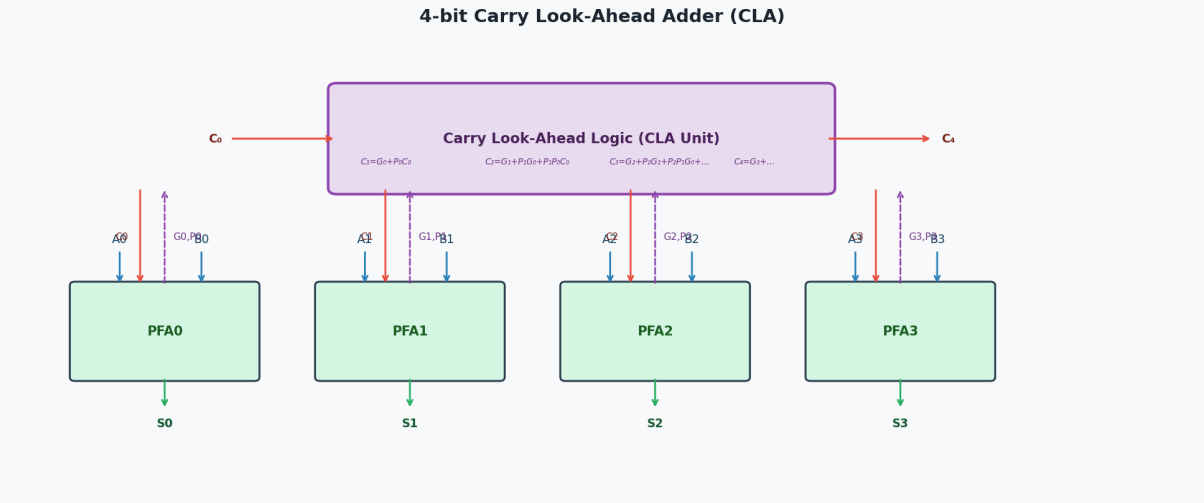


Figure 6: Block diagram of the 4-bit Carry Look-Ahead Adder.

3.3.2 Technology Schematic Observations

The synthesised CLA schematic shows:

- A **fan-in explosion**: gates in the CLA logic unit have increasingly large fan-ins (up to $n + 1$) for higher carry bits, which can force the synthesiser to add buffer trees.
- The carry logic is a wide two-level AND-OR network, clearly distinct from the narrow serial chain seen in the RCA.
- Sum outputs are produced by XOR-ing the partial-FA outputs with the pre-computed carries, so their delay is also nearly constant.
- Overall gate count is *significantly higher* than the RCA.

3.4 Comparison & Inferences

Table 1 summarises the key differences.

Table 1: RCA vs. CLA — Summary Comparison

Metric	Ripple Carry (RCA)	Carry Look-Ahead (CLA)
Delay	$O(n)$ — linear with bits	$O(\log n)$ — parallel carry computation
Gate Count	Low ($\approx 5n$ gates)	Higher (extra propagate/generate logic)
Wiring	Simple, serial carry chain	Complex, wide carry network
Fan-in	Small (max 2–3)	Large (grows with block size)
Scalability	Poor for large n	Better; hierarchical design used
FPGA Mapping	Uses fast dedicated carry chain	Often reduces to carry-chain logic
Area	Compact	Larger; more routing resources
Power (Dynamic)	Lower switching activity	Higher due to wide carry logic
Glitch Power	Minimal	Higher (reconvergent paths)
Logical Depth	Linear ($\approx 2n$ gate levels)	Logarithmic ($\log n$)
Layout Regularity	Highly regular	Irregular routing
Maximum Frequency	Limited	Higher (ASIC implementation)
Design Complexity	Simple	Complex carry network
Preferred Word Size	Small/medium	Medium/large (hierarchical)

3.4.1 Inferences from the Technology Schematic

1. **Speed vs. Area trade-off.** The CLA schematic is visibly larger and more densely connected than the RCA, yet its critical path is shorter. This is a classic trade-off: speed is bought with additional gates and wiring.

2. **Ripple chain visibility in RCA.** The RCA schematic shows a straight carry chain with no parallelism. Any single slow gate on this path becomes the timing bottleneck for the entire adder.

3. **Fan-in limits CLA scalability.** As n grows, the CLA carry equations require AND gates with up to n inputs. For $n > 4$, the synthesiser must split large gates into multi-level trees, which partially erodes the speed advantage. Practical large adders therefore use *hierarchical* CLA or Kogge–Stone / Brent–Kung trees.

3.5 VHDL Implementation of 4-Bit RCA Adder

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

entity RCA_4bit is
    Port (
        A     : in  STD_LOGIC_VECTOR(3 downto 0);
        B     : in  STD_LOGIC_VECTOR(3 downto 0);
        Cin   : in  STD_LOGIC;
        SUM   : out STD_LOGIC_VECTOR(3 downto 0);
        Cout  : out STD_LOGIC
    );
```

```
end RCA_4bit;

architecture structural of RCA_4bit 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(3 downto 0);

begin

    FA0: FA
        port map (
            A    => A(0),
            B    => B(0),
            Cin  => Cin,
            SUM  => SUM(0),
            Cout => C(0)
        );

    FA1: FA
        port map (
            A    => A(1),
            B    => B(1),
            Cin  => C(0),
            SUM  => SUM(1),
            Cout => C(1)
        );

    FA2: FA
        port map (
            A    => A(2),
            B    => B(2),
            Cin  => C(1),
            SUM  => SUM(2),
            Cout => C(2)
        );


    FA3: FA
        port map (
            A    => A(3),
            B    => B(3),
            Cin  => C(2),
            SUM  => SUM(3),
            Cout => C(3)
        );

    Cout <= C(3);

end structural;
```

Listing 4: VHDL Code for 4-Bit RCA Adder

3.6 VHDL Implementation of 4-Bit CLA Adder

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

entity CLA_4bit is
    Port (
        A     : in  STD_LOGIC_VECTOR(3 downto 0);
        B     : in  STD_LOGIC_VECTOR(3 downto 0);
        Cin   : in  STD_LOGIC;
        SUM   : out STD_LOGIC_VECTOR(3 downto 0);
        Cout  : out STD_LOGIC
    );
end CLA_4bit;

architecture structural of CLA_4bit 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 P, G : STD_LOGIC_VECTOR(3 downto 0);
    signal C : STD_LOGIC_VECTOR(4 downto 0);

begin
    C(0) <= Cin;
```

```
31      P <= A xor B;
32      G <= A and B;
33
34      C(1) <= G(0) or (P(0) and C(0));
35
36      C(2) <= G(1) or
37              (P(1) and G(0)) or
38              (P(1) and P(0) and C(0));
39
40      C(3) <= G(2) or
41              (P(2) and G(1)) or
42              (P(2) and P(1) and G(0)) or
43              (P(2) and P(1) and P(0) and C(0));
44
45      C(4) <= G(3) or
46              (P(3) and G(2)) or
47              (P(3) and P(2) and G(1)) or
48              (P(3) and P(2) and P(1) and G(0)) or
49              (P(3) and P(2) and P(1) and P(0) and C(0));
50
51      FA0: FA port map(A(0), B(0), C(0), SUM(0), open);
52      FA1: FA port map(A(1), B(1), C(1), SUM(1), open);
53      FA2: FA port map(A(2), B(2), C(2), SUM(2), open);
54      FA3: FA port map(A(3), B(3), C(3), SUM(3), open);
55
56      Cout <= C(4);
57  end structural;
```

Listing 5: VHDL Code for 4-Bit CLA Adder

3.7 Results

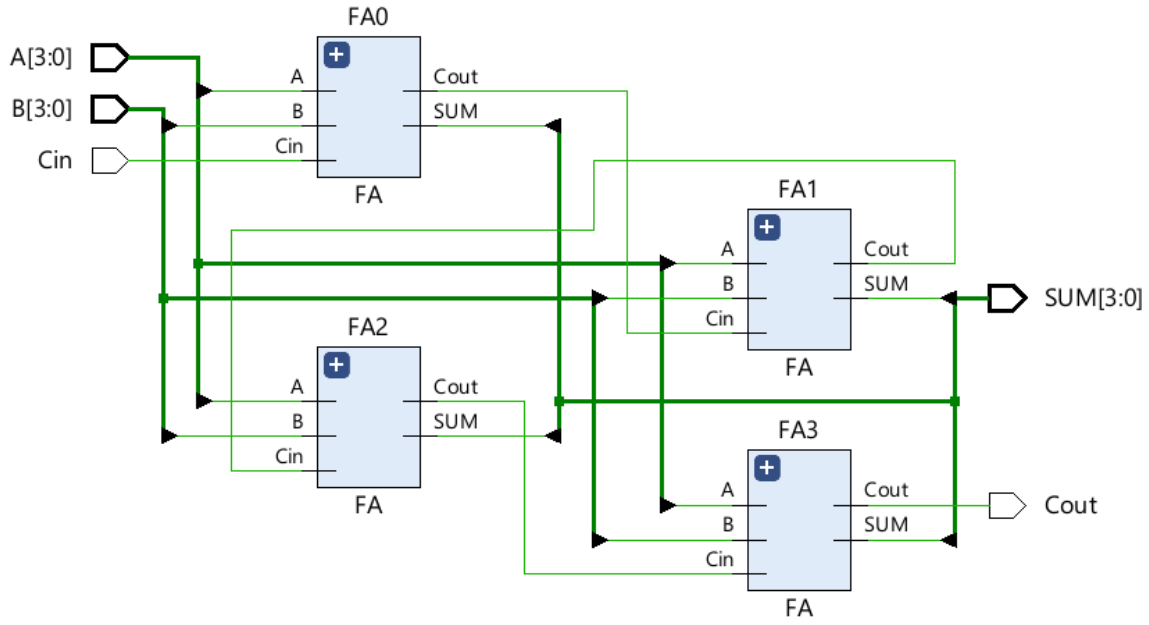


Figure 7: Realized Schematic for 4-bit RCA

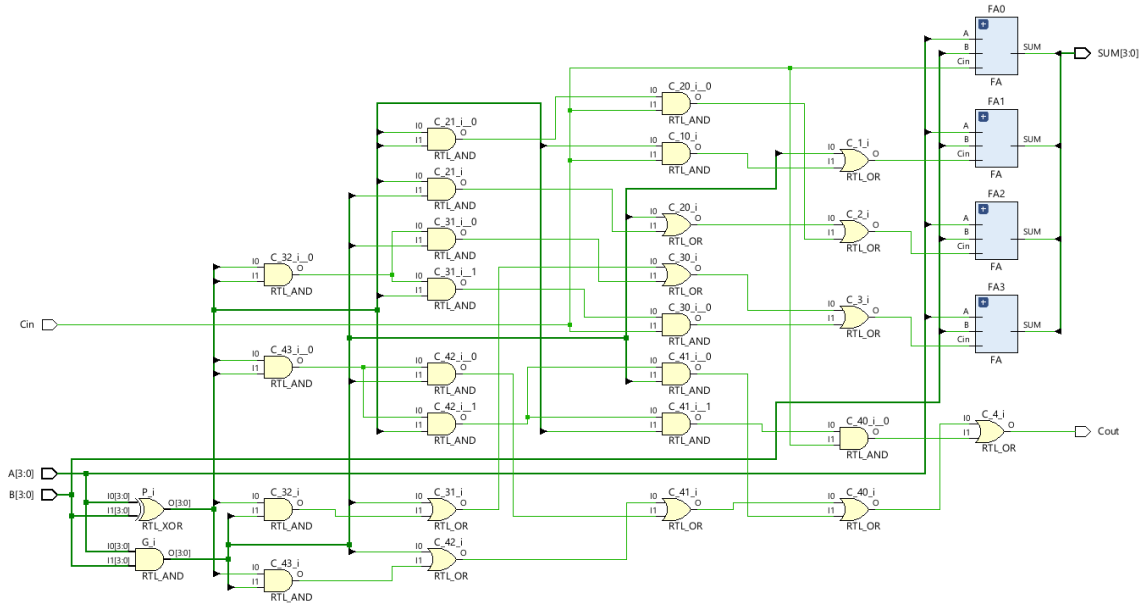


Figure 8: Realized Schematic for 4-bit CLA

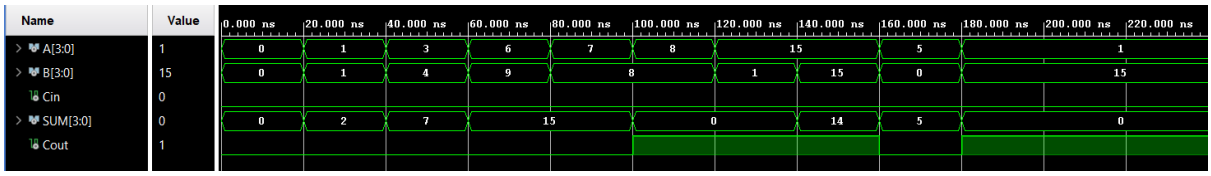


Figure 9: Simulation Results for 4-bit RCA

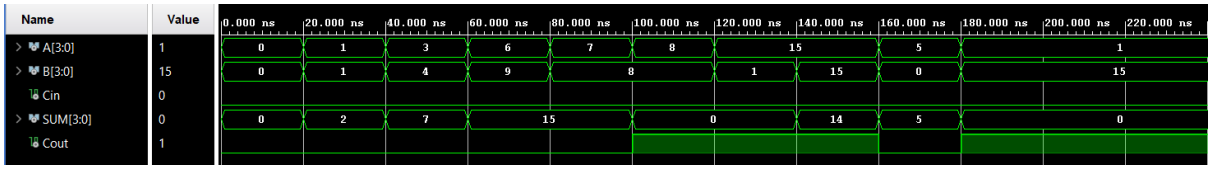


Figure 10: Simulation Results for 4-bit CLA

4 Question 4

4.1 Theory

4.1.1 Seven-Segment Display

A seven-segment display is an electronic display device used to represent decimal and hexadecimal numerals. It consists of seven individual LED segments arranged in a figure-eight pattern, each of which can be independently turned ON or OFF to form digits and characters.

The seven segments are conventionally labeled **a** through **g** as shown in Figure 11:

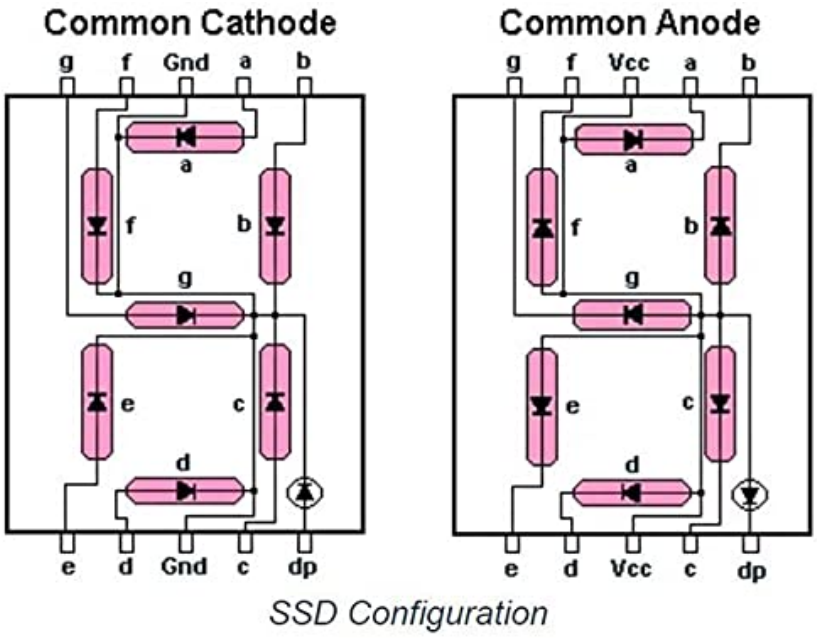


Figure 11: Seven-segment display internal segment labeling (a–g). *(Insert image here)*

4.1.2 BCD to Seven-Segment Decoding

A BCD (Binary Coded Decimal) to seven-segment decoder takes a 4-bit binary input and produces the appropriate 7-bit output to illuminate the correct segments. The truth table for hexadecimal digits 0–F is given in Table 2.

Table 2: Truth table for 4-bit input to seven-segment output (Common Cathode). Logic 1 = segment ON.

Digit	**SW[3:0]**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0000	1	1	1	1	1	1	0
1	0001	0	1	1	0	0	0	0
2	0010	1	1	0	1	1	0	1
3	0011	1	1	1	1	0	0	1
4	0100	0	1	1	0	0	1	1
5	0101	1	0	1	1	0	1	1
6	0110	1	0	1	1	1	1	1
7	0111	1	1	1	0	0	0	0
8	1000	1	1	1	1	1	1	1
9	1001	1	1	1	1	0	1	1
A	1010	1	1	1	0	1	1	1
b	1011	0	0	1	1	1	1	1
C	1100	1	0	0	1	1	1	0
d	1101	0	1	1	1	1	0	1
E	1110	1	0	0	1	1	1	1
F	1111	1	0	0	0	1	1	1

4.2 Hardware

4.2.1 Equipment

- Digilent Zybo Z7-10/Z7-20 FPGA development board
- Common-cathode 7-segment LED display
- Jumper wires
- Breadboard

4.2.2 PMOD JA Pin Configuration

The PMOD JA header on the Zybo Z7 board provides 8 signal pins plus VCC and GND. The physical layout of the 12-pin connector, viewed from the front of the board, is shown in Table 3.

Table 3: Physical pin layout of PMOD JA connector (viewed from outside the board).

VCC	**GND**	**JA4**	**JA3**	**JA2**	**JA1**
VCC	**GND**	**JA10**	**JA9**	**JA8**	**JA7**
← *Left (VCC/GND side) Right (Signal side)* →					

The signal-to-segment mapping used in this experiment is given in Table 4.

Table 4: PMOD JA pin to 7-segment display wiring.

PMOD Pin	FPGA Package Pin	SEG Index	Display Segment
JA1	N15	SEG[0]	a
JA2	L14	SEG[1]	b
JA3	K16	SEG[2]	c
JA4	K14	SEG[3]	d
JA7	N16	SEG[4]	e
JA8	L15	SEG[5]	f
JA9	J16	SEG[6]	g
JA10	J14	SEG[7]	dp
GND	–	–	Common Cathode

4.2.3 Physical Setup

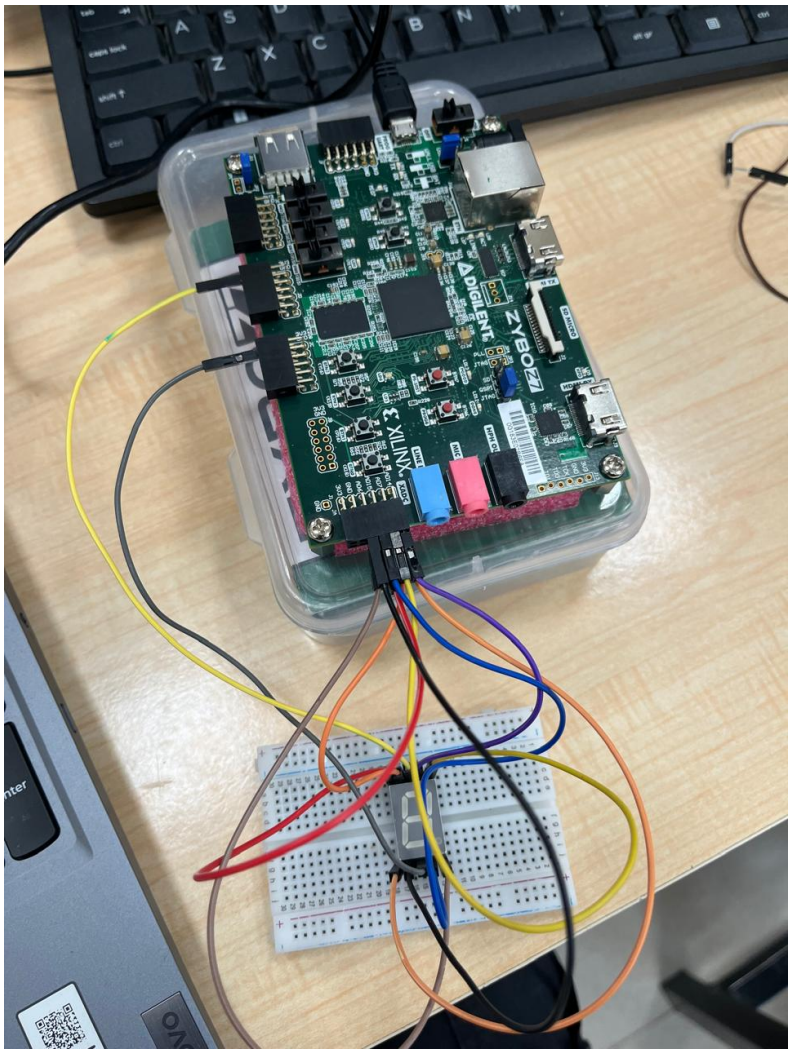


Figure 12: Seven-segment display and FPGA Setup

4.3 VHDL Implementation

4.3.1 VHDL Design Code

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

entity switch_to_7seg is
    Port (
        SW  : in  STD_LOGIC_VECTOR(3 downto 0);
        SEG : out STD_LOGIC_VECTOR(7 downto 0)
    );
end switch_to_7seg;

architecture Behavioral of switch_to_7seg is
begin
    process(SW)
    begin
        SEG(7) <= '0';
        case SW is
            when "0000" => SEG(6 downto 0) <= "0111111"; -- 0: abcdef on,  g off
            when "0001" => SEG(6 downto 0) <= "0000110"; -- 1: bc on
            when "0010" => SEG(6 downto 0) <= "1011011"; -- 2: abdeg on
            when "0011" => SEG(6 downto 0) <= "1001111"; -- 3: abcdg on
            when "0100" => SEG(6 downto 0) <= "1100110"; -- 4: bcfg on
            when "0101" => SEG(6 downto 0) <= "1101101"; -- 5: acdfg on
            when "0110" => SEG(6 downto 0) <= "1111101"; -- 6: acdefg on
            when "0111" => SEG(6 downto 0) <= "0000111"; -- 7: abc on
            when "1000" => SEG(6 downto 0) <= "1111111"; -- 8: all on
            when "1001" => SEG(6 downto 0) <= "1101111"; -- 9: abcdfg on
            when "1010" => SEG(6 downto 0) <= "1110111"; -- A: abcefg on
            when "1011" => SEG(6 downto 0) <= "1111100"; -- b: cdefg on
            when "1100" => SEG(6 downto 0) <= "0111001"; -- C: adef on
            when "1101" => SEG(6 downto 0) <= "1011110"; -- d: bcdeg on
            when "1110" => SEG(6 downto 0) <= "1111001"; -- E: adefg on
            when "1111" => SEG(6 downto 0) <= "1110001"; -- F: aefg on
            when others => SEG(6 downto 0) <= "0000000";
        end case;
    end process;
end Behavioral;
```

Listing 6: VHDL implementation of the switch-to-seven-segment decoder

4.3.2 XDC Constraint File

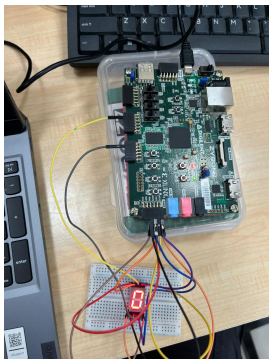
```
###Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L19N_T3_VREF_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L24P_T3_34 Sch=sw[1]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L4N_T0_34 Sch=sw[2]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { SW[3]  }]; #IO_L9P_T1_DQS_34 Sch=sw[3]

## PMOD Header JA used as GPIO for 7-Segment
set_property -dict { PACKAGE_PIN N15 IOSTANDARD LVCMOS33 } [get_ports { SEG[0] }]; # JA1
set_property -dict { PACKAGE_PIN L14 IOSTANDARD LVCMOS33 } [get_ports { SEG[1] }]; # JA2
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { SEG[2] }]; # JA3
set_property -dict { PACKAGE_PIN K14 IOSTANDARD LVCMOS33 } [get_ports { SEG[3] }]; # JA4
set_property -dict { PACKAGE_PIN N16 IOSTANDARD LVCMOS33 } [get_ports { SEG[4] }]; # JA7
set_property -dict { PACKAGE_PIN L15 IOSTANDARD LVCMOS33 } [get_ports { SEG[5] }]; # JA8
set_property -dict { PACKAGE_PIN J16 IOSTANDARD LVCMOS33 } [get_ports { SEG[6] }]; # JA9
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { SEG[7] }]; # JA10
```

Listing 7: VHDL Constraint File for switch-to-seven-segment decoder

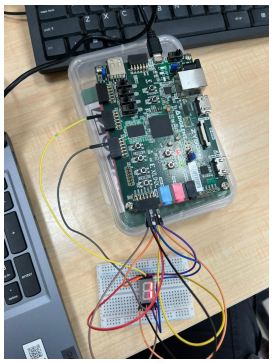
4.4 Results

The design was successfully synthesized, implemented, and programmed onto the Zybo Z7 FPGA board. Upon toggling the four slide switches, the corresponding hexadecimal digit (0–F) was correctly displayed on the seven-segment display.

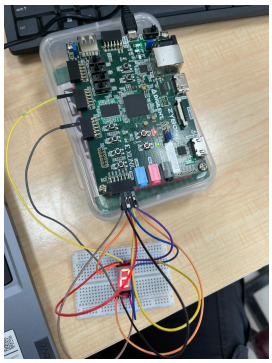
4.4.1 Observed Outputs



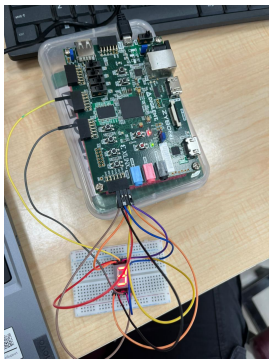
(a) SW = 0000 (0)



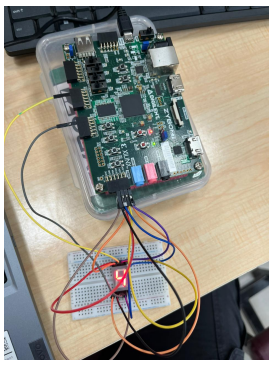
(b) SW = 0001 (1)



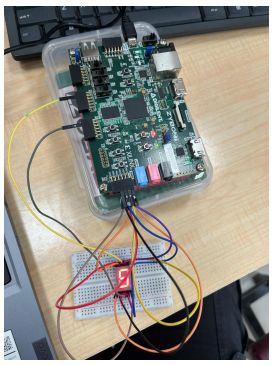
(c) SW = 0010 (2)



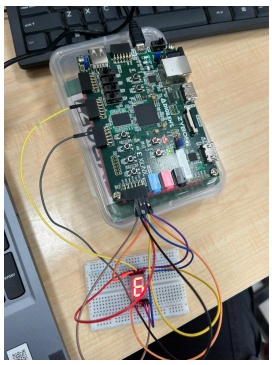
(d) SW = 0011 (3)



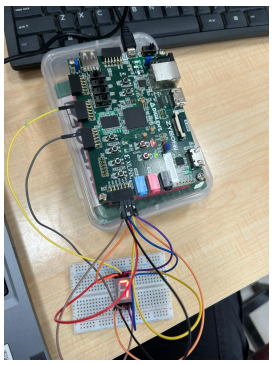
(e) SW = 0100 (4)



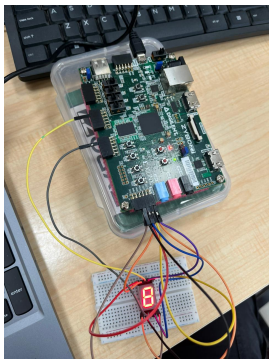
(f) SW = 0101 (5)



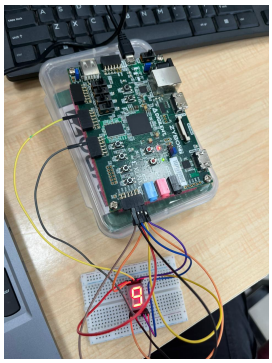
(g) SW = 0110 (6)



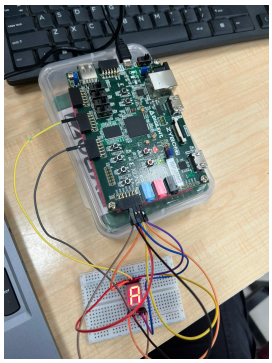
(h) SW = 0111 (7)



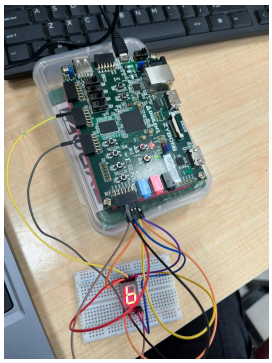
(i) SW = 1000 (8)



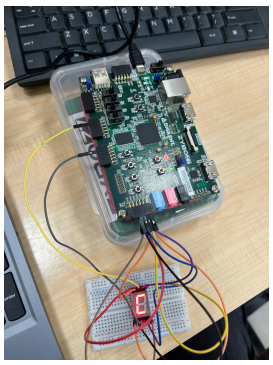
(j) SW = 1001 (9)



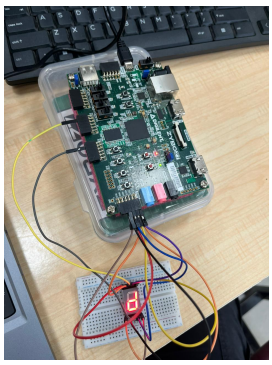
(k) SW = 1010 (A)



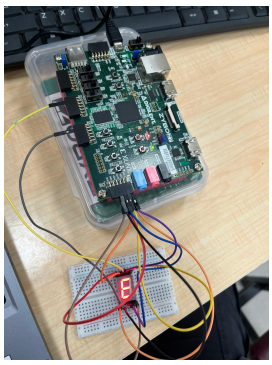
(l) SW = 1011 (b)



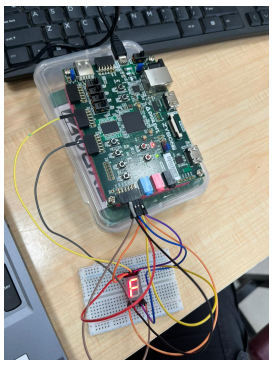
(m) SW = 1100 (C)



(n) SW = 1101 (d)



(o) SW = 1110 (E)



(p) SW = 1111 (F)

Figure 13: Observed seven-segment display output for all 16 hexadecimal inputs (0–F).

5 Question 5

In this experiment, the half adder circuit is implemented on the FPGA using push button switches as inputs instead of slide switches. The logical functionality of the circuit remains unchanged; only the physical input interface is modified.

Push buttons are momentary switches, meaning they generate a logic level only while being pressed. Therefore, the primary modification required in this problem is in the constraint file, where the FPGA pin assignments are updated to map the half adder inputs (A and B) to the push button pins instead of the slide switch pins. The LED output pin assignments remain the same as in Problem 1.

The RTL schematic and technology schematic generated after synthesis also remain unchanged, as the internal implementation within the FPGA configurable logic blocks (CLBs) is identical. Only the external input pin mapping differs due to the use of push button switches.

The VHDL code remains the same as in Problem 1.

```
##Buttons
set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports { A }]; #IO_L10P_T1_AD11P_35 Sch=btn[2]
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { B }]; #IO_L7P_T1_34 Sch=btn[3]

##LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { SUM }]; #IO_L23P_T3_35 Sch=led[0]
set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { CARRY }]; #IO_L23N_T3_35 Sch=led[1]
```

Listing 8: Constraint File for Half Adder

6 Question 6

6.1 Theory

A 4-bit adder-subtractor combines both addition and subtraction into a single circuit controlled by a mode signal M. The design exploits the two's complement representation of negative numbers, where subtraction $A - B$ is equivalent to:

$$A - B = A + \overline{B} + 1 \tag{6}$$

That is, the subtrahend B is bitwise complemented (one's complement) and a 1 is added through the carry-in, yielding the two's complement negation of B.

6.1.1 Working Principle

A single XOR gate per bit acts as a programmable inverter controlled by the mode signal M:

$$B'_i = B_i \oplus M \tag{7}$$

- When $M = 0$: $B'_i = B_i$ (unchanged), and $C_{in} = 0$, so the circuit computes $A + B$.
- When $M = 1$: $B'_i = \overline{B_i}$ (complemented), and $C_{in} = 1$, so the circuit computes $A + \overline{B} + 1 = A - B$.

The carry-in of the RCA is tied directly to M, providing the +1 needed for two's complement in subtraction mode. The unified expression for the circuit's output is:

$$S = A + (B \oplus M) + M \tag{8}$$

6.1.2 Overflow and Borrow Detection

For **addition** ($M = 0$): the carry-out C_{out} indicates an unsigned overflow.
For **subtraction** ($M = 1$): the borrow is the logical complement of C_{out}, i.e., Borrow = $\overline{C_{out}}$. When $C_{out} = 0$, the result is negative (a borrow occurred); when $C_{out} = 1$, the subtraction was exact (no borrow).

6.2 VDHL Implementation

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

entity subtractor_4bit is
    Port (
        A      : in  STD_LOGIC_VECTOR(3 downto 0);
        B      : in  STD_LOGIC_VECTOR(3 downto 0);
        DIFF   : out STD_LOGIC_VECTOR(3 downto 0);
        Borrow : out STD_LOGIC
    );
end subtractor_4bit;

architecture structural of subtractor_4bit is

    component RCA_4bit
        Port (
            A     : in  STD_LOGIC_VECTOR(3 downto 0);
            B     : in  STD_LOGIC_VECTOR(3 downto 0);
            Cin   : in  STD_LOGIC;
            SUM   : out STD_LOGIC_VECTOR(3 downto 0);
            Cout  : out STD_LOGIC
        );
    end component;

    signal B_comp : STD_LOGIC_VECTOR(3 downto 0);
    signal Cout_int : STD_LOGIC;

begin

    B_comp <= not B;

    U1: RCA_4bit
        port map(
            A     => A,
            B     => B_comp,
            Cin   => '1',
            SUM   => DIFF,
            Cout  => Cout_int
        );

    Borrow <= not Cout_int;

end structural;
```

Listing 9: 4 bit subtractor using 4 bit RCA

6.3 Results

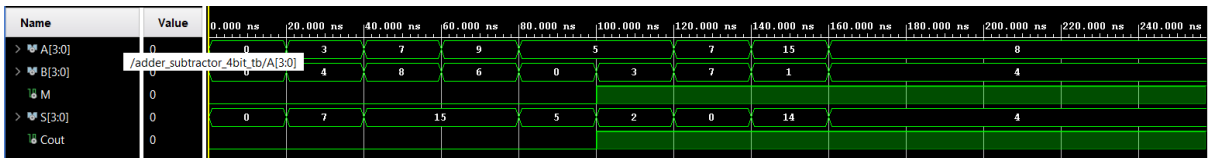


Figure 14: Simulation Results

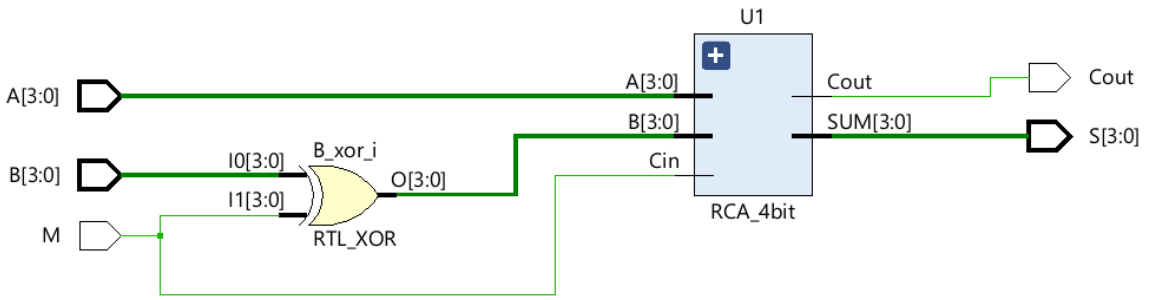


Figure 15: Realized Schematic