Ada Zağyapan

EEE102-02

05 March 2025

LAB-4: Arithmetic Logic Unit

PURPOSE

This lab aimed to develop an arithmetic logic unit (ALU) that could execute eight distinct operations defined by 4-bit inputs. Each operation is implemented in its own module, and a multiplexer is used to select one of the operation results for display via LEDs.

METHODOLOGY

Before beginning the design process, eight specific operations were selected for implementation in the ALU. These operations are: addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, left shift, right shift, and increment.

The ALU takes two 4-bit inputs, A and B, along with a 3-bit select input OP. The select input determines which of the eight operations is executed. The output consists of a 4-bit Result and a single-bit Cout, which is used for operations involving carry propagation.

The ALU is designed using a structural approach, where each operation is implemented as a separate component. These components include a 4-bit adder, 4-bit subtractor, AND module, OR module, XOR module, left shift module, right shift module, and an increment module. The operations corresponding to the select inputs can be observed in Table 1.1.

Each operation's output is computed separately and stored in a signal. These signals are then passed to a multiplexer-like process controlled by the select input OP. Based on the value of OP, the appropriate result is assigned to Result, and if applicable, the corresponding carry output (Cout) is also assigned.

Once the design was completed, RTL schematics were generated for the main ALU module and other operation modules (Figure 1-9). Constraint files were created to map the inputs and outputs to the physical buttons and LEDs on the FPGA board. After generating the bitstream, the code was uploaded to the FPGA for testing. Additionally, a testbench was written in order to check if the values in the FPGA aligns with the code, and if the code aligns with the intended values. The ALU's functionality was verified by providing different input values and ensuring the correct results were displayed on the board and the simulation.

OP (Select Input)	Function	Input	Output
“000”	Addition	A, B	Result, Cout
“001”	Subtraction	A, B	Result, Cout
“010”	Bitwise AND	A,B	Result
“011”	Bitwise OR	A,B	Result
“100”	Bitwise XOR	A,B	Result
“101”	Left Shift	A	Result
“110”	Right Shift	A	Result
“111”	Increment	A	Result, Cout

Table 1: Command table for ALU operations

DESIGN SPECIFICATIONS

ALU: The ALU is the central unit that integrates multiple sub-components which are each responsible for executing a specific operation. The ALU in this lab consists of eight functional units, which include addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, left shift, right shift, and increment.

Each of these operations is implemented as a separate module, and their outputs are routed to a multiplexer-like process that selects the appropriate result based on the 3-bit OP input. This ensures that the ALU performs the desired computation dynamically based on the selected operation.

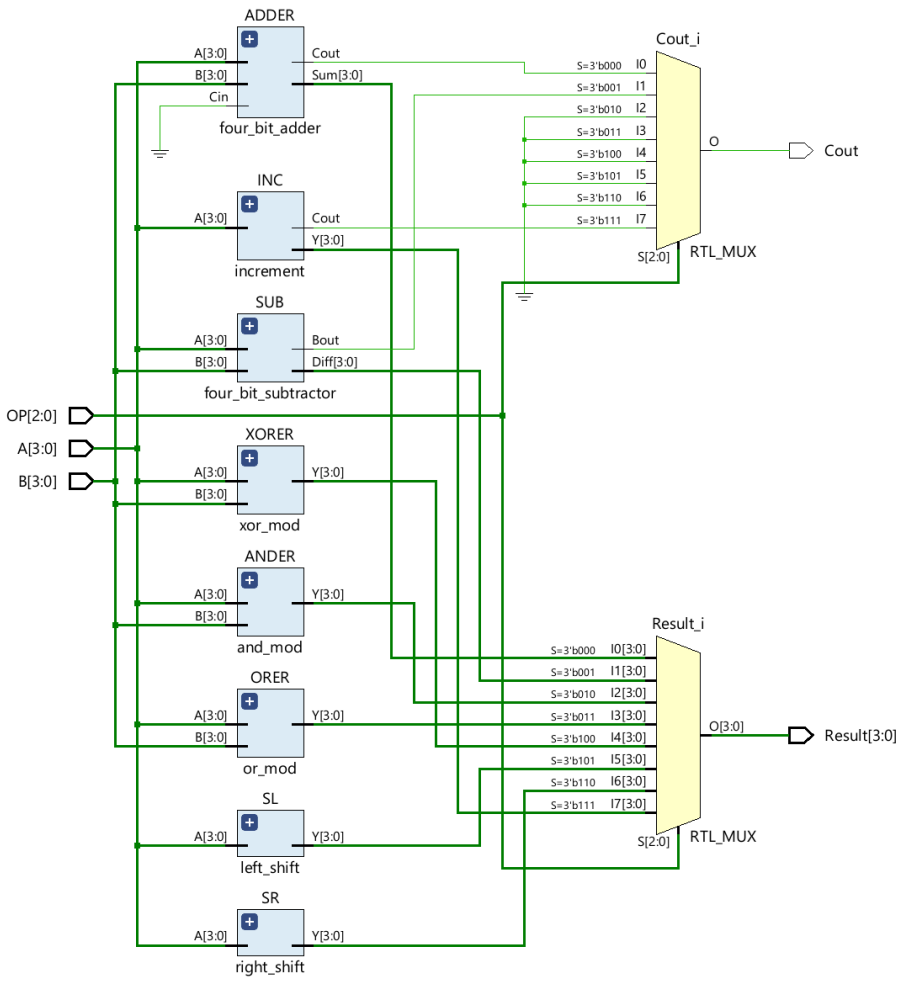

Figure 1: ALU design

4-bit adder: The 4-bit full adder operates with three inputs: A, B, and Cin. The inputs A and B are used for the addition operation, while Cin represents the carry-in value, which is always set to '0'. Since the addition operation is selected when OP = "000", the Cin input is hardcoded to '0' to ensure correct operation.

The 4-bit adder consists of four full-adder stages, each responsible for computing a bit-wise sum and a carry out. The sum output is stored in add_result, which represents the final

summation. The most significant bit's carry-out is stored in Cout, while the possibility of overflow is indicated by monitoring the carry-out behavior.

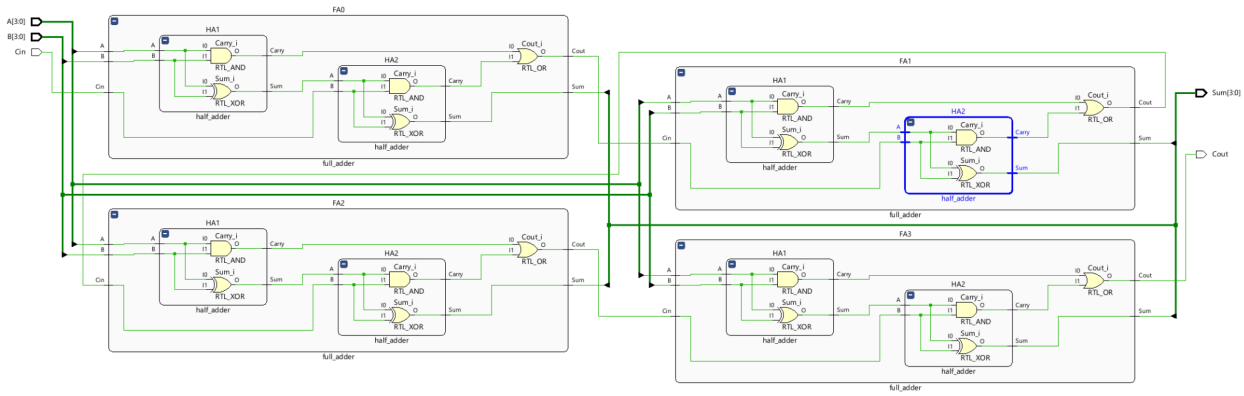

Figure 2: 4-bit adder design

4-bit subtractor: The 4-bit subtractor performs bitwise subtraction between the two 4-bit inputs, A and B. Each bit of B is subtracted from the corresponding bit of A, with borrow propagation handled across all four bits. The result of the subtraction is stored in sub_result.

This operation is selected when OP = "001", ensuring that sub_result is passed to the Result output. The most significant bit's borrow-out is stored in Cout, indicating whether a borrow was needed in the subtraction process.

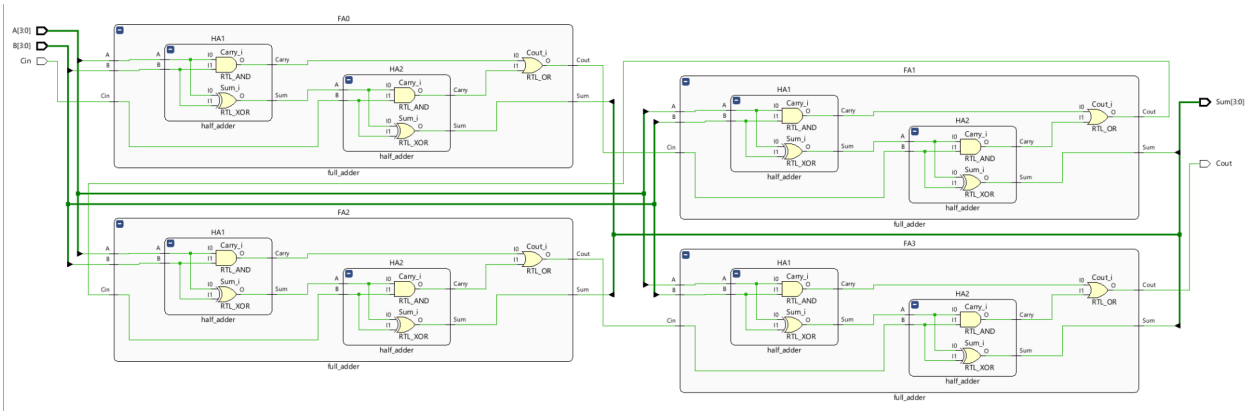

Figure 3: 4-bit subtractor design

AND gate: The 4-bit AND operation performs a bitwise AND between the two 4-bit inputs, A and B. Each bit of A is logically ANDed with the corresponding bit of B, and the result is stored in and_result.

Since the AND operation is selected when OP = "010", the multiplexer-like selection process ensures that and_result is passed to the Result output when this operation is chosen. The carry-out (Cout) is not relevant for this operation and is set to '0'.

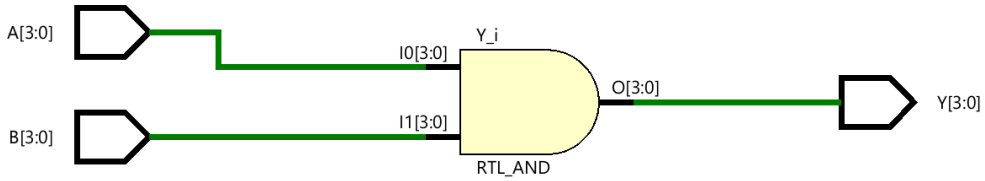

Figure 4: AND gate design

OR gate: The 4-bit OR operation performs a bitwise OR between the two 4-bit inputs, A and B. Each bit of A is logically ORed with the corresponding bit of B, and the result is stored in or_result.

When the operation is selected with OP = "011", the or_result is passed to the Result output. Since the OR operation does not involve carry propagation, the Cout output is set to '0'.

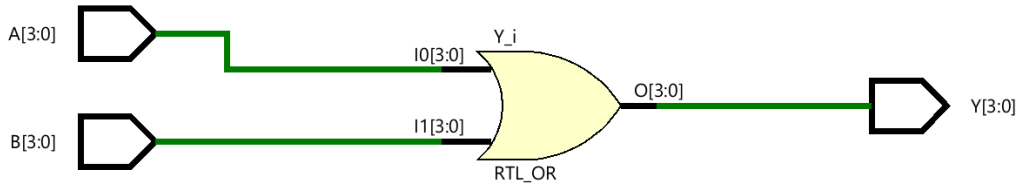

Figure 5: OR gate design

XOR gate: The 4-bit XOR operation performs a bitwise exclusive OR between the two 4-bit inputs, A and B. Each bit of A is XORed with the corresponding bit of B, and the result is stored in xor_result.

When the operation is selected with OP = "100", the xor_result is passed to the Result output. Since XOR does not involve carry propagation, the Cout output is set to '0'.

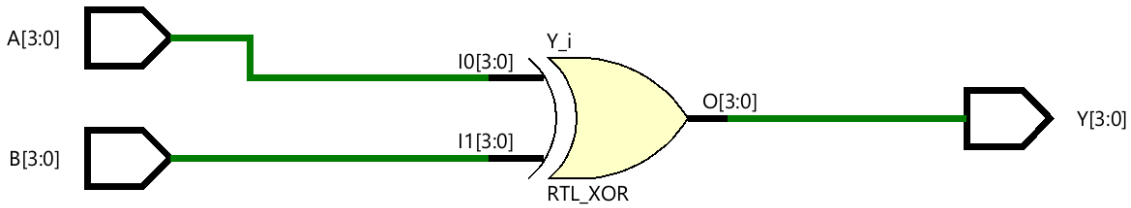

Figure 6: XOR gate design

Right shift: The 4-bit right shift operation shifts the bits of A one position to the right, with the leftmost bit filled with '0'. The result of this operation is stored in shr_result.

When the operation is selected with OP = "110", the shr_result is passed to the Result output. Since shifting does not involve carry propagation, the Cout output is set to '0'.

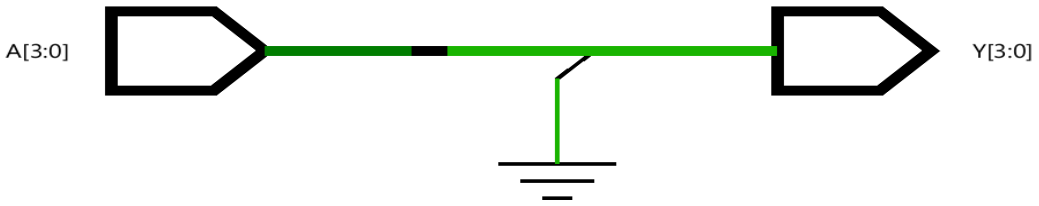

Figure 7: Shift right design

Left shift: The 4-bit left shift operation shifts the bits of A one position to the left, with the rightmost bit filled with '0'. The result of this operation is stored in shl_result.

When the operation is selected with OP = "101", the shl_result is passed to the Result output. Since shifting does not involve carry propagation, the Cout output is set to '0'.

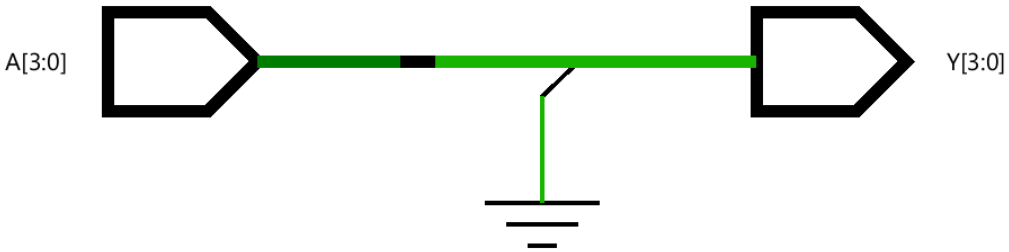

Figure 8: Shift left design

Increment: The 4-bit increment operation increases the value of A by 1. The result of this operation is stored in inc_result.

When the operation is selected with OP = "111", the inc_result is passed to the Result output. Since incrementing may produce a carry-out, the Cout output is set to inc_cout, which indicates if an overflow occurs during the operation.

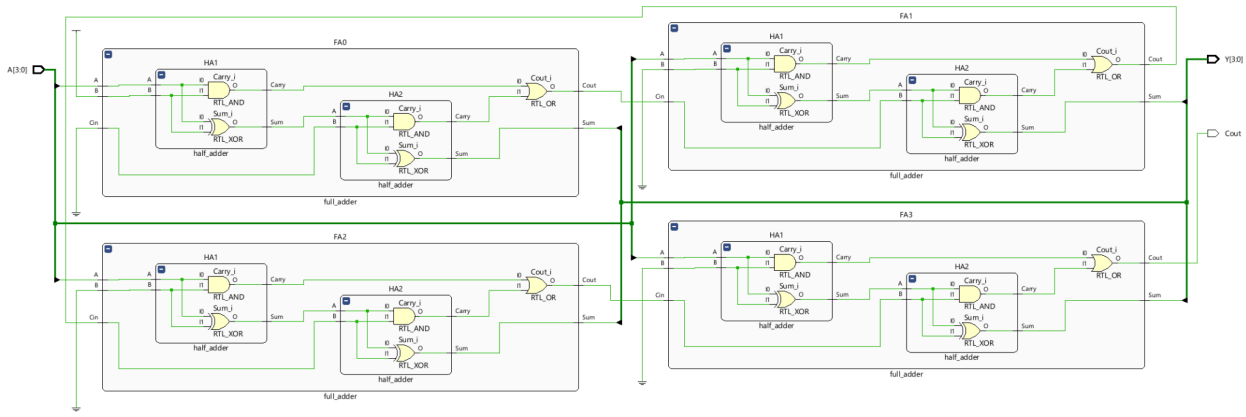

Figure 9: Increment design

RESULTS

After implementing the ALU, its functionality was verified through both simulation and hardware testing. The ALU correctly executed all eight operations, which are addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, left shift, right shift, and increment, based on the OP selection input. The Result output displayed the expected values, and the Cout signal correctly showed the carry-out and borrow-outs.

The ALU was simulated using a testbench to verify its correctness before hardware implementation. Each operation was tested with different input values, ensuring that the results matched theoretical expectations. The simulation confirmed proper bitwise operations, correct carry-out handling for addition and subtraction, and accurate shift and increment operations.

After successful simulation, the design was synthesized and implemented on the Basys3 FPGA board. Inputs were controlled using switches SW5-SW15, and outputs were observed via LEDs LED0-LED4. The ALU functioned as expected in real-time testing, confirming that the multiplexer-based selection and sub-unit integrations were correctly implemented.

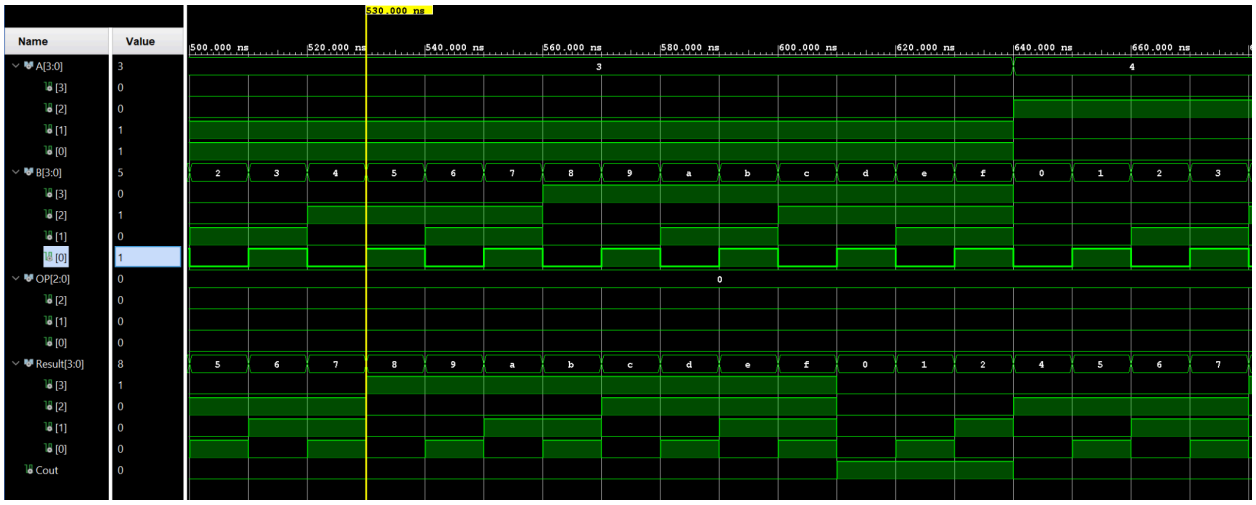

Figure 10: Addition simulation results (OP= “000”, A= “0011”, B= “0101”, Result= “1000”, Cout= ‘0’)

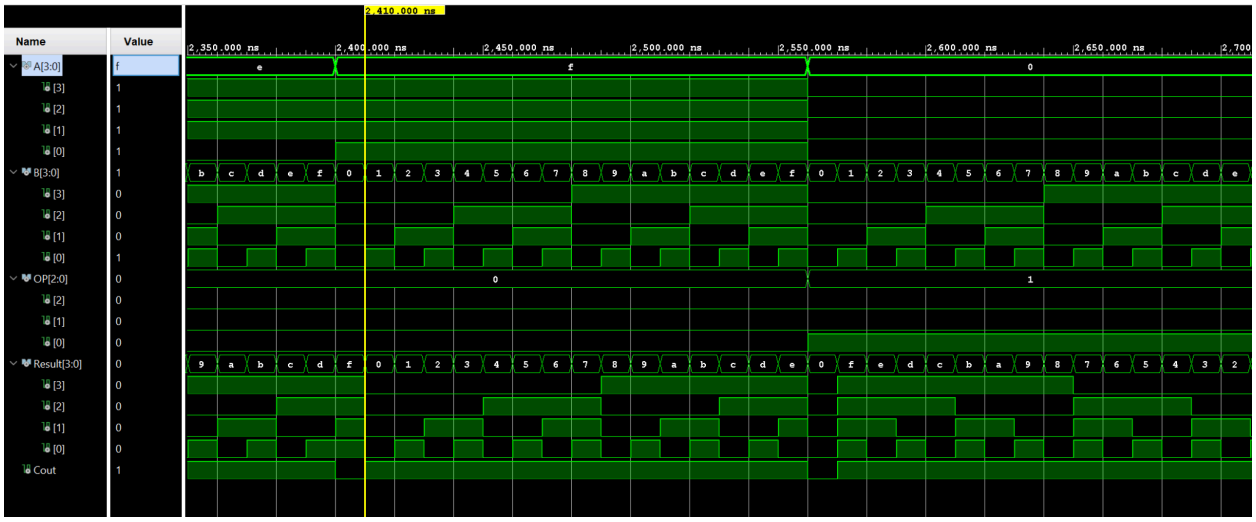

Figure 11: Addition simulation results (OP= “000”, A= “1111”, B= “0001”, Result= “0000”, Cout= ‘1’)

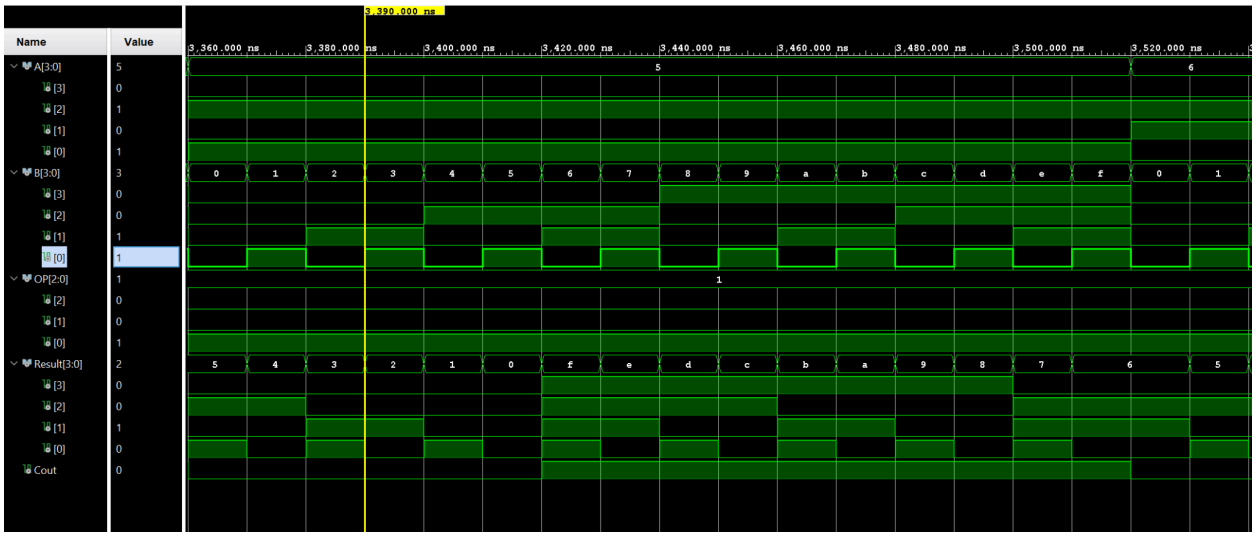

Figure 12: Subtraction simulation results (OP= “001”, A= “0101”, B= “0011”, Result= “0010”, Cout= ‘0’)

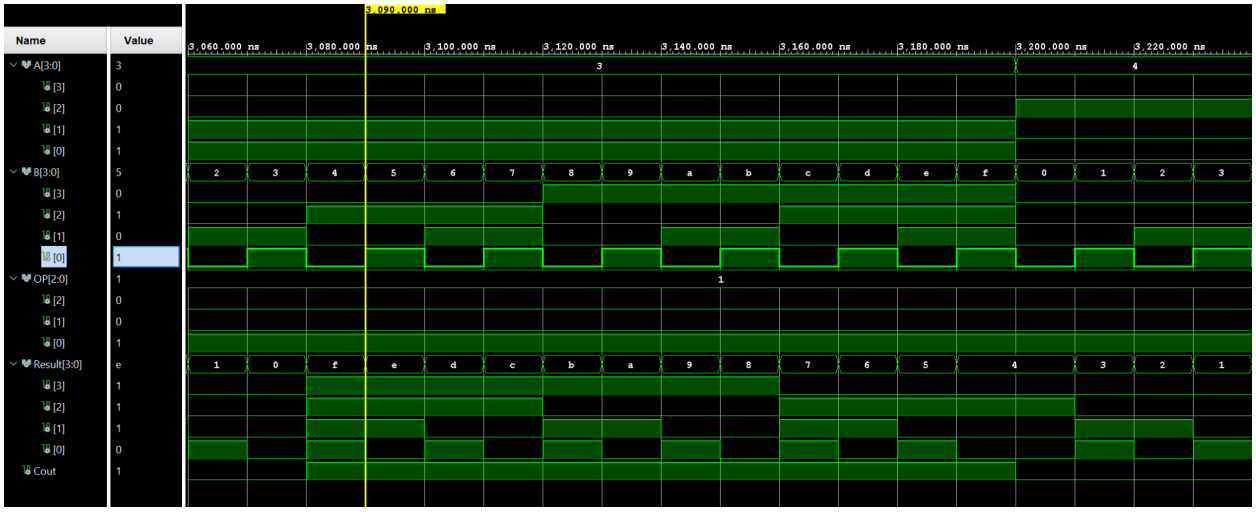

Figure 13: Subtraction simulation results (OP= “001”, A= “0011”, B= “0101”, Result= “1110”, Cout= ‘1’)

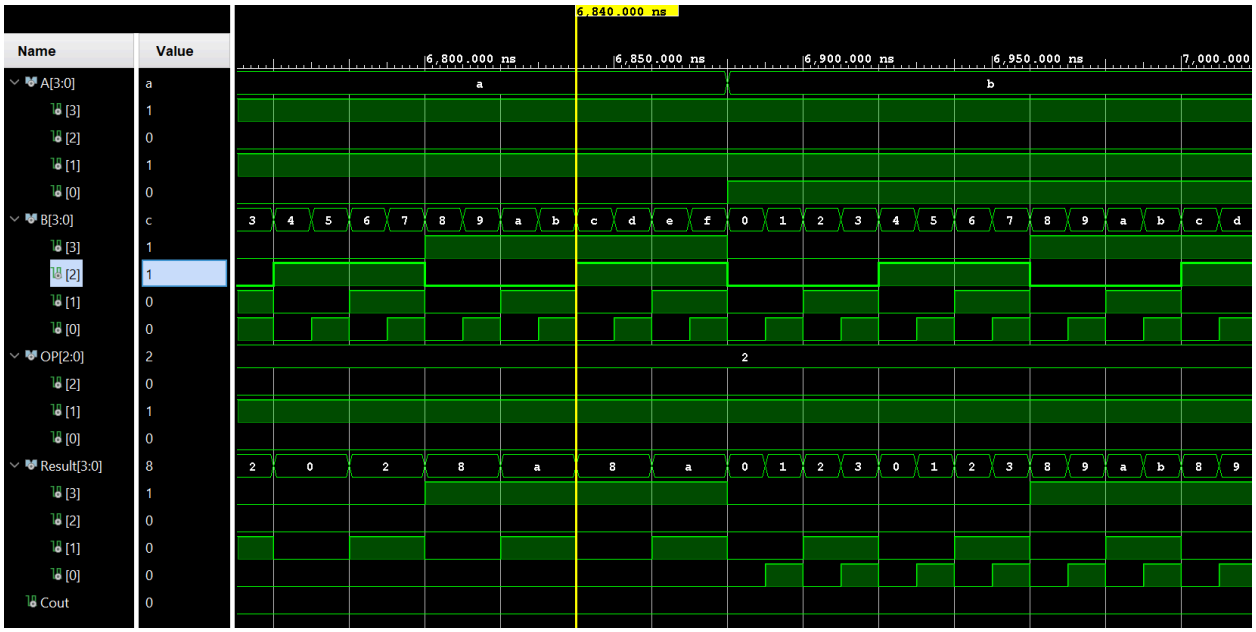

Figure 14: AND gate simulation results (OP= “010”, A= “1010”, B= “1100”, Result= “1000”, Cout= ‘0’)

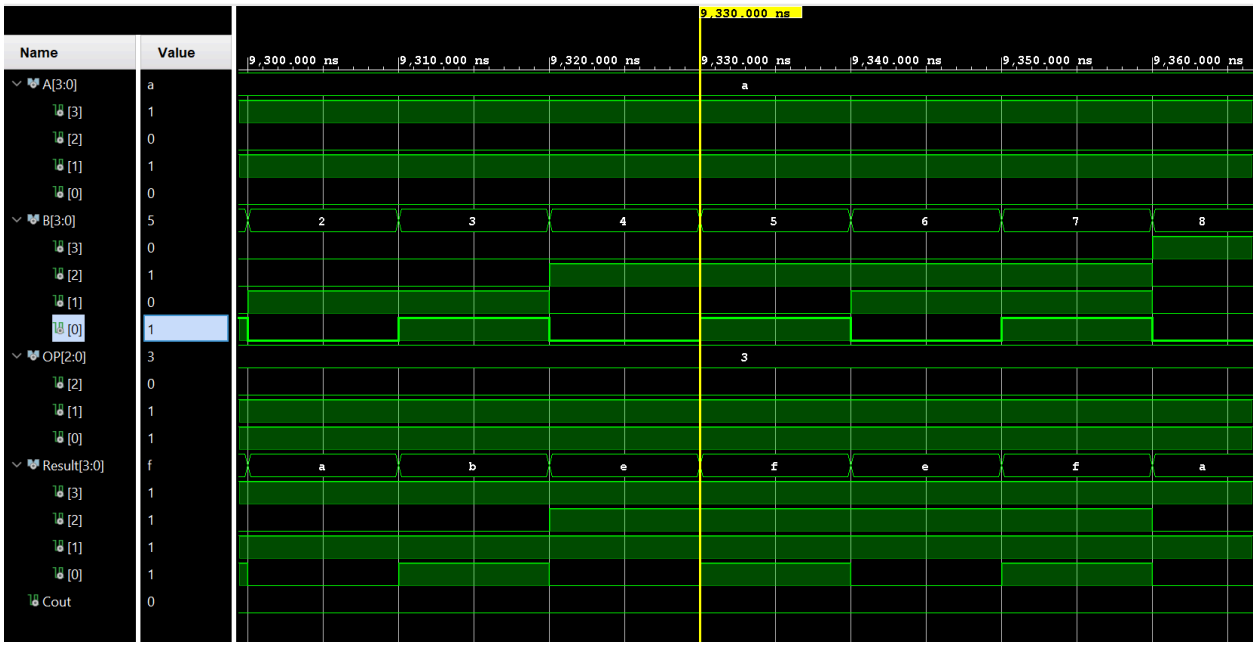

Figure 15: OR gate simulation results (OP= “011”, A= “1010”, B= “0101”, Result= “1111”, Cout= ‘0’)

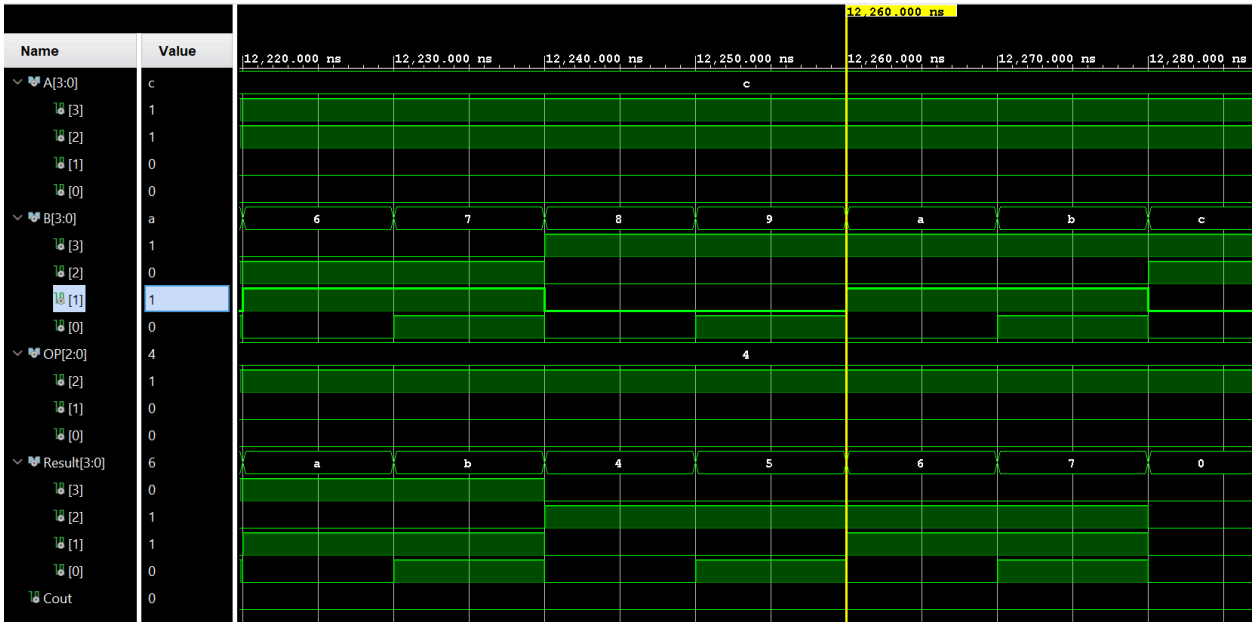

Figure 16: XOR gate simulation results (OP= “100”, A= “1100”, B= “1010”, Result= “0110”, Cout= ‘0’)

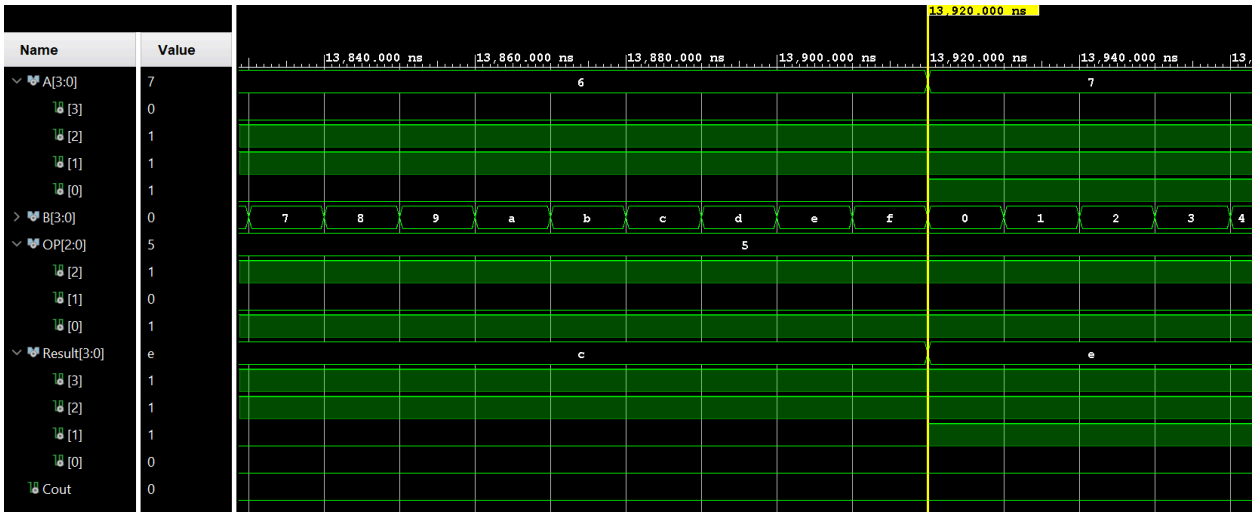

Figure 17: Left shift simulation results (OP= “101”, A= “0111”, Result= “1110”, Cout= ‘0’)

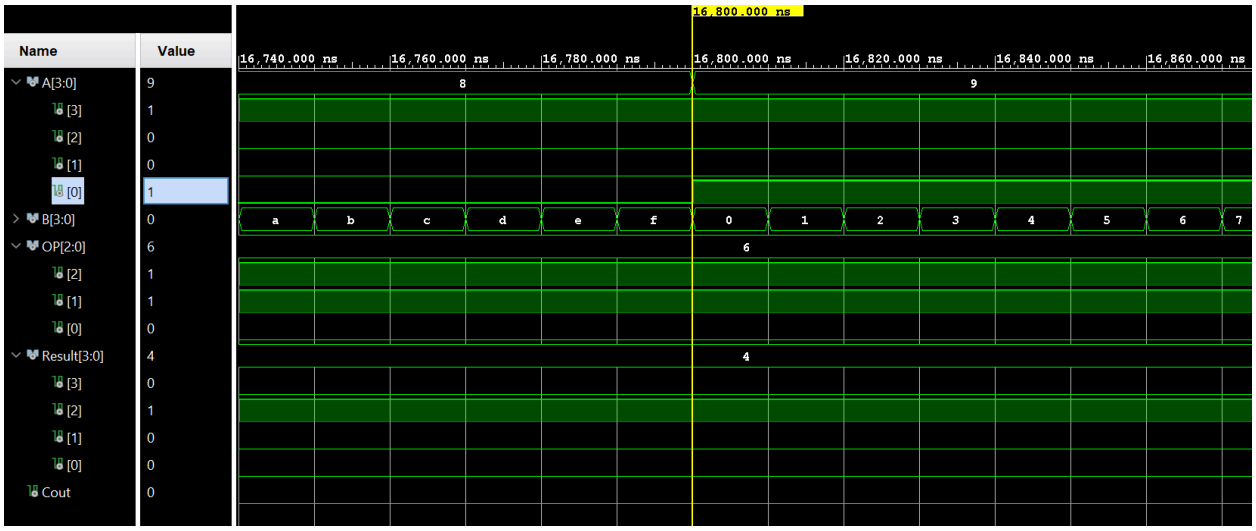

Figure 18: Right shift simulation results (OP= “110”, A= “1001”, Result= “0100”, Cout= ‘0’)

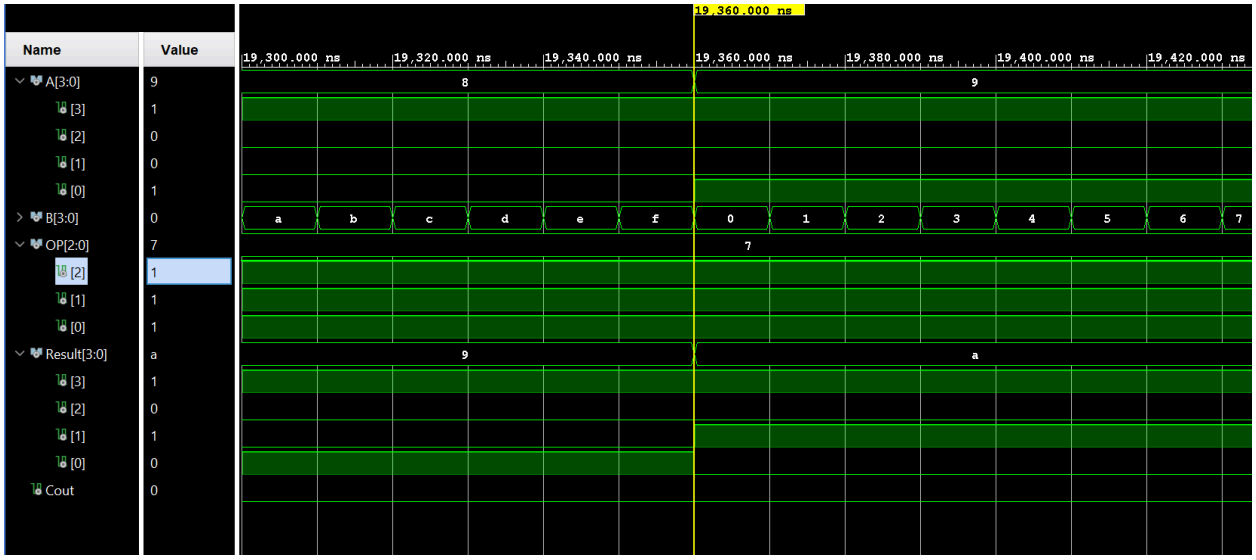

Figure 19: Increment simulation results (OP= “111”, A= “1001”, Result= “1010”, Cout= ‘0’)

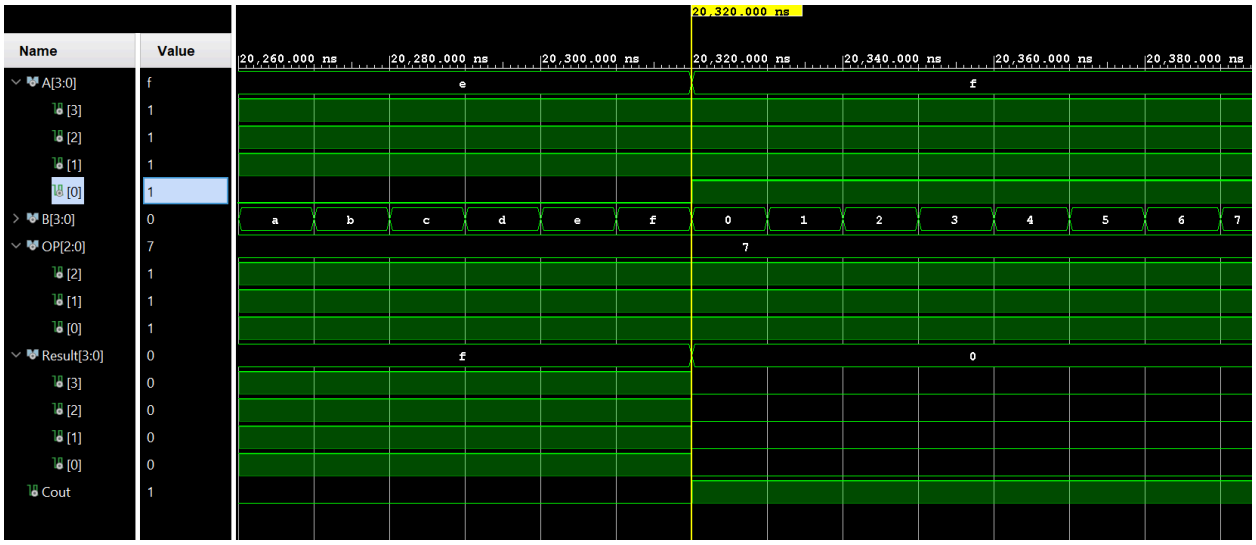

Figure 20: Increment simulation results (OP= “111”, A= “1111”, Result= “0000”, Cout= ‘1’)

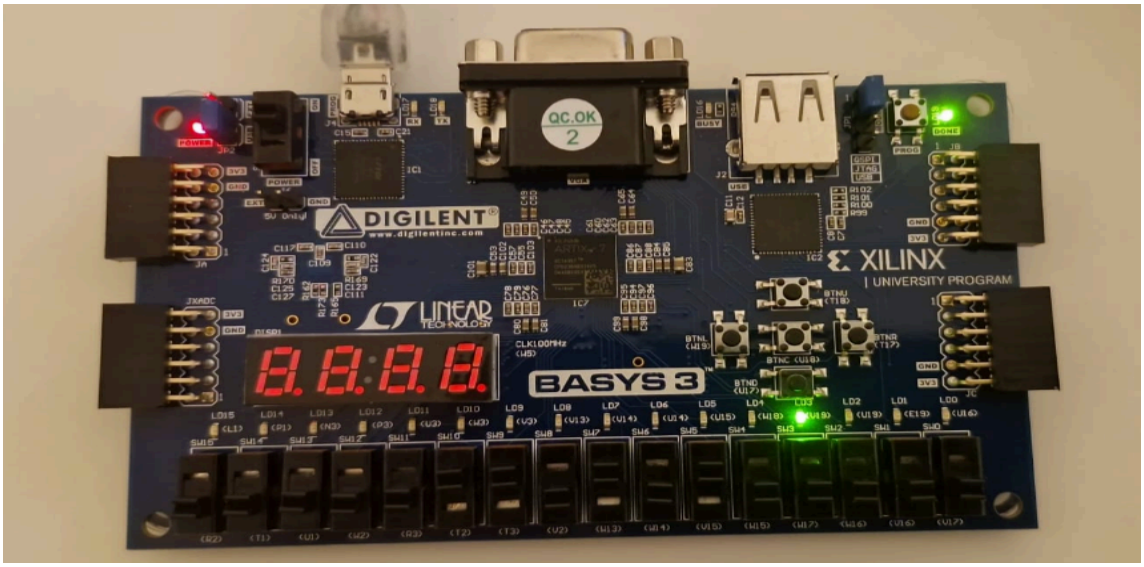

Figure 21: OP= “000”, A= “0011”, B= “0101” Result= “1000”, Cout= ‘0’

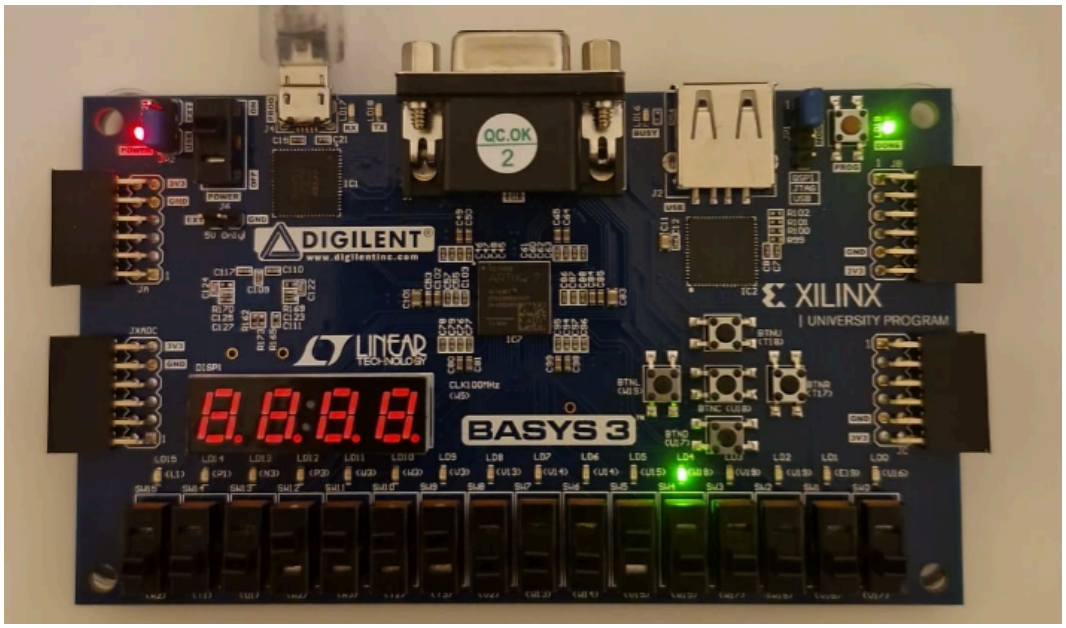

Figure 22: OP= “000”, A= “1111”, B= “0001” Result= “0000”, Cout= ‘1’

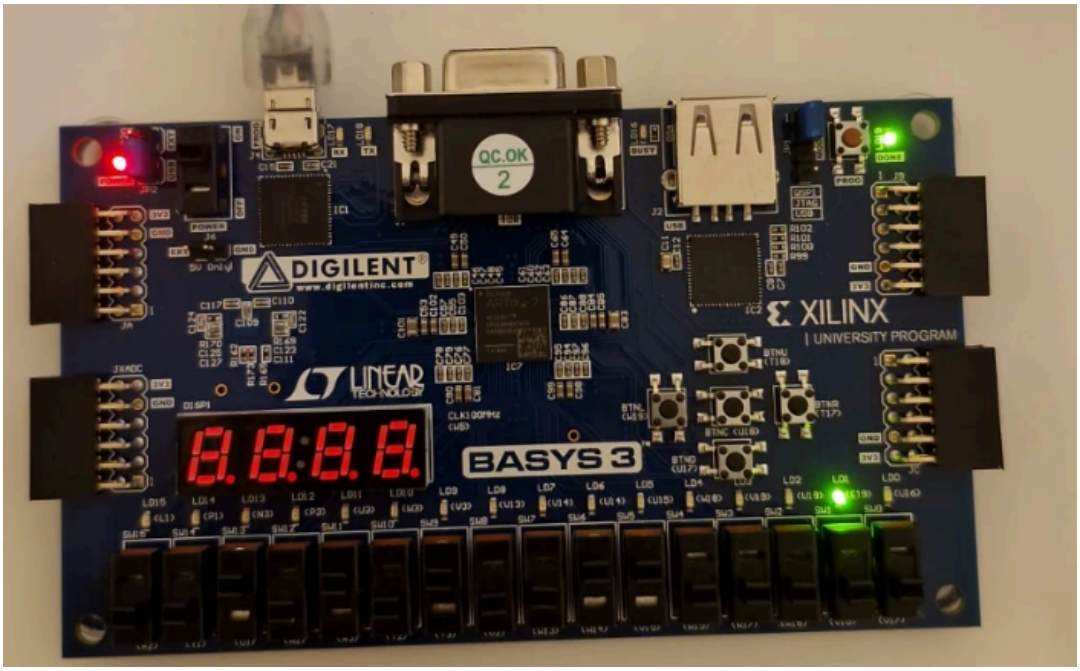

Figure 23: OP= “001”, A= “0101”, B= “0011” Result= “0010”, Cout= ‘0’

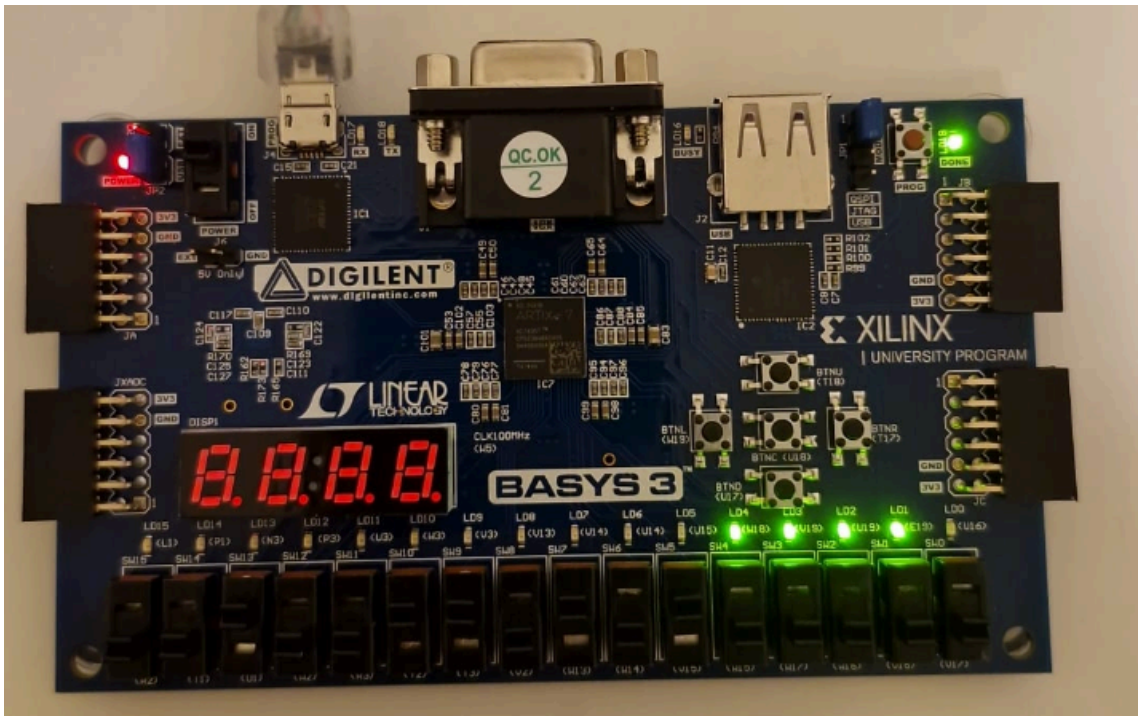

Figure 24: OP= “001”, A= “0011”, B= “0101” Result= “1110”, Cout= ‘1’

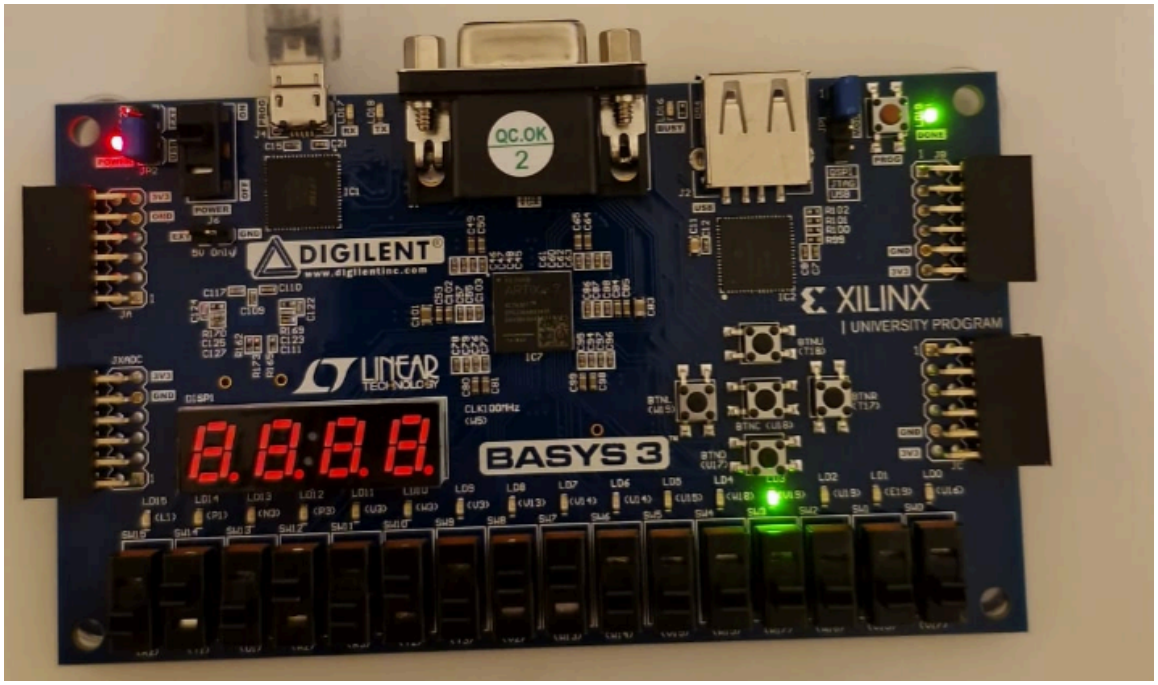

Figure 25: OP= “010”, A= “1010”, B= “1100” Result= “1000”, Cout= ‘0’

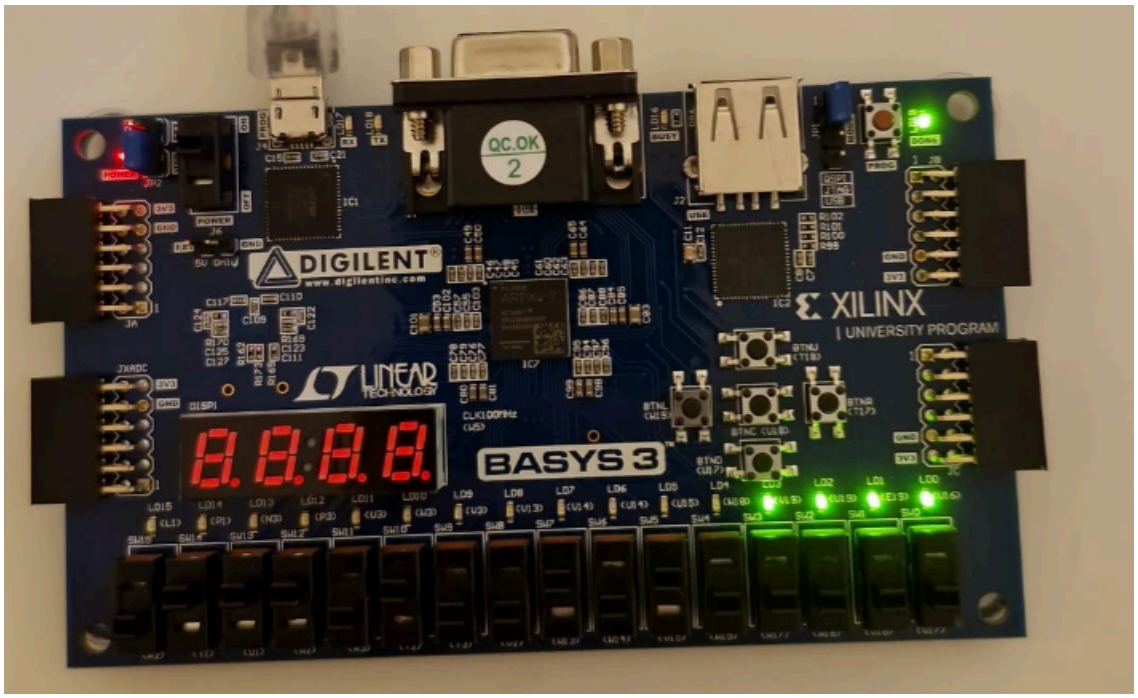

Figure 26: OP= “011”, A= “1010”, B= “0101” Result= “1111”, Cout= ‘0’

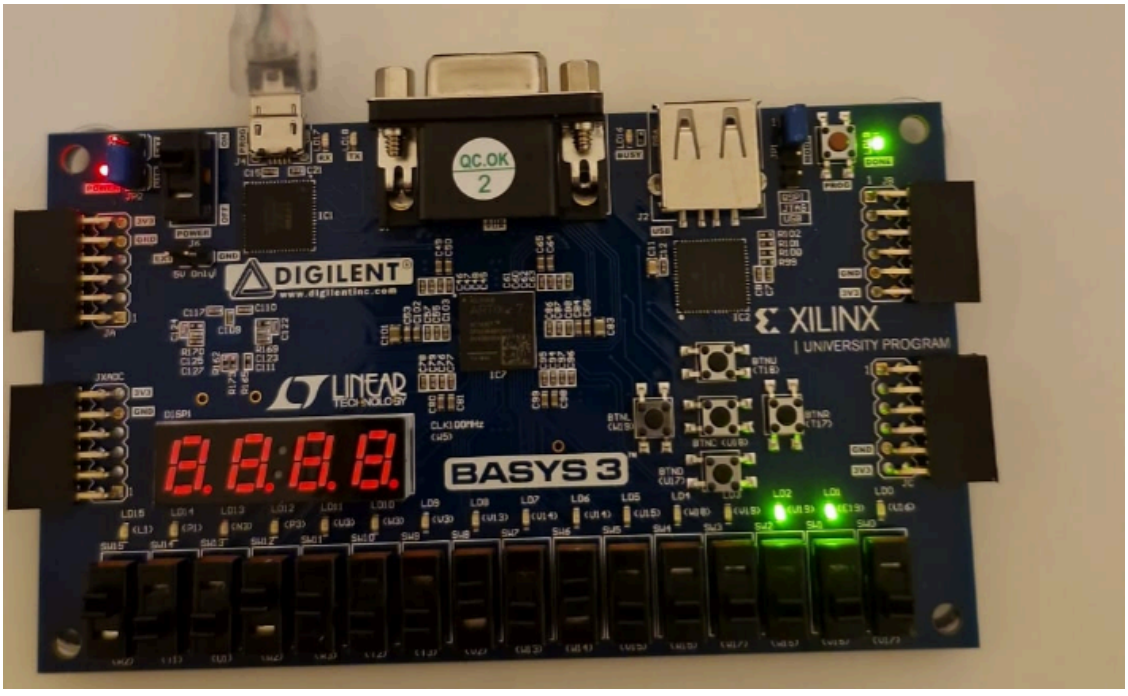

Figure 27: OP= “100”, A= “1100”, B= “1010” Result= “0110”, Cout= ‘0’

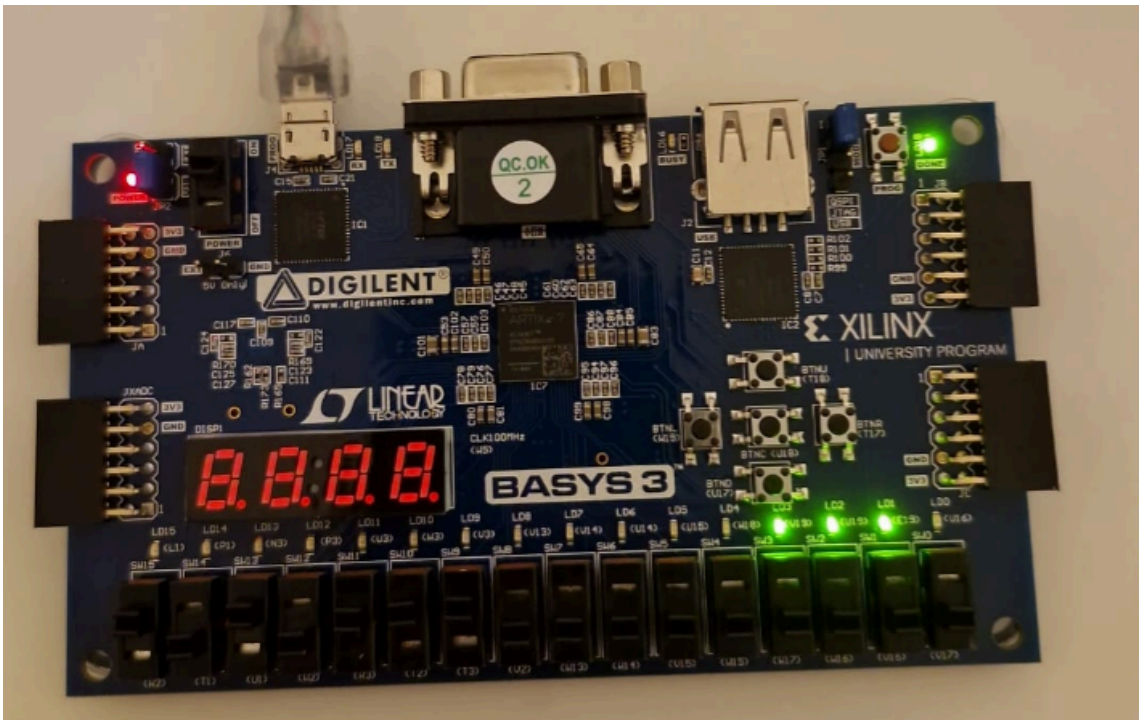

Figure 28: OP= “101”, A= “0111”, Result= “1110”, Cout= ‘0’

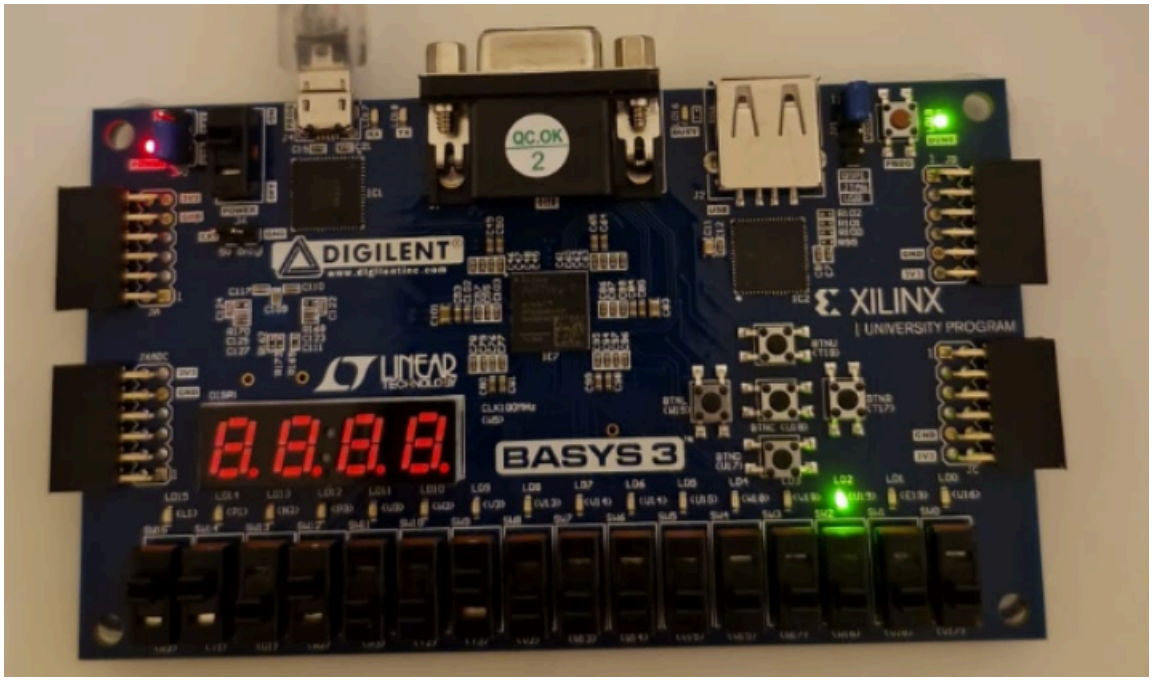

Figure 29: OP= “110”, A= “1001”, Result= “0100”, Cout= ‘0’

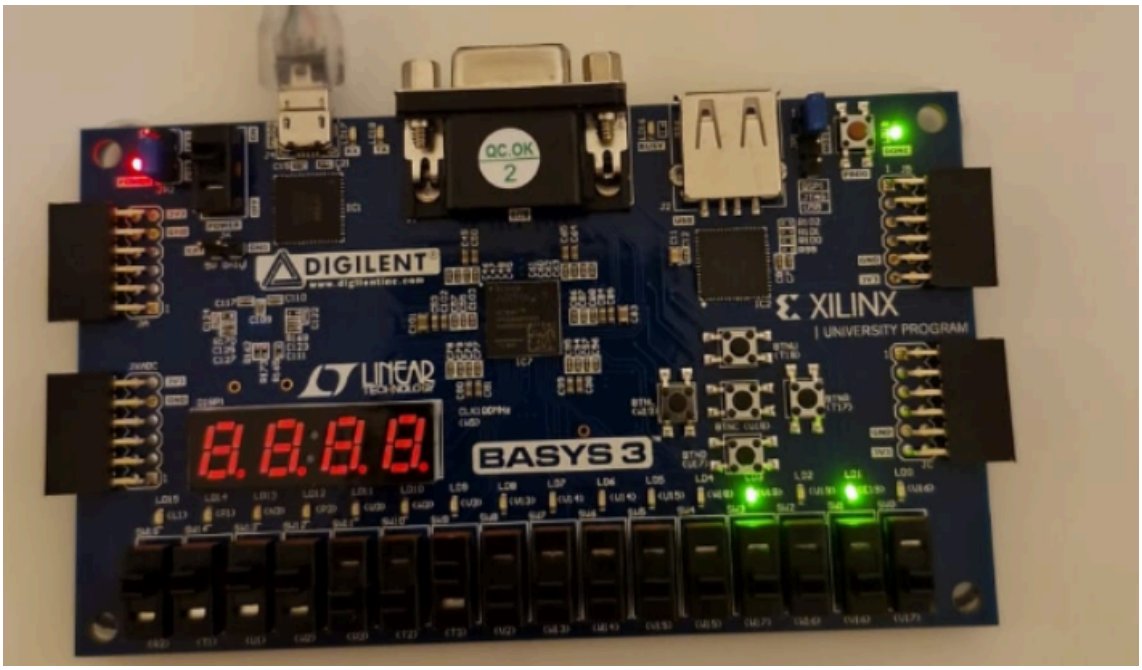

Figure 30: OP= “111”, A= “1001”, Result= “1010”, Cout= ‘0’

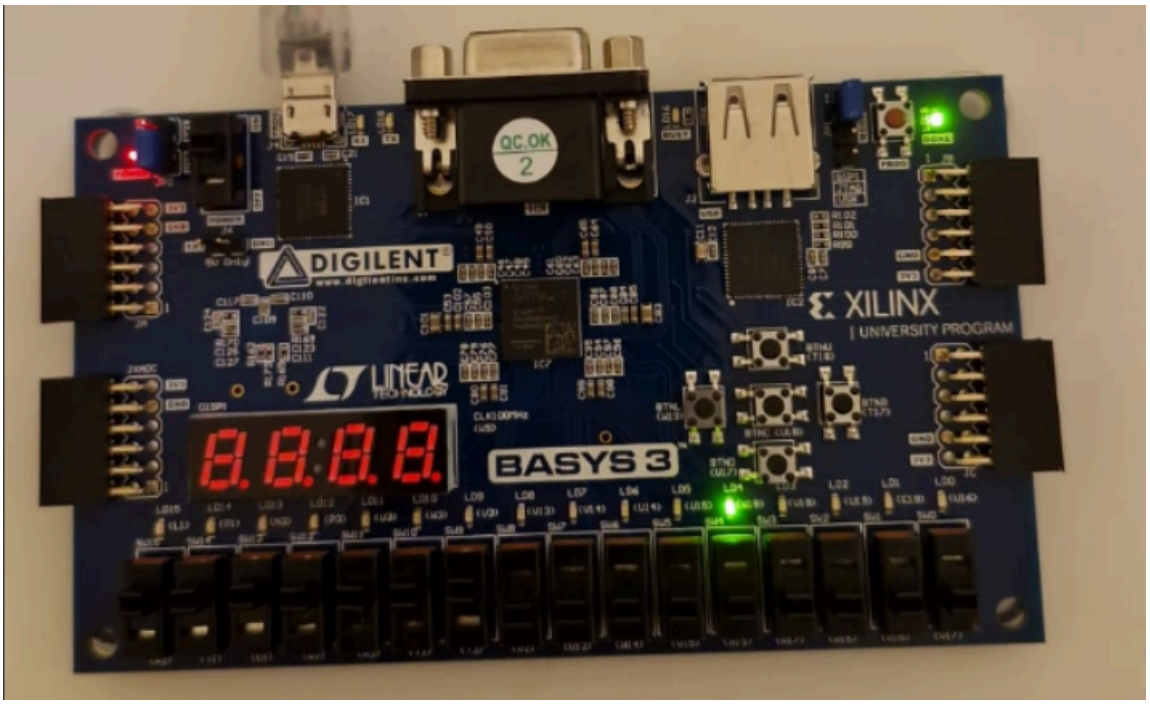

Figure 31: OP= “111”, A= “1111”, Result= “0000”, Cout= ‘1’

CONCLUSION

The 4-bit ALU was successfully designed and tested, and it integrated eight arithmetic and logic operations. Using a multiplexer-based selection, the ALU efficiently executed addition, subtraction, bitwise operations, shifts, and increment. Both simulation and FPGA testing confirmed its correct design, with inputs controlled via switches and results displayed on LEDs. Over the course of this lab, I gained firsthand experience constructing a 4-bit ALU, which refined my VHDL and FPGA testing abilities. It also helped with understanding of concepts such as arithmetic operations, bitwise logic, and carry and overflow handling.

half_adder.vhd

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

entity half_adder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Carry : out STD_LOGIC);
end half_adder;

architecture Behavioral of half_adder is

begin
    Sum <= A xor B;
    Carry <= A and B;

end Behavioral;
```

full_adder.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Cout : out STD_LOGIC);
end full_adder;

architecture Structural of full_adder is

    signal sum1, sum2, carry1, carry2: std_logic;

    component half_adder is
    port
        (
           A : in STD_LOGIC;
           B : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Carry : out STD_LOGIC
        );
    end component;

begin
    HA1: half_adder
    port map (
        A   => A,
        B   => B,
        Sum => sum1,
        Carry   => carry1
    );

    HA2: half_adder
    port map (
        A   => sum1,
        B   => Cin,
        Sum => sum2,
        Carry   => carry2
    );

    Sum <= sum2;
```

```
    Cout <= carry1 or carry2;

end Structural;
```

four_bit_adder.vhd

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

entity four_bit_adder 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 four_bit_adder;

architecture Behavioral of four_bit_adder is
    signal carry: std_logic_vector(3 downto 1);
    component full_adder is
        Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Cout : out STD_LOGIC
    );
    end component;
begin
    FA0: full_adder port map(A(0), B(0), Cin, Sum(0), carry(1));
    FA1: full_adder port map(A(1), B(1), carry(1), Sum(1), carry(2));
    FA2: full_adder port map(A(2), B(2), carry(2), Sum(2), carry(3));
    FA3: full_adder port map(A(3), B(3), carry(3), Sum(3), Cout);

end Behavioral;
```

four_bit_subtractor.vhd

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

```
entity four_bit_subtractor 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);
           Bout : out STD_LOGIC);
end four_bit_subtractor;

architecture Behavioral of four_bit_subtractor is
    signal B_compl : std_logic_vector (3 downto 0);
    signal carry : std_logic_vector(4 downto 0);
    component full_adder is
        Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Cout : out STD_LOGIC
    );
    end component;
begin
    B_compl(0) <= not B(0);
    B_compl(1) <= not B(1);
    B_compl(2) <= not B(2);
    B_compl(3) <= not B(3);

    carry(0) <= '1';

    FA0: full_adder port map(A(0), B_compl(0), carry(0), Diff(0),
carry(1));
    FA1: full_adder port map(A(1), B_compl(1), carry(1), Diff(1),
carry(2));
    FA2: full_adder port map(A(2), B_compl(2), carry(2), Diff(2),
carry(3));
    FA3: full_adder port map(A(3), B_compl(3), carry(3), Diff(3),
carry(4));

    Bout <= not carry(4);

end Behavioral;
```

and_mod.vhd

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

entity and_mod is
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
end and_mod;

architecture Behavioral of and_mod is

begin
    Y <= A and B;

end Behavioral;
```

or_mod.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity or_mod is
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
end or_mod;

architecture Behavioral of or_mod is

begin
    Y <= A or B;

end Behavioral;
```

xor_mod.vhd

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

entity xor_mod is
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
end xor_mod;

architecture Behavioral of xor_mod is

begin
    Y <= A xor B;

end Behavioral;
```

left_shift.vhd

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

entity left_shift is
    Port ( A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0));
end left_shift;

architecture Behavioral of left_shift is

begin
    Y <= A(2 downto 0) & '0';

end Behavioral;
```

right_shift.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity right_shift is
    Port (
        A : in  std_logic_vector(3 downto 0);
        Y : out std_logic_vector(3 downto 0)
    );
end right_shift;

architecture Behavioral of right_shift is
begin
    Y <= '0' & A(3 downto 1);
end Behavioral;
```

increment.vhd

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

entity increment is
    Port ( A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0);
           Cout : out STD_LOGIC);
end increment;

architecture Structural of increment is
    component full_adder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Cout : out STD_LOGIC
         );
    end component;
    signal carry : std_logic_vector(4 downto 0);
begin
    carry(0) <= '0';

    FA0: full_adder port map (A(0), '1', carry(0), Y(0), carry(1));
    FA1: full_adder port map (A(1), '0', carry(1), Y(1), carry(2));
    FA2: full_adder port map (A(2), '0', carry(2), Y(2), carry(3));
```

```
    FA3: full_adder port map (A(3), '0', carry(3), Y(3), carry(4));

    Cout <= carry(4);

end Structural;
```

ALU.vhd

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

entity ALU is
    Port ( A : in std_logic_vector(3 downto 0);
           B : in std_logic_vector(3 downto 0);
           OP : in std_logic_vector(2 downto 0);
           Result : out std_logic_vector(3 downto 0);
           Cout : out std_logic);
end ALU;

architecture Structural of ALU is

    component left_shift
    Port ( A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0));
    end component;

    component right_shift
    Port (
        A : in  std_logic_vector(3 downto 0);
        Y : out std_logic_vector(3 downto 0)
    );
    end component;


    component increment
    Port ( A : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(3 downto 0);
           Cout : out STD_LOGIC);
    end component;

    component four_bit_subtractor
```

```
    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);
           Bout : out STD_LOGIC);
    end component;

    component four_bit_adder
    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;

    component and_mod
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
    end component;

    component or_mod
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
    end component;

    component xor_mod
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Y: out std_logic_vector(3 downto 0)
           );
    end component;

    signal add_result, sub_result, and_result, or_result, xor_result :
std_logic_vector(3 downto 0);
    signal shl_result, shr_result, inc_result                        :
std_logic_vector(3 downto 0);

    signal add_cout, sub_cout, inc_cout : std_logic;
```

```
begin
    ADDER: four_bit_adder
        port map (
        A     => A,
        B     => B,
        Cin   => '0',
        Sum   => add_result,
        Cout  => add_cout
        );

    SUB: four_bit_subtractor port map (A, B, sub_result, sub_cout);
    ANDER: and_mod port map (A, B, and_result);
    ORER: or_mod port map (A, B, or_result);
    XORER: xor_mod port map (A, B, xor_result);
    SL: left_shift port map (A, shl_result);
    SR: right_shift port map (A, shr_result);
    INC: increment port map(A, inc_result, inc_cout);

    process(OP, add_result, sub_result, and_result, or_result, xor_result,
shl_result, shr_result, inc_result, add_cout, sub_cout, inc_cout)
    begin
        case OP is
            when "000" =>
                Result <= add_result;
                Cout   <= add_cout;
            when "001" =>
                Result <= sub_result;
                Cout   <= sub_cout;
            when "010" =>
                Result <= and_result;
                Cout   <= '0';
            when "011" =>
                Result <= or_result;
                Cout   <= '0';
            when "100" =>
                Result <= xor_result;
                Cout   <= '0';
            when "101" =>
                Result <= shl_result;
                Cout   <= '0';
            when "110" =>
                Result <= shr_result;
```

```
                Cout   <= '0';
            when "111" =>
                Result <= inc_result;
                Cout   <= inc_cout;
            when others =>
                Result <= (others => '0');
                Cout   <= '0';
        end case;
    end process;
end Structural;
```

tb_alu.vhd

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

entity tb_ALU is
end tb_ALU;

architecture Behavioral of tb_ALU is
    signal A      : std_logic_vector(3 downto 0);
    signal B      : std_logic_vector(3 downto 0);
    signal OP     : std_logic_vector(2 downto 0);
    signal Result : std_logic_vector(3 downto 0);
    signal Cout   : std_logic;
begin
    UUT: entity work.ALU
        port map (
            A      => A,
            B      => B,
            OP     => OP,
            Result => Result,
            Cout   => Cout
        );

    stimulus: process
        variable op_int : integer;
        variable a_int  : integer;
        variable b_int  : integer;
    begin
```

```
        for op_int in 0 to 7 loop
            OP <= std_logic_vector(to_unsigned(op_int, 3));
            for a_int in 0 to 15 loop
                A <= std_logic_vector(to_unsigned(a_int, 4));
                for b_int in 0 to 15 loop
                    B <= std_logic_vector(to_unsigned(b_int, 4));
                    wait for 10 ns;
                end loop;
            end loop;
        end loop;
        wait;
    end process stimulus;
end Behavioral;
```

alu_constraint.xdc

```
set_property PACKAGE_PIN T3 [get_ports {A[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[0]}]

set_property PACKAGE_PIN T2 [get_ports {A[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[1]}]

set_property PACKAGE_PIN R3 [get_ports {A[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[2]}]

set_property PACKAGE_PIN W2 [get_ports {A[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {A[3]}]

set_property PACKAGE_PIN V15 [get_ports {B[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[0]}]

set_property PACKAGE_PIN W14 [get_ports {B[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[1]}]

set_property PACKAGE_PIN W13 [get_ports {B[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[2]}]

set_property PACKAGE_PIN V2 [get_ports {B[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {B[3]}]

set_property PACKAGE_PIN U1 [get_ports {OP[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP[0]}]
```

```
set_property PACKAGE_PIN T1 [get_ports {OP[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP[1]}]

set_property PACKAGE_PIN R2 [get_ports {OP[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OP[2]}]

set_property PACKAGE_PIN U16 [get_ports {Result[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Result[0]}]

set_property PACKAGE_PIN E19 [get_ports {Result[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Result[1]}]

set_property PACKAGE_PIN U19 [get_ports {Result[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Result[2]}]

set_property PACKAGE_PIN V19 [get_ports {Result[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Result[3]}]

set_property PACKAGE_PIN W18 [get_ports {Cout}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cout}]
```