DIGITAL SYSTEM DESIGN

POCKET CALCULATOR

Implemented and projected by:

Mozacu Ștefania-Cristina

Technical University of Cluj Napoca

Faculty of Automation and Computer Science

Department of Computer Science and Information Technology

Group 30414

Registration Number 210/6430

Course Professor:

Octavian Augustin Cret

Project Supervisor:

Albu Cristian

Table of Contents

Contents

1. Specifications

Calculators are indispensable tools in our daily life, from educational purposes to professional and personal use. For example, they are used by students for solving mathematical problems, by engineers to do quick computations and even by financial professionals for handling numeric data.

The Pocket Calculator Project is designed to provide a simple, yet functional arithmetic tool using VHDL (VHSIC Hardware Description Language). This calculator performs basic operations such as addition, subtraction, multiplication, and division. It leverages FPGA (Field-Programmable Gate Array) technology to perform real-time calculations and display results on the seven-segment display.

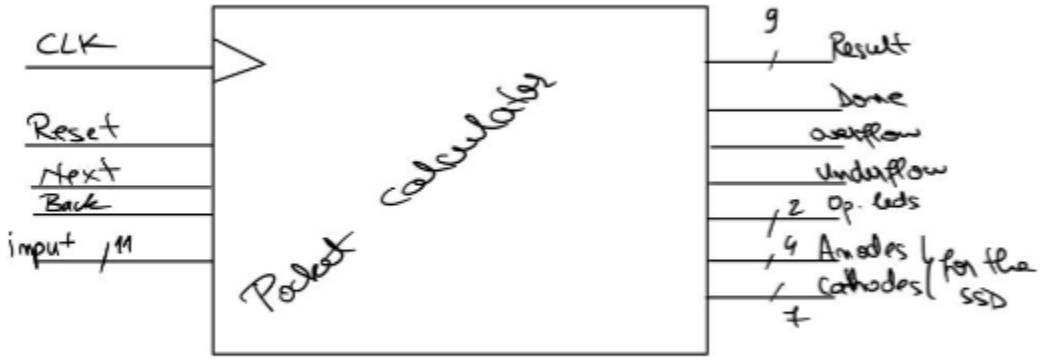
The operation flow of the calculator involves several stages: input stage (user inputs the first number, then he selects the desired arithmetic operation, then he inputs the second number), calculation stage (it processes the inputs and performs the operation, then the result is displayed), output stage (the result is displayed on the seven-segment display and the overflow and underflow conditions are indicated on LEDs), reset and back operations (the reset button clears all inputs and resets the calculator, the back button restores the previous state or operation, allowing corrections or adjustments).

Implementing the calculator in VHDL on a FPGA offers speed (due to parallel processing capabilities), customization (the design can be easily modified or expanded to include more complex operations or additional features) and reliability.

The pocket calculator demonstrates the practical application of VHDL in creating a functional tool. This calculator serves as an educational tool, a professional aid, and for personal use, illustrating the utility of digital design in our daily lives.

2. Design

a) Black Box



b) Inputs/Outputs

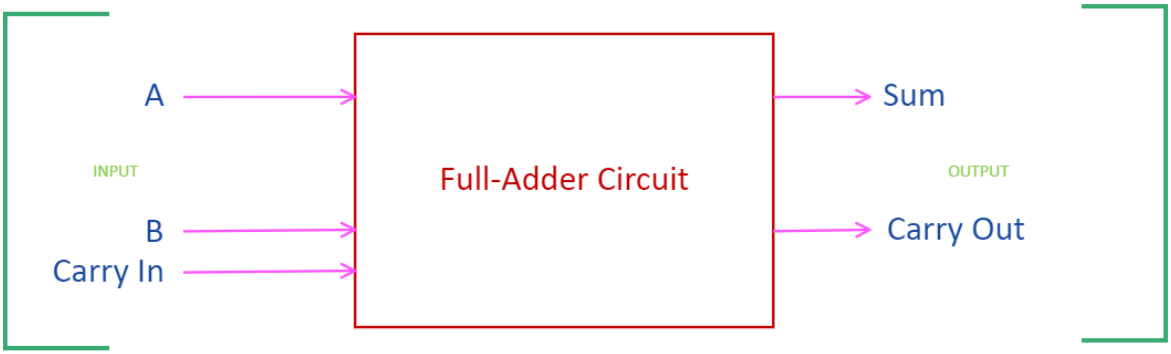
The clock signal provides the timing reference for all synchronous operations within the calculator and is of type STD_LOGIC. Then, the reset signal resets the calculator to its initial state and clears all the registers and outputs. The next button is used for going through the states of the machine, it advances the input sequence and triggers arithmetic operations. To go to the previous state or operation, allowing the user to correct inputs or review previous results, we have the back button. The input has 12 bits: the first 8 bits store the number, the 9th bit is the sign (0 for positive and 1 for negative) and the other 2 bits are used as selection for the arithmetic operations (00 for addition, 01 for subtraction, 10 for multiplication, 11 for division).

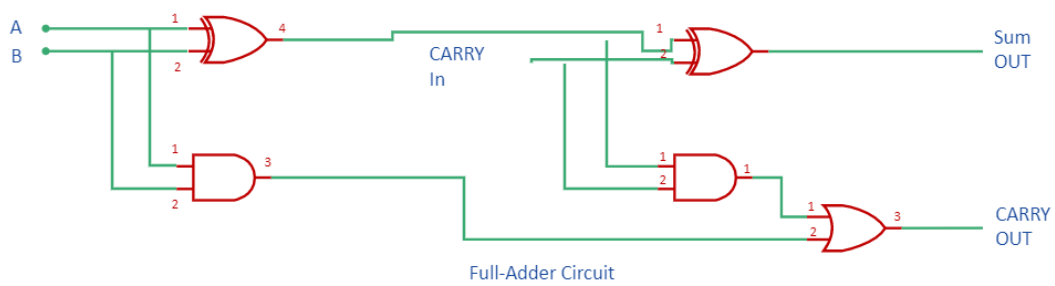
The result output displays the result of the arithmetic operation. The op LEDs indicate the current operation selected. The anodes and cathodes control the seven-segment display. The done signal indicates the completion of an arithmetic operation. The overflow and underflow detect if the conditions occur during the arithmetic operations.

c) Components

i. Adder

The 1-bit adder is the fundamental building block for the larger adder components. It is designed to add two single-bit binary numbers, along with an input carry bit, and produce a sum and a carry-out.





The truth table is as follows.

Full-Adder Truth Table

A	B	Cin	S	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

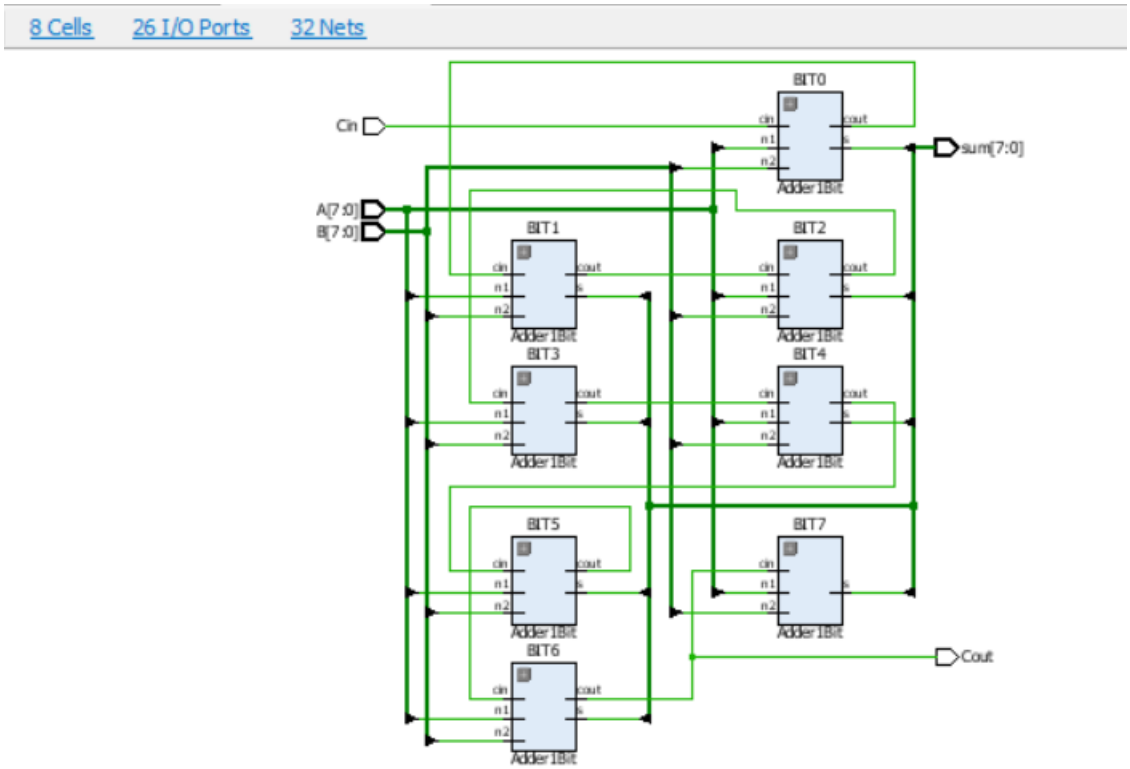
From the truth table we deduce the Karnaugh map.

```
        AB
        00 01 11 10
Cin  +------------
  0  |  0  0  1  0
  1  |  0  1  1  1
```

The 1 bit full adder takes 3 inputs: n1,n2 and cin. Then, it calculates the sum and the carry out.

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

entity Adder1Bit is
   Port ( n1 : in STD_LOGIC;
          n2 : in STD_LOGIC;
          cin : in STD_LOGIC;
          cout : out STD_LOGIC;
          s : out STD_LOGIC);
end Adder1Bit;

architecture Behavioral of Adder1Bit is
begin
   process(n1,n2,cin)
   begin
      s<=(n1 xor n2) xor cin; -- Sum Bit
      cout<=(n1 and n2) or (cin and (n1 xor n2)); --Cout Bit
   end process;
end Behavioral;
```

The 8-bit adder extends the functionality of the 1-bit adder to handle 8-bit binary numbers. It consists of eight 1-bit adders connected in series, where the carry-out of each adder becomes the carry-in of the next.




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

entity Adder8Bit is
   Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
          B : in STD_LOGIC_VECTOR (7 downto 0);
          Cin : in STD_LOGIC;
          Cout : out STD_LOGIC;
          sum : out STD_LOGIC_VECTOR (7 downto 0));
       --   enable : in STD_LOGIC);
end Adder8Bit;

architecture Behavioral of Adder8Bit is
component Adder1Bit is
   Port ( n1 : in STD_LOGIC;
          n2 : in STD_LOGIC;
          cin : in STD_LOGIC;
          cout : out STD_LOGIC;
          s : out STD_LOGIC);
end component;

signal carry : STD_LOGIC_VECTOR (7 downto 0):= (others=>'0');
```

```
signal internal_sum : STD_LOGIC_VECTOR (7 downto 0):= (others=>'0');
signal internal_cout : STD_LOGIC;
begin
-- Cascading 1Bit Full adders
BIT0: Adder1Bit Port map(n1 => A(0), n2 => B(0), cin => Cin, s => internal_sum(0), cout => carry(0)); -- least significant bit
BIT1: Adder1Bit Port map(n1 => A(1), n2 => B(1), cin => carry(0), s => internal_sum(1), cout => carry(1));
BIT2: Adder1Bit Port map(n1 => A(2), n2 => B(2), cin => carry(1), s => internal_sum(2), cout => carry(2));
BIT3: Adder1Bit Port map(n1 => A(3), n2 => B(3), cin => carry(2), s => internal_sum(3), cout => carry(3));
BIT4: Adder1Bit Port map(n1 => A(4), n2 => B(4), cin => carry(3), s => internal_sum(4), cout => carry(4));
BIT5: Adder1Bit Port map(n1 => A(5), n2 => B(5), cin => carry(4), s => internal_sum(5), cout => carry(5));
BIT6: Adder1Bit Port map(n1 => A(6), n2 => B(6), cin => carry(5), s => internal_sum(6), cout => carry(6));
BIT7: Adder1Bit Port map(n1 => A(7), n2 => B(7), cin => carry(6), s => internal_sum(7), cout => internal_cout); -- most significat bit
sum <= internal_sum;
Cout <= carry(7);
end Behavioral;
```

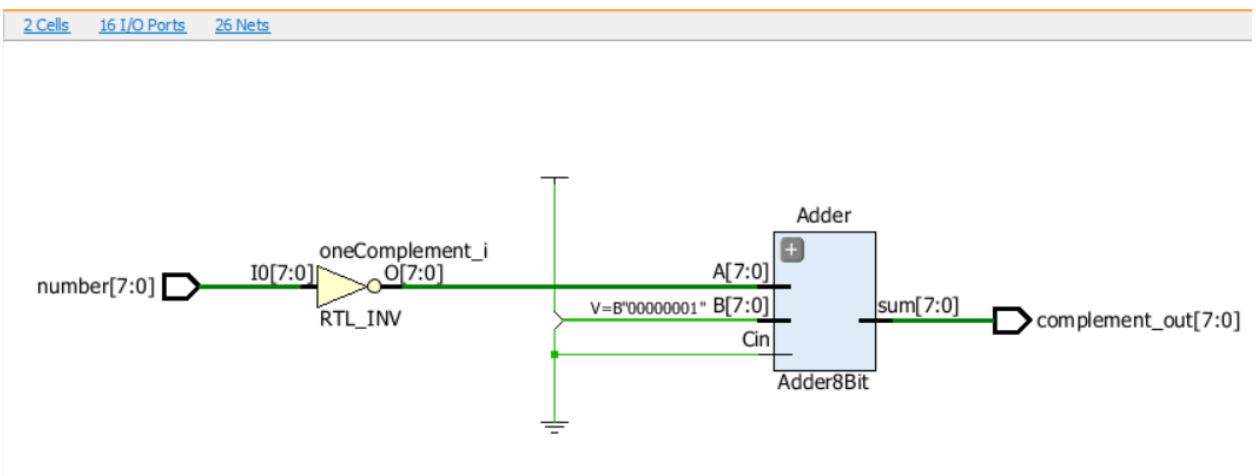
ii. Complement of 2

The 2's complement is a mathematical operation used to represent negative binary numbers and perform binary subtraction. We use it for the subtraction in order to use the same adder as before.

To find the 2's complement of a binary number, we need to invert all the bits and add 1 to the least significant bit of the result.

To subtract B from A, we compute the 2's complement of B and add it to A.

Binary Input	1's Complement	+1 (2's Complement)
0000000	1111111	10000000
0000001	1111110	1111111
0000010	1111101	1111110
...	...	...
1111111	0000000	0000001




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

entity Complement2 is
   Port ( number : in STD_LOGIC_VECTOR (7 downto 0);
          complement_out : out STD_LOGIC_VECTOR (7 downto 0));
end Complement2;

architecture Behavioral of Complement2 is
   component Adder8Bit is
      Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
             B : in STD_LOGIC_VECTOR (7 downto 0);
             Cin : in STD_LOGIC;
```

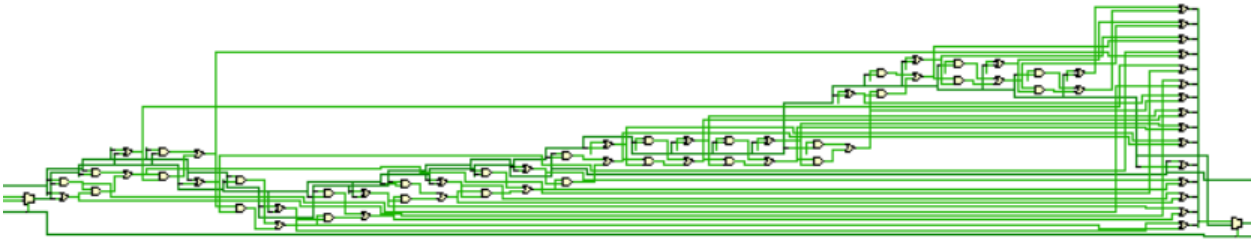
```
        Cout : out STD_LOGIC;
        Sum : out STD_LOGIC_VECTOR (7 downto 0));
    end component;

    signal oneComplement : STD_LOGIC_VECTOR (7 downto 0);
    signal carry_out : STD_LOGIC;
    signal one : STD_LOGIC_VECTOR (7 downto 0) := "00000001"; -- to add 1 to the number
    signal adder_sum : STD_LOGIC_VECTOR (7 downto 0);

begin
    -- Adder8Bit instantiation
    Adder: Adder8Bit Port map (A => oneComplement, B => one, Cin => '0', Cout => carry_out, Sum => adder_sum);

    -- 1's complement
    oneComplement <= not number; -- negate all the bits from the input

    -- Output assignment
    complement_out <= adder_sum;

end Behavioral;
```

iii. Multiplier

The multiplier component is responsible for performing the multiplication of two 8-bit binary numbers, producing a 16-bit binary product. This component is designed to handle multiplication using a combination of shifting and addition operations.

The multiplier component performs binary multiplication by iteratively adding shifted versions of the first number based on the bits of the second number. This process, combined with full adder logic for handling binary addition and carry propagation, produces the correct 16-bit product.

The schematic is build by 8 cascaded elements that look like this.



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

entity Multiplicator is
   Port ( n1 : in STD_LOGIC_VECTOR (7 downto 0);
          n2 : in STD_LOGIC_VECTOR (7 downto 0);
          product : out STD_LOGIC_VECTOR (15 downto 0));
         -- enable : in STD_LOGIC); -- Enable signal
end Multiplicator;
architecture Behavioral of Multiplicator is
begin
process(n1, n2)
   -- we use variables, so the changes happen instantly, instead of signals
   variable P : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); -- product
   variable N1Extended : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); -- extended n1 to 16 bits
   variable N2Reg : STD_LOGIC_VECTOR (7 downto 0); -- Register for the second number
   variable carry : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); -- carry signals for addition
   variable N1Shifted : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); -- signal for shifted n1
   variable temp_sum : STD_LOGIC;
   variable temp_carry : STD_LOGIC;
begin
--if enable='1' then
   -- Extend the first number to 16 bits
   N1Extended(7 downto 0) := n1;
   N1Extended(15 downto 8) := (others => '0');
```

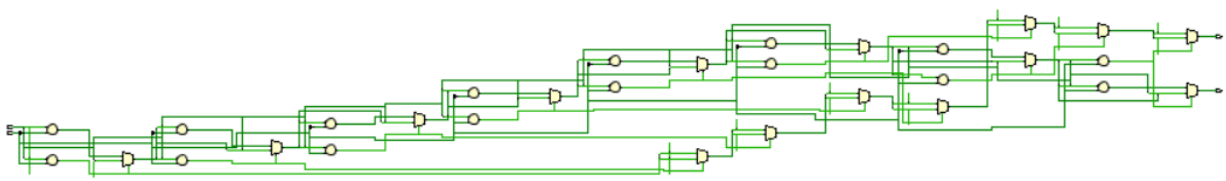
```
N2Reg := n2;

-- Initialize product to 0 and set the initial shifted N1
P := (others => '0');
N1Shifted := N1Extended;

-- Iterate over each bit of the second number
for i in 0 to 7 loop
  if N2Reg(i) = '1' then
    -- Add shifted N1Shifted to the product
    carry(0) := '0';
    for j in 0 to 15 loop
      --we can't instantiate the full adder in a process, so I rewrote the equations
      temp_sum := N1Shifted(j) xor P(j) xor carry(j);
      temp_carry := (N1Shifted(j) and P(j)) or (carry(j) and (N1Shifted(j) xor P(j)));

      P(j) := temp_sum;
      if j < 15 then
        carry(j+1) := temp_carry;
      end if;
    end loop;
  end if;

  -- Shift N1Extended left by 1 bit for the next iteration
  N1Shifted(15 downto 1) := N1Shifted(14 downto 0);
  N1Shifted(0) := '0'; -- Fill Least Significant Bit with 0
end loop;
```

```
    product <= P; -- Assign final product
  --end if;
  end process;
  end Behavioral;
```

iv. Divider

The divider component in the Pocket Calculator performs the division of two 8-bit unsigned binary numbers, producing an 8-bit quotient and an 8-bit remainder.

The Subtract-and-Shift Algorithm repeatedly subtracts the divisor from a portion of the dividend and shifts the result. It works similarly to how you might manually divide numbers using long division.



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

-- Entity definition for an 8-bit divider
entity SimpleDivider8Bit is
  Port (
    dividend  : in  STD_LOGIC_VECTOR(7 downto 0); -- 8-bit dividend input
    divisor   : in  STD_LOGIC_VECTOR(7 downto 0); -- 8-bit divisor input
    quotient  : out STD_LOGIC_VECTOR(7 downto 0); -- 8-bit quotient output
    remainder : out STD_LOGIC_VECTOR(7 downto 0) -- 8-bit remainder output
  );
end SimpleDivider8Bit;
```

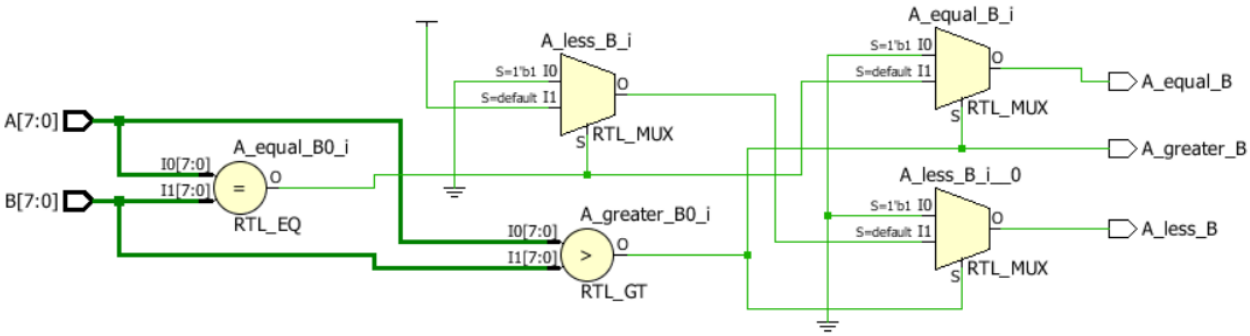
```
architecture Behavioral of SimpleDivider8Bit is
begin
  -- Process to perform the division
  process(dividend, divisor)
    -- Temporary variables for calculations
    variable temp_dividend : UNSIGNED(15 downto 0);  -- 16-bit temporary dividend
    variable temp_divisor  : UNSIGNED(7 downto 0);  -- 8-bit temporary divisor
    variable temp_quotient : UNSIGNED(7 downto 0) := (others => '0'); -- 8-bit temporary quotient, initialized to 0
    variable temp_remainder: UNSIGNED(7 downto 0);  -- 8-bit temporary remainder
  begin
    -- Extend the 8-bit dividend to 16 bits by appending 8 zeros
    temp_dividend := UNSIGNED(dividend) & "00000000";
    -- Convert divisor to unsigned type
    temp_divisor := UNSIGNED(divisor);
    -- Initialize quotient and remainder to 0
    temp_quotient := (others => '0');
    temp_remainder := (others => '0');

    -- Loop to perform the subtract-and-shift division algorithm
    for i in 7 downto 0 loop
      -- Shift left the remainder and bring down the next bit of the dividend
      temp_remainder := temp_remainder(6 downto 0) & temp_dividend(15);
      -- Shift left the dividend
      temp_dividend := temp_dividend(14 downto 0) & '0';
```

```
        -- Check if the current remainder is greater than or equal to the divisor
        if temp_remainder >= temp_divisor then
            -- If so, subtract the divisor from the remainder
            temp_remainder := temp_remainder - temp_divisor;
            -- Set the corresponding bit of the quotient to 1
            temp_quotient := temp_quotient(6 downto 0) & '1';
        else
            -- Otherwise, set the corresponding bit of the quotient to 0
            temp_quotient := temp_quotient(6 downto 0) & '0';
        end if;
    end loop;

    -- Assign the calculated quotient and remainder to the output ports
    quotient <= STD_LOGIC_VECTOR(temp_quotient);
    remainder <= STD_LOGIC_VECTOR(temp_remainder);
  end process;
end Behavioral;
```

v. Comparator

The comparator is responsible for determining the numbers' relative magnitudes. It outputs signals indicating whether the first number is greater than, equal to, or less than the second number.

In subtraction, the sign of the result depends on the sign of the biggest number.

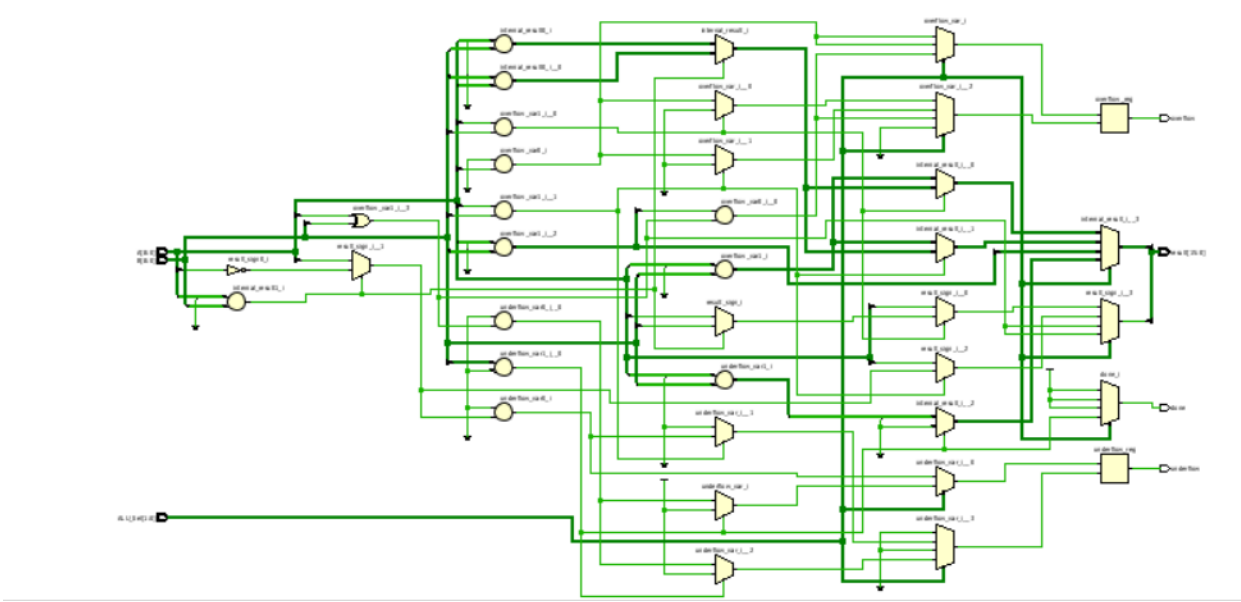


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Comparator8Bit is
   Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
          B : in STD_LOGIC_VECTOR (7 downto 0);
          A_greater_B : out STD_LOGIC;
          A_equal_B : out STD_LOGIC;
          A_less_B : out STD_LOGIC);
end Comparator8Bit;

architecture Behavioral of Comparator8Bit is
begin
   process(A, B)
   begin
      if A > B then
         A_greater_B <= '1';
         A_equal_B <= '0';
         A_less_B <= '0';
      elsif A = B then
         A_greater_B <= '0';
         A_equal_B <= '1';
         A_less_B <= '0';
      else
         A_greater_B <= '0';
         A_equal_B <= '0';
         A_less_B <= '1';
      end if;
   end process;
end Behavioral;
```

vi. ALU

The Arithmetic Logic Unit performs all the arithmetic operations, while also handling the sign of the result.

Operation	ALU_Sel	A_sign	B_sign	Result_sign	Description
Addition	"00"	0	0	0	Regular addition
	"00"	1	1	1	Regular addition with negative sign
	"00"	0	1	0	Subtract B from A
	"00"	1	0	1	Subtract A from B
Subtraction	"01"	0	0	0	Regular subtraction
	"01"	1	1	1	Regular subtraction with negative sign
Multiplication	"10"	0	0	0	Regular multiplication
	"10"	1	1	0	Multiplication with positive result
	"10"	0	1	1	Multiplication with negative result
	"10"	1	0	1	Multiplication with negative result
Division	"11"	0	0	0	Regular division
	"11"	1	1	0	Division with positive result
	"11"	0	1	1	Division with negative result
	"11"	1	0	1	Division with negative result



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

entity ALU is
   Port (
      A : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
      B : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
      ALU_Sel : in STD_LOGIC_VECTOR (1 downto 0); -- 2-bit control
signal for selecting operation
      clk : in STD_LOGIC;
      result : out STD_LOGIC_VECTOR (15 downto 0); -- 16-bit result
including sign for multiplication
      done : out STD_LOGIC; -- operation complete signal
      overflow : out STD_LOGIC; -- Overflow signal
      underflow : out STD_LOGIC -- Underflow signal
   );
end ALU;

architecture Behavioral of ALU is
   signal A_unsigned, B_unsigned : STD_LOGIC_VECTOR (7 downto 0);
   signal A_sign, B_sign : STD_LOGIC;
   signal result_sign : STD_LOGIC;
   signal quotient, remainder : STD_LOGIC_VECTOR (7 downto 0);
   signal quotient_unsigned, remainder_unsigned : unsigned(7 downto
0);
   signal complement_result : STD_LOGIC_VECTOR (7 downto 0);
   signal adder_result : STD_LOGIC_VECTOR (7 downto 0);
   signal adder_carry_out : STD_LOGIC;
   signal mult_result : STD_LOGIC_VECTOR (15 downto 0);
   signal div_done : STD_LOGIC;
   signal A_greater_B, A_equal_B, A_less_B : STD_LOGIC;
   signal internal_result : STD_LOGIC_VECTOR (15 downto 0);

   -- Components for different operations
   component Adder8Bit is
      Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
            B : in STD_LOGIC_VECTOR (7 downto 0);
            Cin : in STD_LOGIC;
            Cout : out STD_LOGIC;
            sum : out STD_LOGIC_VECTOR (7 downto 0));
   end component;

   component Complement2 is
      Port ( number : in STD_LOGIC_VECTOR (7 downto 0);
            complement_out : out STD_LOGIC_VECTOR (7 downto 0));
   end component;
```

```
    component Multiplicator is
      Port ( n1 : in STD_LOGIC_VECTOR (7 downto 0);
             n2 : in STD_LOGIC_VECTOR (7 downto 0);
             product : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component Divider8Bit is
      Port (
        clk : in std_logic;
        start : in std_logic;
        dividend : in unsigned(7 downto 0);
        divisor : in unsigned(7 downto 0);
        quotient : out unsigned(7 downto 0);
        remainder : out unsigned(7 downto 0);
        done : out std_logic
      );
    end component;

    component Comparator8Bit is
      Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
             B : in STD_LOGIC_VECTOR (7 downto 0);
             A_greater_B : out STD_LOGIC;
             A_equal_B : out STD_LOGIC;
             A_less_B : out STD_LOGIC);
    end component;

  begin
    -- Extract sign and unsigned parts
    A_unsigned <= A(7 downto 0);
    B_unsigned <= B(7 downto 0);
    A_sign <= A(8);
    B_sign <= B(8);

    -- Instantiate components with intermediate signals
    Addition: Adder8Bit Port map (
      A => A_unsigned,
      B => B_unsigned,
      Cin => '0',
      Cout => adder_carry_out,
      sum => adder_result
    );

    Complement: Complement2 Port map (
      number => B_unsigned,
      complement_out => complement_result
    );
```

```
Multiplication: Multiplicator Port map (
   n1 => A_unsigned,
   n2 => B_unsigned,
   product => mult_result
);

Division: Divider8Bit Port map (
   clk => clk,
   start => '1', -- Ensure proper management of start signal
   dividend => unsigned(A_unsigned),
   divisor => unsigned(B_unsigned),
   quotient => quotient_unsigned,
   remainder => remainder_unsigned,
   done => div_done
);

Comparison: Comparator8Bit Port map (
   A => A_unsigned,
   B => B_unsigned,
   A_greater_B => A_greater_B,
   A_equal_B => A_equal_B,
   A_less_B => A_less_B
);

process (clk)
begin
   if rising_edge(clk) then
      -- Reset done signal
      done <= '0';
      overflow <= '0';
      underflow <= '0';

      case ALU_Sel is
         when "00" =>
            -- Addition
            if (A_sign xor B_sign) = '1' then
               -- Signs are different, so it's effectively a subtraction
               if A_sign = '1' then
                  -- A is negative, B is positive
                  result_sign <= A_sign;
                  internal_result  <=  std_logic_vector(resize(signed('0'  &
adder_result) - signed('0' & complement_result), 16));
               else
                  -- A is positive, B is negative
                  result_sign <= A_sign;
```

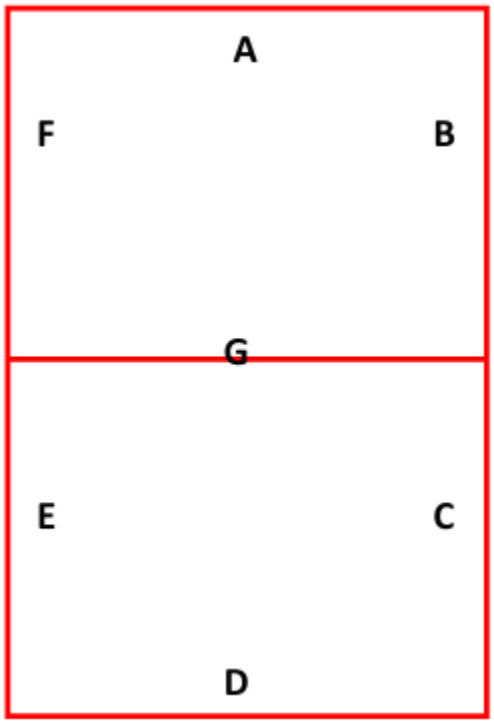
```
                internal_result <= std_logic_vector(resize(signed('0' & adder_result) - signed('0' & complement_result), 16));
            end if;
        else
            -- Both have the same sign, it's a regular addition
            result_sign <= A_sign;
            internal_result <= std_logic_vector(resize(signed('0' & adder_result), 16));
        end if;
        -- Check for overflow
        if internal_result(15) /= result_sign then
            overflow <= '1';
        end if;
        done <= '1';

    when "01" =>
        -- Subtraction
        if A_unsigned >= B_unsigned then
            result_sign <= A_sign;
            internal_result <= std_logic_vector(resize(signed('0' & A_unsigned) - signed('0' & B_unsigned), 16));
        else
            result_sign <= B_sign;
            internal_result <= std_logic_vector(resize(signed('0' & B_unsigned) - signed('0' & A_unsigned), 16));
        end if;
        -- Check for underflow
        if internal_result(15) /= result_sign then
            underflow <= '1';
        end if;
        done <= '1';

    when "10" =>
        -- Multiplication
        result_sign <= A_sign xor B_sign; -- Result sign is determined by the XOR of input signs
        internal_result <= mult_result;
        -- Check for overflow
        if internal_result(15) /= result_sign then
            overflow <= '1';
        end if;
        done <= '1';

    when "11" =>
        -- Division
        result_sign <= A_sign xor B_sign; -- Result sign is determined by the XOR of input signs
```

```
                if div_done = '1' then
                  internal_result <=
std_logic_vector(resize(quotient_unsigned, 16));
                  -- Check for underflow
                  if internal_result(15) /= result_sign then
                    underflow <= '1';
                  end if;
                  done <= '1';
                end if;

              when others =>
                internal_result <= (others => '0');
                done <= '0';
            end case;
          end if;
        end process;

        -- Convert the quotient and remainder back to std_logic_vector
for output
        quotient <= std_logic_vector(quotient_unsigned);
        remainder <= std_logic_vector(remainder_unsigned);

        result <= internal_result;

      end Behavioral;
```

vii. Seven-Segment Display

The Seven-Segment Display is responsible for displaying numerical and operational outputs. The architecture of the SSD defines the processes for controlling the display, refreshing the digits and converting binary values to BCD for display.

A seven-segment display consists of 7 LEDs (labeled A to G) arranged in a rectangular to form the number 8. An additional LD (labeled DP) is used for decimal point. The segments are controlled by the cathode's signals, and the digit to bedisplayed is selected by the anode signals.



The clock_100Mhz is necessary for a smooth display refreshing, to avoid flickering and to create an illusion of a steady display. High-frequency clocks allow precise control over timing intervals, which is critical for generating accurate delays and refresh periods.

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

entity seven_segment_display is
  Port (
    clock_100Mhz : in STD_LOGIC; -- 100Mhz clock on Basys 3 FPGA board
    reset : in STD_LOGIC; -- reset
```

```
        Anode_Activate : out STD_LOGIC_VECTOR (3 downto 0); -- 4 Anode signals
        Cathode_out : out STD_LOGIC_VECTOR (6 downto 0); -- Cathode patterns of 7-segment display
        Display_Value : in STD_LOGIC_VECTOR (15 downto 0); -- Value to display (including sign)
        Display_Type : in STD_LOGIC -- Type of display: 0 for number, 1 for operation
    );
end seven_segment_display;

architecture Behavioral of seven_segment_display is
    signal one_second_counter : unsigned (27 downto 0) := (others => '0');
    signal one_second_enable : std_logic := '0';
    signal displayed_number : STD_LOGIC_VECTOR (15 downto 0) := (others => '0');
    signal LED_BCD : STD_LOGIC_VECTOR (3 downto 0);
    signal refresh_counter : unsigned (19 downto 0) := (others => '0');
    signal LED_activating_counter : std_logic_vector(1 downto 0);

    -- Signals for BCD conversion
    signal thousands, hundreds, tens, unit : STD_LOGIC_VECTOR (3 downto 0);
    signal sign : STD_LOGIC; -- Signal to store the sign of the number
begin
-- BCD to 7-segment decoder
process(LED_BCD, Display_Type, Display_Value)
begin
    if Display_Type = '1' then
        case Display_Value(2 downto 0) is
            when "000" => Cathode_out <= "0000001"; -- "0"
            when "001" => Cathode_out <= "1001111"; -- "1"
```

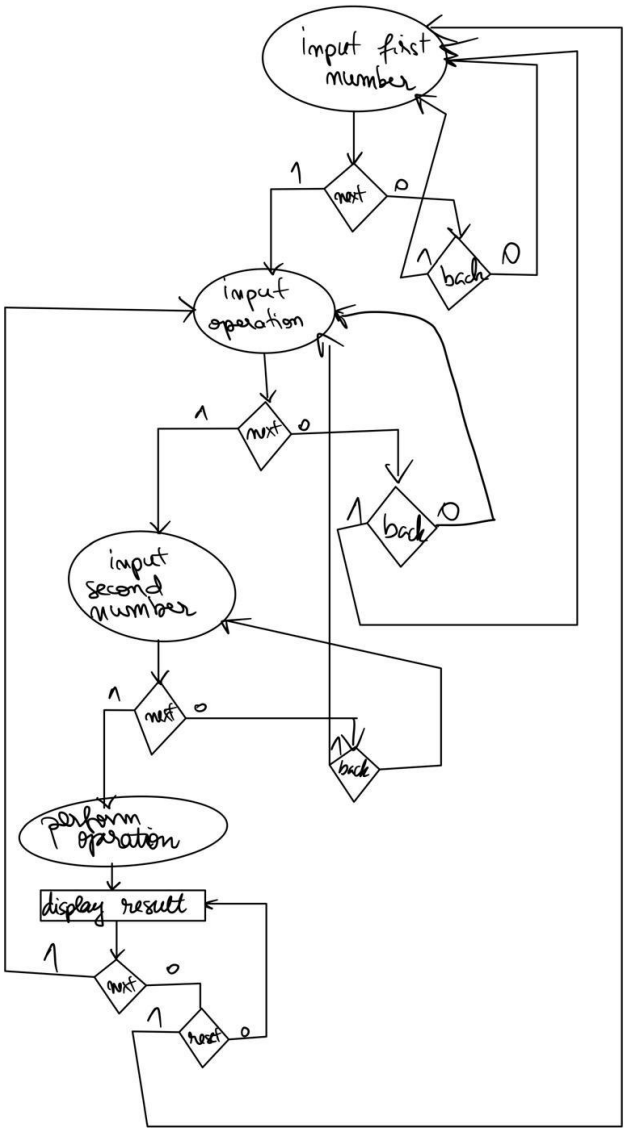
```
            when "010" => Cathode_out <= "0010010"; -- "2"
            when "011" => Cathode_out <= "0000110"; -- "3"
            when "100" => Cathode_out <= "1001100"; -- "4"
            when "101" => Cathode_out <= "0100100"; -- "5"
            when "110" => Cathode_out <= "0100000"; -- "6"
            when "111" => Cathode_out <= "0001111"; -- "7"
            when others => Cathode_out <= "1111111"; -- off
        end case;
    else
        case LED_BCD is
            when "0000" => Cathode_out <= "0000001"; -- "0"
            when "0001" => Cathode_out <= "1001111"; -- "1"
            when "0010" => Cathode_out <= "0010010"; -- "2"
            when "0011" => Cathode_out <= "0000110"; -- "3"
            when "0100" => Cathode_out <= "1001100"; -- "4"
            when "0101" => Cathode_out <= "0100100"; -- "5"
            when "0110" => Cathode_out <= "0100000"; -- "6"
            when "0111" => Cathode_out <= "0001111"; -- "7"
            when "1000" => Cathode_out <= "0000000"; -- "8"
            when "1001" => Cathode_out <= "0000100"; -- "9"
            when others => Cathode_out <= "1111111"; -- off
        end case;
    end if;
end process;

-- Generate refresh period of 10.5ms
process(clock_100Mhz, reset)
begin
    if reset = '1' then
```

```
        refresh_counter <= (others => '0');
    elsif rising_edge(clock_100Mhz) then
        refresh_counter <= refresh_counter + 1;
    end if;
end process;

LED_activating_counter <= std_logic_vector(refresh_counter(19 downto
18));

-- 4-to-1 MUX to generate anode activating signals for 4 LEDs
process(LED_activating_counter, thousands, hundreds, tens, unit, sign)
begin
    case LED_activating_counter is
        when "00" =>
            Anode_Activate <= "0111";
            if sign = '1' then
                LED_BCD <= "1010"; -- Display "-" for negative numbers
            else
                LED_BCD <= thousands;
            end if;
        when "01" =>
            Anode_Activate <= "1011";
            LED_BCD <= hundreds;
        when "10" =>
            Anode_Activate <= "1101";
            LED_BCD <= tens;
        when "11" =>
            Anode_Activate <= "1110";
            LED_BCD <= unit;
        when others =>
```

```
            Anode_Activate <= "1111"; -- Turn off all segments
        end case;
    end process;

    -- Counting the number to be displayed on 4-digit 7-segment Display
    process(clock_100Mhz, reset)
    begin
        if reset = '1' then
            one_second_counter <= (others => '0');
        elsif rising_edge(clock_100Mhz) then
            if one_second_counter >= x"5F5E0FF" then
                one_second_counter <= (others => '0');
            else
                one_second_counter <= one_second_counter + 1;
            end if;
        end if;
    end process;
    one_second_enable <= '1' when one_second_counter = x"5F5E0FF" else '0';

    -- Binary to BCD conversion process
    process(Display_Value, Display_Type)
        variable value : integer;
        variable bcd_thousands, bcd_hundreds, bcd_tens, bcd_units : integer;
        variable temp_value : integer;
    begin
        if Display_Type = '0' then
            -- Handle the sign
            if Display_Value(15) = '1' then
                value := to_integer(signed(Display_Value(14 downto 0)));
                sign <= '1'; -- Negative number
```

```
        else
          value := to_integer(unsigned(Display_Value(14 downto 0)));
          sign <= '0'; -- Positive number
        end if;
        -- Extract BCD digits
        bcd_thousands := value / 1000;
        temp_value := value mod 1000;
        bcd_hundreds := temp_value / 100;
        temp_value := temp_value mod 100;
        bcd_tens := temp_value / 10;
        bcd_units := temp_value mod 10;
        -- Assign to signals
        thousands <= std_logic_vector(to_unsigned(bcd_thousands, 4));
        hundreds <= std_logic_vector(to_unsigned(bcd_hundreds, 4));
        tens <= std_logic_vector(to_unsigned(bcd_tens, 4));
        unit <= std_logic_vector(to_unsigned(bcd_units, 4));
      else
        thousands <= "0000";
        hundreds <= "0000";
        tens <= "0000";
        unit <= Display_Value(3 downto 0);
        sign <= '0';
      end if;
    end process;

    end Behavioral;
```

d) State Diagram

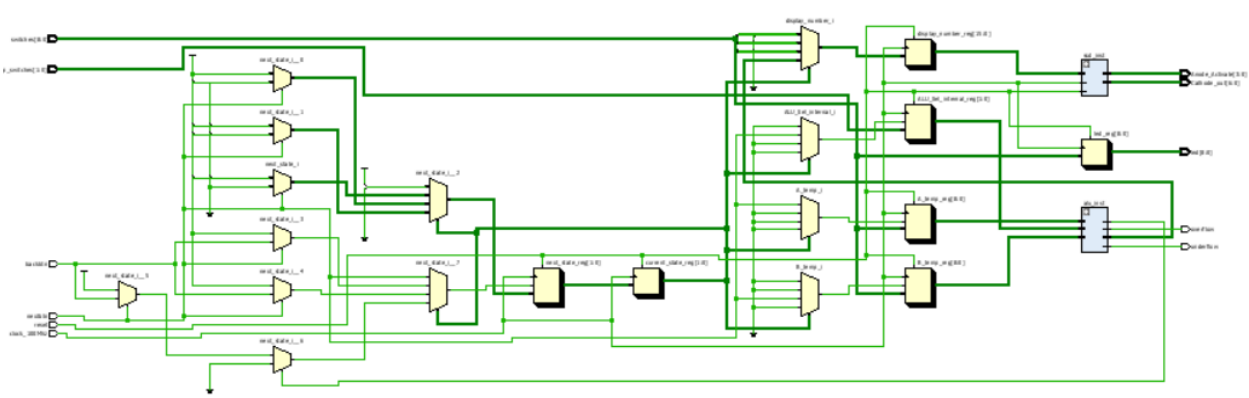


When the reset button is pressed, it goes from any state to the initial one. When the next button is pressed, it goes from INIT to Input_A, from Input_A to Input_OP, from Input_OP to Input_B, from Input_B to Perform_OP, from Perform_OP to Display_Rez, from Display_Rez to Done. The back button transitions from any state to the previous one.

e) Calculator code

The calculator entity integrates two primary components: the ALU and the Seven-Segment Display controller. This ensures that the calculator can perform arithmetic operations and display the results in a user-friendly manner.

```
+----------------------+
|      Calculator      |
|                      |
| +--------+  +------+ |
| |  ALU   |  | SSD  | |
| |        |  |      | |
| |  A_reg |-->display_value--->Anodes
| |  B_reg |  |      |  Cathodes
| | ALU_Sel|  |      |
| +--------+  +------+ |
+----------------------+
```



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity ALU_System is

Port (

```
    clock_100Mhz : in STD_LOGIC; -- 100Mhz clock on Basys 3 FPGA board
    reset : in STD_LOGIC; -- reset button
    nextbtn : in STD_LOGIC; -- button to move to the next state
    backbtn : in STD_LOGIC; -- button to move to the previous state
    switches : in STD_LOGIC_VECTOR (8 downto 0); -- 8 bits for number, 1 for sign
    op_switches : in STD_LOGIC_VECTOR (1 downto 0); -- 2-bit control signal for selecting operation
    Anode_Activate : out STD_LOGIC_VECTOR (3 downto 0); -- 4 Anode signals
    Cathode_out : out STD_LOGIC_VECTOR (6 downto 0); -- Cathode patterns of 7-segment display
    overflow : out STD_LOGIC; -- Overflow signal
    underflow : out STD_LOGIC; -- Underflow signal
    led : out STD_LOGIC_VECTOR (8 downto 0) -- LEDs for switch status
  );
end ALU_System;

architecture Behavioral of ALU_System is

  type state_type is (wait_for_first_num, wait_for_operation, wait_for_second_num, display_result);
  signal current_state, next_state : state_type;

  signal A, B : STD_LOGIC_VECTOR (8 downto 0);
  signal result : STD_LOGIC_VECTOR (15 downto 0);
  signal ALU_Sel_internal : STD_LOGIC_VECTOR (1 downto 0);
  signal alu_done, alu_overflow, alu_underflow : STD_LOGIC;

  signal A_temp, B_temp : STD_LOGIC_VECTOR (8 downto 0);
  signal display_number : STD_LOGIC_VECTOR (15 downto 0);

  component ALU is
```

```
    Port (
        A : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
        B : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
        ALU_Sel : in STD_LOGIC_VECTOR (1 downto 0); -- 2-bit control signal for selecting operation
        result : out STD_LOGIC_VECTOR (15 downto 0); -- 16-bit result including sign for multiplication
        done : out STD_LOGIC; -- operation complete signal
        overflow : out STD_LOGIC; -- Overflow signal
        underflow : out STD_LOGIC -- Underflow signal
    );
  end component;

  component SevenSegmentDisplay is
    Port (
        clock_100Mhz : in STD_LOGIC; -- 100Mhz clock on Basys 3 FPGA board
        reset : in STD_LOGIC; -- reset
        Anode_Activate : out STD_LOGIC_VECTOR (3 downto 0); -- 4 Anode signals
        Cathode_out : out STD_LOGIC_VECTOR (6 downto 0); -- Cathode patterns of 7-segment display
        Number_Display : in STD_LOGIC_VECTOR (15 downto 0) -- Input Number
    );
  end component;

begin

  -- Instantiate ALU
  alu_inst : ALU
    port map (
        A => A_temp,
        B => B_temp,
```

```
        ALU_Sel => ALU_Sel_internal,
        result => result,
        done => alu_done,
        overflow => alu_overflow,
        underflow => alu_underflow
    );

-- Instantiate Seven Segment Display
ssd_inst : SevenSegmentDisplay
    port map (
        clock_100Mhz => clock_100Mhz,
        reset => reset,
        Anode_Activate => Anode_Activate,
        Cathode_out => Cathode_out,
        Number_Display => display_number
    );

process (clock_100Mhz, reset)
begin
    if reset = '1' then
        led <= (others => '0');
    elsif rising_edge(clock_100Mhz) then
        led <= switches; -- Map the switches directly to the LEDs
    end if;
end process;

-- State transition and signal assignment
process (clock_100Mhz, reset)
begin
    if reset = '1' then
```

```
        current_state <= wait_for_first_num;
            next_state <= wait_for_first_num;
            A <= (others => '0');
            B <= (others => '0');
            A_temp <= (others => '0');
            B_temp <= (others => '0');
            ALU_Sel_internal <= (others => '0');
            display_number <= (others => '0');
    elsif rising_edge(clock_100Mhz) then
        current_state <= next_state;

        -- State-specific assignments
        case current_state is
          when wait_for_first_num =>
            display_number <= std_logic_vector(resize(signed(('0' & switches(7 downto 0))),
16));
            display_number(15) <= switches(8); -- Assign sign bit to 16th bit

            if nextbtn = '1' then
              A_temp <= switches;
              A <= switches;
              next_state <= wait_for_operation;
            end if;

          when wait_for_operation =>
            display_number <= std_logic_vector(resize(signed(('0' & switches(7 downto 0))),
16));
            display_number(15) <= switches(8); -- Assign sign bit to 16th bit

            if nextbtn = '1' then
              ALU_Sel_internal <= op_switches;
```

```
            next_state <= wait_for_second_num;
        elsif backbtn = '1' then
            next_state <= wait_for_first_num;
        end if;

    when wait_for_second_num =>
        display_number <= std_logic_vector(resize(signed(('0' & switches(7 downto 0))),
16));
        display_number(15) <= switches(8); -- Assign sign bit to 16th bit

        if nextbtn = '1' then
            B_temp <= switches;
            B <= switches;
            next_state <= display_result;
        elsif backbtn = '1' then
            next_state <= wait_for_operation;
        end if;

    when display_result =>
        display_number <= result;

        if alu_done = '1' then
            if nextbtn = '1' then
                A <= result(8 downto 0); -- Store result as the first number for the next
operation
                next_state <= wait_for_operation;
            elsif backbtn = '1' then
                next_state <= wait_for_second_num;
            end if;
        end if;
```

```
            when others =>
                next_state <= wait_for_first_num;
        end case;
    end if;
end process;

-- Output overflow and underflow signals
overflow <= alu_overflow;
underflow <= alu_underflow;

end Behavioral;
```

f) Constraints file

Below is the constraints file that maps the GPGA pins to the appropriate ports of the Calculator entity. This file ensures that the signals are correctly routed to the physical pins on the FPGA board.

```
## Clock signal
set_property PACKAGE_PIN W5 [get_ports clock_100Mhz]
set_property IOSTANDARD LVCMOS33 [get_ports clock_100Mhz]

## Reset button
set_property PACKAGE_PIN U18 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

## Next button
set_property PACKAGE_PIN T17 [get_ports nextbtn]
set_property IOSTANDARD LVCMOS33 [get_ports nextbtn]

## Previous button
set_property PACKAGE_PIN W19 [get_ports backbtn]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports backbtn]

## Switches (8 bits for number + 1 bit for sign)
set_property PACKAGE_PIN V17 [get_ports {switches[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[0]}]

set_property PACKAGE_PIN V16 [get_ports {switches[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[1]}]

set_property PACKAGE_PIN W16 [get_ports {switches[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[2]}]

set_property PACKAGE_PIN W17 [get_ports {switches[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[3]}]

set_property PACKAGE_PIN W15 [get_ports {switches[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[4]}]

set_property PACKAGE_PIN V15 [get_ports {switches[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[5]}]

set_property PACKAGE_PIN W14 [get_ports {switches[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[6]}]

set_property PACKAGE_PIN W13 [get_ports {switches[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[7]}]

## Sign bit
set_property PACKAGE_PIN V2 [get_ports {switches[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[8]}]
```

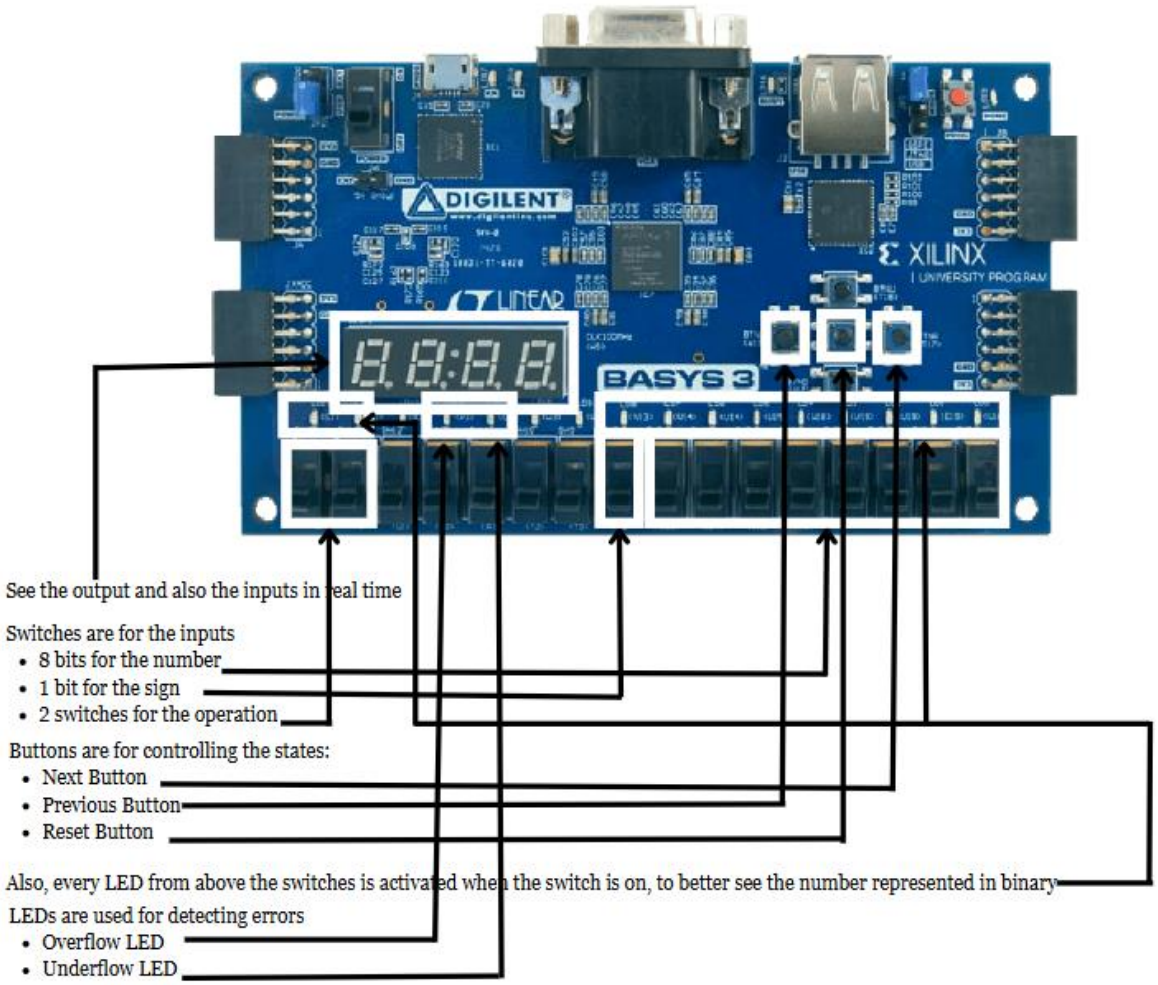
```
## Operation switches (2 bits)
set_property PACKAGE_PIN R3 [get_ports {op_switches[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {op_switches[0]}]

set_property PACKAGE_PIN W2 [get_ports {op_switches[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {op_switches[1]}]

## LEDs for switch status
set_property PACKAGE_PIN U16 [get_ports {led[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]

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

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

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

set_property PACKAGE_PIN W18 [get_ports {led[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[4]}]

set_property PACKAGE_PIN U15 [get_ports {led[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[5]}]

set_property PACKAGE_PIN U14 [get_ports {led[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[6]}]
```

```
set_property PACKAGE_PIN V14 [get_ports {led[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[7]}]

## Sign bit LED
set_property PACKAGE_PIN V13 [get_ports {led[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[8]}]

## Overflow LED
set_property PACKAGE_PIN U3 [get_ports overflow]
set_property IOSTANDARD LVCMOS33 [get_ports overflow]

## Underflow LED
set_property PACKAGE_PIN P3 [get_ports underflow]
set_property IOSTANDARD LVCMOS33 [get_ports underflow]

## Seven-segment LED display
set_property PACKAGE_PIN W7 [get_ports {Cathode_out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[6]}]
set_property PACKAGE_PIN W6 [get_ports {Cathode_out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[5]}]
set_property PACKAGE_PIN U8 [get_ports {Cathode_out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[4]}]
set_property PACKAGE_PIN V8 [get_ports {Cathode_out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[3]}]
set_property PACKAGE_PIN U5 [get_ports {Cathode_out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[2]}]
set_property PACKAGE_PIN V5 [get_ports {Cathode_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[1]}]
set_property PACKAGE_PIN U7 [get_ports {Cathode_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Cathode_out[0]}]
```

```
set_property PACKAGE_PIN V7 [get_ports dp]
set_property IOSTANDARD LVCMOS33 [get_ports dp]
set_property PACKAGE_PIN U2 [get_ports {Anode_Activate[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[0]}]
set_property PACKAGE_PIN U4 [get_ports {Anode_Activate[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[1]}]
set_property PACKAGE_PIN V4 [get_ports {Anode_Activate[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[2]}]
set_property PACKAGE_PIN W4 [get_ports {Anode_Activate[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[3]}]

## Configuration Settings
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

## Done signal
set_property PACKAGE_PIN T17 [get_ports done]
set_property IOSTANDARD LVCMOS33 [get_ports done]
```

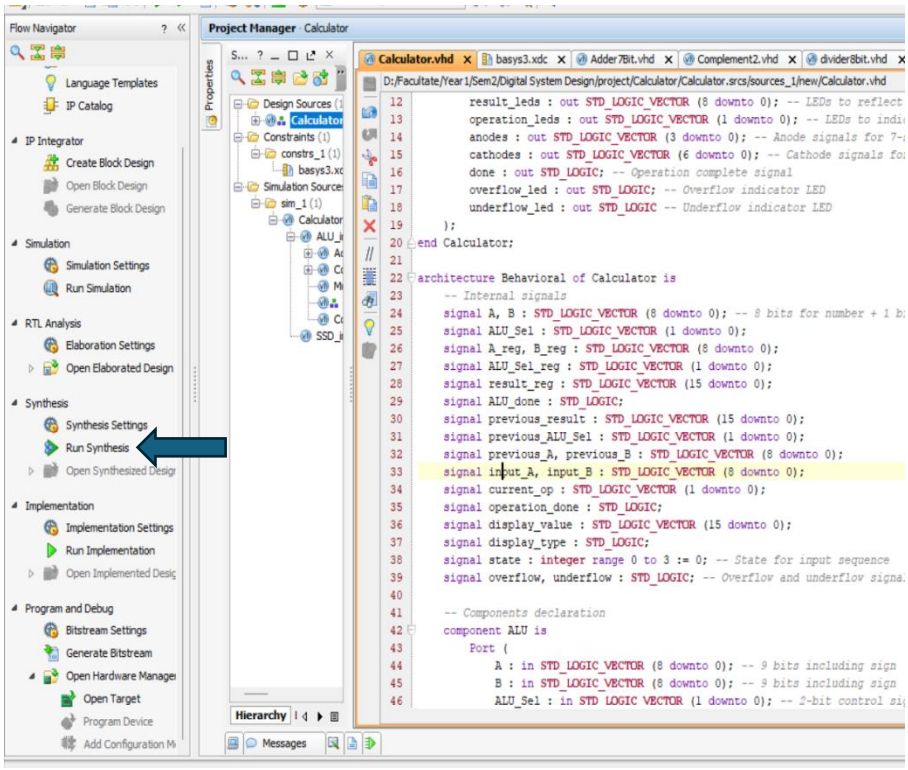
3. User Manual

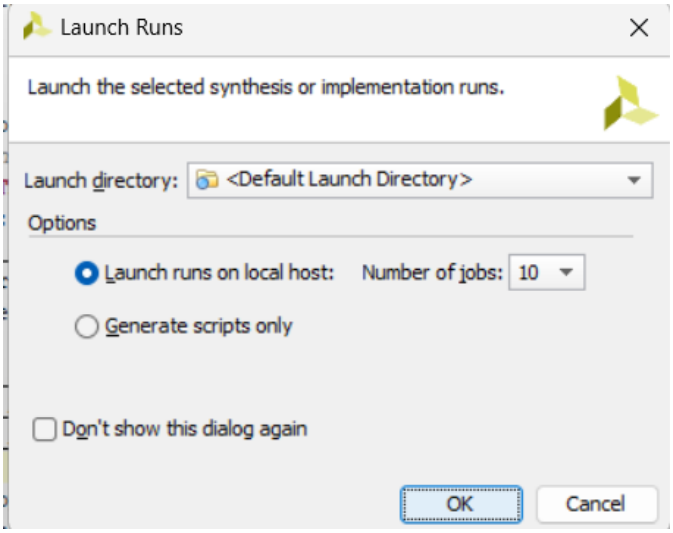
The Basys 3 board is an FPGA development board designed by Digilent, which is equipped with a Xilinx Artix-7 FPGA. This board is widely used in educational environments and by hobbyists for learning and implementing digital logic designs.

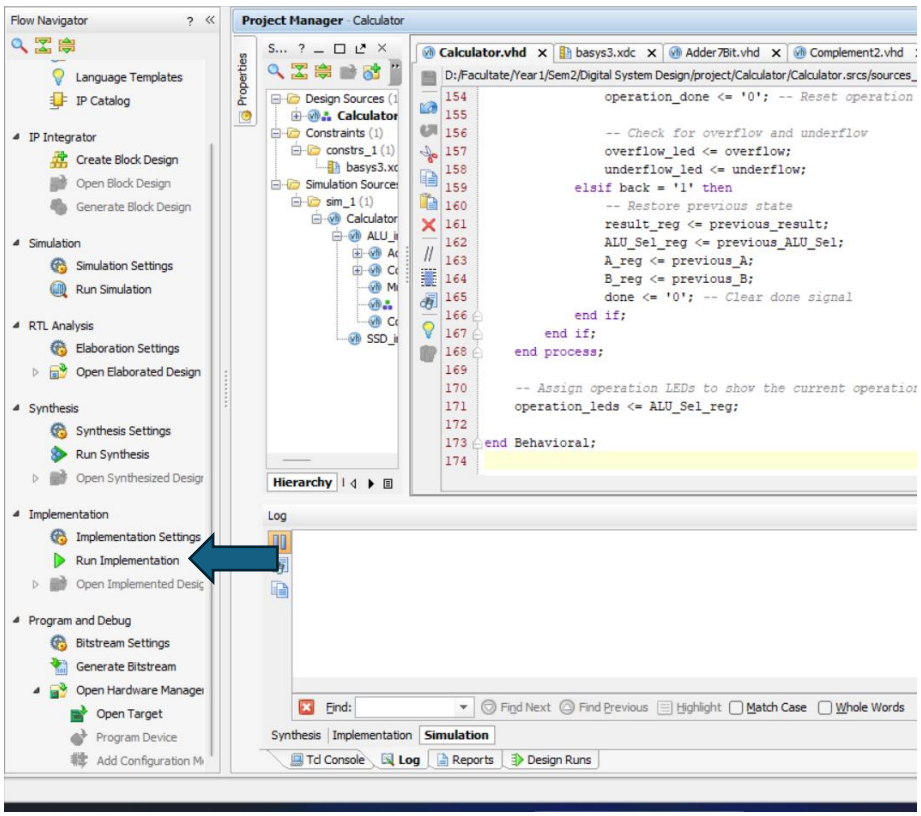


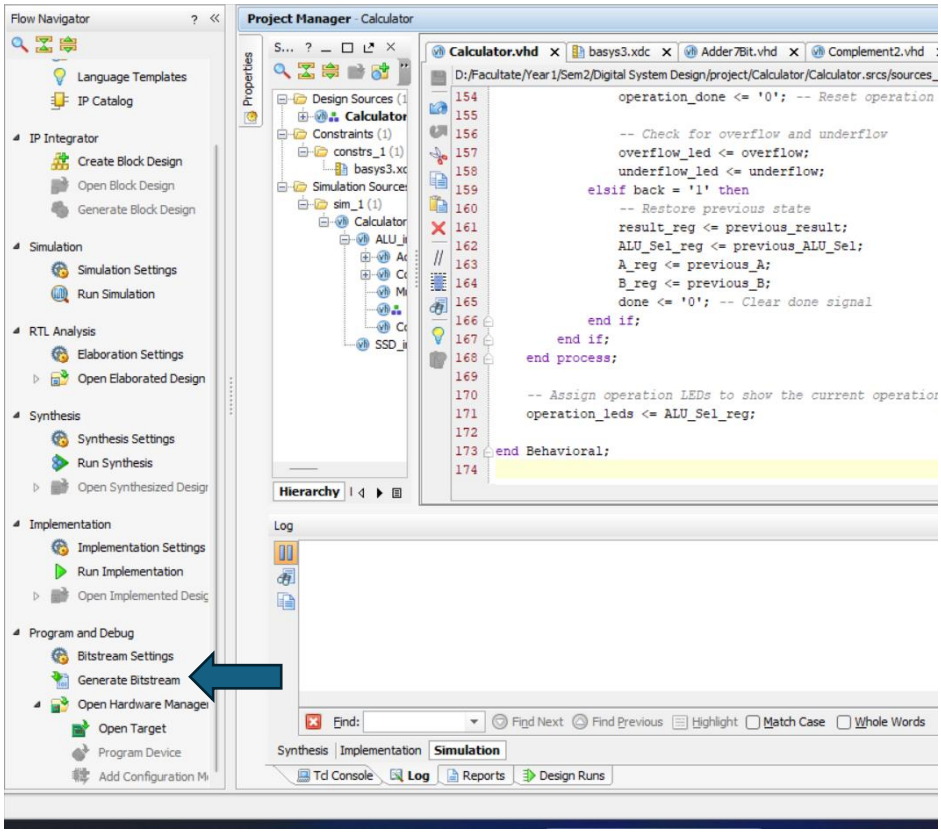
The calculator uses various features of the board to perform operations and display the result.

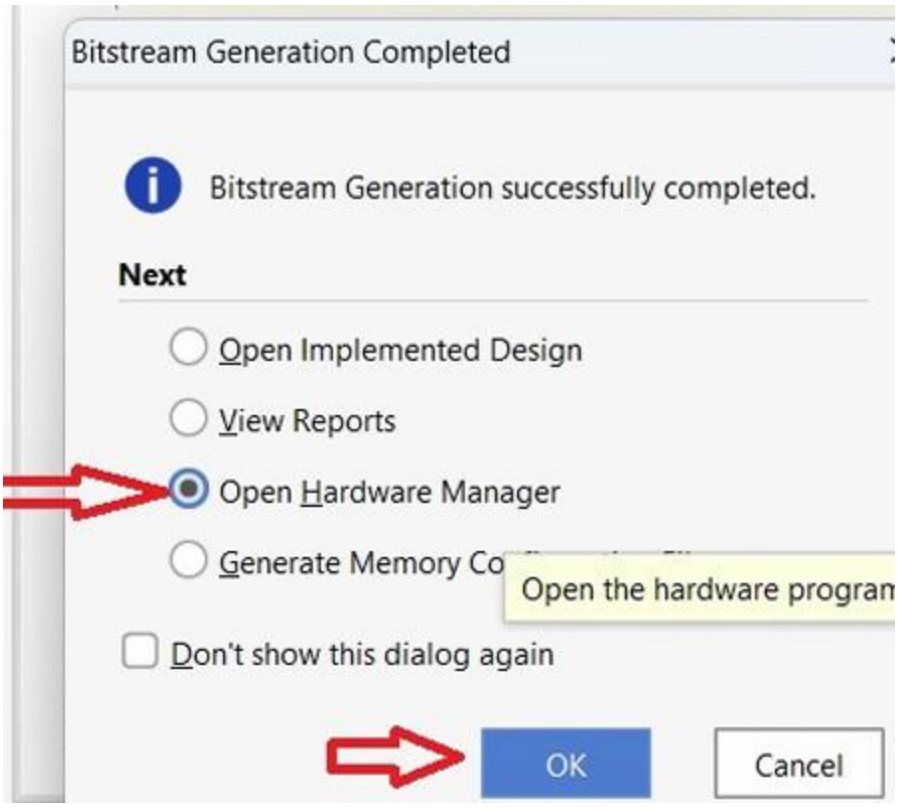
The steps of programming the board are the following: Synthesis, Implementation, Bitstream, Hardware Manager, Open Target and Program Device.

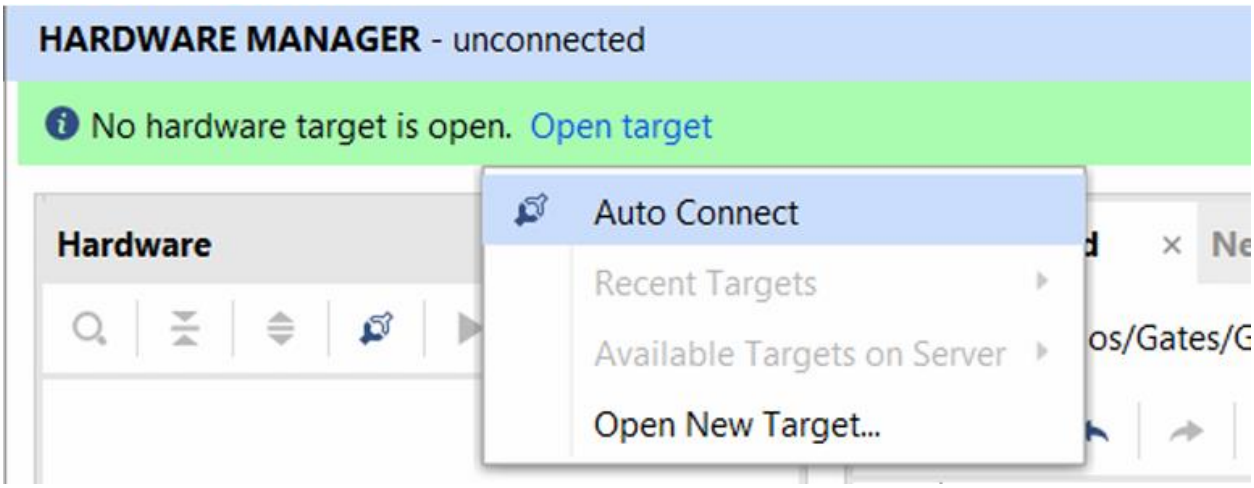

Flow Navigator
Project Manager - Calculator
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Implementation
Implementation Settings
Run Implementation
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Open Target
Program Device
Design Sources
Constraints
Simulation Sources
Hierarchy
Messages
Calculator.vhd
basys3.xdc
Adder7Bit.vhd
Complement2.vhd
divider8bit.vhd
D:/Facultate/Year1/Sem2/Digital System Design/project/Calculator/Calculator.srcs/sources_1/new/Calculator.vhd
result_leds : out STD_LOGIC_VECTOR (8 downto 0); -- LEDs to reflect
operation_leds : out STD_LOGIC_VECTOR (1 downto 0); -- LEDs to indi
anodes : out STD_LOGIC_VECTOR (3 downto 0); -- Anode signals for 7-
cathodes : out STD_LOGIC_VECTOR (6 downto 0); -- Cathode signals fo
done : out STD_LOGIC; -- Operation complete signal
overflow_led : out STD_LOGIC; -- Overflow indicator LED
underflow_led : out STD_LOGIC -- Underflow indicator LED
);
end Calculator;
architecture Behavioral of Calculator is
-- Internal signals
signal A, B : STD_LOGIC_VECTOR (8 downto 0); -- 8 bits for number + 1 b
signal ALU_Sel : STD_LOGIC_VECTOR (1 downto 0);
signal A_reg, B_reg : STD_LOGIC_VECTOR (8 downto 0);
signal ALU_Sel_reg : STD_LOGIC_VECTOR (1 downto 0);
signal result_reg : STD_LOGIC_VECTOR (15 downto 0);
signal ALU_done : STD_LOGIC;
signal previous_result : STD_LOGIC_VECTOR (15 downto 0);
signal previous_ALU_Sel : STD_LOGIC_VECTOR (1 downto 0);
signal previous_A, previous_B : STD_LOGIC_VECTOR (8 downto 0);
signal input_A, input_B : STD_LOGIC_VECTOR (8 downto 0);
signal current_op : STD_LOGIC_VECTOR (1 downto 0);
signal operation_done : STD_LOGIC;
signal display_value : STD_LOGIC_VECTOR (15 downto 0);
signal display_type : STD_LOGIC;
signal state : integer range 0 to 3 := 0; -- State for input sequence
signal overflow, underflow : STD_LOGIC; -- Overflow and underflow signa
-- Components declaration
component ALU is
Port (
A : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
B : in STD_LOGIC_VECTOR (8 downto 0); -- 9 bits including sign
ALU_Sel : in STD_LOGIC_VECTOR (1 downto 0); -- 2-bit control si


Launch Runs
Launch the selected synthesis or implementation runs.
Launch directory:
<Default Launch Directory>
Options
Launch runs on local host:
Number of jobs:
10
Generate scripts only
Don't show this dialog again
OK
Cancel


Flow Navigator
Project Manager - Calculator
Calculator.vhd
basys3.xdc
Adder7Bit.vhd
Complement2.vhd
D:/Facultate/Year 1/Sem2/Digital System Design/project/Calculator/Calculator.srcs/sources_
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Implementation
Implementation Settings
Run Implementation
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Open Target
Program Device
Design Sources
Constraints
Simulation Sources
Hierarchy
Log
Find:
Find Next
Find Previous
Highlight
Match Case
Whole Words
Synthesis
Implementation
Simulation
Tcl Console
Log
Reports
Design Runs


Flow Navigator
Project Manager - Calculator
Calculator.vhd
basys3.xdc
Adder7Bit.vhd
Complement2.vhd
D:/Facultate/Year 1/Sem2/Digital System Design/project/Calculator/Calculator.srcs/sources_
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Implementation
Implementation Settings
Run Implementation
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Open Target
Program Device
Design Sources
Constraints
Simulation Sources
Hierarchy
Log
Find:
Find Next
Find Previous
Highlight
Match Case
Whole Words
Synthesis
Implementation
Simulation
Tcl Console
Log
Reports
Design Runs


Bitstream Generation Completed
Bitstream Generation successfully completed.
Next
Open Implemented Design
View Reports
Open Hardware Manager
Generate Memory Co
Open the hardware progran
Don't show this dialog again
OK
Cancel


HARDWARE MANAGER - unconnected
No hardware target is open. Open target
Hardware
Auto Connect
Recent Targets
Available Targets on Server
Open New Target...
os/Gates/G

4. Simulation Captures

To better see how the ALU works, we take the numbers 2 and 6 and do all the possible combinations of operations and of signs

First, 6+2 = 8

Name	Value	Data Type
A[8:0]	006	Array
B[8:0]	002	Array
ALU_Sel[1:0]	0	Array
result[15:0]	0008	Array
done	1	Logic
overflow	0	Logic
underflow	0	Logic

Then, 6-2 = 4

Name	Value	Data Type
A[8:0]	006	Array
B[8:0]	002	Array
ALU_Sel[1:0]	1	Array
result[15:0]	0004	Array
done	1	Logic
overflow	0	Logic
underflow	0	Logic

Now, 6/2 = 3

Name	Value	Data Type
A[8:0]	006	Array
B[8:0]	002	Array
ALU_Sel[1:0]	3	Array
result[15:0]	0003	Array
done	1	Logic
overflow	0	Logic
underflow	0	Logic

Now, 6*2 = 12 = c

Name	Value	Data Type
A[8:0]	006	Array
B[8:0]	002	Array
ALU_Sel[1:0]	2	Array
result[15:0]	000c	Array
done	1	Logic
overflow	0	Logic
underflow	0	Logic

Now, if we add the sign – to 6, we redo the operations and see that the addition turns into a subtraction and so on, so the sign is carefully handled.

Name	Value	Data Type
A[8:0]	106	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	1	Logic
[1]	1	Logic
[0]	0	Logic
B[8:0]	002	Array
ALU_Sel[1:0]	0	Array
result[15:0]	8004	Array
[15]	1	Logic
[14]	0	Logic
[13]	0	Logic
[12]	0	Logic
[11]	0	Logic
[10]	0	Logic
[9]	0	Logic
[8]	0	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic

Name	Value	Data Type
A[8:0]	106	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	1	Logic
[1]	1	Logic
[0]	0	Logic
B[8:0]	002	Array
ALU_Sel[1:0]	1	Array
result[15:0]	8008	Array
[15]	1	Logic
[14]	0	Logic
[13]	0	Logic
[12]	0	Logic
[11]	0	Logic
[10]	0	Logic
[9]	0	Logic
[8]	0	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic

Name	Value	Data Type
A[8:0]	106	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	1	Logic
[1]	1	Logic
[0]	0	Logic
B[8:0]	002	Array
ALU_Sel[1:0]	2	Array
result[15:0]	800c	Array
[15]	1	Logic
[14]	0	Logic
[13]	0	Logic
[12]	0	Logic
[11]	0	Logic
[10]	0	Logic
[9]	0	Logic
[8]	0	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic

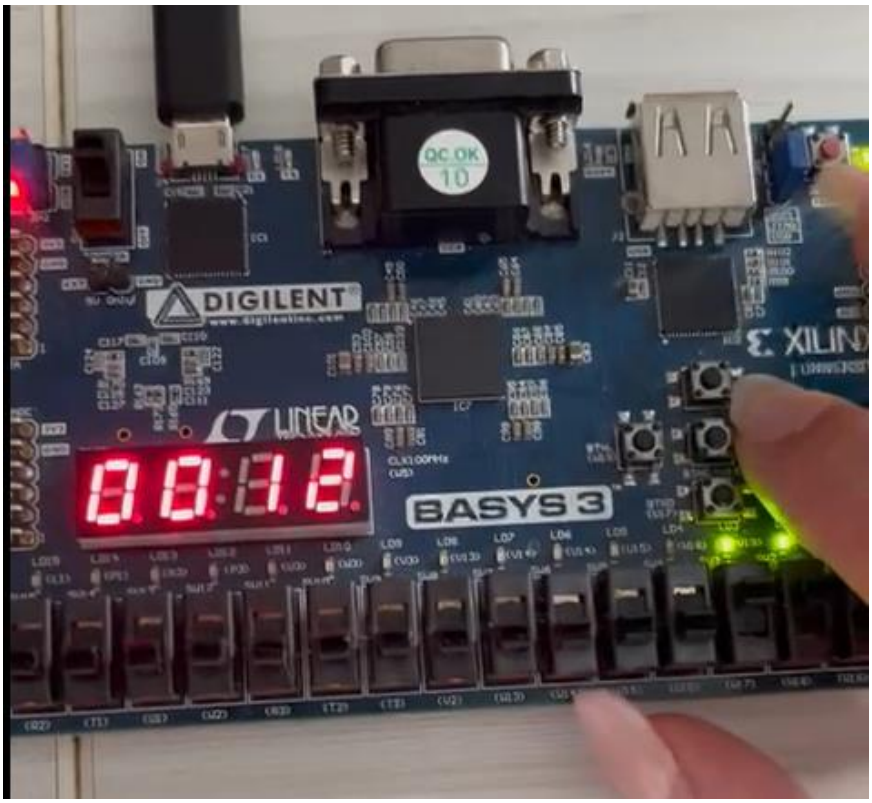
A[8:0]	106	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	1	Logic
[1]	1	Logic
[0]	0	Logic
B[8:0]	002	Array
ALU_Sel[1:0]	3	Array
result[15:0]	8003	Array
[15]	1	Logic
[14]	0	Logic
[13]	0	Logic
[12]	0	Logic
[11]	0	Logic
[10]	0	Logic
[9]	0	Logic
[8]	0	Logic
[7]	0	Logic

Also, if they both are negative, the sign is still handled correctly.

Name	Value	Data Type
A[8:0]	106	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	1	Logic
[1]	1	Logic
[0]	0	Logic
B[8:0]	102	Array
[8]	1	Logic
[7]	0	Logic
[6]	0	Logic
[5]	0	Logic
[4]	0	Logic
[3]	0	Logic
[2]	0	Logic
[1]	1	Logic
[0]	0	Logic
ALU_Sel[1:0]	3	Array
result[15:0]	0003	Array
[15]	0	Logic
[14]	0	Logic
[13]	0	Logic

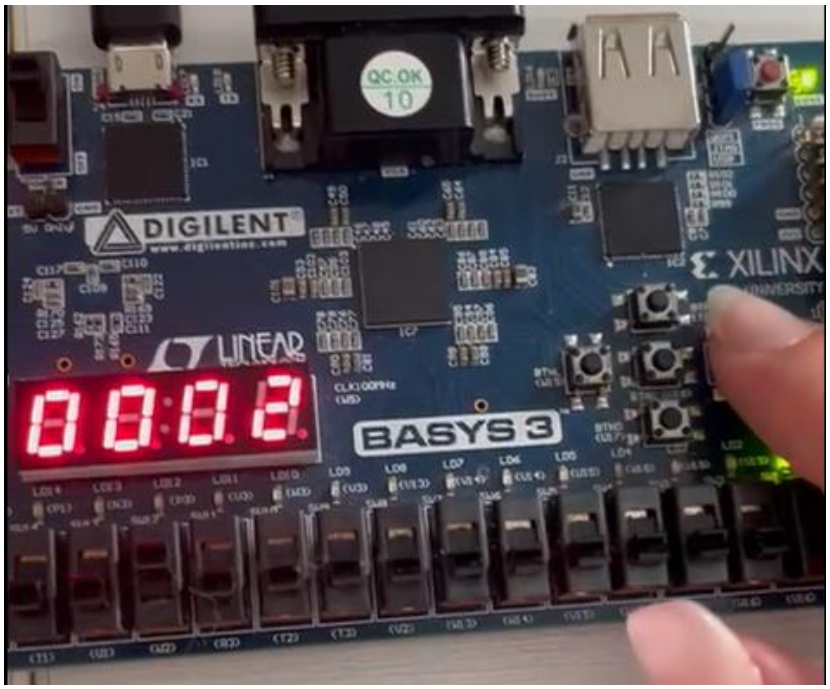
5. Board Pictures

So, first we input the number 12 from the switches and press the next button.

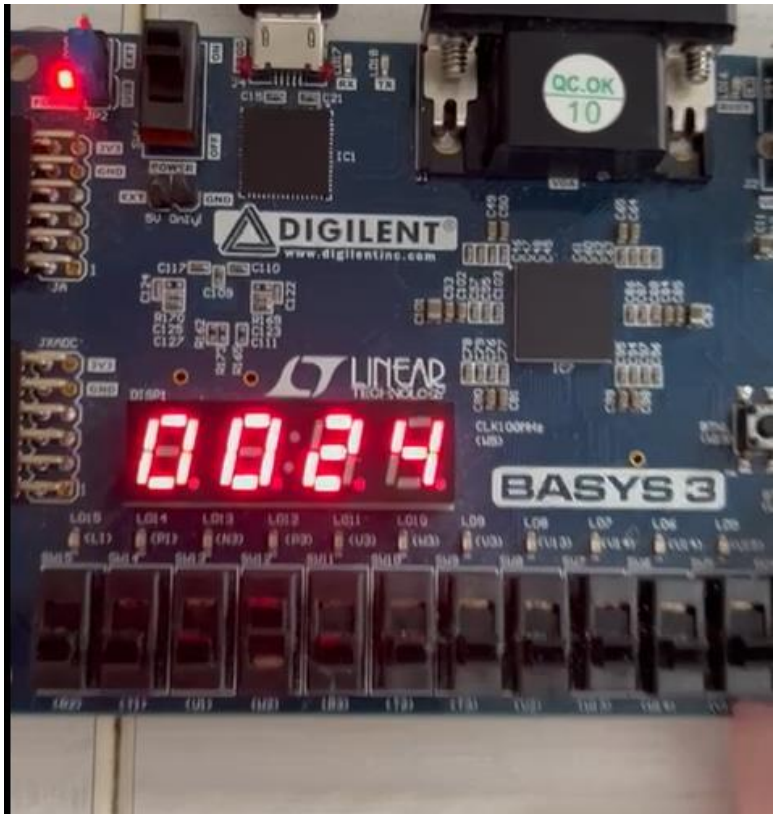


Then, we choose from the operation switches the multiplication (which is 10).

Then, we choose the second number which is 2.



When I press next, the result is displayed. So, for 12*2=24.



For more examples of how it works on the board, I have made a drive with more videos and instructions on how to handle the input and how to interact with the project. The link to the drive is the following: https://drive.google.com/drive/folders/1zth5YlL4Pb6luQxA79LSMZCuKK3O6Idp?usp=sharing .

6. Future Improvements

While the current implementation of the calculator on the Basys 3 FPGA board provides a functional system for performing basic arithmetic operations with integer numbers, as future improvements I want the project to be able to manipulate floating-point numbers.
The user interface consists of switches and LEDs, but I want to integrate a more intuitive user interface, such as keyboard input and LCD screen.
Also, I want to add advanced arithmetic functions such as trigonometric expressions, logarithmic and exponential functions.

7. Conclusion

The Pocket Calculator project on the Basys 3 FPGA development board represents a significant achievement in digital design and FPGA programming. This implementation utilizes the powerful features of the Basys 3 board, including switches, LEDs, buttons, and a seven-segment display, to provide an interactive and educational tool.
As I look to the future, the planned improvements will further enhance the calculator's functionality, ensuring it remains a relevant and valuable tool for a wide range of applications

8. References

- Course Logic Design Year 1, sem 1
- Course Digital System Design, Year 1, sem 2
- Laboratory Digital System Design
- Udemy
- IEEE Standard VHDL Language Reference Manual