Department of Electrical and Computer Engineering

University of Canterbury

ENEL 373: Digital Electronics and Devices

Formal Report ALU + FSM + REGS Project

by

Tim Hadler,

and Joseph Green

Report due on 26th May 2019

Table of Contents

1. Introduction 2

2. Block Diagram 3

3. Expanded Design Summary 4

4. Testing 7

5. Problems and Solutions 7

6. Learning, Outcomes and Improvements 8

7. Conclusion 8

8. Appendix and References 9

1. Introduction

This project involved designing an 8-bit Arithmetic Logic Unit (ALU), and implementing the design on an FPGA board. The final project must allow for the user to input two 8-bit operands and a 4-bit operation code to specify the operation to be carried out by the ALU. The ALU is required to be able to perform arithmetic add and subtract functions, and bitwise logical AND and OR functions. Other design requirements included having data registers for the inputs and the ALU output, and having user feedback to confirm the inputs have been successfully stored.

The ALU is of central importance to modern electronics. What follows is a description of a designed, programmed and tested ALU made with an FPGA in the VHDL programming language. A Digilent NEXYS 4 board was used with Vivado 2016.2 to generate an 8-bit ALU, that has 4 arithmetic/logic operators and displays on a Binary Coded Decimal (BCD) display. It uses a Finite State Machine (FSM) to take input from 8 input switches and store the values and operands into storage registers.

2. Top Level Block Diagram

Figure 1 shows a schematic of the project design. The design includes a data bus to carry data between components, and a control unit to control the data flow and state of the ALU. There are switch inputs, used by the user to input data, and a display output to show the ALU output. Registers are used to store data.

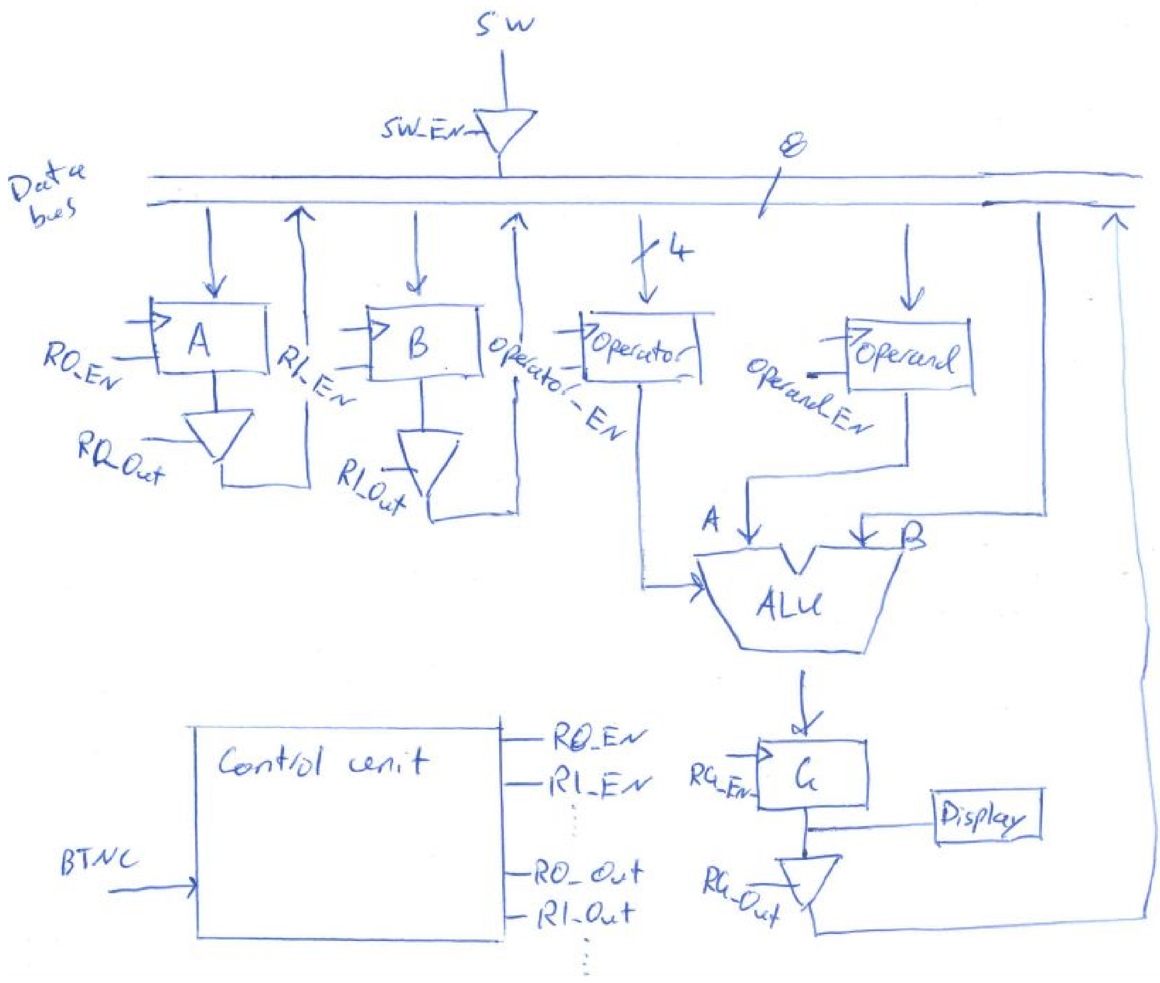


Figure 1: Schematic overview of the ALU design

3. Expanded Design Summary

This section provides a detailed explanation of how each of the components used in the project function.

3.1 Main project

The main project module is the structural component of the ALU, and maps all the signals that go in and out of each of the behavioural modules. The main project module is where the signals are passed to and from the physical peripherals, like the timers and LEDs. The inputs to main_project are the 100MHz clock, button BTNC, and the first 8 slider switches on the FPGA board. The module outputs a 7-bit signal to control the 7 segments on the display, an 8-bit signal to control the 8 digits of the display, and a 16-bit signal for the 16 LEDs. Lines 42 to 100 (Appendix A - main_project) declares the components and their inputs/outputs that make up the ALU. To create connections between components, there are signals created in the main module, and mapped to the inputs and outputs of each of the behavioural modules, this is done on lines 104 to 164. Mapping outputs from components to the external peripherals is also done this way. An important signal in main project is the data bus, which is passed to most of the other modules, and is how input data is passed around the ALU modules.

3.2 Control unit

The control unit module determines what state the ALU is in, and controls some of the other modules by setting tristate buffers, and enable signals for the data registers. The inputs are a 100 Hz clock signal and a debounced button signal (exe). The module outputs the current state, as a 3-bit data bus signal. The control unit outputs single bit signals, labelled R0_EN, R1_EN etc and R1_Out, R2_out etc. The enable signals control when data on the data bus is stored into each register. The Out signals control when the contents of a register is put onto the data bus, via a tristate buffer. The control unit operates as a state machine, using the exe signal to change states.

3.3 Finite State machine

To increase the user friendliness of the ALU, the current state is displayed on LED’s 13 to 15. The displayed state indicates, to the user, what data has been input. To encode the state, a one-hot method was used for the first three states, where only one of the three bits is set at a time. This allows users to know when data needed to be input. If the LSB is set, operand A needs to be input, if the second bit is set, operand B is needed, and the MSB indicates the operator code is needed. When more than one bit is set, no data will be read from the switches. Encoding the state this way, meant five states could be encoded using only three bits, but still takes advantage of the simplicity of the one-hot method.

The state machine is part of the control unit module. There are five states, which are iterated through by pressing BTNC. The first three states allow data to be passed from the data bus into register A, B and the operand register. In the control unit module, R0 and R1 correspond to registers A and B in the main module, respectively. The control unit loads data into the registers by setting SW_EN high, which allows input from the user to be put on the data bus via the 8 switches, and setting the appropriate register enable signal high to allow data to be stored in that register, from the data bus (see Figure 1). All other control signals are set low. The fourth state puts the contents of register A into the operand register in preparation for the ALU to do the user specified operation. The fifth state puts the second operand, the contents of register B, onto the data bus, and sets the register G (ALU output) enable high. In the fifth state the ALU has the two operands and the operator code as inputs (see Figure 1), and the result is stored in register G.

3.4 Button debounce

In order to operate the ALU reliably, button debouncing must be implemented. The BTNC button on the FPGA is used to input data to the ALU and to change states. Button debouncing is important to make sure the state changes only once when BTNC is pressed for a sufficiently long time. Debouncing is implemented in software, using the rising edge of a 5 Hz clock signal to poll BTNC. The debouncing component outputs a std logic signal that is set when BTNC is polled as pressed.

3.5 Display

The FPGA board has an eight digit 7-segment display, only the first two digits are used in this project. The ALU output which is stored in register G, is constantly displayed in hexadecimal. The display component uses a seven segment decoder to convert the 8-bit input value into two hexadecimal digits. To display the two digit number on the BCD display, time division multiplexing is used (Appendix A - hex_display, lines 51 to 60).

3.6 Clock Divider

The my_divider module was taken from example code given by Dr Steve Weddell. It was modified to output three clock signals at different frequencies. The module takes a 100 MHz clock and outputs a 100 Hz clock signal for the ALU to run on, a 50Hz clock for multiplexing the display, and a 5 Hz signal for the button debouncing.

3.7 Registers

This project uses 8-bit and 4-bit registers, in the modules reg_8_en and reg_4_en. The operand inputs and the ALU output (G) is stored in 8-bit registers, the operator code input is stored in a 4-bit register. The registers have an enable input to control when the input data, D, is stored. The registers also have a clear input to clear the stored values asynchronously. In this project the clear input was not implemented and was always set to low by the control unit. A second button would need to be set up to use the clear functionality.

3.8 Tri-state Buffers

The tristate_8_buf module implements an 8-bit tristate buffer used to control when the output of each register is passed onto the data bus. Tri-state buffers are important to make sure only one signal is being put onto the bus at any one time. The buffers have an enable input, which when set high by the controller, will allow the input to go through to the output.

3.9 Arithmetic Logic Unit

The ALU module executes an operation specified by the user. The operations supported are signed addition and subtraction, and logical AND and OR operations. The input operator code is a 4-bit signal that uses a one-hot method of encoding. This method allows a simple way of specifying what operation the user wants to be executed. The operations are assigned to the op_code as in lines 32 to 35 in Appendix A - ALU. The ALU outputs the result of the specified operation executed on the two inputs A and B, if no operation is specified, the output is zero.

3.10 LEDs

In order to provide user feedback when entering the inputs, the FPGA's LEDs were used. Once an input has been entered and BTNC pressed, the data stored in the appropriate register is shown in binary on the LEDs. The user feedback means the user can check and verify that the input is correctly stored. The led_mux module takes the register contents and the current state as inputs, and outputs the binary value to be displayed. The module determines what value should be displayed based on the current state (lines 37 to 54, Appendix A - Led Mux).

4. Testing

Testing electronic circuits can be troublesome. Because of the complexities, circuit simulation and programm debugging are formidable opponents to the electronic engineer. Fortunately Vivado offers a solution, the testbench allows significantly reduced time in software/hardware development, by simulating code through software without the need to reprogram the FPGA again and again. A module forms what is known as a Unit Under Test (UUT). The port mappings of the UUT are taken directly from the module itself, the input can be assigned and then all values in the UUT can be directed, via a signal, to a versatile waveform generator. The generator may display different data types including binary, hex or digital signals. Two such testbenches were implemented in this project. Portions of the code and their output are given in appendix B, along with their output waveforms.

The reader can verify that the Buttons Debounce testbench implements circuitry which “debounces”, meaning that sudden changes in the button input value do not affect the output, but significantly long changes do effect it. When the clock (clk) is high and the output is asserted for a sufficient amount of time, the output butt_signal is set high.

Furthermore, the ALU Testbench provides confirmation that all ALU functions are realised. Signed addition, subtraction, bitwise AND and OR can be verified by analysing the waveform output. Operands A,B along with the operator code (op_code) are used to compute the output G. These datum were analysed and confirmed correct; the first operation performed in the first 100 nanoseconds, as an example, corresponds to ‘01000111’ + ‘00010001’ = ‘01011000’ or 0x47 + 0x11 = 0x58.

5. Problems and Solutions

A major design problem was how to get data into and out of the FPGA, in a temporally sequential way. Three user inputs had to be stored in memory. It was decided that a push button (BTNC) would be used to cycle through a FSM to sequentially store the inputs. In the design a control unit was used to allow data to flow in, through and out of the FPGA.

An important problem to solve in almost every electronic application that involves buttons was button debouncing. Our design involved making something which behaves “like” a CPU, but not a CPU which has to deal with peripheral devices. Because of this, interrupt driven peripheral communications was not important, thus polling the buttons was the decided solution. Through testing, it was decided that a 5 Hz clock signal would be used for the polling. See Button_Debounce.vhd in Appendix A for this implementation.

6. Learning, Outcomes & Improvements

This project has been an exciting and rewarding educational experience. Both electronics and computer programming skills have been applied to a real life problem with the desired outcome. As an improvement, the code could be easily modified to contain a register with status flags such as carry in/out, negative, zero result and overflow. Eventually opcode style input could be implemented to mimic a modern CPU.

7. Conclusion

To conclude, the requirements outlined in the introduction have been meet by the ALU design. The design uses a FSM to sequence user inputs, and a data bus to carry data between modules and peripherals. Registers are used to store data, and the ALU performs arithmetic and logic functions specified by the user. The FPGA display and LEDs are used to give user feedback and display the result of the ALU. Modules provide structure, abstraction and data exchange between components. Testing has proven to be a valuable part designing this project to ensure reliable and correct function of the ALU. This project was a great learning experience that gave in depth knowledge of how most electronic devices require a robust ALU, and how ALUs work.

8. References and Appendices

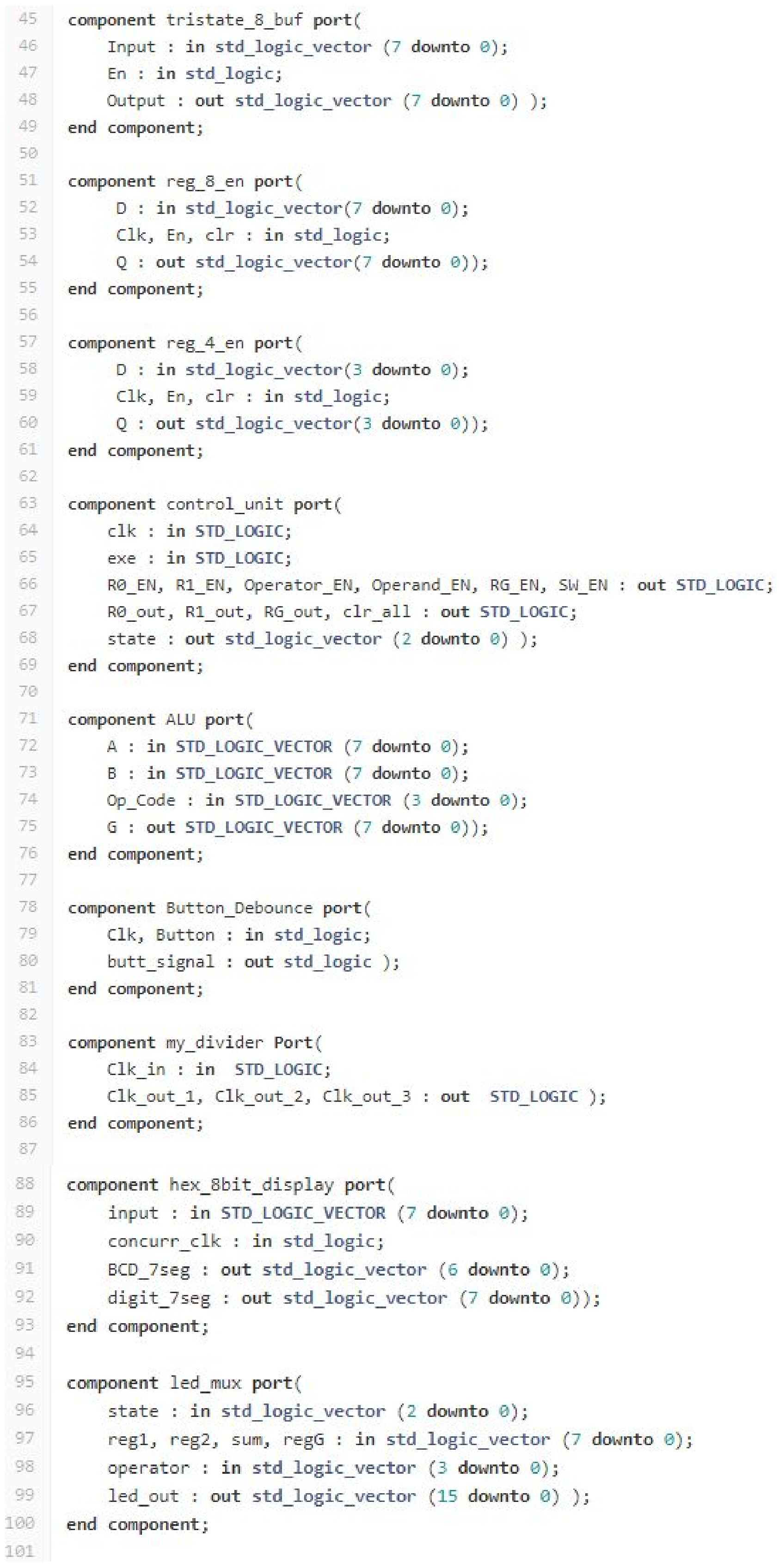
Thanks to Dr Steve Weddell, University of Canterbury ECE Department, who provided countless resources for this project and dedication to teaching in the labs.

https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/

https://www.fpga4student.com/2017/06/vhdl-code-for-arithmetic-logic-unit-alu.html

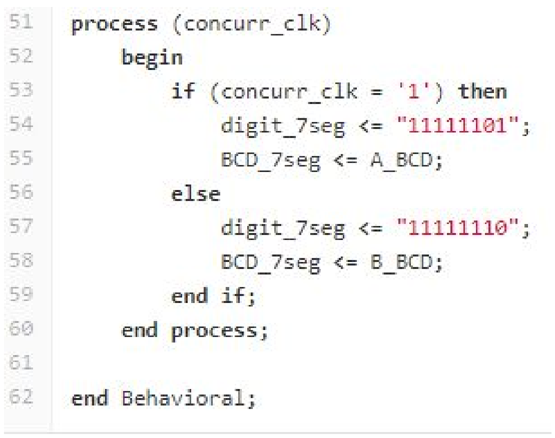
Appendix A

Main Project



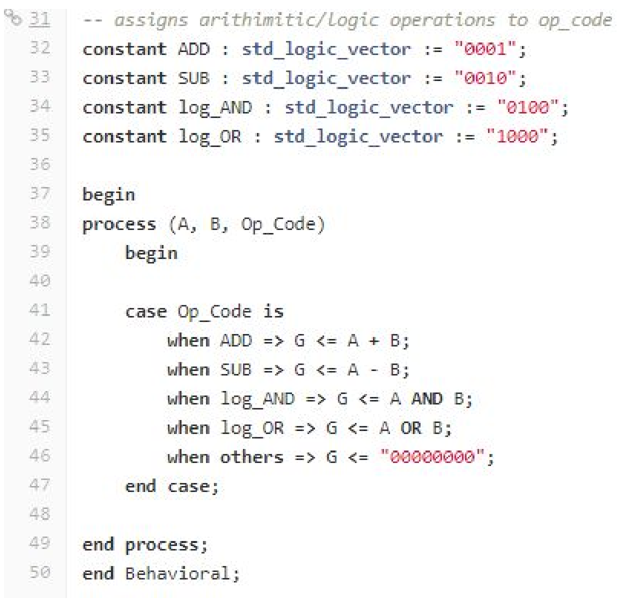
```vhdl
component tristate_8_buf port(
    Input : in std_logic_vector (7 downto 0);
    En : in std_logic;
    Output : out std_logic_vector (7 downto 0) );
end component;

component reg_8_en port(
     D : in std_logic_vector(7 downto 0);
     Clk, En, clr : in std_logic;
     Q : out std_logic_vector(7 downto 0));
end component;

component reg_4_en port(
     D : in std_logic_vector(3 downto 0);
     Clk, En, clr : in std_logic;
     Q : out std_logic_vector(3 downto 0));
end component;

component control_unit port(
    clk : in STD_LOGIC;
    exe : in STD_LOGIC;
    R0_EN, R1_EN, Operator_EN, Operand_EN, RG_EN, SW_EN : out STD_LOGIC;
    R0_out, R1_out, RG_out, clr_all : out STD_LOGIC;
    state : out std_logic_vector (2 downto 0) );
end component;

component ALU port(
    A : in STD_LOGIC_VECTOR (7 downto 0);
    B : in STD_LOGIC_VECTOR (7 downto 0);
    Op_Code : in STD_LOGIC_VECTOR (3 downto 0);
    G : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component Button_Debounce port(
    Clk, Button : in std_logic;
    butt_signal : out std_logic );
end component;

component my_divider Port(
    Clk_in : in  STD_LOGIC;
    Clk_out_1, Clk_out_2, Clk_out_3 : out  STD_LOGIC );
end component;

component hex_8bit_display port(
    input : in STD_LOGIC_VECTOR (7 downto 0);
    concurr_clk : in std_logic;
    BCD_7seg : out std_logic_vector (6 downto 0);
    digit_7seg : out std_logic_vector (7 downto 0));
end component;

component led_mux port(
    state : in std_logic_vector (2 downto 0);
    reg1, reg2, sum, regG : in std_logic_vector (7 downto 0);
    operator : in std_logic_vector (3 downto 0);
    led_out : out std_logic_vector (15 downto 0) );
end component;
```

Hex Display



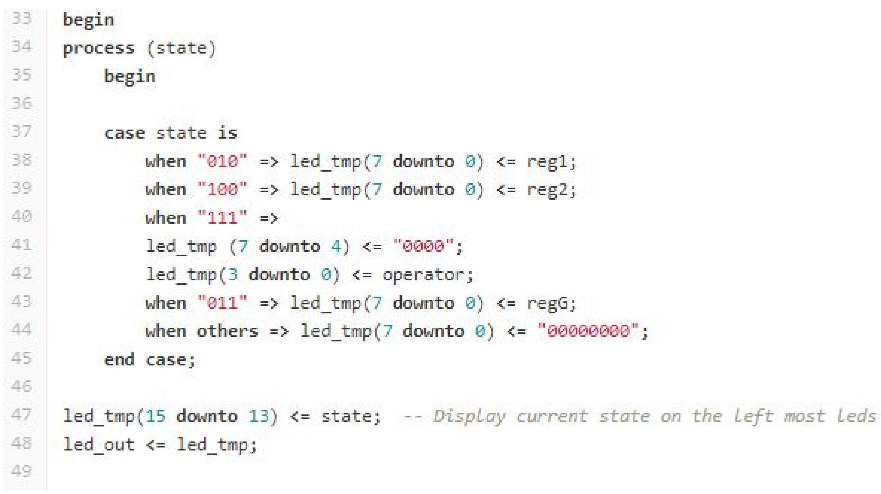
```vhdl
process (concurr_clk)
    begin
        if (concurr_clk = '1') then
            digit_7seg <= "11111101";
            BCD_7seg <= A_BCD;
        else
            digit_7seg <= "11111110";
            BCD_7seg <= B_BCD;
        end if;
    end process;

end Behavioral;
```

ALU



```vhdl
-- assigns arithimitic/logic operations to op_code
constant ADD : std_logic_vector := "0001";
constant SUB : std_logic_vector := "0010";
constant log_AND : std_logic_vector := "0100";
constant log_OR : std_logic_vector := "1000";

begin
process (A, B, Op_Code)
    begin

    case Op_Code is
        when ADD => G <= A + B;
        when SUB => G <= A - B;
        when log_AND => G <= A AND B;
        when log_OR => G <= A OR B;
        when others => G <= "00000000";
    end case;

end process;
end Behavioral;
```

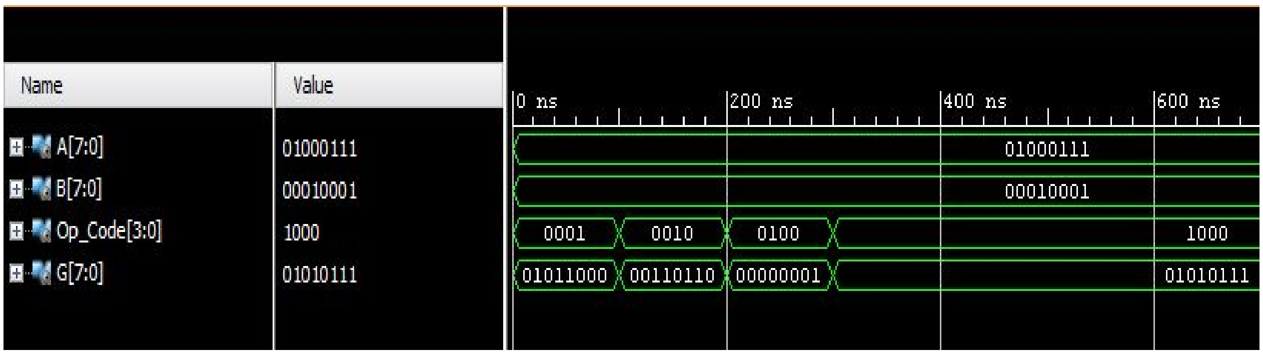
Led Mux

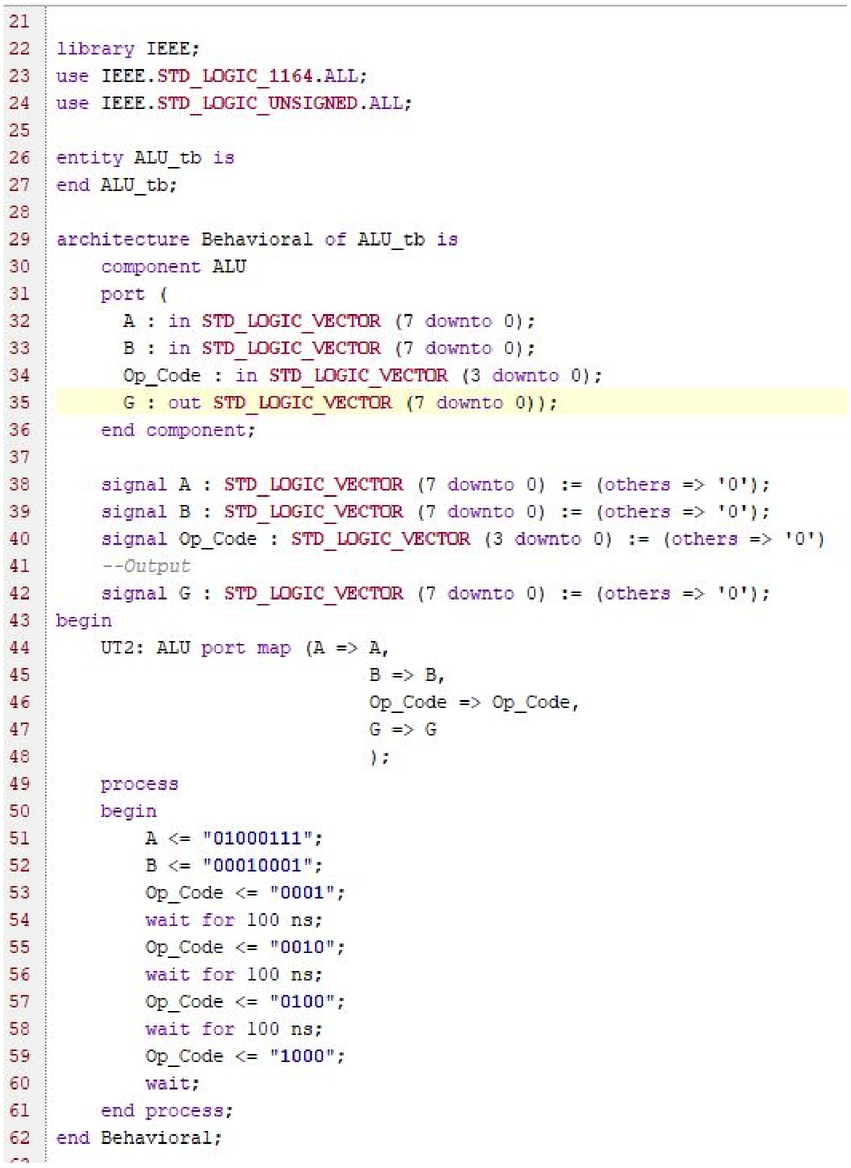


```vhdl
begin
process (state)
    begin

    case state is
        when "010" => led_tmp(7 downto 0) <= reg1;
        when "100" => led_tmp(7 downto 0) <= reg2;
        when "111" =>
        led_tmp (7 downto 4) <= "0000";
        led_tmp(3 downto 0) <= operator;
        when "011" => led_tmp(7 downto 0) <= regG;
        when others => led_tmp(7 downto 0) <= "00000000";
    end case;

led_tmp(15 downto 13) <= state;  -- Display current state on the left most leds
led_out <= led_tmp;

```

Appendix B

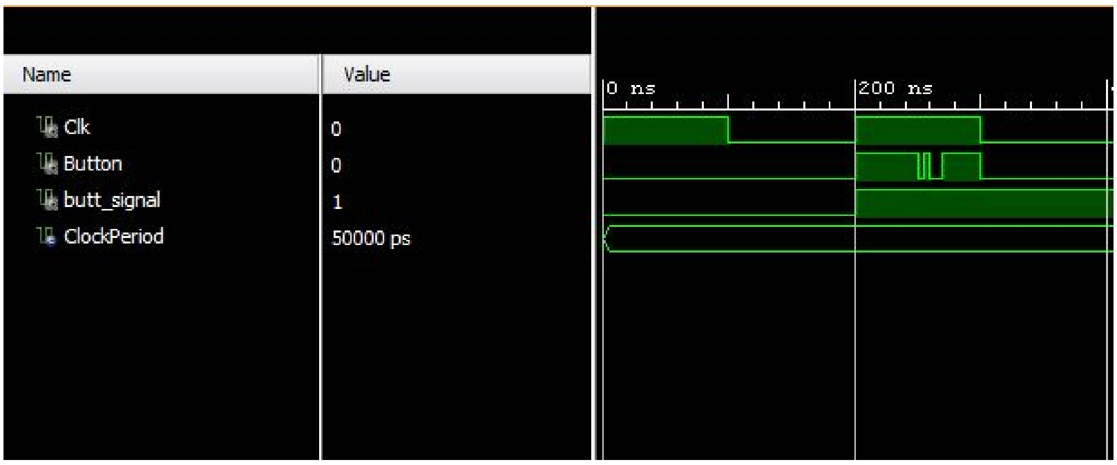
ALU Testbench Waveform and code



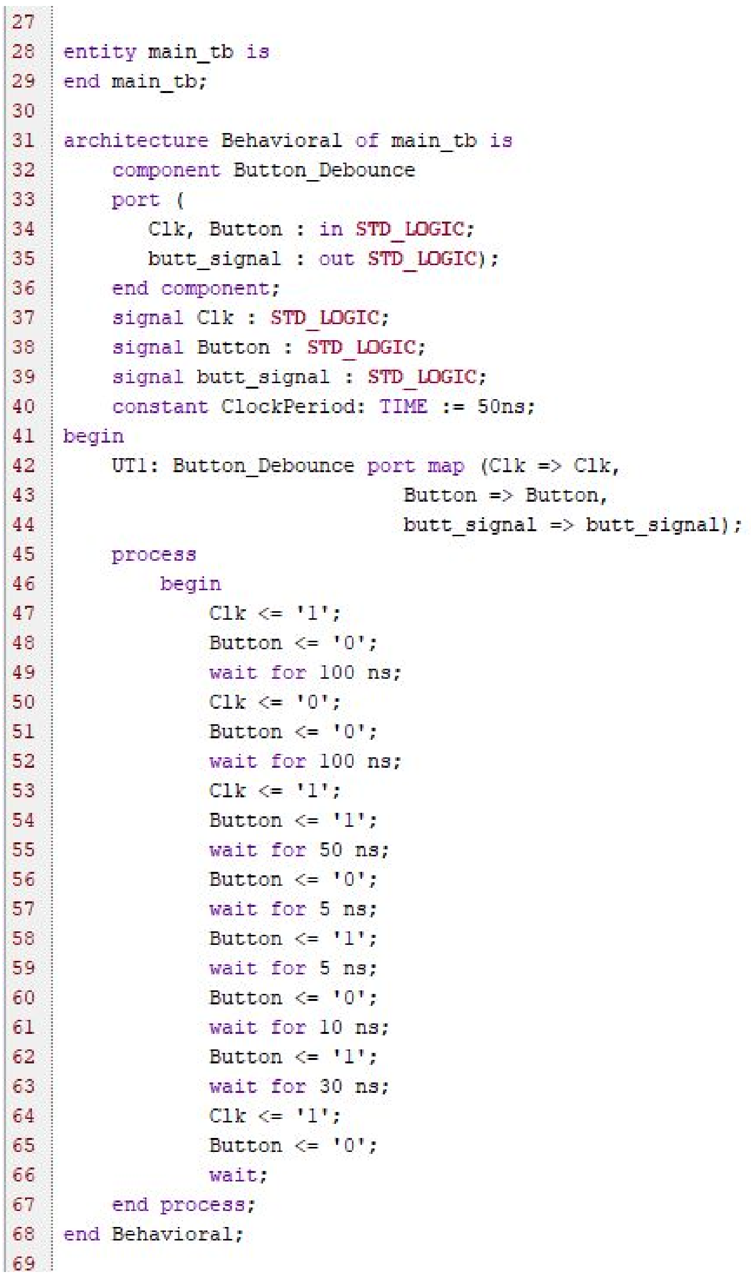


```vhdl

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU_tb is
end ALU_tb;

architecture Behavioral of ALU_tb is
    component ALU
    port (
      A : in STD_LOGIC_VECTOR (7 downto 0);
      B : in STD_LOGIC_VECTOR (7 downto 0);
      Op_Code : in STD_LOGIC_VECTOR (3 downto 0);
      G : out STD_LOGIC_VECTOR (7 downto 0));
    end component;

    signal A : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
    signal B : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
    signal Op_Code : STD_LOGIC_VECTOR (3 downto 0) := (others => '0')
    --Output
    signal G : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
begin
    UT2: ALU port map (A => A,
                        B => B,
                        Op_Code => Op_Code,
                        G => G
                        );

    process
    begin
        A <= "01000111";
        B <= "00010001";
        Op_Code <= "0001";
        wait for 100 ns;
        Op_Code <= "0010";
        wait for 100 ns;
        Op_Code <= "0100";
        wait for 100 ns;
        Op_Code <= "1000";
        wait;
    end process;
end Behavioral;
```

Button Debounce Testbench Waveform and Code



Name	Value
Clk	0
Button	0
butt_signal	1
ClockPeriod	50000 ps



```vhdl

entity main_tb is
end main_tb;

architecture Behavioral of main_tb is
    component Button_Debounce
    port (
       Clk, Button : in STD_LOGIC;
       butt_signal : out STD_LOGIC);
    end component;
    signal Clk : STD_LOGIC;
    signal Button : STD_LOGIC;
    signal butt_signal : STD_LOGIC;
    constant ClockPeriod: TIME := 50ns;
begin
    UT1: Button_Debounce port map (Clk => Clk,
                            Button => Button,
                            butt_signal => butt_signal);
    process
        begin
            Clk <= '1';
            Button <= '0';
            wait for 100 ns;
            Clk <= '0';
            Button <= '0';
            wait for 100 ns;
            Clk <= '1';
            Button <= '1';
            wait for 50 ns;
            Button <= '0';
            wait for 5 ns;
            Button <= '1';
            wait for 5 ns;
            Button <= '0';
            wait for 10 ns;
            Button <= '1';
            wait for 30 ns;
            Clk <= '1';
            Button <= '0';
            wait;
    end process;
end Behavioral;

```