Report on LAB 2

(Topics in Adv Comp Eng: Embedded Systems Hardware EE 493)

NAME: Prince K. Bose
NET ID: pkb44
RUID: 186008149

DATE: 3/7/19

Purpose

The main purpose of this lab was to build a 4 bit, 16 function ALU from ground up. In the pre-lab we were introduced to a concept similar to abstraction/ classes in programming. Here, we created a 1 bit adder once, then used 4 different instances of the same adder to create a 4 bit ripple carry adder.

There are 4 main components to the circuit of Lab 2:
1. Debounce Switch
2. MY_ALU
3. ALU_Tester

PRE LAB

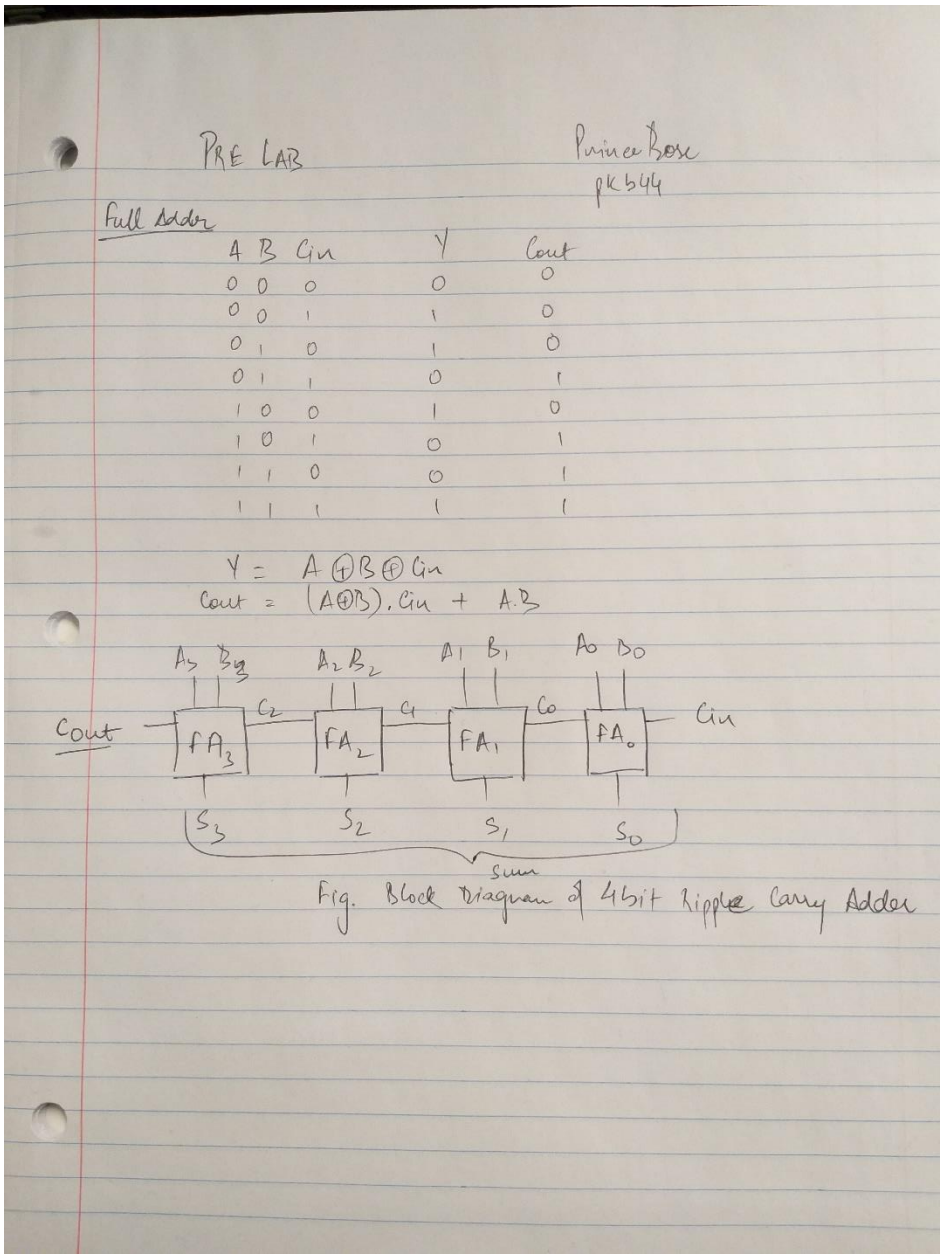

Fig. 1 : Pre Lab Working

1 Bit Adder

The 1 bit adder has 3 inputs A, B and Cin and produces 2 outputs S and Cout.
The Truth Table is as shown in Figure 1.

VHDL CODE

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


entity Adder_1bit is
Port (  Cin : in std_logic;
        A : in std_logic;
        B : in std_logic;
        Sumout : out std_logic;
        Cout: out std_logic);
end Adder_1bit;

architecture Behavioral of Adder_1bit is
begin

Sumout <= A xor B xor Cin;
Cout <= (((A xor B) and Cin) or (A and B));

end Behavioral;
```

SCHEMATIC

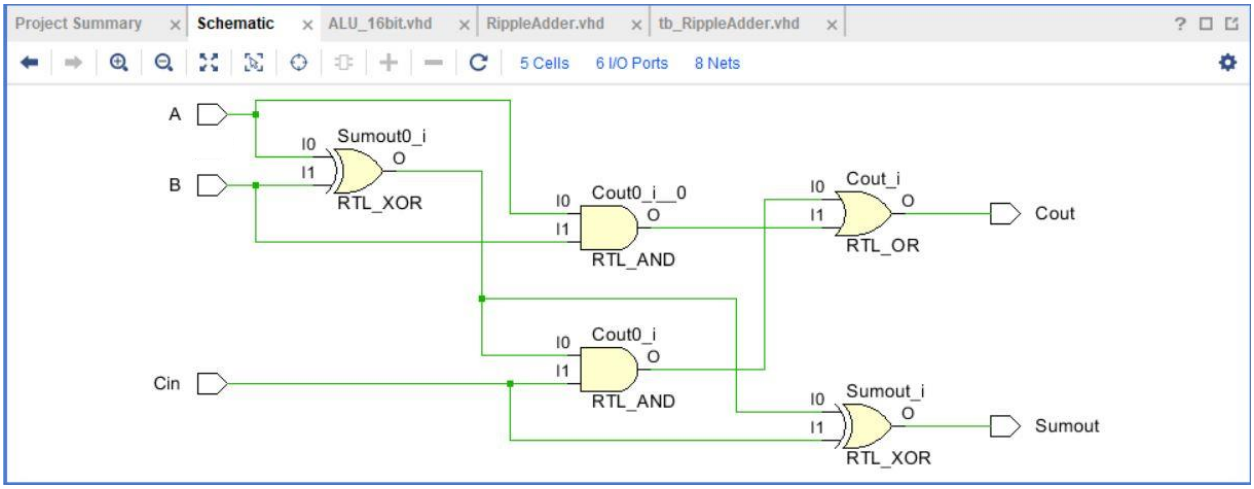

Fig. 2: 1 Bit Adder Schematic

Ripple Carry Adder

The ripple carry adder has a functional block diagram as shown in Figure 1.
It basically cascades four 1 bit adders, Cout of each stage is passed on to the input of the next stage.

VHDL CODE

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

entity RippleAdder is
    Port ( A : in std_logic_vector (3 downto 0);
           B : in std_logic_vector (3 downto 0);
           Cin: in std_logic;
           Cout : out std_logic;
           S : out std_logic_vector (3 downto 0));
end RippleAdder;

architecture Behavioral of RippleAdder is
component Adder_1bit
Port (  Cin : in std_logic;
        A : in std_logic;
        B : in std_logic;
        Sumout : out std_logic;
        Cout: out std_logic);
end component;

signal c1,c2,c3 : std_logic;

begin

FA0: Adder_1bit
Port map(   Cin => Cin,
            A => A(0),
            B => B(0),
            Sumout => S(0),
            Cout => c1);

FA1: Adder_1bit
Port map(   Cin => c1,
            A => A(1),
            B => B(1),
            Sumout => S(1),
```

```
        Cout => c2);

FA2: Adder_1bit
Port map(   Cin => c2,
        A => A(2),
        B => B(2),
        Sumout => S(2),
        Cout => c3);

FA3: Adder_1bit
Port map(   Cin => c3,
        A => A(3),
        B => B(3),
        Sumout => S(3),
        Cout => Cout);

end Behavioral;
```

VHDL TESTBENCH

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


entity tb_RippleAdder is
end tb_RippleAdder;

architecture Behavioral of tb_RippleAdder is
component RippleAdder
Port ( A : in std_logic_vector (3 downto 0);
        B : in std_logic_vector (3 downto 0);
        Cin: in std_logic;
        Cout : out std_logic;
        S : out std_logic_vector (3 downto 0));
end component;

signal tb_A : std_logic_vector(3 downto 0) := "0101";
signal tb_B : std_logic_vector(3 downto 0) := "1010";
signal tb_C : std_logic := '0';
signal S : std_logic_vector(3 downto 0) := "0000";
signal Cout : std_logic;

begin

inputgen: process
```

```
begin

wait for 100 ns;
tb_A <= "0000";
tb_B <= "0001";
tb_C <= '0';

wait for 100 ns;
tb_A <= "1000";
tb_B <= "0101";
tb_C <= '1';

wait for 100 ns;
tb_A <= "1111";
tb_B <= "0100";
tb_C <= '1';

wait for 100 ns;
tb_A <= "0000";
tb_B <= "0000";
tb_C <= '1';

end process;

testDev: RippleAdder
Port map(   A => tb_A,
            B => tb_B,
            Cin => tb_C,
            Cout => Cout,
            S => S);

end Behavioral;
```

SCHEMATIC

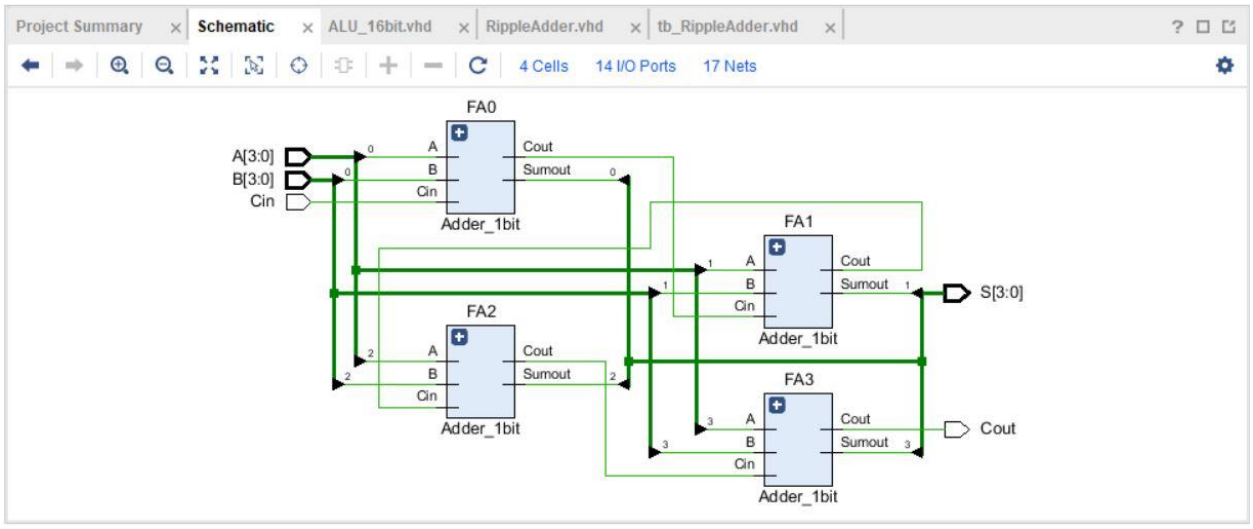

Fig. 3: Ripple Carry Adder showing four 1 bit Adders

SIMULATION

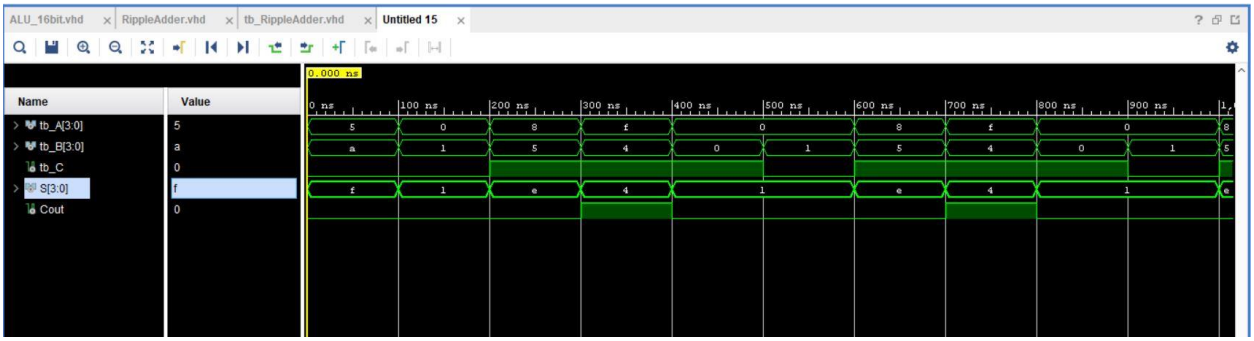

Fig. 4: Simulation showing waveforms for A,B,Cin, Cout and S.

As seen in the above simulation wave, S produces the desired Sum and Cout, the desired Carry output.

LAB 2

Each of these components are discussed in the following sections.

1. DEBOUNCE

Due to being mechanical in nature, they are often the bane of existence for digital designers when not dealt with correctly. The problem lies in that mechanical nature. When the button is pushed or released the spring inside causes the contacts to behave like a damped oscillator, which creates spikes in the signal.

This means, in between a swap from ‘0’ to ‘1’ or LOW to HIGH or ‘1’ to ‘0’ or HIGH to LOW, the switch goes through a few meta stable stages in between. During these meta stable stages, the output value may fluctuate.

It looks something like this :

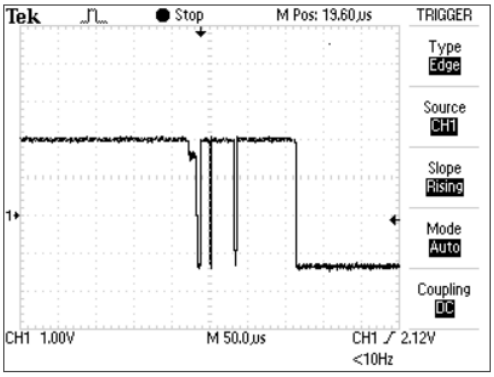

Fig 5. Switch Debouncing

DESIGN

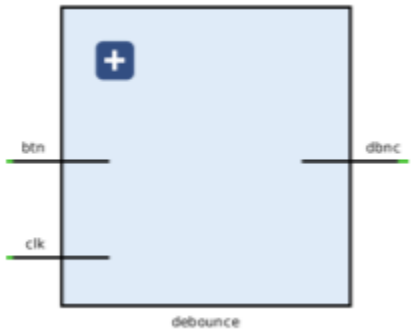

Fig. 6 Block Diagram of a Debounce Circuit

When a state changes from ‘0’ to ‘1’, the circuit waits for 20 ms for the switch to come to a stable state, before finally changing the output of the debounce at ‘1’.

In order to do this, we set the count value to 2499999.

VHDL Code

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


entity debounce is
    Port( clk : in STD_LOGIC;
          btn : in STD_LOGIC;
          dbnc : out STD_LOGIC);
end debounce;

architecture Behavioral of debounce is
signal sh : std_logic_vector(1 downto 0) := "00";
signal cnt_value : std_logic_vector(22 downto 0):=(others => '0');
begin
process(clk,btn)
begin

if (rising_edge(clk)) then
    sh(1) <= sh(0);
    sh(0) <= btn;
    if (unsigned(cnt_value) < 2499999) then
        dbnc <= '0';
```

```
        if (sh(1) = '1') then
            cnt_value <= std_logic_vector(unsigned(cnt_value)+1);
        else
            cnt_value <= (others => '0');
        end if;
    else
        dbnc <= '1';
        if(btn = '0') then
            dbnc <= '0';
            cnt_value <= (others => '0');
        end if;
    end if;
end if;
end process;
end Behavioral;
```

VHDL Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tb_debounce is
end tb_debounce;

architecture Behavioral of tb_debounce is
component debounce
    Port (clk: in std_logic;
        btn: in std_logic;
        dbnc: out std_logic );
end component;
signal tb_clk: std_logic := '0';
signal tb_btn: std_logic := '0';
signal tb_dbnc: std_logic := '0';
begin
devut: debounce port map(   clk => tb_clk,
btn => tb_btn,
dbnc => tb_dbnc);
clock:  process
begin
    tb_clk <= '1';
    wait for 4 ns;
    tb_clk <= '0';
    wait for 4 ns;
end process;
button: process
begin
tb_btn <= '0';
```

```
wait for 200us;
tb_btn <= '1';
wait for 60ms;
tb_btn <= '1';
wait for 2ms;
tb_btn <= '1';
wait for 1ms;
tb_btn <= '0';
tb_btn <= '1';
wait for 22ms;
tb_btn <= '0';
wait for 1ms;
tb_btn <= '1';
wait for 50 ms;
end process;
end Behavioral;
```

Simulation Results

Fig 7. Simulation Waveform for Debounce

As seen in the simulation waveform above, the circuit waits for 20ms till the output is stable at '1'. After 20s, the output is pulled up to '1'. As soon a '0' is received, the output is pulled down to '0'.

2. MY_ALU

This component takes 3 inputs, A, B and Opcode and generates a 4 bit output ALUOut.
The functionalities are based on the following table:

opcode	function	opcode	function
x"0"	$A + B$	x"8"	$A >>> 1$ (right shift arithmetic)
x"1"	$A - B$	x"9"	not A
x"2"	$A + 1$	x"A"	A and B
x"3"	$A - 1$	x"B"	A or B
x"4"	$0 - A$	x"C"	A xor B
x"5"	$A > B$ (greater than - see note)	x"D"	A xnor B
x"6"	$A << 1$ (left shift logical)	x"E"	A nand B
x"7"	$A >> 1$ (right shift logical)	x"F"	A nor B

Fig. 8: Functionality Table

VHDL Code

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


entity MY_ALU is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Opcode : in STD_LOGIC_VECTOR (3 downto 0);
           ALUOut : out STD_LOGIC_VECTOR (3 downto 0));
end MY_ALU;

architecture Behavioral of MY_ALU is

begin
opcodeselect:process(A,B,opcode)
begin
case opcode is
    when "0000" => ALUOut <= std_logic_vector(unsigned(A) + unsigned(B));
    when "0001" => ALUOut <= std_logic_vector(unsigned(A) - unsigned(B));
    when "0010" => ALUOut <= std_logic_vector(unsigned(A) + 1);
    when "0011" => ALUOut <= std_logic_vector(unsigned(A) - 1);
    when "0100" => ALUOut <= std_logic_vector(0 - unsigned(A));
```

```
    when "0101" =>
        if A > B then
            ALUOut <= "0001";
        else
            ALUOut <= "0000";
        end if;
    when "0110" => ALUOut <= std_logic_vector(unsigned(A(2 downto 0)) & '0');
    when "0111" => ALUOut <= std_logic_vector('0' & unsigned(A(3 downto 1)));
    when "1000" => ALUOut <= std_logic_vector(A(3) & unsigned(A(3 downto 1)));
    when "1001" => ALUOut <= std_logic_vector(not (unsigned(A)));
    when "1010" => ALUOut <= std_logic_vector(unsigned(A) and unsigned(B));
    when "1011" => ALUOut <= std_logic_vector(unsigned(A) or unsigned(B));
    when "1100" => ALUOut <= std_logic_vector(unsigned(A) xor unsigned(B));
    when "1101" => ALUOut <= std_logic_vector(unsigned(A) xnor unsigned(B));
    when "1110" => ALUOut <= std_logic_vector(unsigned(A) nand unsigned(B));
    when "1111" => ALUOut <= std_logic_vector(unsigned(A) nor unsigned(B));
end case;
end process;

end Behavioral;
```

Schematic

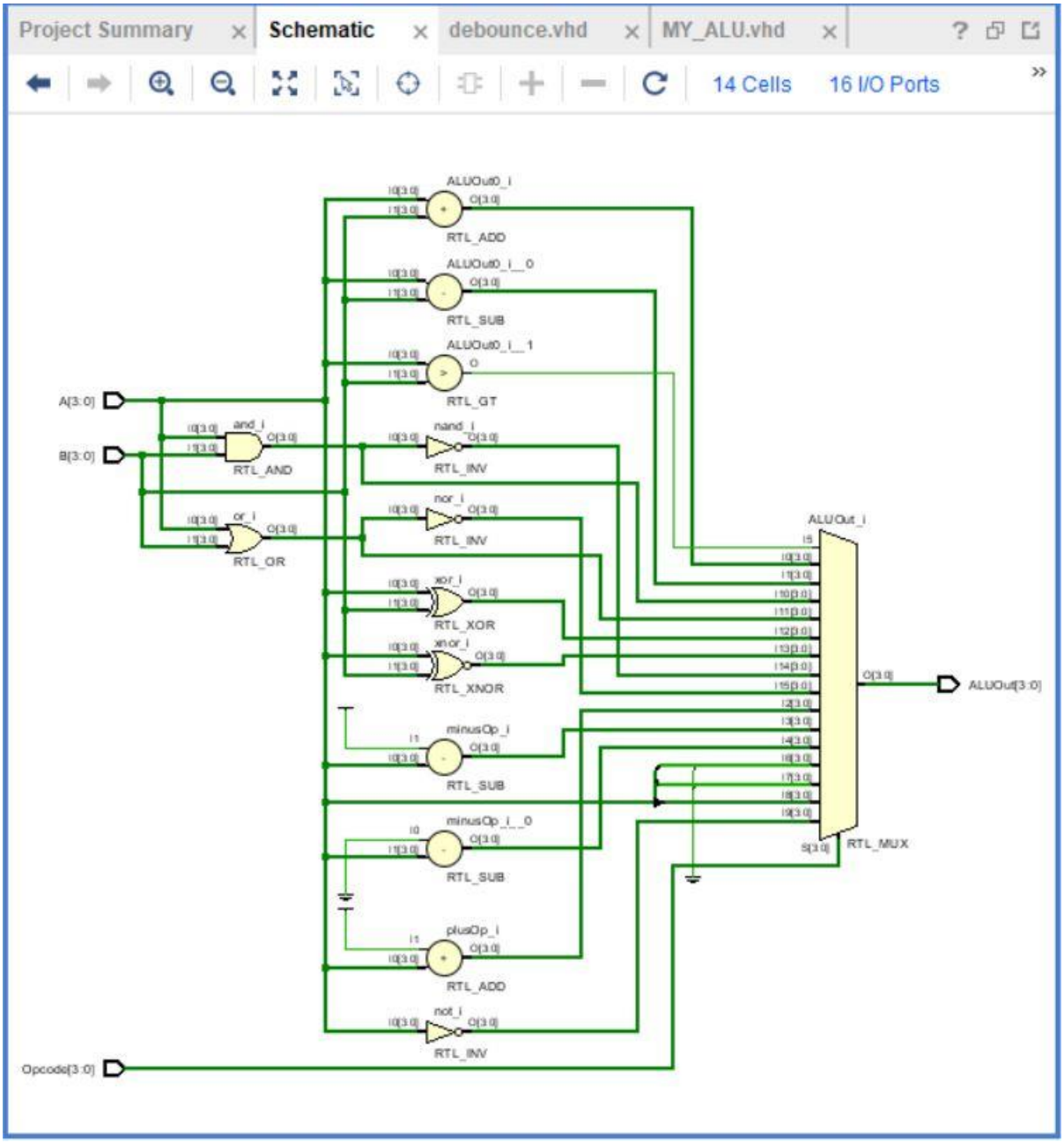

Fig. 9: My ALU Schematic

3. ALU_Tester

The top level synth only combines all available and pre made devices/ systems and creates instances of each component.
The intermediate wiring is done using locally created signals.
The ALU_Tester uses MY_ALU and Debounce as components. The debounce is used to assign functionalities to buttons.
BTN0 : LOAD A
BTN1 : LOAD B
BTN2 : LOAD OPCODE
BTN3 : RESET

VHDL Code

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

entity ALU_Tester is
    Port ( led : out STD_LOGIC_VECTOR (3 downto 0);
           btn : in STD_LOGIC_VECTOR (3 downto 0);
           sw : in STD_LOGIC_VECTOR (3 downto 0);
           clk : in STD_LOGIC);
end ALU_Tester;

architecture Behavioral of ALU_Tester is
signal debounced: std_logic_vector(3 downto 0);
signal temp_A, temp_b, temp_op, temp_reset: std_logic_vector(3 downto 0);

component MY_ALU
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           Opcode : in STD_LOGIC_VECTOR (3 downto 0);
           ALUOut : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component debounce
    Port( clk : in STD_LOGIC;
      btn : in STD_LOGIC;
      dbnc : out STD_LOGIC);
end component;
begin

assign:Process(clk)
```

```
begin
   if(rising_edge(clk)) then
      if(debounced(3) = '1') then
         temp_a <= "0000";
         temp_b <= "0000";
         temp_op <= "0000";
      elsif(debounced(0) = '1') then
         temp_b <= sw(3 downto 0);
      elsif(debounced(1) = '1') then
         temp_a <= sw(3 downto 0);
      elsif(debounced(2) = '1') then
         temp_op <= sw(3 downto 0);
      end if;
   end if;
end process;

Button_Debounce0: debounce
port map(clk => clk,
        btn => btn(0),
        dbnc => debounced(0));

Button_Debounce1: debounce
port map(clk => clk,
        btn => btn(1),
        dbnc => debounced(1));

Button_Debounce2: debounce
port map(clk => clk,
        btn => btn(2),
        dbnc => debounced(2));

Button_Debounce3: debounce
port map(clk => clk,
        btn => btn(3),
        dbnc => debounced(3));

ALU: MY_ALU
port map(A(3 downto 0) => temp_A(3 downto 0),
        B(3 downto 0) => Temp_B(3 downto 0),
        opcode(3 downto 0) => Temp_op(3 downto 0),
        ALUOut(3 downto 0) => led(3 downto 0));

end Behavioral;
```

Schematic

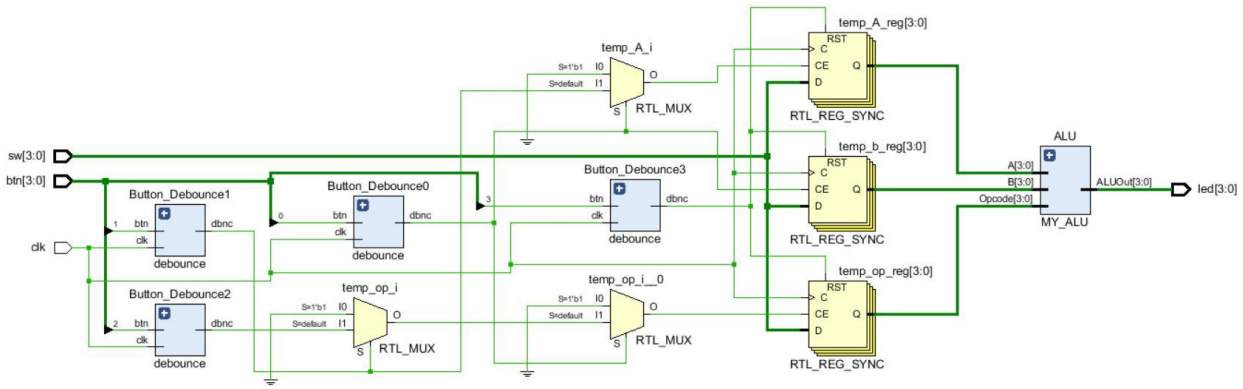

Fig. 10: ALU Tester Schematic

Post-Synthesis Schematic

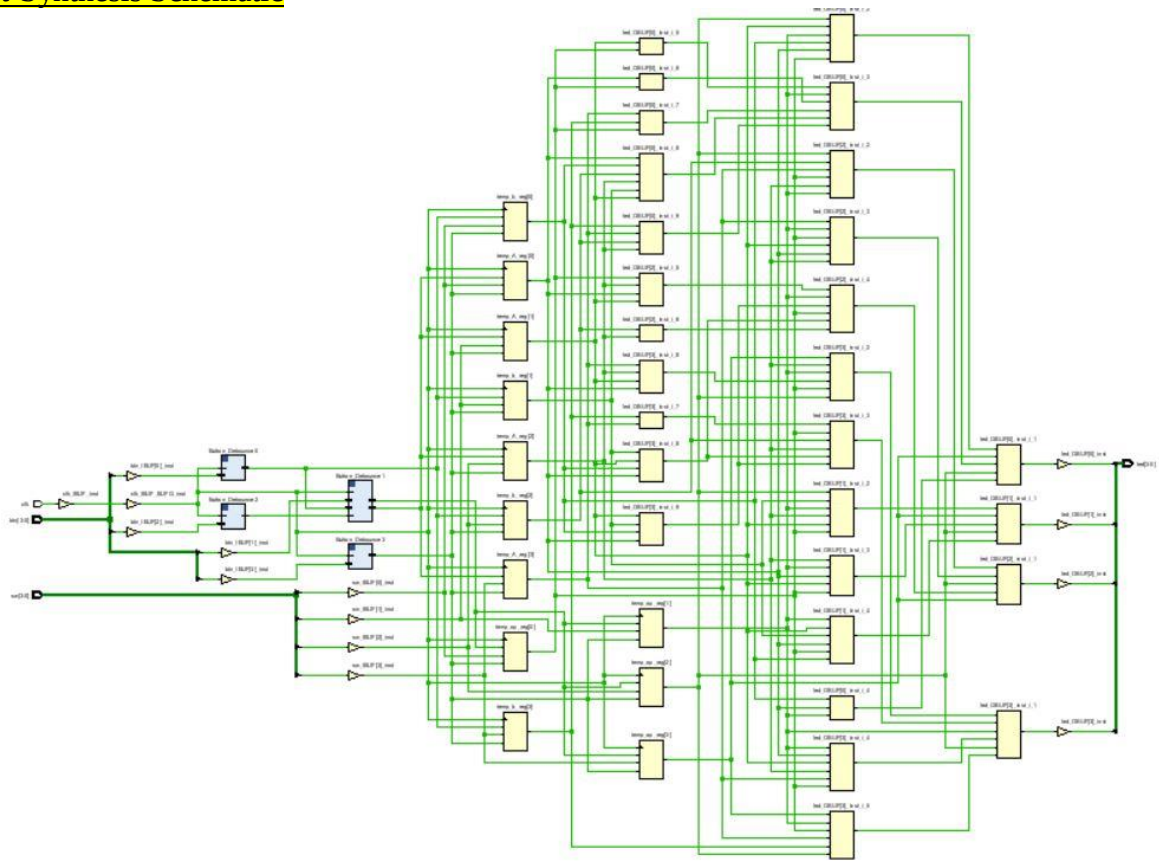

Fig. 11: Post Synthesis Schematic of ALU Tester

Implemented Design

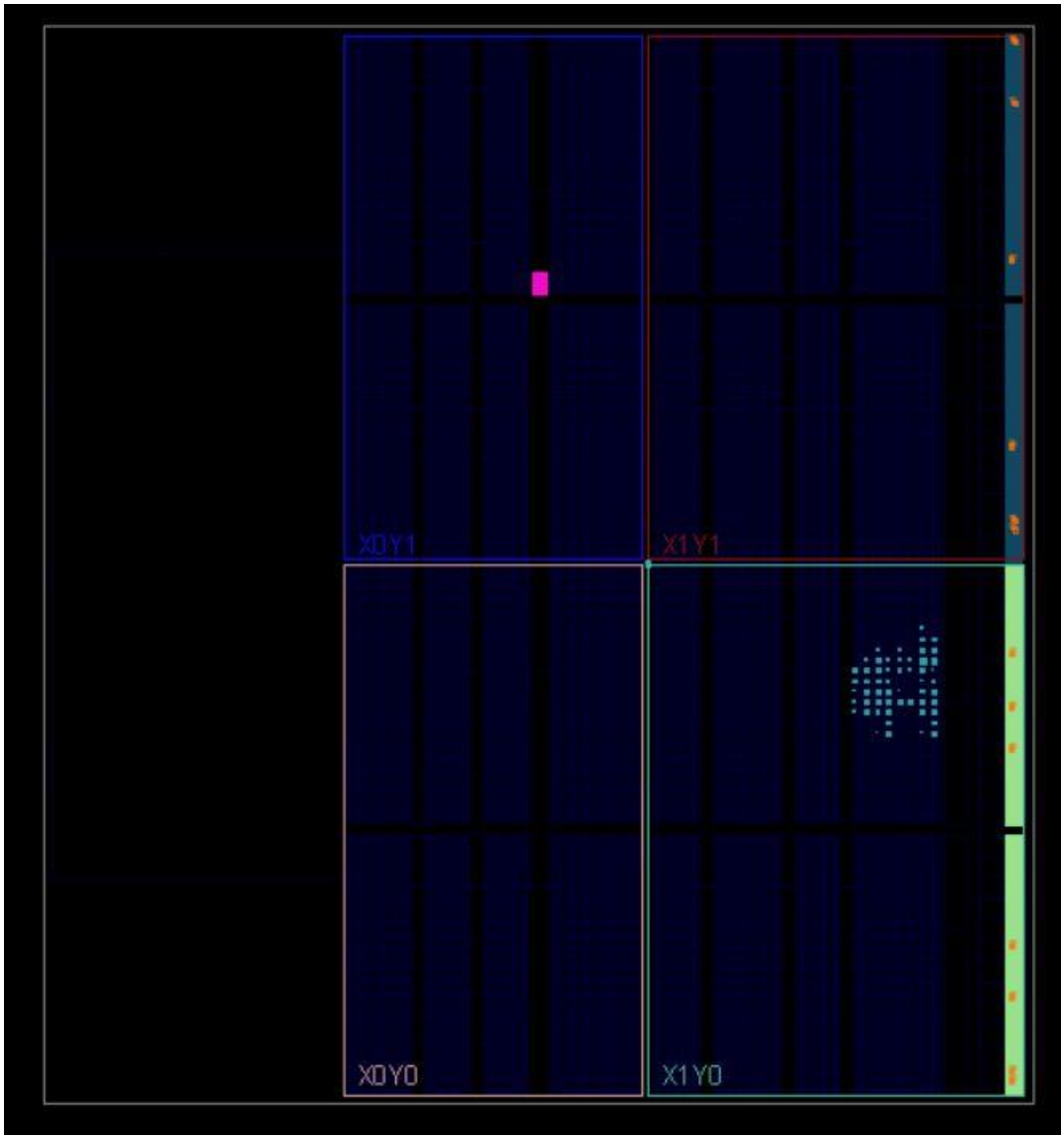

Fig 12. Implemented Design showing the part of FPGA used to implement the ALU

Resource Utilization

Hierarchy

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT Flip Flop Pairs (17600)	Bonded IOB (100)	BUFGCTRL (32)
ALU_Tester	68	112	55	68	5	13	1
Button_Debounce0 (d...	11	25	13	11	1	0	0
Button_Debounce1 (d...	12	25	13	12	1	0	0
Button_Debounce2 (d...	11	25	11	11	1	0	0
Button_Debounce3 (d...	11	25	12	11	1	0	0

Fig. 13: Table showing Resources utilized in making the ALU

Power Utilization

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.096 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.1°C**
Thermal Margin:	58.9°C (5.0 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

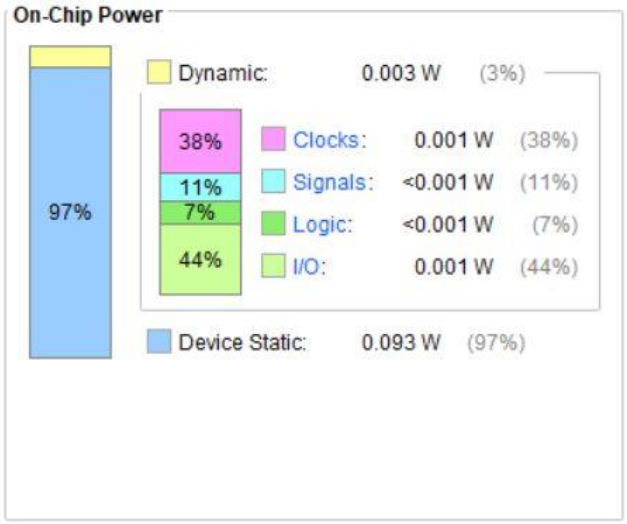

Fig. 14: Power Utilization Report Summary

Discussion

Observations

1. In the XDC file, while using multiple switches/leds/buttons we use '[' to denote each bit. While in the VHDL source files, we use '('.
2. A conditional in the code is converted into a MUX.
3. Any register that stores a value is represented as a D Flip Flop.
4. All inputs and outputs have buffers.
5. Overall Power Utilization for my design is 0.096 Watts, which can be further reduced by using a DSP Slice.
6. A total of 68 LUTs out of a 17600 are used for this design.

Questions/ Follow Up:

Concepts Understood:

1. Switches, Buttons, LEDs
2. Debouncing
3. Instantiation of components

Concepts Unsure of:

1. Why do we not debounce a switch when it is pulled down?
2. Can we implement pipelining on this design?
3. How can we differentiate our design from the existing designs?