Course Name: __Embedded Systems____

Course Number and Section: **14:332:368:01**

Lab: [Lab # [2] – The only time you have to do math]

Professor: Phil Southard

Date Submitted: 03/07/2019

Submitted by: [Timothy Langer RUID# 189005424]

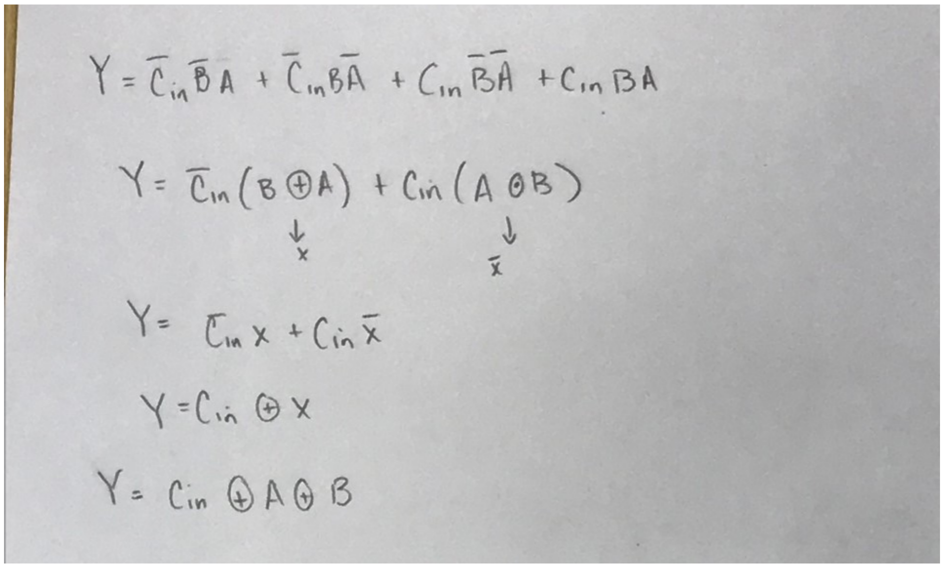
PRE-LAB

Single Bit Full Adder Truth Table

Cin	B	A	Y	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	1	1
1	1	0	1	1
1	1	1	1	1

Write the logic equations for a single bit full adder with inputs A, B, Cin, and outputs Y, Cout.

Y:



$$Y = \bar{C}_{in}\bar{B}A + \bar{C}_{in}B\bar{A} + C_{in}\bar{B}\bar{A} + C_{in}BA$$

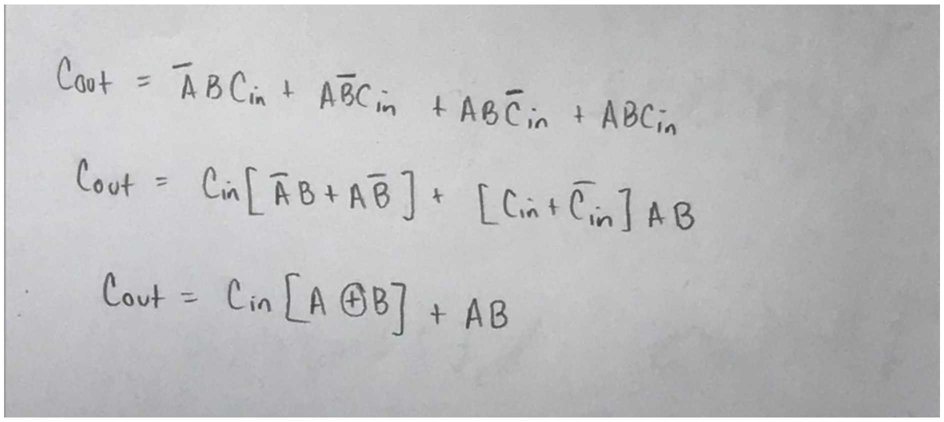
$$Y = \bar{C}_{in}\underset{\substack{\downarrow \\ x}}{(B \oplus A)} + C_{in}\underset{\substack{\downarrow \\ \bar{x}}}{(A \odot B)}$$

$$Y = \bar{C}_{in}x + C_{in}\bar{x}$$

$$Y = C_{in} \oplus x$$

$$Y = C_{in} \oplus A \oplus B$$

Cout :

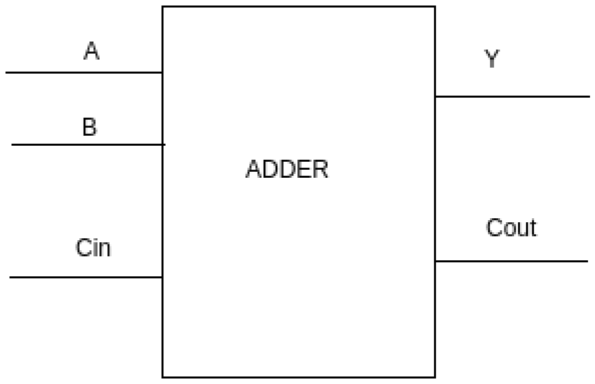


$$C_{out} = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in}$$

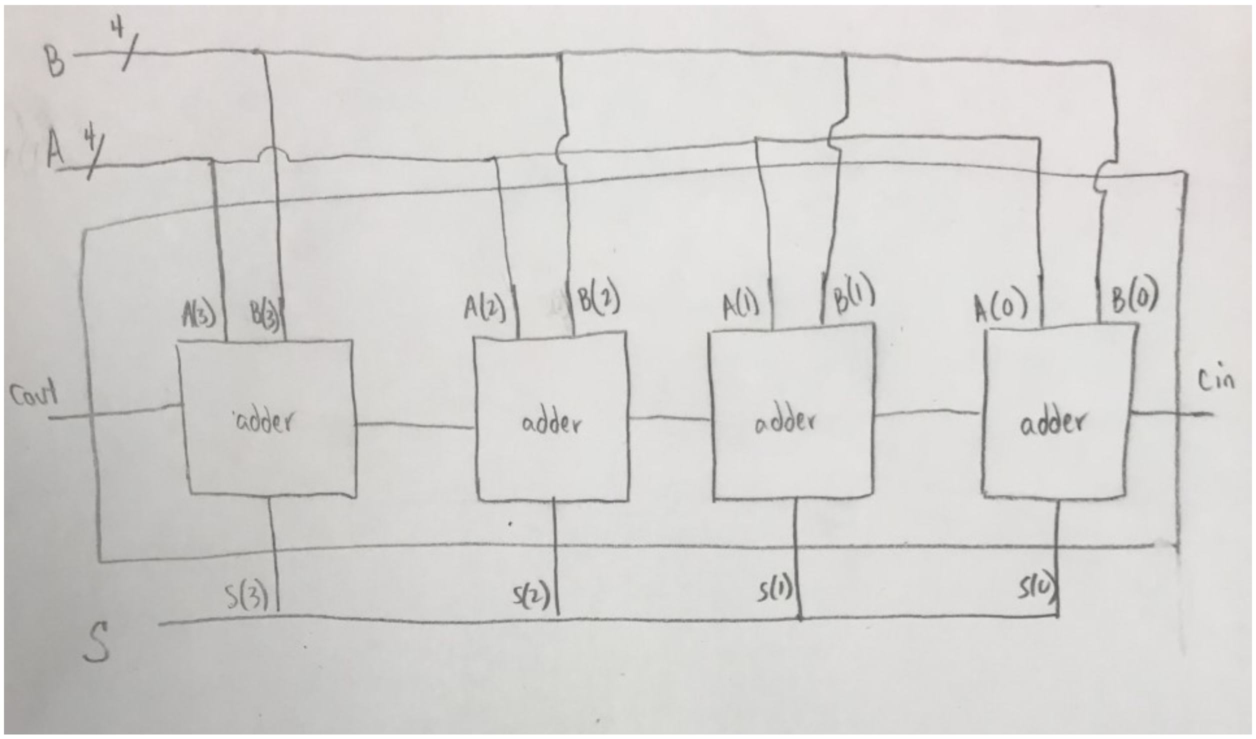
$$C_{out} = C_{in}[\bar{A}B + A\bar{B}] + [C_{in} + \bar{C}_{in}]AB$$

$$C_{out} = C_{in}[A \oplus B] + AB$$

Single Bit Adder(ADDER) Black Box Diagram:



ripple_adder Box Diagram:



PURPOSE:

The purpose of this lab is to be able to understand and create a multi-bit adder and how a multi-function ALU is created in vhdl and implemented in a circuit. Part 1 helped to understand how from scratch, a single bit adder is created and is used to create a 4 bit adder. Starting with truth tables, they are used to create boolean expressions that represent the ouputs such as Y, and Cout. Using a Karanaugh map, the logic can be minizmied and a gate level description can be made to represent the data flow. This lab also helps to further understand structural design methodology by incorporating multiple single bit adders to create a 4-bit adder.

The purpose of part 2 of this lab is to understand the power of VHDL's numeric_std library. The important aspect is that this library contains many of the basic arithmetic and logical functions. They are already defined and synthesize-able without having to create your own VHDL code. Therefore a multi- function ALU can be made without having to spend the time creating a hardware implementation for each function.

PART 1

Theory of Operation:

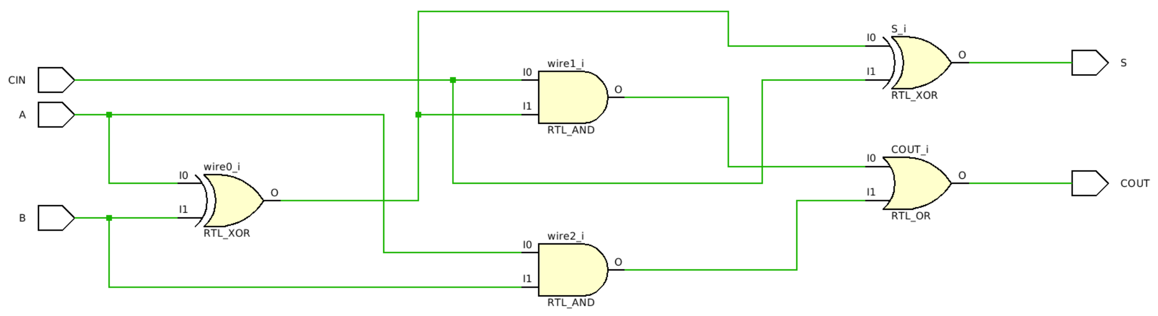
This part begins with the construction of a single-bit full adder called adder. From the logic created from the truth table, which also matches the gate level schematic provided in the lab instructions, the single bit adder was made. Then by connecting four of them together in the top design called ripple_adder, this made a 4-bit adder. This ripple_adder has an input of **Cin**, 4-bit input **A**, and 4-bit input **B**. With an ouput **Cout**, and 4-bit output **S**.

This circuit can be used under a few scenarios. First we can use it to add two 4-bit numbers. To do so, the Cin on the ripple counter should be set to zero. This circuit should then be able to correctly add two 4- bit numbers. If the answer resulted in an overflow, the overflow bit will set the Cout bit high, or a ‘1’. If there is no overflow, then the circuit will run as intended.

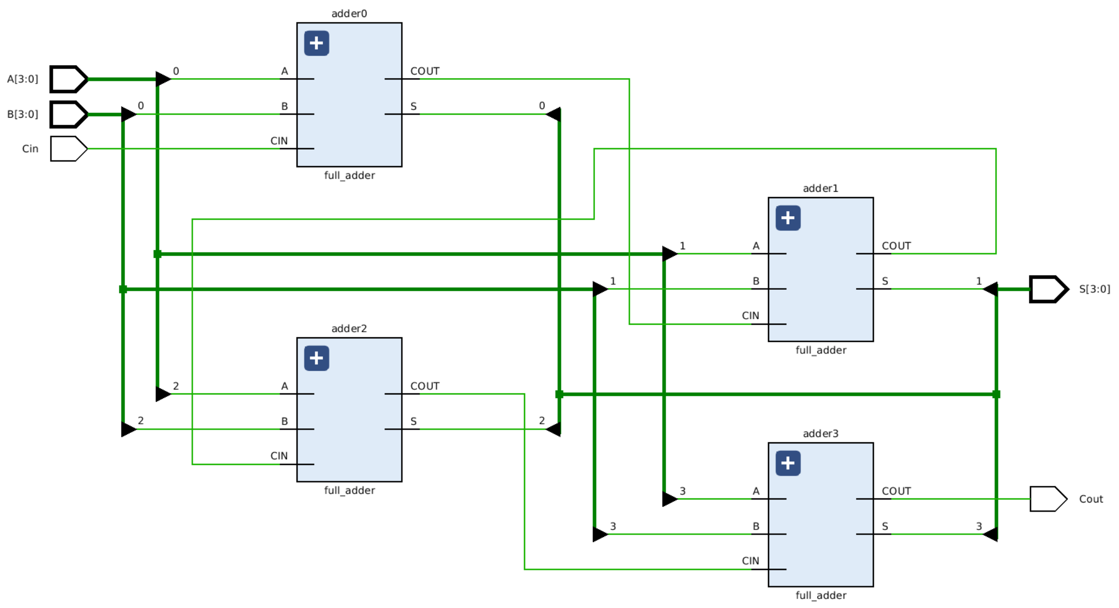
Another situation this ripple_adder can be used for, can be chains of ripple_adders. To connect them together the Cin of a ripple adder would connect to the Cout of another ripple adder. Together they can now represent an 8-bit adder. This 8-bit adder will also have a Cin input and a Cout output. Meaning the chain can be made even longer. The adder can be 4Bit, 8bit, 12bit and so on.

A test bench was then used to simulate the circuit. It successfully simulated the circuit as expected.

Single bit-adder “adder” SCHEMATIC:



4-bit adder, “ripple_adder” SCHEMATIC:



VHDL CODE:

single bit adder “adder”:

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

entity full_adder is
    Port ( B : in STD_LOGIC;
           A : in STD_LOGIC;
           CIN : in STD_LOGIC;
           S : out STD_LOGIC;
           COUT : out STD_LOGIC);
end full_adder;

architecture Behavioral of full_adder is
signal wire0, wire1, wire2 : std_logic;
begin

wire0 <= a xor b;
S<= wire0 xor CIN;
wire1 <= cin and wire0;
wire2 <= a and b;
cout <= wire1 or wire2;

end Behavioral;
```

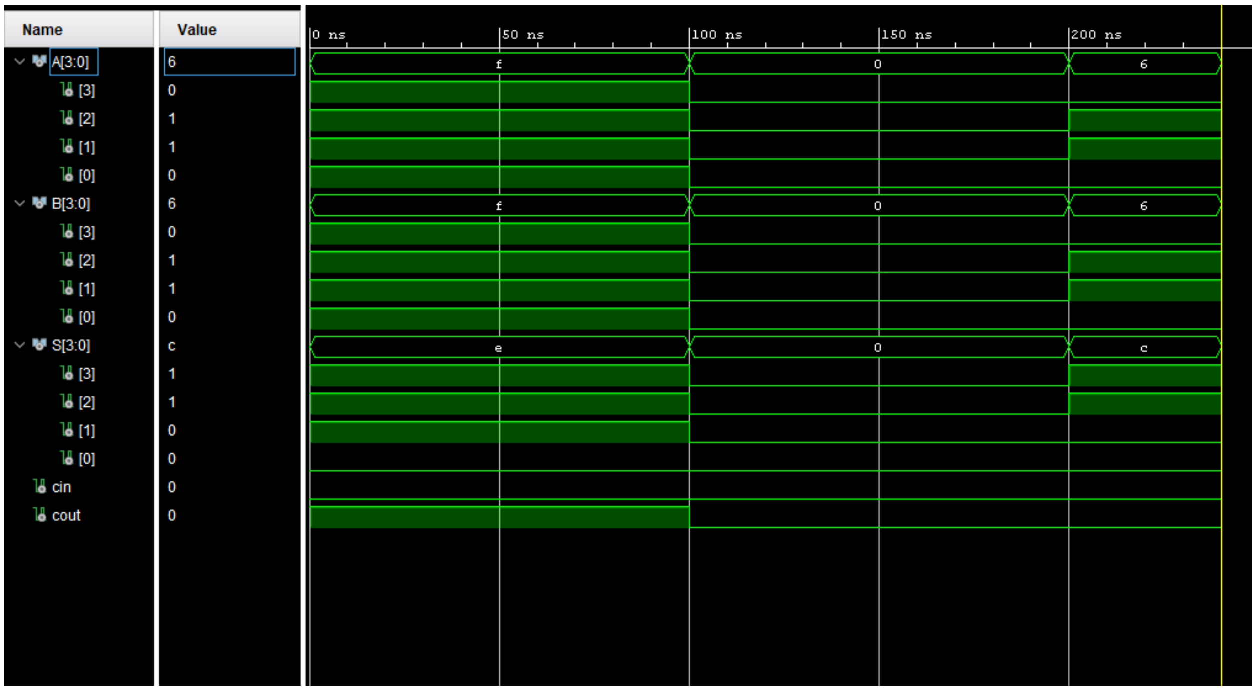
4-bit ripple adder “ripple_adder”:

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

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

architecture Behavioral of ripple_adder is

component full_adder
Port(
B : in STD_LOGIC;
A : in STD_LOGIC;
CIN : in STD_LOGIC;
S : out STD_LOGIC;
COUT : out STD_LOGIC
);
end component;
signal wire : std_logic_vector(3 downto 0);
begin
wire(0)<=Cin;

adder0: full_adder
port map(
B => B(0),
A=> A(0),
CIN=> wire(0),
S=> s(0),
COUT => wire(1)
);
adder1: full_adder
port map(
B => B(1),
A=> A(1),
CIN=> wire(1),
S=> s(1),
COUT => wire(2)
);
adder2: full_adder
port map(
B => B(2),
A=> A(2),
CIN=> wire(2),
S=> s(2),
COUT => wire(3)
);
adder3: full_adder
port map(
B => B(3),
A=> A(3),
CIN=> wire(3),
S=> s(3),
COUT => cout
);

end Behavioral;
```

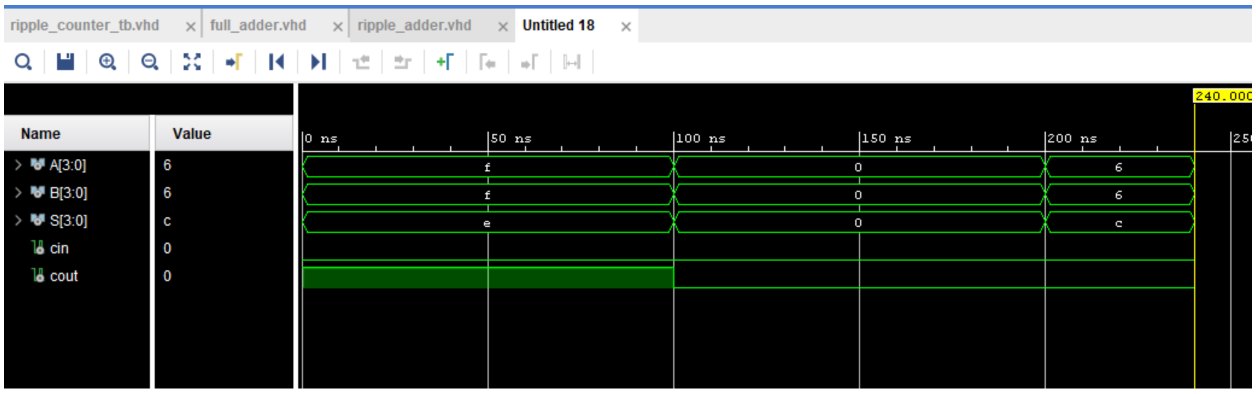
testbench code:

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

entity ripple_counter_tb is
end ripple_counter_tb;

architecture Behavioral of ripple_adder_tb is

component ripple_adder is
Port
(A : in STD_LOGIC_VECTOR (3 downto 0);
 B : in STD_LOGIC_VECTOR (3 downto 0);
 Cin : in STD_LOGIC;
 S : out STD_LOGIC_VECTOR (3 downto 0);
 Cout : out STD_LOGIC);
end component;
signal A,B: std_logic_vector( 3 downto 0) := (others => '0');
signal S: std_logic_vector( 3 downto 0);
signal cin: std_logic := '0';
signal cout: std_logic;
begin

UUT: ripple_adder
Port map(
A => A,
B=> B,
cin => cin,
s=>S,
cout =>cout
);

stimulus : process
begin

cin <='0';
 A <= "1111";
 B <= "1111";
wait for 100 ns;
 A <= "0000";
 B <= "0000";
wait for 100 ns;

 A<= "0110";
 B<= "0110";

wait for 40 ns;
wait;
end process;

end Behavioral;
```

Simulation Window:



Same Simulation Window showing only hex values:



SIMULATIONS EXPLAINED:

Three cases were done to test the ripple_counter:

1) (A = 1111) + (B=1111) will output S = 1110, cout = 1
   - In this case it will result in overflow, so the correct answer won't be in theoutput, what will be displayed is 1110 with a carry bit of 1. The cout bit is taking the overflow bit. The cout bit can be thought of as the 5th bit. The output S in the simulation correctly returned "e" which is equal to "1110", with a cout of 1.
2) (A = 0000) + (B = 0000) will output S = 0000, cout = 0
   - In the simulation window it shows correctly the output 0 and the carry out bit 0
3) (A = 0110) + (B=0110) will output S = 1100, cout = 0
   - In this case the answer should be 12(decimal) = 1100(binary) with a carry out bit, cout = 0.

The simulation correctly output 'c' in hex which is equal to 1100, with a carry outbit, cout = 0.

PART 2

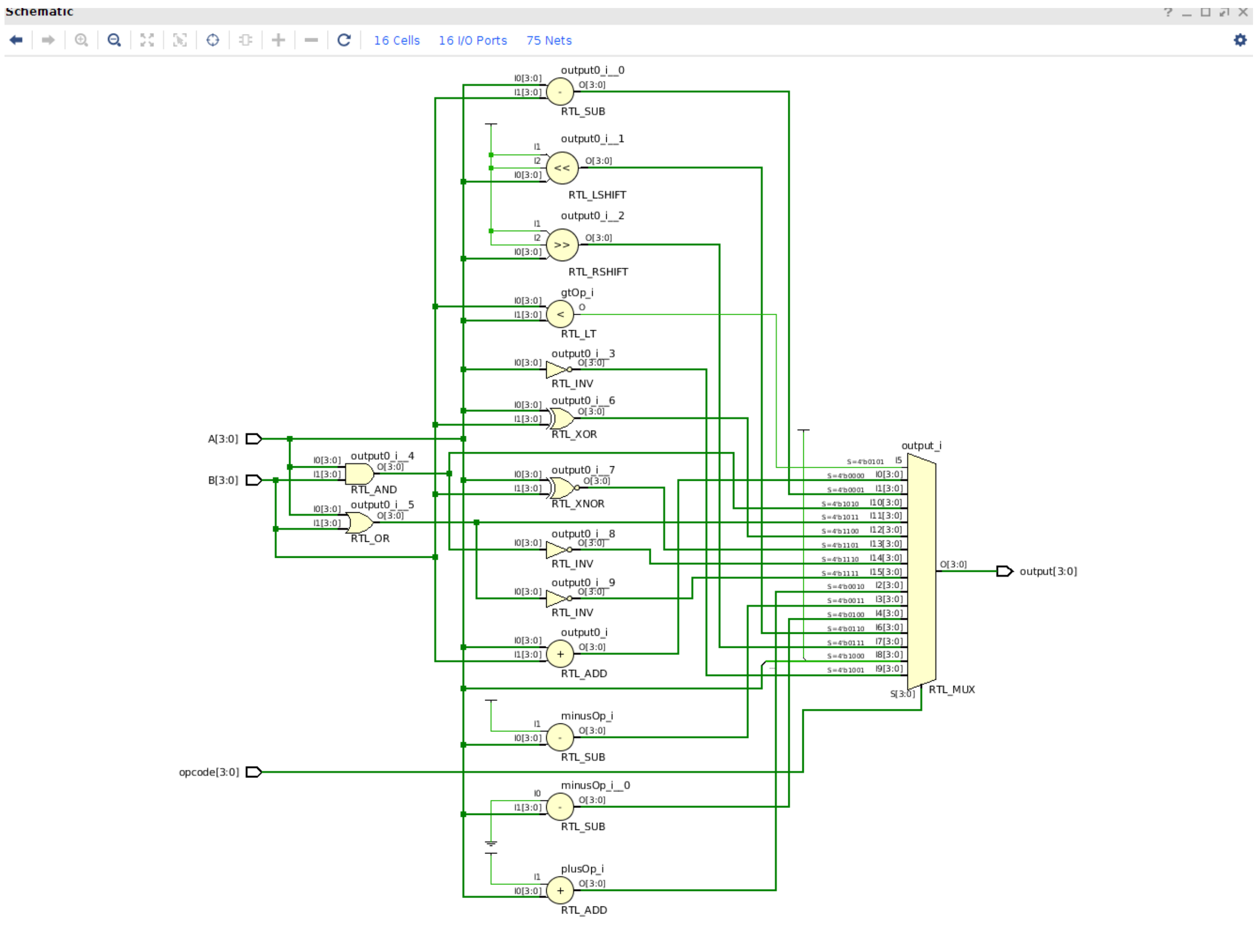
THEORY OF OPERATION:

For this part of the lab, we constructed a 16 function ALU. By taking advantage of the std_numeric library, a multi function ALU can be designed without having to code each function independently. We then created a top design that incorporates the 16 function ALU and the debouncer created from a previous lab. The top design called Alu_tester will have 3 inputs, **switch** and **button** to two different 4-bit std_logic_vectors, and a **clk**. The output will be stored in a 4-bit std_logic_vector to **LED**.
The output of the my_alu will be connected to **LED**. The values for **A**, **B**, and **OPCODE** from my_alu will be assigned to the **SWITCH** inputs. The clock enables for the OPCODE, A, and B registers will be connected to the outputs of the button debouncers. The debouncers are used to stabilize the output of a pressed button. By pressing button(2), and when its Debounced output is high, it will update the **OPCODE** value/register. Button(1) updates **A** value, and Button(0) updates **B** value. Button(3) will reset the signals of A,B, and OPCODE to “0000”. Using the button update functionality, we can use one set of 4 switches to update the three signals when desired. All of these operations will only happen on a rising clock edge.

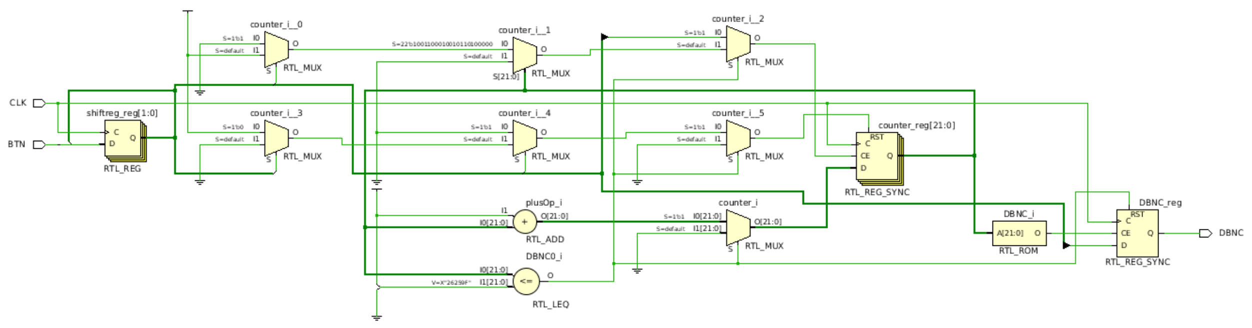
All of this functionality together will enable the use of a 16 function ALU. Assigning values to the A,B, and OPCODE signals are performed by using the buttons and switches. Depending on the opcode, the ALU will perform the corresponding logic/arithmetic operation on A and B and output the result to the LEDS.

A demo of this design was performed on a zybo board and with success behaved as expected.

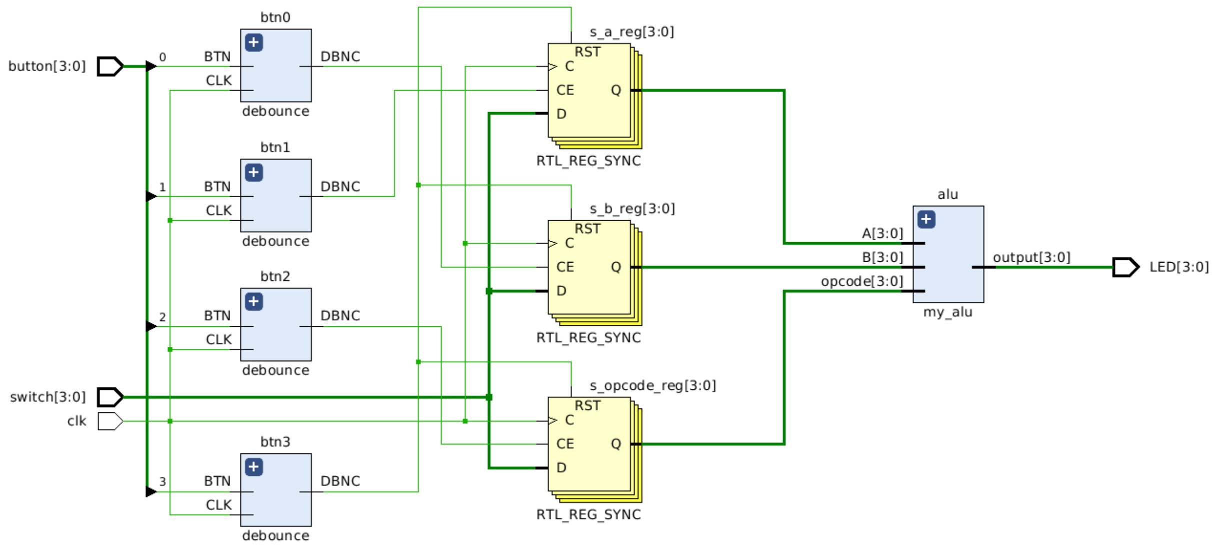
“my_alu” SCHEMATIC:



“debounce” SCHEMATIC:



“alu_tester” (top design) SCHEMATIC:



VHDL CODE

my_alu:

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

use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.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);
           output : out STD_LOGIC_VECTOR (3 downto 0));
end my_alu;

architecture Behavioral of my_alu is

begin
process(A,B,opcode)
begin
 case (opcode) is
     when "0000" => output<= std_logic_vector(unsigned(A) + unsigned(B));
     when "0001" => output<= std_logic_vector(unsigned(A) - unsigned(B));
     when "0010" => output<= std_logic_vector(unsigned(A) + 1);
     when "0011" => output <= std_logic_vector(unsigned(A) -1);
     when "0100" => output<= std_logic_vector(0- unsigned(A));
     when "0101" => if (A>B) then output<="0001"; else output<= "0000"; end if;
     when "0110" => output<= std_logic_vector(unsigned(A) sll 1);
     when "0111" => output<= std_logic_vector(unsigned(A) srl 1);
     when "1000" => output<= '1' & A(3 downto 1);
     when "1001" =>output<= not A;
     when "1010" =>output<= A and B;
     when "1011" =>output<= A or B;
     when "1100" =>output<= A xor B;
     when "1101" =>output<= A xnor B;
     when "1110" =>output<= A nand B;
     when "1111" =>output<= A nor B;
     when others =>output<="0000";
  end case;
  end process;

end Behavioral;
```

debounce:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity debounce is
    Port ( BTN : in STD_LOGIC;
           CLK : in STD_LOGIC;
           DBNC : out STD_LOGIC);
end debounce;

architecture Behavioral of debounce is
signal shiftreg : std_logic_vector (1 downto 0) := (others => '0');
signal counter : std_logic_vector(21 downto 0) := (others =>'0');
begin

process (clk)
begin

if (rising_edge(clk)) then
    shiftreg(1) <= shiftreg(0);
    shiftreg(0) <= BTN;

if (unsigned(counter)<= 2499999) then
    DBNC <= '0';
    if shiftreg(1) = '1' then
    counter <= std_logic_vector(unsigned(counter)+1);
    elsif (shiftreg(1)='0') then
    counter <= (others => '0');

    end if;

elsif (unsigned(counter)= 2500000) then
    if (shiftreg(1)='1') then
    DBNC <='1';
    else
    DBNC <='0';
    counter <= (others =>'0');
    end if;

end if;
end if;

end process;

end Behavioral;
```

alu_tester:

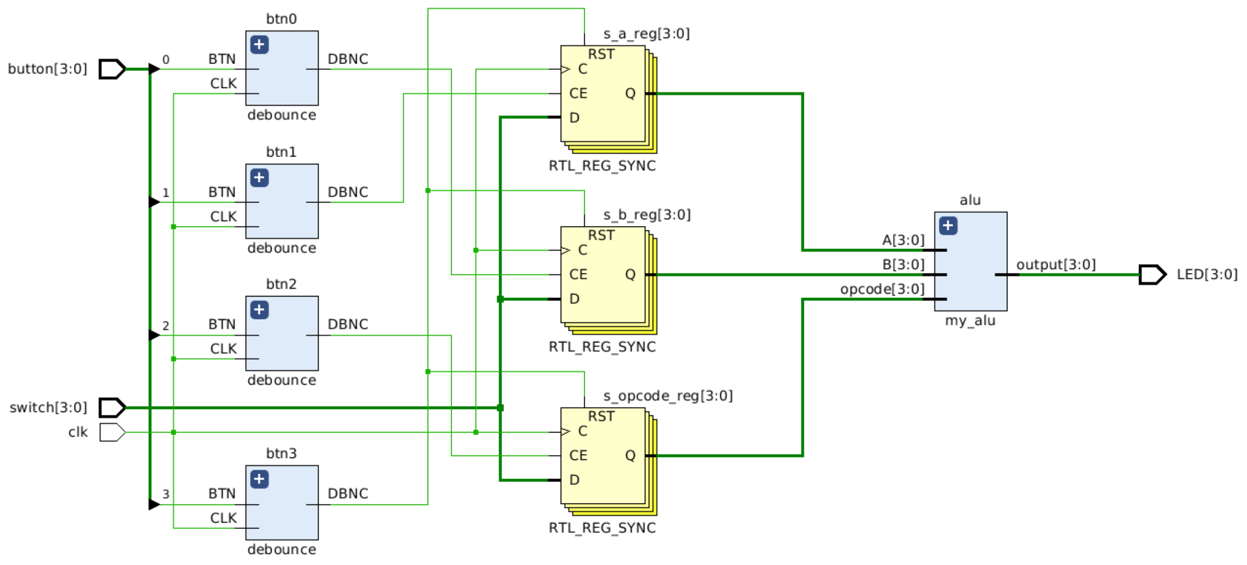
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity alu_tester is
    Port ( switch : in STD_LOGIC_VECTOR (3 downto 0);
           button : in STD_LOGIC_VECTOR (3 downto 0);
           clk : in std_logic;
           LED : out STD_LOGIC_VECTOR (3 downto 0));
    end alu_tester;



architecture Behavioral of alu_tester is

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);
output : out STD_LOGIC_VECTOR (3 downto 0)
);
end component;

component debounce
    Port ( BTN : in STD_LOGIC;
           CLK : in STD_LOGIC;
           DBNC : out STD_LOGIC);
end component;

signal s_dbnc : std_logic_vector(3 downto 0);
signal s_opcode, s_a, s_b : std_logic_vector ( 3 downto 0);
begin

process(clk)
begin
if (rising_edge(clk)) then
if s_dbnc(3) ='1' then
s_opcode <="0000";
s_a <="0000";
s_b<="0000" ;
else
    if s_dbnc(2) ='1' then s_opcode <= switch; end if;
    if s_dbnc(1) ='1' then s_a<= switch; end if;
    if s_dbnc(0) ='1' then s_b <= switch; end if;
```

```
        end if;
end if;
end process;


btn3: debounce
port map(
btn => button(3),
clk => clk,
dbnc => s_dbnc(3)
);
btn2: debounce
port map(
btn => button(2),
clk => clk,
dbnc => s_dbnc(2)
);
btn1: debounce
port map(
btn => button(1),
clk => clk,
dbnc => s_dbnc(1)
);
btn0: debounce
port map(
btn => button(0),
clk => clk,
dbnc => s_dbnc(0)
);
alu: my_alu
port map(
A => s_a,
B => s_b,
opcode => s_opcode,
output => LED
);


end Behavioral;
```

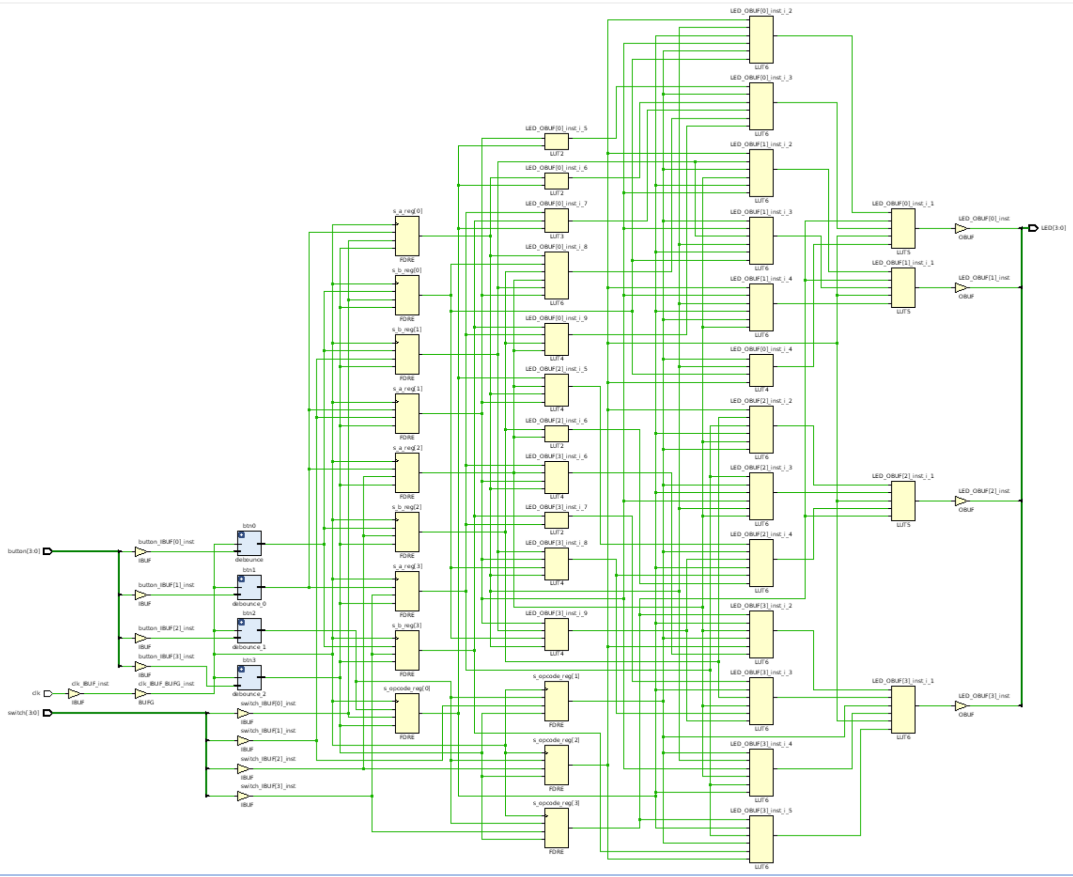
XDC FILE:

```
##Clock signal
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports
{ clk }]; #IO_L11P_T1_SRCC_35 Sch=sysclk
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports
{ clk }];


##Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports
{ switch[0] }]; #IO_L19N_T3_VREF_35 Sch=SW0
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports
{ switch[1] }];  #IO_L24P_T3_34 Sch=SW1
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports
{ switch[2] }]; #IO_L4N_T0_34 Sch=SW2
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports
{ switch[3] }]; #IO_L9P_T1_DQS_34 Sch=SW3


##LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports
{ LED[0] }]; #IO_L23P_T3_35 Sch=LED0
set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports
{ LED[1] }]; #IO_L23N_T3_35 Sch=LED1
set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports
{ LED[2] }]; #IO_0_35=Sch=LED2
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports
{ LED[3] }]; #IO_L3N_T0_DQS_AD1N_35 Sch=LED3
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports
{ button[0] }]; #IO_L23P_T3_35 Sch=LED0
set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports
{ button[1] }]; #IO_L23N_T3_35 Sch=LED1
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports
{ button[2] }]; #IO_0_35=Sch=LED2
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports
{ button[3] }]; #IO_L3N_T0_DQS_AD1N_35 Sch=LED3
```

ELABORATION SCHEMATIC:



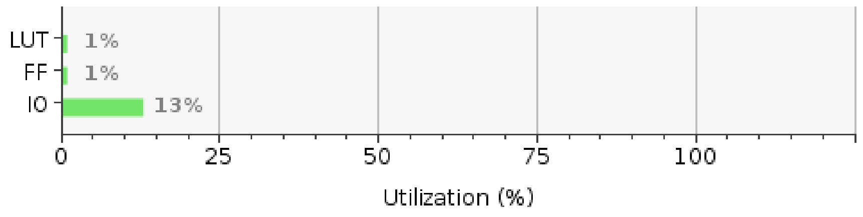
SYNTHESIS SCHEMATIC:



UTILIZATION:

Summary

Resource	Utilization	Available	Utilization %
LUT	171	17600	0.97
FF	112	35200	0.32
IO	13	100	13.00



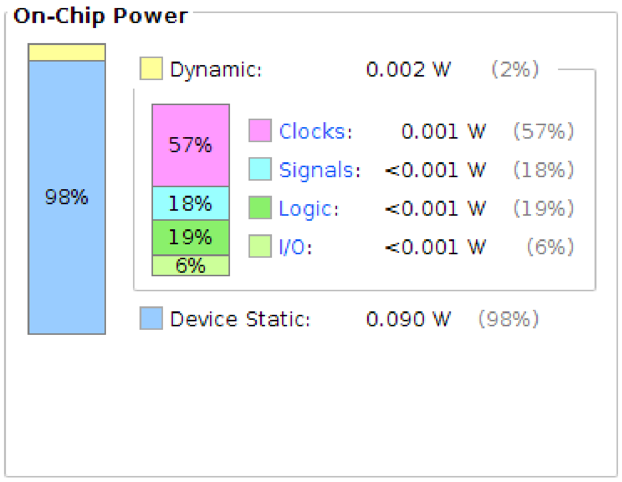
POWER:

Summary

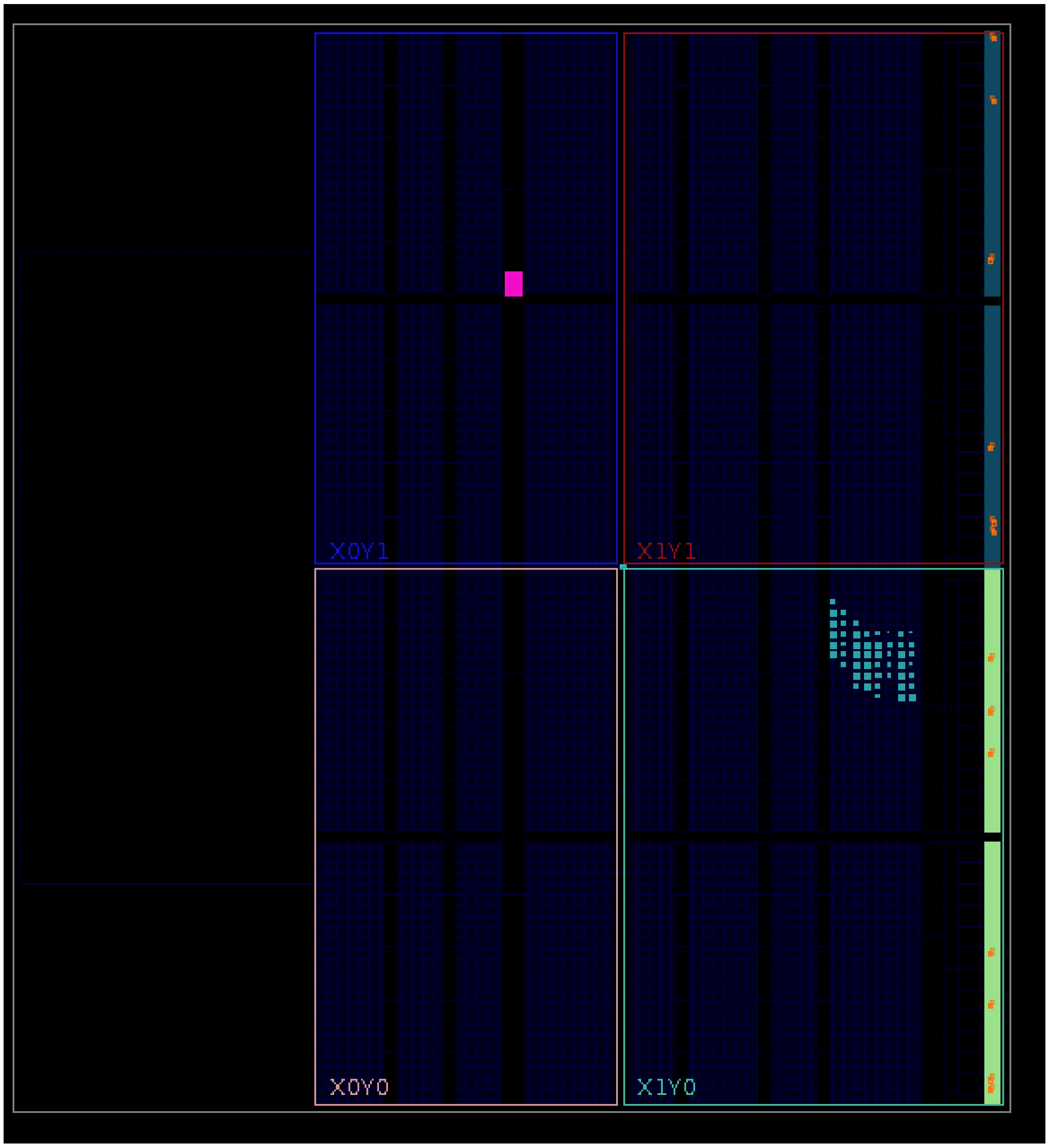
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.091 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

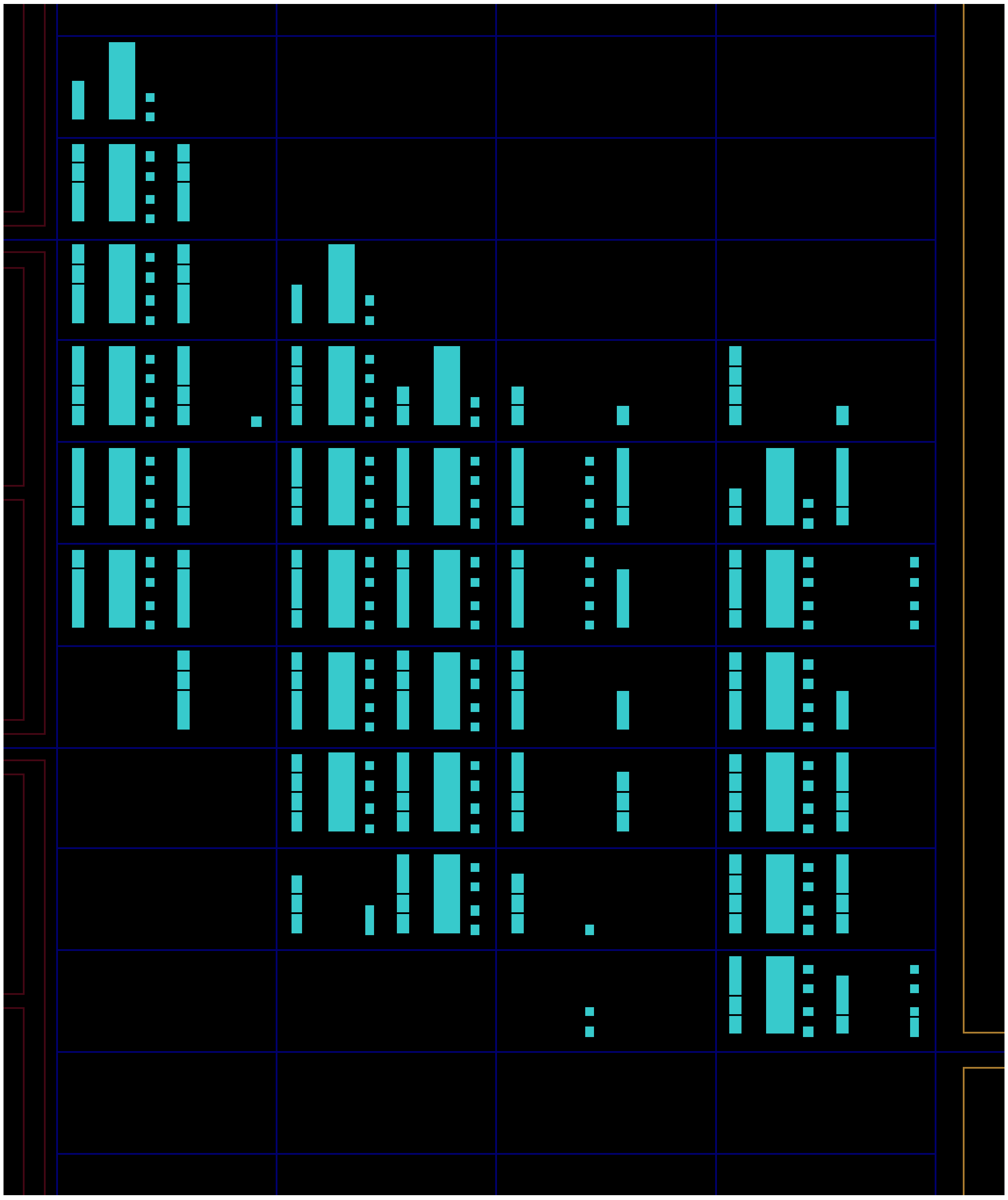
Launch Power Constraint Advisor to find and fix invalid switching activity



DEVICE:



DEVICE ZOOMED IN:



XDC FILE CHANGES:

The clk input is connected to pin L16 (125 mHz clock).
The switch input bits 0-4 are connected respectively to pin G15, P15, W13, T16.
The LED output bits are conncected respectively to the boards led pins M14, M15, G14, D18.
The 4 button inputs are respectively connected to pins R18, P16, V16, Y16.
These inputs and outputs cover all the I/O for this design.

DISCUSSION:

It was interesting how the buttons were used to update the OPCODE, A, and B signals. It enabled us to not need three sets of switches to update each signal. By using the buttons we only needed one set of switches. Seeing this implementation made me think of reduction in components and therefore save money on design.