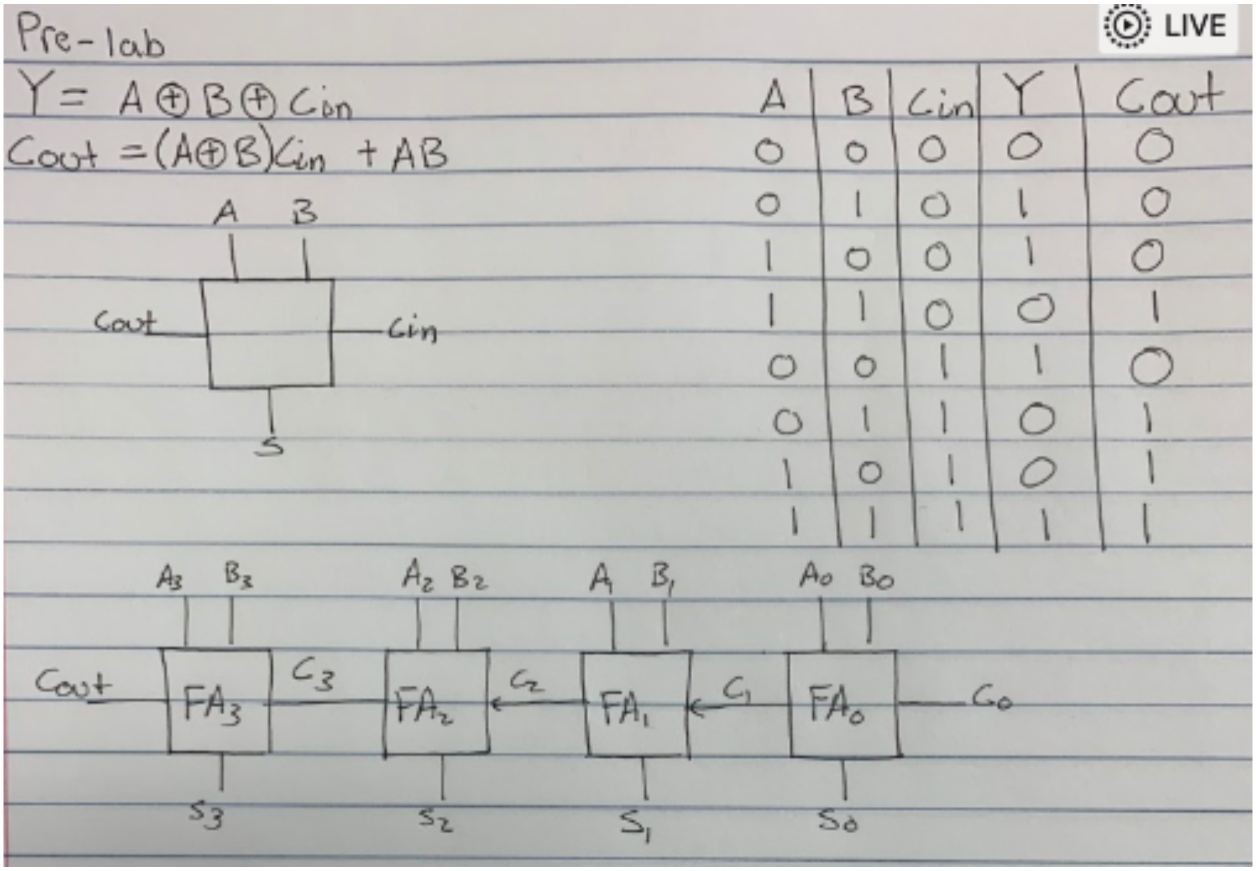
Lab 2 - The only time you Have to do math

Embedded Systems
Lab Report 2
Anthony Lau
March 7, 2019

Purpose:

In this lab we are going to look at a 4-bit ALU. The first function that we will look at and understand is the ripple carry adder, when can be made from multiple single bit full adders in a structural model in VHDL. Next we will create a 16 function 4-bit ALU behaviorally.

Pre-Lab:



Pre-lab

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

$C_{out} = (A \oplus B)C_{in} + AB$

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

LIVE
A
B
Cout
Cin
S
A3
B3
A2
B2
A
B1
A0
B0
Cout
C3
C2
C1
C0
FA3
FA2
FA1
FA0
S3
S2
S1
S0

1) Back to Digital Logic Design

Theory:

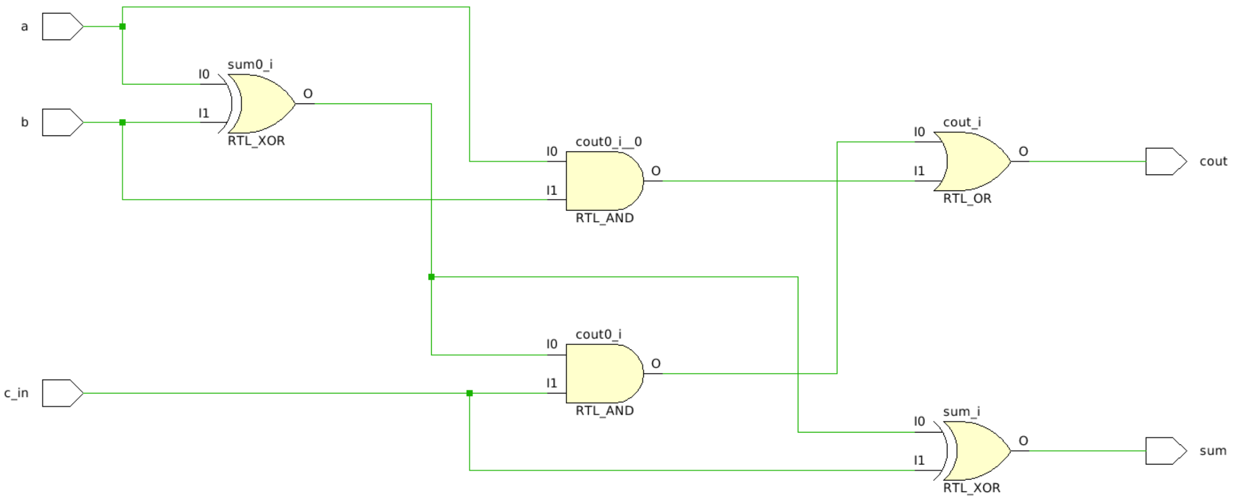
In this part of the lab we are making a 4-bit ripple carry adder from a 4 single bit full adders. We can construct the truth table for a single bit full adder to write the logic eequations for the sum and the carry out. Once we know the equations we can write the VHDL to model the full adder and instantiate it multiple times to create a ripple adder. The ripple adders carry in will be the carry out of the previous full adder.

Truth Table:

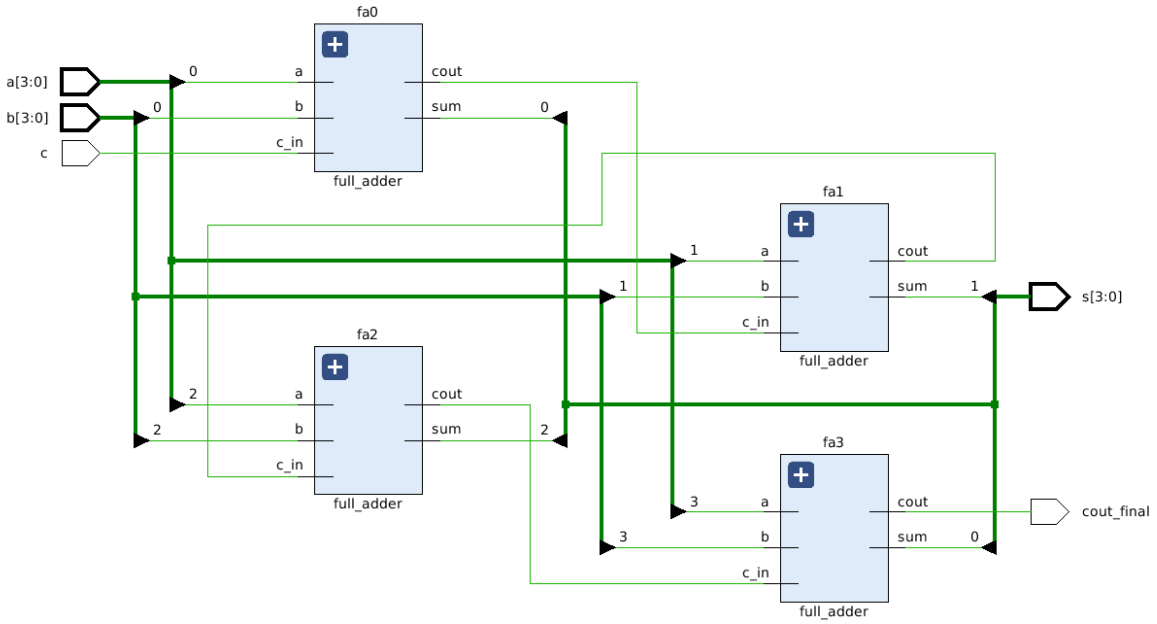
A	B	Cin	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Schematic Diagram:

Full Adder



Ripple Adder



Design:

Full Adder

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

entity full_adder is
Port (a, b, c_in: in std_logic;
      sum, cout: out std_logic);
end full_adder;

architecture Behavioral of full_adder is

begin

sum <= a xor b xor c_in;
cout <= ((a xor b) and c_in) or (a and b);

end Behavioral;
```

Ripple Adder

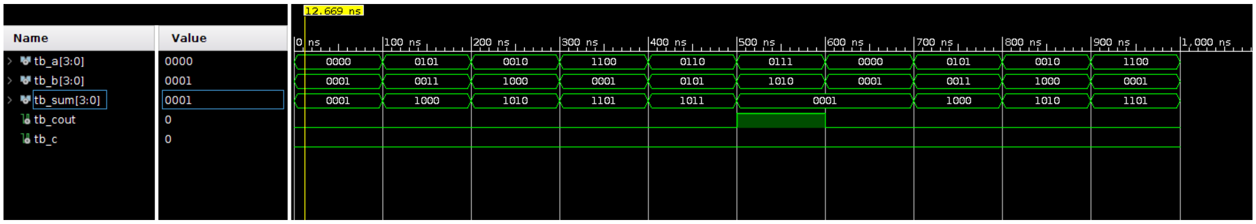
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ripple_adder is
    port(a, b: in std_logic_vector(3 downto 0);
         c: in std_logic;
         s: out std_logic_vector(3 downto 0);
         cout_final: out std_logic);
end ripple_adder;

architecture four_bit of ripple_adder is

component full_adder is
    Port (a, b, c_in: in std_logic;
          sum, cout: out std_logic);
end component;

signal c1 : std_logic;
signal c2 : std_logic;
signal c3 : std_logic;

begin
fa0: full_adder
port map(a => a(0),
         b => b(0),
         c_in => c,
         sum => s(0),
         cout => c1);

fa1: full_adder
port map(a => a(1),
         b => b(1),
         c_in => c1,
         sum => s(1),
         cout => c2);

fa2: full_adder
port map(a => a(2),
         b => b(2),
         c_in => c2,
         sum => s(2),
         cout => c3);

fa3: full_adder
port map(a => a(3),
         b => b(3),
         c_in => c3,
         sum => s(3),
         cout => cout_final);


end four_bit;
```

Test:

Testbench

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

entity ripple_adder_tb is
end ripple_adder_tb;

architecture testbench of ripple_adder_tb is

    signal tb_a, tb_b: std_logic_vector(3 downto 0);
    signal tb_sum: std_logic_vector(3 downto 0);
    signal tb_cout, tb_c: std_logic;

    component ripple_adder is
        port(a, b: in std_logic_vector(3 downto 0);
             s: out std_logic_vector(3 downto 0);
             c: in std_logic;
             cout_final: out std_logic);
    end component;

begin

    generate_inputs: process
    begin
        tb_c <= '0';
        tb_a <= "0000";
        tb_b <= "0001";
        wait for 100 ns;

        tb_c <= '0';
        tb_a <= "0101";
        tb_b <= "0011";
        wait for 100 ns;

        tb_c <= '0';
        tb_a <= "0010";
        tb_b <= "1000";
        wait for 100 ns;

        tb_c <= '0';
        tb_a <= "1100";
        tb_b <= "0001";
        wait for 100 ns;


        tb_c <= '0';
        tb_a <= "0110";
        tb_b <= "0101";
        wait for 100 ns;

        tb_c <= '0';
        tb_a <= "0111";
        tb_b <= "1010";
        wait for 100 ns;
    end process generate_inputs;

    dut: ripple_adder
    port map(a => tb_a,
             c => tb_c,
             b => tb_b,
             s => tb_sum,
             cout_final => tb_cout);

end testbench;
```

Simulation



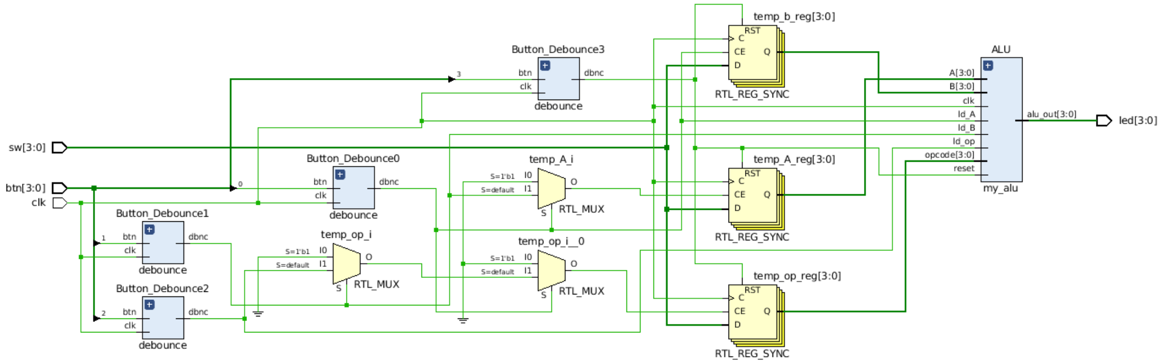
Name	Value
tb_a[3:0]	0000
tb_b[3:0]	0001
tb_sum[3:0]	0001
tb_cout	0
tb_c	0

2) Somebody did the work already

Theory:

In VHDL, many basic arithmetic and logical functions are already defined and synthesizable without us having to make our own hardware implementation. For this part we are going to take advantage of that t0 create a 4-bit 16 function ALU. The ALU will take 3 4-bit inputs A, B and opcode from the switches on the board and perform the operation based on the opcode. The result should be displayed on the leds.

Schematic Diagram:



Design:

Debounce

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity debounce is
Port (clk, btn: in std_logic;
      dbnc: out std_logic);
end debounce;

architecture Behavioral of debounce is

signal counter: std_logic_vector(21 downto 0);
signal count_set: std_logic;
signal shift_register: std_logic_vector(1 downto 0);

begin
    process(clk)
    begin
    count_set <= shift_register(1) xor shift_register(0);
        if(rising_edge(clk)) then
            shift_register(0) <= btn;
            shift_register(1) <= shift_register(0);
                if(count_set = '1') then
                    counter <= (others => '0');
                elsif(counter(21) = '0') then
```

```
                    counter <= std_logic_vector(unsigned(counter) + 1);
                else
                    dbnc <= shift_register(1);
                end if;
        end if;
     end process;
end Behavioral;
```

My_ALU

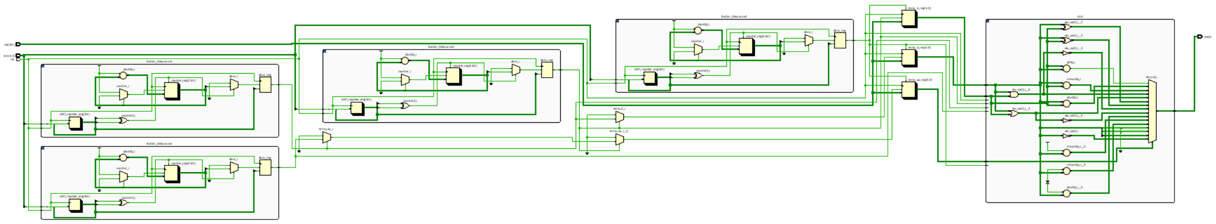
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use IEEE.NUMERIC_STD.ALL;

entity my_alu is
Port(clk, ld_A, ld_B, ld_op, reset: in std_logic;
      A, B, opcode: in std_logic_vector(3 downto 0);
      alu_out: out std_logic_vector(3 downto 0));
end my_alu;

architecture Behavioral of my_alu is

begin
    process(A, B, opcode, reset)
    begin

        case(opcode) is
        when "0000" => alu_out <= A + B;
        when "0001" => alu_out <= A - B;
        when "0010" => alu_out <= A + 1;
        when "0011" => alu_out <= A - 1;
        when "0100" => alu_out <= 0 - A;
        when "0101" =>
            if(A > B) then
                alu_out <= "0001";
            else
                alu_out <= "0000";
            end if;
        when "0110" => alu_out <= A(2 downto 0) & '0';
        when "0111" => alu_out <= '0' & A(3 downto 1);
        when "1000" => alu_out <= A(3) & A(3 downto 1);
        when "1001" => alu_out <= not A;
        when "1010" => alu_out <= A and B;
        when "1011" => alu_out <= A or B;
        when "1100" => alu_out <= A xor B;
        when "1101" => alu_out <= A xnor B;
        when "1110" => alu_out <= A nand B;
        when "1111" => alu_out <= A nor B;
        end case;
    end process;
end Behavioral;
```

ALU_Tester

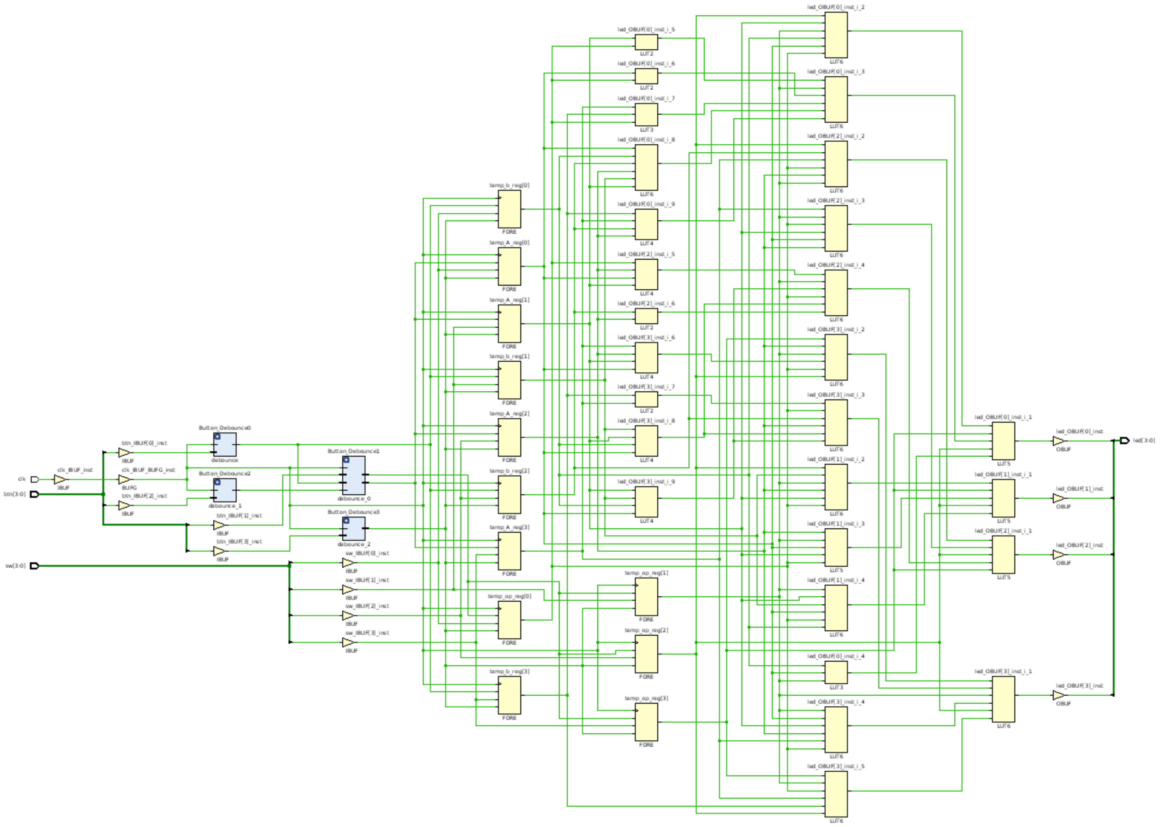
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity alu_tester is
Port (btn, sw: in std_logic_vector(3 downto 0);
      clk: std_logic;
      led: out std_logic_vector(3 downto 0));
end alu_tester;

architecture Behavioral of alu_tester is

----------------------Intermediate Signals------------------------------
signal debounced: std_logic_vector(3 downto 0);
signal temp_A, temp_b, temp_op, temp_reset: std_logic_vector(3 downto 0);

--------------------ALU Component-------------------------------
component my_alu is
Port (clk, ld_A, ld_B, ld_op, reset: in std_logic;
      A, B, opcode: in std_logic_vector(3 downto 0);
      alu_out: out std_logic_vector(3 downto 0));
end component;

---------------------Debounce Component---------------------------
component debounce is
Port (clk, btn: in std_logic;
      dbnc: out std_logic);
end component;

begin

        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(clk => clk,
         ld_A => debounced(0),
         ld_B => debounced(1),
         ld_op => debounced(2),
         reset => debounced(3),
         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),
         alu_out(3 downto 0) => led(3 downto 0));
end Behavioral;
```

Implementation

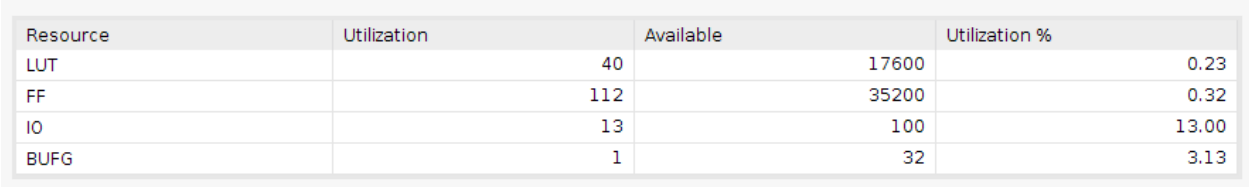
Elaboration Schematic



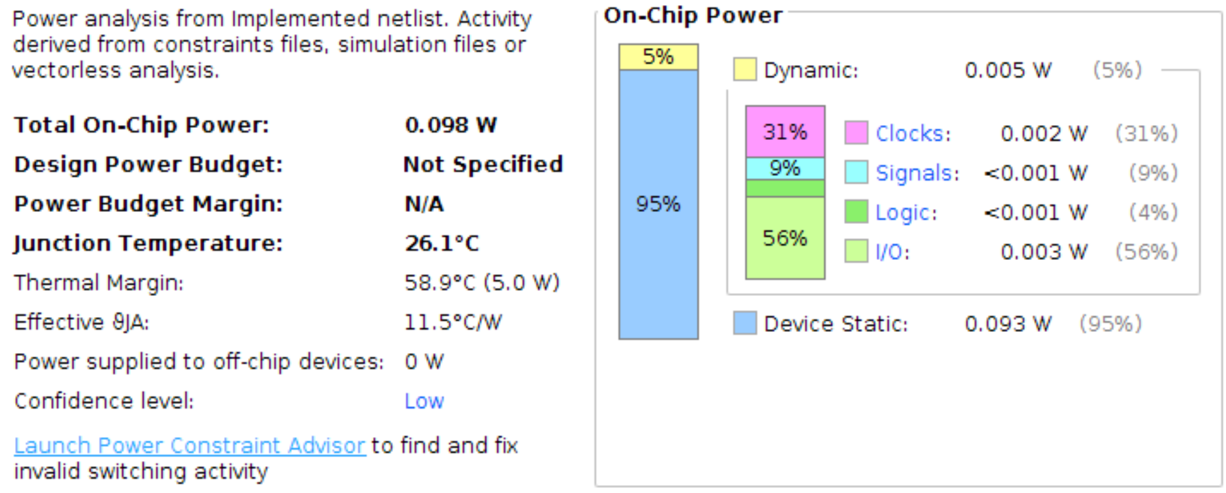
Synthesis Schematic



Project Summary



Resource	Utilization	Available	Utilization %
LUT	40	17600	0.23
FF	112	35200	0.32
IO	13	100	13.00
BUFG	1	32	3.13



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

Total On-Chip Power:	**0.098 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

On-Chip Power

Dynamic:	0.005 W	(5%)
Clocks:	0.002 W	(31%)
Signals:	<0.001 W	(9%)
Logic:	<0.001 W	(4%)
I/O:	0.003 W	(56%)
Device Static:	0.093 W	(95%)

XDC File

## This file is a general .xdc for the ZYBO Rev B board

```
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used signals according to the project


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


##Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L20N_T3_34
Sch=BTN0
set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L24N_T3_34
Sch=BTN1
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }]; #IO_L18P_T2_34
Sch=BTN2
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }]; #IO_L7P_T1_34
Sch=BTN3



##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
```

In my XDC file I uncommented all the buttons, switches, and leds. I had to rename the port names with the outputs that I have in my VHDL code.

Discussion:

Overall, this lab was not too difficult to figure out. What I learned was that the std_logic_unsigned library made it easier to deal with any logic type conversions and allowed me to perform operations directly on the inputs. When there is a negative input, the result shows the two's compliment of the negative value. The concepts that I completely understand is how a ripple adder can be made of multiple full adders, and how to implement a 16 function ALU.