Course Name:Embedded Systems Hardware/Software

Course Number and Section: **14:332493:03**

Experiment: Lab #2, The only time you have to do math

Lab Instructor: Phillip Southard

Date Performed: 02/28/19

Date Submitted: 03/07/19

Submitted by: Raul Mori 183001439

Purpose:

The purpose of this lab is to create a Full_adder (Simple 1-bit Adder), which we use to generate a full 4-bit RIPPLE-ADDER. Using this Ripple-Adder, we create an ALU which can process 16 functions out of its two 2 4-bit inputs fed into it.

Prelab:

LAB 2: THE ONLY TIME YOU HAVE TO DO MATH

PRE LAB

SALMAN HOQUE & MUSHFIQ ATUL

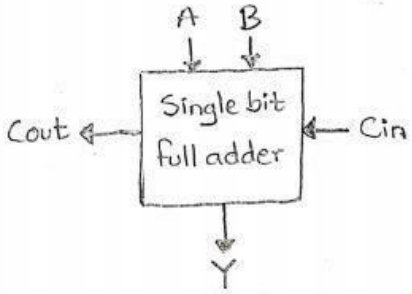
10/09/18

1) Single bit full adder logic equation:

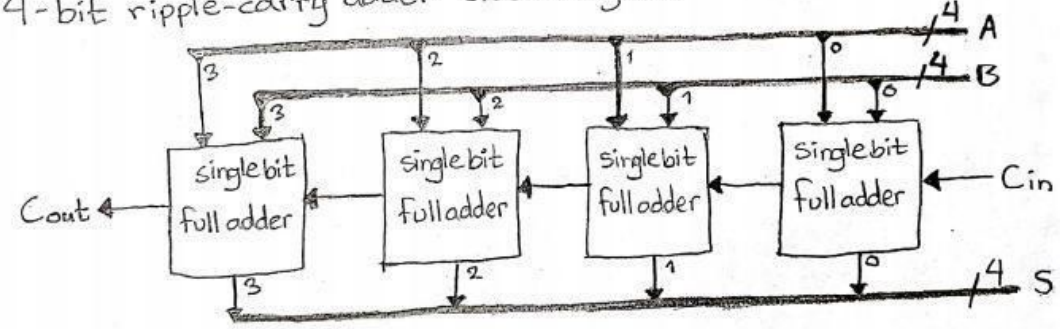
$$Y = A \oplus B \oplus Cin$$

$$Cout = Cin(A \oplus B) + AB$$

2) Single bit Full adder black box diagram:



3) 4-bit ripple-carry adder block diagram



Part 1: Back to digital logic design

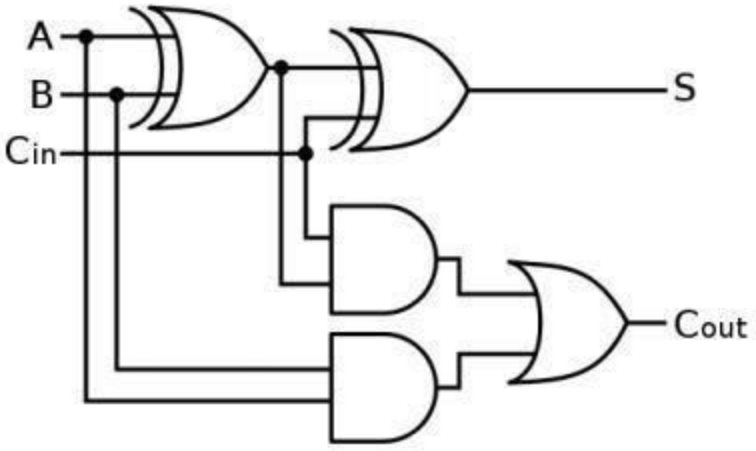
1. Theory of Operation

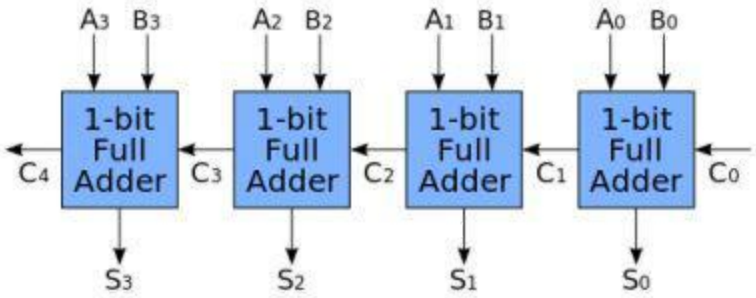
We created a 1-bit full-adder and use it to produce a 4-bit RIPPLE-ADDER. This is an extension to the prelab with the addition of VHDL and Vivado implementation. Finally a a testbench for the RIPPLE-ADDER is made to confirm it works according to specifications.

2. 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

3. Schematic Diagram





4. Design

a. VHDL Code

----------1-Bit_FULLADDER----------

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;



entity adder is     --declare adder
     Port (A, B, Cin: in std_logic;
          S, Cout: out std_logic );
end adder;

architecture Behavioral of adder is

       begin

          S <= (A XOR B) XOR Cin;
          Cout <= ((A XOR B) AND Cin) OR (A AND B);

end Behavioral;
```

----------Ripple Adder----------

```
--Notice that in this code. We are just connecting ports

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ripple_adder is
    Port (A, B: in std_logic_vector(3 downto 0); --This made up 4-bit Entity of "A" is to help us save the component
                    1-bit Entity of "A"
          Cin: in std_logic;
          Cout: out std_logic;
          S: out std_logic_vector(3 downto 0));
end ripple_adder;



architecture Behavioral of ripple_adder is

      signal C_out: std_logic_vector(3 downto 0) := (others => '0');          --Remember the semicolon at the end of a
                    temporary signal


      component adder             --declare component
           Port (A, B, Cin: in std_logic;
                 S, Cout: out std_logic );
      end component;


      begin
                --When working with STRUCTURAL DESIGN,it is just about the hierarchy of connecting
                    Entity-Component values to Main-Entity values (Regardless of logical order)
                 --Since this is not about order calculation, it does not matter if the Full Adder's calculation is done first,
                    before the values are transferred
                --Remember we assign something small into something bigger (Component-Entity to
                    Main-Entity)bookie216


         Adder_1 : adder port map(A => A(0),         --Here we pass the Ripple Adder's Entity 0th-bit "A" in the
                    Entity-Component "A"
                          B => B(0),         --Here we pass the Ripple Adder's Entity 0th-bit "B" in the Entity-Component
                    "B"
                          Cin => Cin,        --Remember this is  the Ripple Adder's "Cin" being put into the
                    Entity-Component "Cin"
                          Cout => C_out(0),      --Here we pass the Entity 4-bit "Cout" into the temporary 1-bit "Cout" for
                    the 0-bit position
                          S => S(0));        --This just means that the entity 4-bit "S" will be equal to temporary 0th bit "S"


         Adder_2 : adder port map(A => A(1),     --Notice here that we take the 1th bit of Entity "A"
                          B => B(1),
```

```
            Cin => C_out(0),   --This just connects he 0th-bit "Cout" of the Full-Adder#1 in the "Cin" of
        Full-Adder#2
            Cout => C_out(1),  --Here we saved the Entity "Cout" obtained from the Full-Adder calculation
        into temporary "1th bit Cout"
            S => S(1));

    Adder_3 : adder port map(A => A(2),
            B => B(2),
            Cin => C_out(1),
            Cout => C_out(2),  --Remember the "Cout" on the left is from the -Entity-component
            S => S(2));


    Adder_4 : adder port map(A => A(3),
            B => B(3),
            Cin => C_out(2),
            Cout => C_out(3),      --Here We assigned the entity-component of Full-Adder#4 into
        Ripple-Carry Entity's 3rd bit "C_out(3)
            S => S(3));


  Cout <= C_out(3);    --In this case since we dont need any more Full-Adders, we simply pass the temporary
            "Cout(3)" into the Main-Entity "Cout"


end Behavioral;
```

5. TEST

a. Testbench

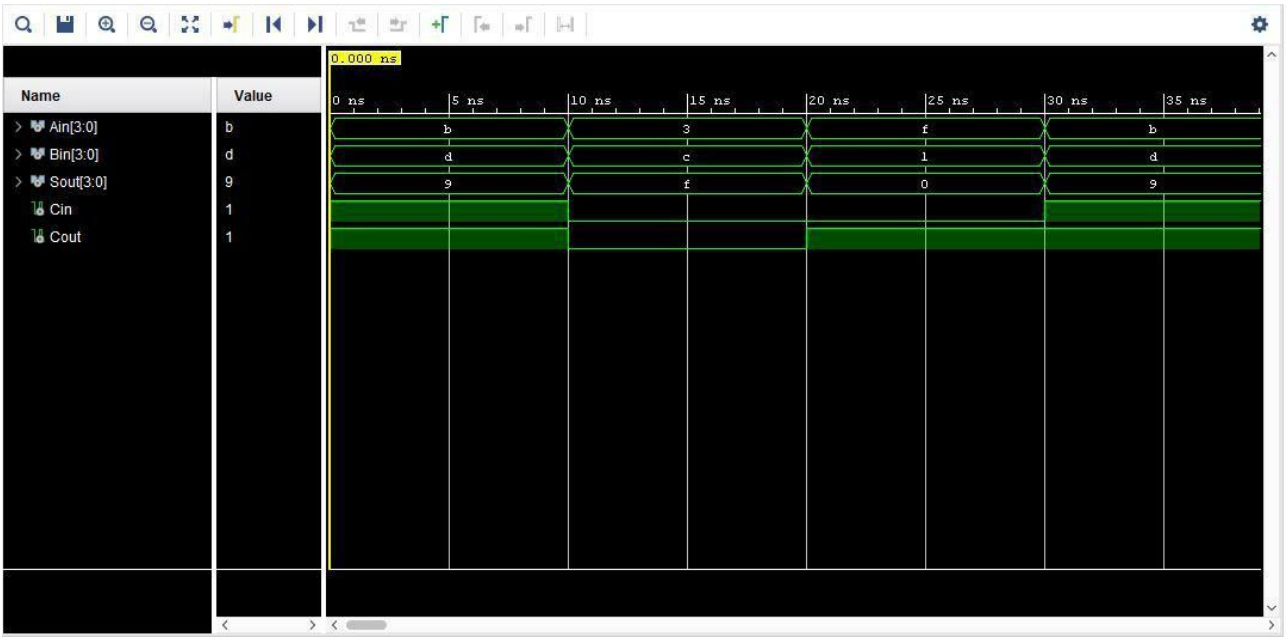
----------Ripple Adder (Testbench)----------

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ripple_TB is
--  Port ( );
end ripple_TB;


architecture Behavioral of ripple_TB is

    signal A, B, S : std_logic_vector(3 downto 0) := (others => '0');       --create intermediate signals/initialize them
    signal Cin, Cout : std_logic := '0';
```

```
--------------------------------------------------------------------------------

	component ripple_adder                          --declare ripple adder component
	     Port (
	          A, B: in std_logic_vector(3 downto 0);
	          Cin: std_logic;
	          Cout: out std_logic;
	          S: out std_logic_vector(3 downto 0));
	end component;
--------------------------------------------------------------------------------

	begin
	   process                                  --declare process for sim
	        begin
	             A <= "0011";
	             B <= "0100";
	             Cin <= '1';

	             wait for 1ms;   --note: expect S = "1000" = 8

	             Cin <= '0';
	             A <= "0101";
	             B <= "1010";

	             wait for 1ms;           --note: expect S = "1111" = 15 -> f


	             A <= "0111";
	             B <= "0001";

	             wait for 1ms;                  --note: expect S = "1000" = 8


	             A <= "1000";
	             B <= "0010";

	             wait for 1ms;          --note: expect S = "1010" = 10 -> a


	              A <= "1111";                      --lets try to overload
	              B <= "0011";

	              --note: expect S = 15 + 3 = 18 -> 2
	              --this happens because it cycles back through the set 0 to 15
	              -- carry out should be 1 at this point
	              wait for 1ms;

	             --do nothing/ reset
	             A <= "0000";
	             B <= "0000";
	             wait;

	        end process;

	        dut: ripple_adder port map(A => A,              --set signals from ENTITY-COMPONENT (Design) to
MAIN-TEMPORARY (Testbench)

	             B => B,
	             Cin => Cin,
	             Cout => Cout,
	             S => S);
```

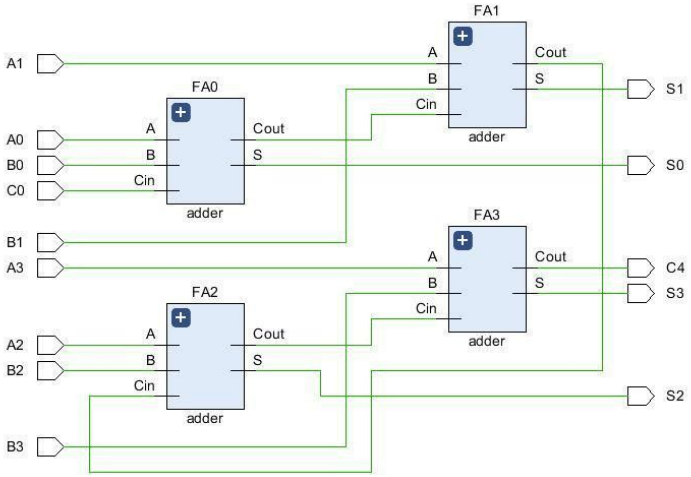
end Behavioral;

b. Simulation Results

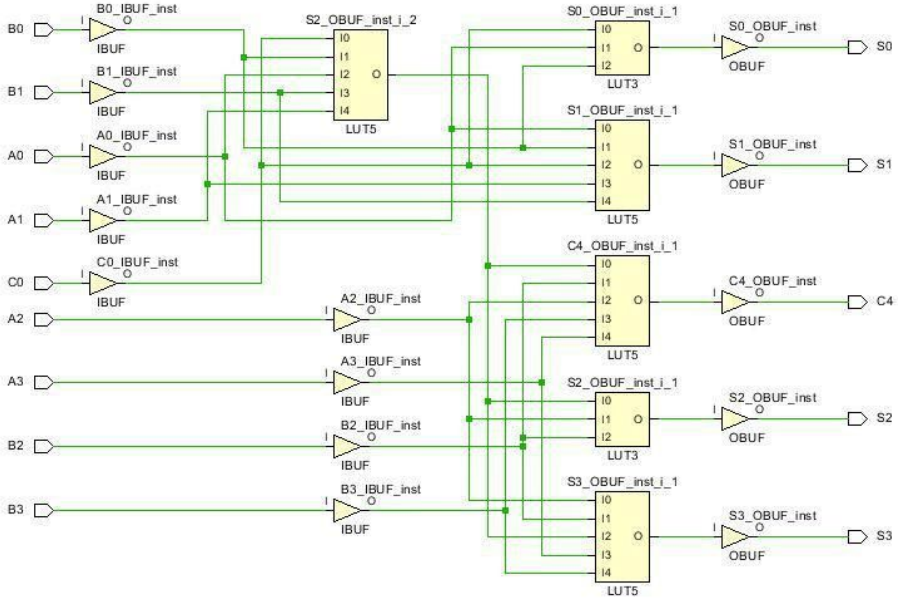


6. Implementation

a. Elaboration Schematic



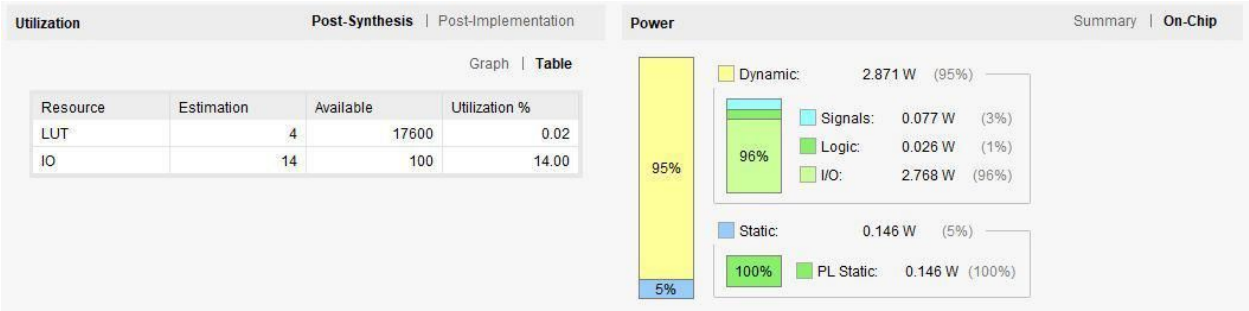
b. Synthesis Schematic



c. Project Summary Images

i. Post-Synthesis Utilization table

ii. On-chip Power Graphs



Part 2: Somebody did the work already

1. Theory of Operation

In this section, we create an ALU with 16 different functions. We feed two 4-bit inputs to it based on our Opcode,. It computes the desired function. Asserting button-0 loads the 4-bit switch-0 into input B. Pressing button-1 and button-2 does the same for input A and Opcode. Pressing button-4 resets the 3 inputs (Opcode, A and B) to “0000”. To make the alu functions work, some require concatenation and the use of type-casting such as the UNSIGNED-FUNCTION.

2. Design

a. VHDL Code

--**MY_ALU----------**

```
--Notice that in this code we use the BEHAVIORAL MODEL". This is why we call a "process" instead of "component-portmap"

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity my_alu is
      Port (A, B, Opcode: in std_logic_vector(3 downto 0);  --Remember that we were given an input template. We use this template
to make out 4-bt ALU
           output: out std_logic_vector(3 downto 0) );
end my_alu;


architecture Behavioral of my_alu is

      signal inputA, inputB, alu_out : unsigned(3 downto 0);  --Here we create three temporaries (intermediate signals). Remember
that when dealing with "temporary" signals we don't need to label them as "input" or "output"

         begin

               inputA <= unsigned(A);      --Here we put the entity values into the "temporary" values
               inputB <= unsigned(B);      --Notice, we don't use portmaps to assign signals in this case (We don't use portmaps
because this is not structural design)

               ALU:   process(inputA, inputB, Opcode)     --Here we use the new-temporary "inputA" and new-temporary "inputB" and
Entity "opcode"

                        begin   --Notice that we are creating a 4bit input multiplexer, where we manually enter its logic on the right

                        case Opcode is                        --In this case Entity "Opcode" is just the left 4-bit binary value on the left

                              when "0000" => alu_out <= inputA + inputB;      --This is the first output. We created a 4-bit artificial number
                              when "0001" => alu_out <= inputA - inputB;      --Notice in this case the Entity "opcode" is "0001". Also this
subtraction

                              when "0010" => alu_out <= inputA + "0001";       -- Notice that in this case we add a "1", but, it is not in
integer form, it is in binary form
                              when "0011" => alu_out <= inputA - "0001";      -- We add a "1" here in binary form as well
                              when "0100" => alu_out <= "0000" - inputA;     -- Subtracts "A" from 0. The "0" is in binary form

                              when "0101" => -- A > B comparison         --Note: sll, srl, sra are deemed as incorrect
                                    if(A > B) then                    --Checks if "A" is greater than "B"
                                       alu_out <= "0001";               -- Outputs an output of "1" in binary form
                                    else
                                       alu_out <= "0000";               -- Outputs an output of "0" in binary form
                                    end if;


                              -- they have been replaced by functions
                              -- unsigned performs logical shift
```

```
                        -- signed performs arithmetic shifts
                        when "0110" => alu_out <= shift_left(inputA, 1);              --Here we use a "shift left logical" special function
                        when "0111" => alu_out <= shift_right(inputA, 1);             --Here we use a "shift right logical" special
function
                        when "1000" => alu_out <= unsigned(shift_right(signed(inputA), 1));     --Here we use "right shift arithmetic"
special function but we add "unsigned" and "signed"

                        when "1001" => alu_out <= NOT inputA;                         --This just inverts "A"
                        when "1010" => alu_out <= inputA AND inputB;                  --This just checks to see if "A and B" is true,
then prints an output. Notice a shorcut "AND" function already exists
                        when "1011" => alu_out <= inputA OR inputB;                   --Same but with  "OR", where we have "A or B"
                        when "1100" => alu_out <= inputA XOR inputB;                  --Same but with "XOR", for "A" and "B"
                        when "1101" => alu_out <= inputA XNOR inputB;                 --Same but with "XNOR"
                        when "1110" => alu_out <= inputA NAND inputB;                 --Same with "NAND"
                        when "1111" => alu_out <= inputA NOR inputB;                  --SAME with "NOR"

                        when others => alu_out <= "0000";       --This is just a part of the multiplexer that always has to be
there.For everything else the output defaults to "0"
                    end case;


                    --output <= std_logic_vector(alu_out);          --this is the 4-bit temporary "alu_out" (we added a "std_logic_vector"
to it to change it from binary to 1-bit) being put into the entity "out"

            end process;

            output <= std_logic_vector(alu_out);        --This is the line just above  except that we put it outside the "Behavioral
process"


end Behavioral;
```

5. TEST

a. Testbench

----------ALU_TESTER (Testbench)----------

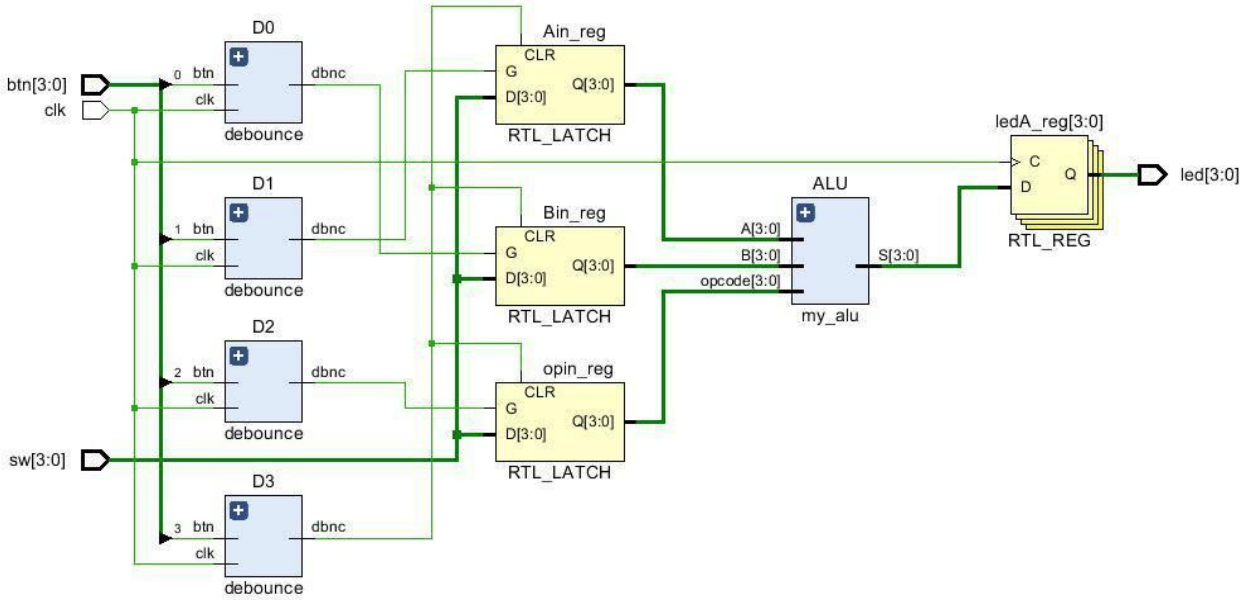
```
--Basically what we do here is we make a Main circuit, and we create a "structure model" then we use port-maps to connect four
single "ALU's"

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity alu_tester is
      Port ( sw, btn : in std_logic_vector(3 downto 0);       --The button values will be loaded into the "my_alu" component values for
each bit
           clk: in std_logic;                          --Notice we created a clock for the debounced buttons
           led : out std_logic_vector(3 downto 0));  --Notice these are four 1-bit outputs of this "Main"
end alu_tester;


architecture Behavioral of alu_tester is        --We will use structural behavior
```

```
        signal db_btn : std_logic_vector(3 downto 0);
        signal inA, inB, OP, led_out : std_logic_vector(3 downto 0);
        signal tempA, tempB, tempOP : std_logic_vector(3 downto 0);
        signal test : integer;


        component my_alu        --This calls the individual "my_alu"'s
          Port (A, B, Opcode: in std_logic_vector(3 downto 0);      --Notice that because we have 4-bit inputs and outputs we will call
"my_alu" four times
               output: out std_logic_vector(3 downto 0) );
        end component;

        component debounce          --This is the debounced clock
          Port (clk: in std_logic;
               btn: in std_logic;
               dbnc: out std_logic );
        end component;



        begin                --portmap

              process(clk)             --This is the process for making the switches work
                 begin

                 if(rising_edge(clk)) then          --on rising edge clock

                    if(db_btn(2) = '1') then        --check if logic true then, declare respective loads
                       tempOP <= sw;                --load opcode
                    end if;


                    if(db_btn(1) = '1') then         --loadA
                       tempA <= sw;                 --report "in A";
                    end if;


                    if(db_btn(0) = '1') then        --loadB
                       tempB <= sw;                  --report "in B";
                    end if;


                    if(db_btn(3) = '1') then           --reset values
                       tempA <= "0000";                --This just sets all the inputs to "0"
                       tempB <= "0000";
                       tempOP <= "0000";
                    end if;

                 end if;

              end process;


              inA <= tempA;       --Here we just connect a temporary "inA" into another temporary "tempA"
              inB <= tempB;
              OP <= tempOP;


               --opcode
               Button2: debounce port map(clk => clk,      --We just connect the component-entity with the corresponding temporary
signal (small to big)
                                  btn => btn(2),
                                  dbnc => db_btn(2));

               --A
               Button1: debounce port map(clk => clk,
```

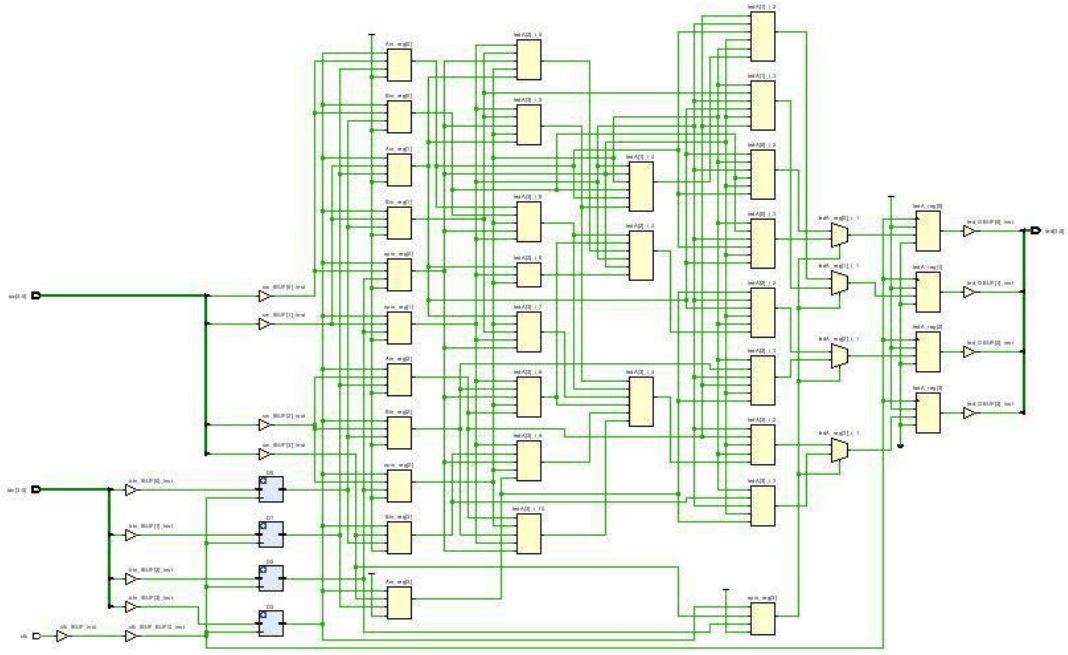
```
                btn => btn(1),
                dbnc => db_btn(1));

        --B
        Button0: debounce port map(clk => clk,
                btn => btn(0),
                dbnc => db_btn(0));


        --reset
        Button3: debounce port map(clk => clk,
                btn => btn(3),
                dbnc => db_btn(3));

        ALU: my_alu port map(A => inA,          --We just connect the component-entity with the corresponding temporary
signal (small to big)
                B => inB,
                Opcode => OP,
                output => led);

end Behavioral;
```

3. Implementation

b. Elaboration Schematic



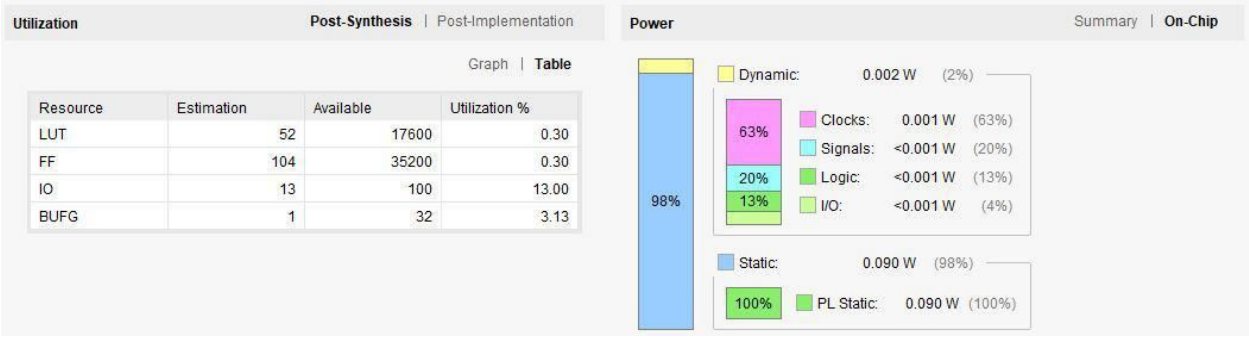
c. Synthesis Schematic



d. Project Summary Images

i. Post-Synthesis Utilization table

ii. On-chip Power Graphs



e. XDC File

The XDC file for this looks similar to that used in the previous lab for the counter_top. We only use the clock, switches, LEDs and the buttons.

```
##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 {04} [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=LE
```

Observations/Discoveries:

We learned to force a bit into simulation, allowing to see simulations in graphs without the need of a testbench. This was quite helpful since errors in the native file were not carried over to the testbench. Once sure the program worked as should, we began to work on the testbench.

Questions/Follow up:

1 What concepts do you fully understand?

It took awhile for the first portion of the lab to implement the ALU because a large amount of time was spent learning concatenation in VHDL. We had to make sure functions worked without typecasting. It was a trial and error process which nonetheless was quite helpful.

2 Any concepts you are unsure of?

Is it possible to choose certain vhdl codes to run and do rtl analysis on them without the need to create another project. Currently we made separate projects for the debounce and the counters in lab 1 and wondered if we could keep everything in one file instead and then synthesize the ones we need.