Course Name: Hardware and Software Design of Embedded Systems

Course Number and Section: **14:332:493**

Experiment: [Experiment #1 – Clocks, Counters, and Buttons]

Lab Instructor: Phil Southard

Date Performed: 2/14/19 – 2/28/19

Date Submitted: 2/28/19

Submitted by: Jonathan Boccardi – 159006465 – jmb779

----------For Lab Instructor Use ONLY----------

GRADE: _____

COMMENTS:

Purpose: The purpose of this lab is to construct a step-up, step-down counter by implementing its components piece-by-piece. The first step is to implement a clock divider to achieve a 2Hz pulse signal from 125MHz and checking it using an LED on the FPGA. Next, we want to make sure that when we press a button on the board, it is interpreted as only one press, and not many presses before the button stabilizes. This is the digital button debouncer portion of the project. Lastly, we want to use a counter to not only divide the input clock signal, but count up or down to the desired value.

Method:

Part 1: Designing a clock divider to achieve a 2Hz pulse from 125MHz.

In order to achieve this, we need to divide the input clock signal of 125MHz by a factor of 62.5M to achieve a 2Hz pulse. The process used to divide the input clock signal is by using a counter, which increments each clock cycle by 1, and when it reaches the full 62.5M, div is set to '1', the counter is reset, and div returns to 0 after one clock cycle. The VHDL code is shown here:

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.  use ieee.numeric_std.all;
4.  use ieee.std_logic_unsigned.all;
5.
6.  entity clock_div is
7.      port (
8.          clk  : in std_logic;
9.          div  : out std_logic
10.     );
11. end clock_div;
12.
13. architecture new_clock of clock_div is
14.     signal count : std_logic_vector(25 downto 0) := (others => '0'); --only needs 26 bits to
    store 62500000
15. begin
16.     divider: process(clk) begin
17.         if (rising_edge(clk)) then   --rising edge means 1 clock cycle
18.             count <= std_logic_vector(unsigned(count) + 1 );    --add a clock cycle
19.             if (unsigned(count) = 62500000) then             -- pulse a '1' when 500ms
    passes
20.                 div <= '1';
21.                 count <= std_logic_vector(to_unsigned(0,26));
22.             else                    --stay low
23.                 div <= '0';
24.             end if;
25.         end if;
26.     end process divider;
27. end new_clock;
```

clock_div.vhd

The RTL Diagram for the clock_div code shown above is in Figure 1.

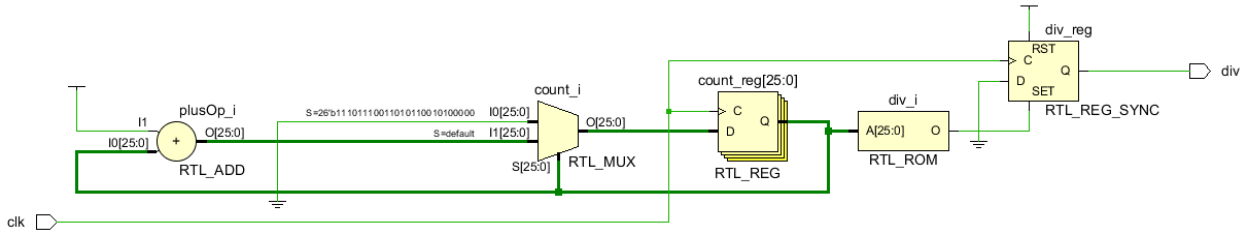

Figure 1: RTL Diagram for clock_div

A simulation run for two seconds is shown in Figure 2, and shows that the divided clock signal div pulses two times per second.

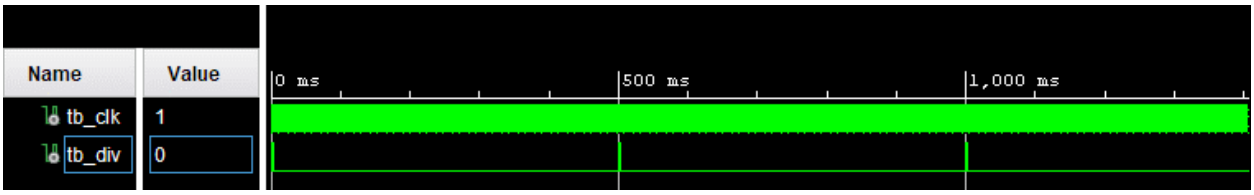

Figure 2: 2Hz pulse signal on div

Here is the testbench code used to produce these simulation results.

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.  entity clock_div_tb is
4.  end clock_div_tb;
5.
6.  architecture tb of clock_div_tb is
7.
8.      component clock_div
9.          port (
10.             clk  : in std_logic;
11.             div  : out std_logic
12.         );
13.     end component;
14.
15.     signal tb_clk : std_logic := '0';
16.     signal tb_div : std_logic := '0';
17.
18. begin
19.
20.     assign: clock_div port map (clk => tb_clk, div => tb_div);
21.     testbench: process begin        --one full clock cycle in 125MHz is 8ns
22.             tb_clk <= '0';
23.             wait for 4 ns;
24.             tb_clk <= '1';
25.             wait for 4 ns;
26.     end process;
27.
28. end tb;
```

clock_div_tb.vhd

Next is to test the divider with an LED on the FPGA. The controls for the LED are clock and the divided clock, which determines whether the LED will be on or off. The RTL diagram for this design is shown in Figure 3, and the code right after. We expect the LED to flip on or off every time div sends a pulse, changing its state two times per second, and completing one full on/off cycle once per second. The test bench code shows this behavior, and so did the testing done on the FPGA.

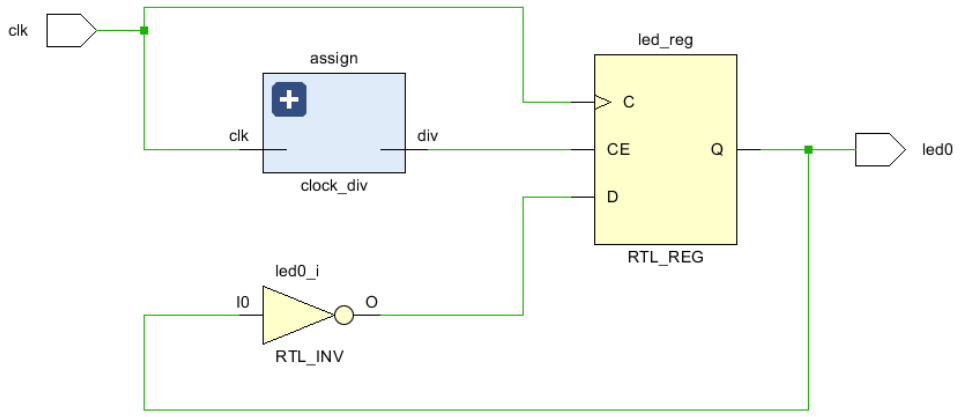

Figure 3: RTL Diagram for divider_top

A simulation run for two seconds using almost the same testbench as clock_div gives the desired result. This is shown in Figure 4.

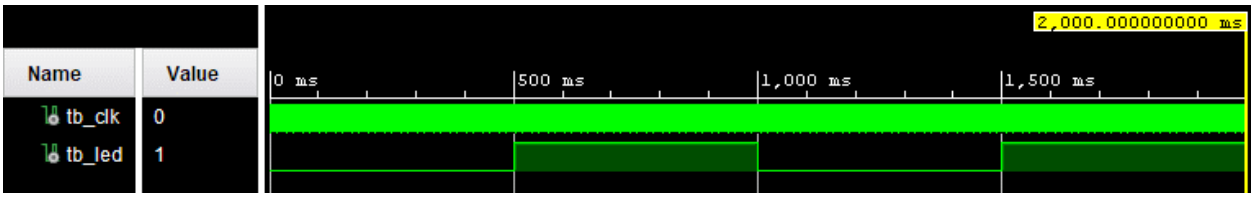

Figure 4: LED switching on and off at the desired rate

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.
4.  entity divider_top is
5.      port(
6.          clk : in std_logic;
7.          led0: out std_logic
8.      );
9.  end divider_top;
10.
11. architecture beh of divider_top is
12.
13.     component clock_div
14.         port (
15.             clk  : in std_logic;
16.             div  : out std_logic
17.         );
18.     end component;
19.
20.     signal led  : std_logic := '0';
21.     signal div  : std_logic;
22. begin
23.
24.     assign: clock_div port map(clk => clk, div => div);
25.
26.     led_reg: process(clk) begin
27.         if (rising_edge(clk)) then
28.             if (div = '1') then
29.                 led <= not led;
30.             end if;
31.        end if;
32.        led0 <= led;
33.     end process;
34.
35. end beh;
```

divider_top.vhd

Part 1 Discussion:

Q1.1: How much do we need to divide our input by to get from 125MHz to 2Hz?

A: 125MHz / 62.5M = 2 Hz

Q1.2: How many bits are required to store a counter that can count up to the value obtained in Q1.1?

A: 2^26 is about 67M, so 62.5M is 26 bits.

Part 2: Fixing Button Bounce

In order to debounce a button digitally, we want to keep track of the input values of the button, and check for a stabilized output. On the ZYNC we use for this lab, a button press is represented as a '1'. So, to debounce our button, we want to detect a certain number of consecutive '1's to verify that the button has been stabilized. Our design goal is to give the button 20ms to stabilize, or, at 125MHz, 2.5 million clock ticks. A steady stream of 2.5 million '1's will guarantee the button has a stable state.

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.  use IEEE.NUMERIC_STD.ALL;
4.
5.  entity debounce is
6.      port(
7.          clk : in std_logic;
8.          btn : in std_logic;
9.          dbnc: out std_logic
10.     );
11. end debounce;
12.
13. architecture db of debounce is
14.     signal count    : std_logic_vector(21 downto 0) := (others => '0');
15.     signal shift    : std_logic_vector(1 downto 0);
16. begin
17.
18.     process(clk, btn) begin
19.         if(rising_edge(clk)) then
20.             if(unsigned(count) < 2500000) then
21.             dbnc <= '0';
22.                 if(shift(1) = '1') then
23.                     count <= std_logic_vector(unsigned(count) + 1);
24.                 else
25.                     count <= (others => '0');
26.                 end if;
27.             else
28.                 dbnc <= '1';
29.                 if (btn = '0') then
30.                     dbnc <= '0';
31.                     count <= (others => '0');
32.                 end if;
33.             end if;
34.             shift(1) <= shift(0);
35.             shift(0) <= btn;
36.         end if;
37.     end process;
38. end db;
```

debounce.vhd

The RTL diagram for debounce is shown in Figure 5.

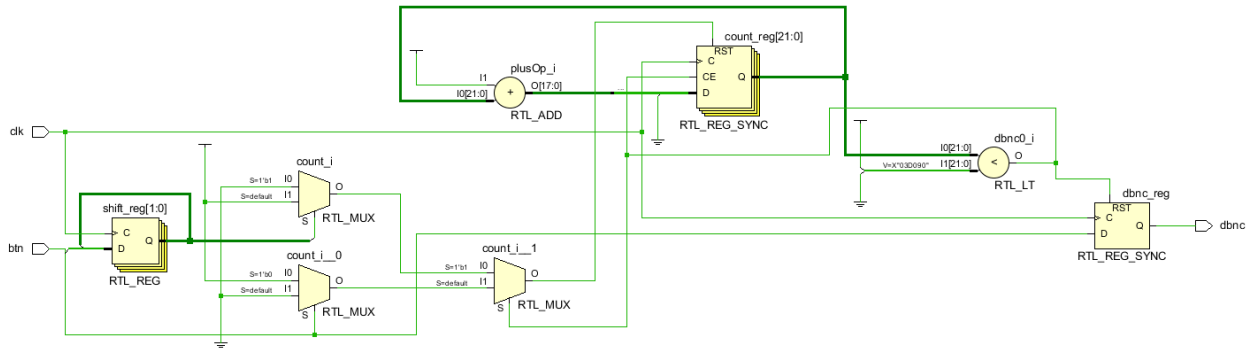

Figure 5: RTL Diagram for debounce

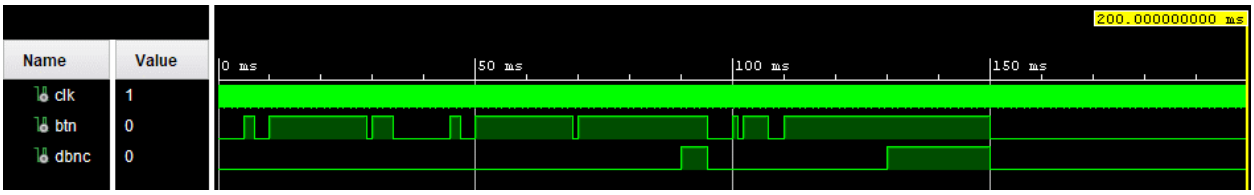

Figure 6: Simulation Results for debouncing

Figure 6 shows the debouncer working. dbnc is only high when btn has been pressed for 20ms or more. Below is the testbench code that produced this simulation result. The debouncer works as expected.

Part 2 Discussion:

Q2.1: What is the value of the button when it is pressed for the Zybo?

A: The button gives a value of 1 when pressed.

Q2.2: If it were the other value when pressed, would we have to alter our debounce design? Why or why not?

A: Yes, instead we would have to check for four consecutive 0's instead of 1's.

Q2.3: If we want our debounce time to be 20 ms, and our system clock is 125 MHz, how many ticks do we need a steady ‘1’ to be read for it to count as a ‘1’?)

A: 20 ms * 125 MHz = 2.5 million clock ticks for 20ms of debounce time.

Q2.4: How many bits are required for a counter that can go that high?

A: $\log_2(2.5M) = 21.25 \Longrightarrow 22$ bits

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

entity debounce_tb is
end debounce_tb;

architecture tb of debounce_tb is
    component debounce
        port (
            clk : in std_logic;
            btn : in std_logic;
            dbnc: out std_logic
            );
    end component;

    signal clk  : std_logic := '0';
    signal btn  : std_logic := '0';
    signal dbnc : std_logic := '0';
begin

assign: debounce port map(clk => clk, btn => btn, dbnc => dbnc);

clock: process begin
    clk <= '1';
    wait for 4 ns;
    clk <= '0';
    wait for 4 ns;
end process;

button: process begin
    --test 1
    btn <= '0';
    wait for 5ms;

    --bounce
    btn <= '1';
    wait for 2ms;
    btn <= '0';
    wait for 3ms;
    btn <= '1';
    wait for 19ms;
    btn <= '0';
    wait for 1ms;
    btn <= '1';
    wait for 4ms;
    btn <= '0';
    wait for 6ms
--40ms passed
    --test 2
    btn <= '0';
    wait for 5ms;

    --bounce
    btn <= '1';
    wait for 2ms;
    btn <= '0';
    wait for 3ms;
    btn <= '1';
    wait for 19ms;
    btn <= '0';
    wait for 1ms;
    btn <= '1';
    wait for 25ms; --on at 90ms

    --95ms passed

    --test 3
    btn <= '0';
    wait for 5ms;

    --bounce
    btn <= '1';
    wait for 1ms;
    btn <= '0';
    wait for 1ms;
    btn <= '1';
    wait for 5ms;
    btn <= '0';
    wait for 3ms;
    btn <= '1';
    wait for 40ms; --on at 130ms

    btn <= '0';
    wait;
end process;
end tb;
```

debounce_tb.vhd

Part 3: Fancy Counter

In this part, we implemented a 4-bit counter with the rules outlined in the lab manual. The behavior of the counter is as follows:

- Unless en is 1, nothing will change in the circuit.
- Even if en is 1, if clk en is 0 nothing can change the circuit except rst
- On the clock rising edge, when rst is asserted the cnt value will become 0.
- It can count either up or down depending on the value of a “direction” register, which is updated at the clock rising edge with the value present at dir when updn is 1.
- On the clock rising edge, if ld is 1, the value present at val will be loaded into the “value” register.
- If counting up, it will count until the number in a 4-bit “value” register has been reached, at which point it will roll over to 0000. If counting down, it will go from 0000 to value when it underflows

This behavior can be translated to an easier-to-understand form:

```
if (en = 1)
        if (rising edge clock)
                if (reset)
                        reset;
                elsif (clock enable)
                        if (ld = 1)
                                value <= val;
                        elsif (updn = 1)
                                direction <= dir;
                                if (dir = 1)
                                        cnt <= cnt + 1;
                                        if (cnt = val)
                                                cnt <= "0000";
                                else
                                        cnt <= cnt - 1;
                                        if (cnt = "0000")
                                                cnt <= val;
```

Pseudocode for fancy counter

Figure 7 shows fancy_counter’s RTL diagram.

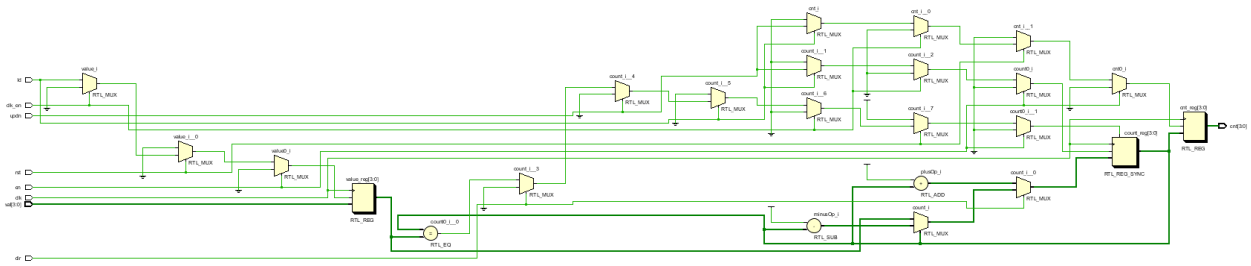

Figure 7: RTL Diagram for fancy_counter

Here is the VHDL implementation of the described behavior:

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.  use ieee.numeric_std.all;
4.
5.  entity fancy_counter is
6.      port (
7.          clk     : in std_logic;
8.          clk_en  : in std_logic;
9.          dir     : in std_logic;
10.         en      : in std_logic;
11.         ld      : in std_logic;
12.         rst     : in std_logic;
13.         updn    : in std_logic;
14.         val     : in std_logic_vector(3 downto 0);
15.         cnt     : out std_logic_vector(3 downto 0)
16.     );
17. end fancy_counter;
18.
19. architecture beh of fancy_counter is
20.     signal count    : std_logic_vector(3 downto 0) := (others => '0');
21.     signal value    : std_logic_vector(3 downto 0) := (others => '0');
22.     signal direction: std_logic := '0';
23. begin
24.     process(clk, clk_en) begin
25.         if (en = '1') then
26.             if (rising_edge(clk)) then
27.                 if (rst = '1') then
28.                     count <= (others => '0');
29.                 elsif (clk_en = '1') then
30.                     if (ld = '1') then
31.                         value <= val;
32.                     elsif (updn = '1') then
33.                         direction <= dir;
34.                         if (dir = '1') then
35.                             count <= std_logic_vector(unsigned(count) + 1);
36.                             cnt <= count;
37.                             if (unsigned(count) = unsigned(value)) then
38.                                 count <= (others => '0');
39.                                 cnt <= count;
40.                             end if;
41.                         else
42.                             count <= std_logic_vector(unsigned(count) - 1);
43.                             cnt <= count;
44.                             if (unsigned(count) = "0000") then
45.                                 count <= value;
46.                                 cnt <= count;
47.                             end if;
48.                         end if;
49.                     end if;
50.                 end if;
51.             end if;
52.         end if;
53.     end process;
54. end beh;
```

fancy_counter.vhd

Part 4: Combining clock dividers, debouncers, and counters for the final product.

Since all the parts work on their own, it makes sense to combine all of them to create a final working product. In this part, we need 4 buttons, 4 switches, 4 LEDs, and the clock. All four buttons will be debounced, and will control reset, enable, up/down, and load. The switches will control the 4-bit value we are counting up to or down from. The four LEDs show the current value we are at. All that needs to be done is create the components and map the ports, and the final product should work together. Figure 8 shows the RTL diagram for the final circuit, and the VHDL code is shown after.

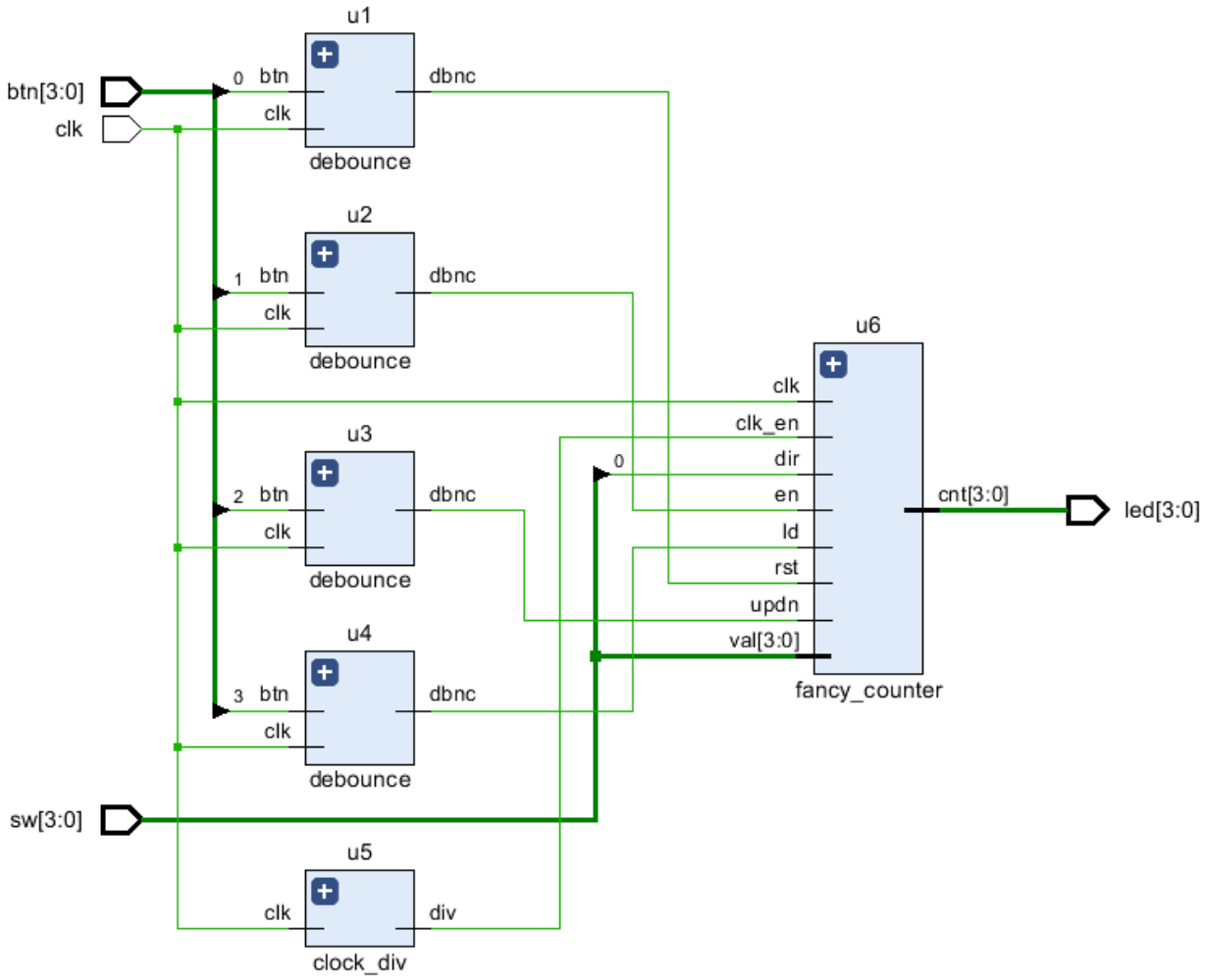

Figure 8: RTL Diagram for counter_top

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.
4.  entity counter_top is
5.      port (
6.          btn : in std_logic_vector(3 downto 0);
7.          clk : in std_logic;
8.          sw  : in std_logic_vector(3 downto 0);
9.          led : out std_logic_vector(3 downto 0)
10.     );
11. end counter_top;
12.
13. architecture ct of counter_top is
14.     component debounce
15.         port(
16.             clk : in std_logic;
17.             btn : in std_logic;
18.             dbnc: out std_logic
19.         );
20.     end component;
21.
22.     component clock_div
23.         port (
24.             clk  : in std_logic;
25.             div  : out std_logic
26.         );
27.     end component;
28.
29.     component fancy_counter
30.         port (
31.             clk     : in std_logic;
32.             clk_en  : in std_logic;
33.             dir     : in std_logic;
34.             en      : in std_logic;
35.             ld      : in std_logic;
36.             rst     : in std_logic;
37.             updn    : in std_logic;
38.             val     : in std_logic_vector(3 downto 0);
39.             cnt     : out std_logic_vector(3 downto 0)
40.         );
41.     end component;
42.
43.     signal db1      : std_logic := '0';
44.     signal db2      : std_logic := '0';
45.     signal db3      : std_logic := '0';
46.     signal db4      : std_logic := '0';
47.     signal slw_clk  : std_logic := '0';
48.
49. begin
50.
51.     u1: debounce port map( btn => btn(0), clk => clk, dbnc => db1);
52.     u2: debounce port map( btn => btn(1), clk => clk, dbnc => db2);
53.     u3: debounce port map( btn => btn(2), clk => clk, dbnc => db3);
54.     u4: debounce port map( btn => btn(3), clk => clk, dbnc => db4);
55.
56.     u5: clock_div port map( clk => clk, div => slw_clk);
57.
58.     u6: fancy_counter port map  (   clk     => clk,
59.                                     clk_en  => slw_clk,
60.                                     dir     => sw(0),
61.                                     en      => db2,
62.                                     ld      => db4,
63.                                     rst     => db1,
64.                                     updn    => db3,
65.                                     val     => sw,
66.                                     cnt     => led
67.                                 );
68. end ct;
```

counter_top.vhd

Implementation on final product:

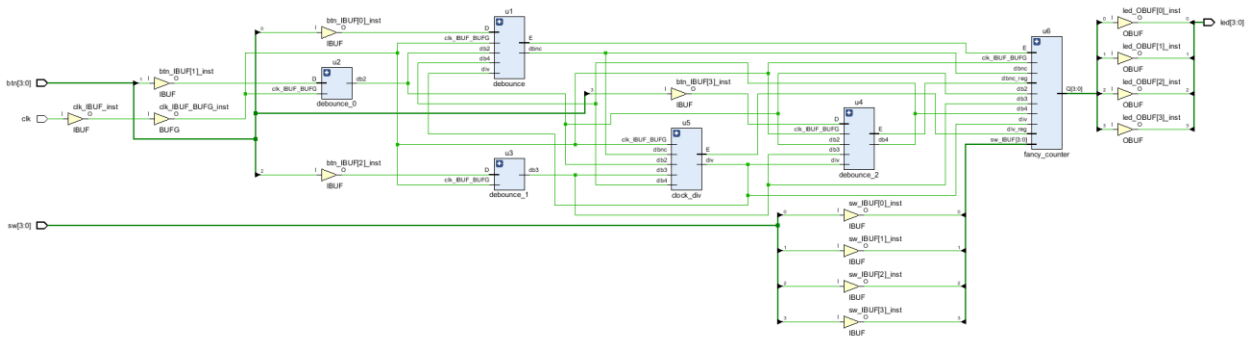

Figure 9: Synthesis and Implementation Schematic

For this design, the Synthesis and Implementation schematics were identical.

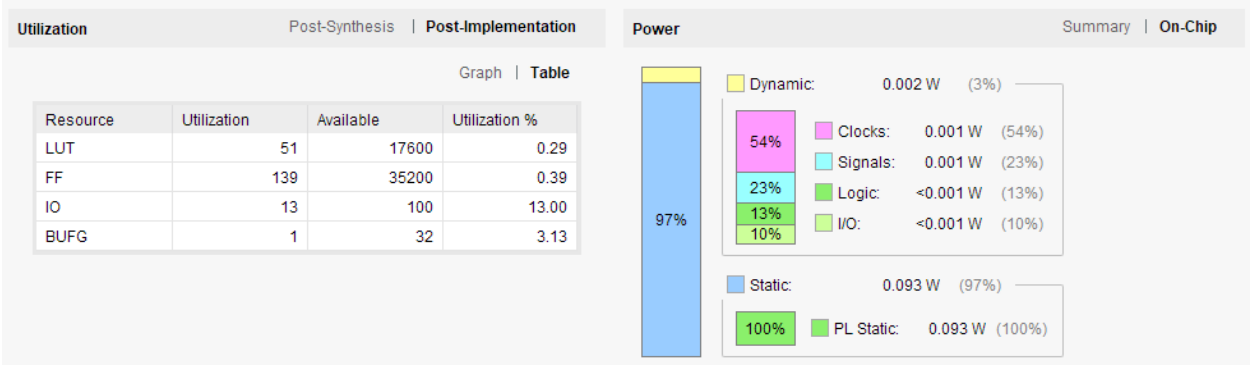

Figure 10: Utilization Table and Power Graphs

This design uses mostly IO on the chip, and doesn't draw much power at all.

Discussion (discussion questions are in their respective sections): I learned quite a lot from this lab, especially regarding the test-bench portions. Aside from gaining a complete understanding of clock dividers, digital debouncing, and this type of counter, I learned a lot about debugging VHDL code, as we had to work with our boards to confirm our designs worked. One thing I would make clearer in the lab manual is that the clock divider is creating a 2Hz pulse, not a clock. It may be my fault for interpreting it as a clock, but many others made the same error, so it may need small clarification, maybe a sample simulation showing the pulse for one clock cycle. Another thing I learned this lab is how to use entities that I already created in another place by creating a component and mapping the corresponding ports. One thing I'm still a bit confused about is why my simulations had uninitialized values and how to fix it. Oddly enough, when I brought my code to the computer lab, the simulation ran fine with no issues. It may have been a one-time thing, but next time I will try restarting Vivado to fix the issue.