Lab #1 Clocks, Counters, & Buttons

Hardware/Software Design of Embedded Systems

Iftidar Miah

28 February 2019

1. Purpose

   Design a counter to divide up the clock so we can implement multiple frequency clocks from one input clock by using a counter. Create a debounce circuit to stabilize analog button presses using shift registers and counters. Design an asynchronous bidirectional counter for counting purposes. Use component instantiation and structural modeling to combine all the parts into a 4-button, 4 switch, and clock inputs into debouncing circuits, a clock divider, and the divided clock, switches, and debounced buttons going the counter.

2. Part 1

   a. Theory

      Clock_div will divide a 125 MHz (T = 8ns) clock into a 2 Hz clock output call div. Div should act as a 2 Hz clk to be used as a clk enable for another device. Need to create a process with a loop that checks clock rising edge and increments a counter that counts up to a division ratio. When the counter reaches the division ratio of 62500000 after 0.5 s, div quickly pulses '1' for 8ns and the counter resets. Then div immediately resets to 0 until the counter reaches the division ratio again. Div becomes a 2 Hz output (T = 0.5 s) with only a T_{ON} = 8 ns, that can be used as an input clock enable in other components.

      We can make clock_div into a component and can instantiate/use it multiple times in another architecture. This requires reusing the entity and arch for clock_div then creating a new encompassing architecture called divider_top. Divider_top will use clock_div as one of the components inside. It takes the clk and the output div as a 2 Hz clk enable and puts them into a DFF. The output Q of the DFF will be inverted and fed back into D. Q also drives LED0. The result

should be that when the clk is on its rising edge, if the enable is on, then the output Q will invert and retain that state. Because div = clk enable and it will only be on for 0.8 ns every 0.5 s, the 125MHz (T = 8ns) clock will only have time for one rising edge in that time frame so multiple inversions can't take place back to back. LED0 will invert every 0.5 s.

b. Design

Clock_div

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity clock_div is
        port(   clk     : in    std_logic; -- 125 Mhz clock = 8ns per rising edge
        div     : out   std_logic); -- output clock enable
end clock_div;

architecture clock_div_arch of clock_div is
  signal counter : std_logic_vector(25 downto 0) := (others => '0');

begin
  count: process(clk)   -- process checks changes in clk
  begin
      if rising_edge(clk) then
          counter <= std_logic_vector(unsigned(counter) + 1);
          if (unsigned(counter) = 62500000) then
              div <= '1';
              counter <= (others => '0');
          else
              div <= '0';
          end if;
      end if;
  end process count;

end clock_div_arch;
```

Divider_top

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity divider_top is
  port( clk     : in    std_logic;
      led0  : out   std_logic);
end divider_top;
```

```
---------- Divider_top architecture ------------
architecture divider_top_arch of divider_top is
   signal D, div, C, CE       : std_logic;
   signal Q                   : std_logic := '0';

   component clock_div
      port(   clk     : in std_logic;
              div     : out std_logic);
   end component;

begin

   C <= clk;
   CE <= div;
   D <= (NOT Q);

   dff: process (C)
   begin
      if (CE = '1') then
        if rising_edge(C) then
          Q <= D;
        end if;
      end if;
   end process dff;

   led0 <= Q;

   U1: clock_div
      Port Map(   clk     => clk,
                  div     => div);

end divider_top_arch;

------------ Entity and arch, defines component clock_div instances ------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity clock_div is
   port(   clk     : in    std_logic;
           div     : out   std_logic);
end clock_div;

architecture clock_div_arch of clock_div is
   signal counter  : std_logic_vector(25 downto 0) := (others => '0');
begin
   count: process(clk)
   begin
      if rising_edge(clk) then
         counter <= std_logic_vector(unsigned(counter) + 1);
         if (unsigned(counter) = 62500000) then
            div <= '1';
            counter <= (others => '0');
```

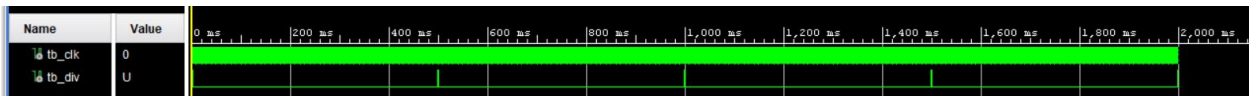
```
            else
                div <= '0';
            end if;
        end if;
    end process count;

end clock_div_arch;
```

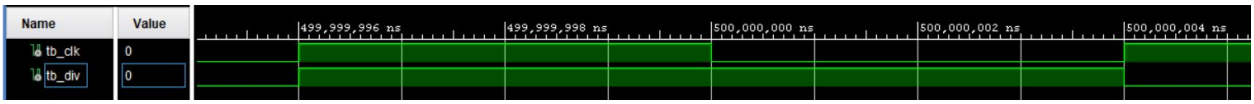
c. Test

Clock_div_tb

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

entity clock_div_tb is
end clock_div_tb;

architecture clock_div_tb_arch of clock_div_tb is
    signal tb_clk   : std_logic := '0';
    signal tb_div   : std_logic := '0';

    component clock_div is
    port(   clk          : in    std_logic;
            div     : out    std_logic);
    end component;

begin
    clk_proc: process
    begin
        wait for 4 ns;
        tb_clk <= '1';
        wait for 4 ns;
        tb_clk <= '0';
    end process clk_proc;

    dut: clock_div
    port map(
        clk => tb_clk,
        div => tb_div
    );

end clock_div_tb_arch;
```

Clock_div simulation

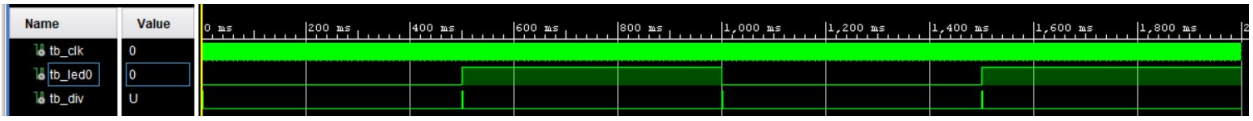




Divider_top_tb

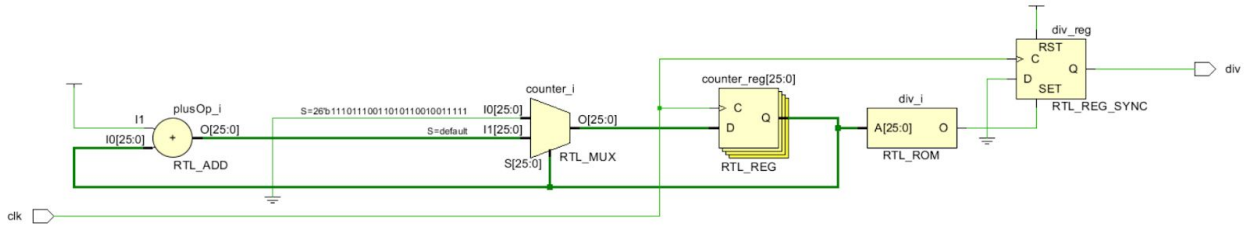
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity divider_top_tb is
end divider_top_tb;

architecture divider_top_tb_arch of divider_top_tb is
   signal tb_clk   : std_logic := '0';
   signal tb_led0   : std_logic := '0';
   signal tb_div   : std_logic;

   component divider_top is
   port(   clk          : in    std_logic;
          led0    : out    std_logic);
   end component;

   component clock_div is
   port(   clk     : in    std_logic;
          div     : out   std_logic);
   end component;

begin
   clk_proc: process
   begin
      wait for 4 ns;
      tb_clk <= '1';
      wait for 4 ns;
      tb_clk <= '0';
   end process clk_proc;

   dut1: divider_top
   port map(
      clk => tb_clk,
      led0 => tb_led0
   );

   dut2: clock_div
   port map(
      clk => tb_clk,
      div => tb_div
   );

end divider_top_tb_arch;
```

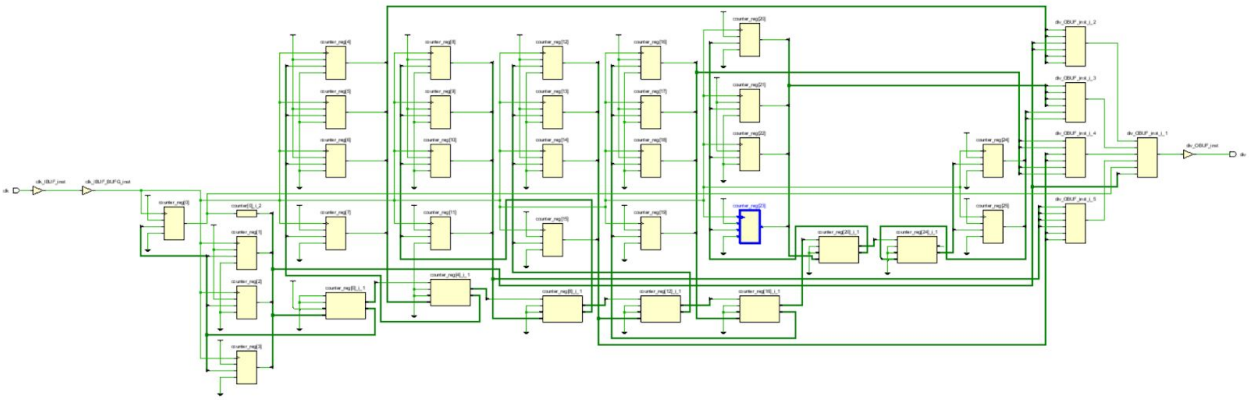
Divider_top simulation

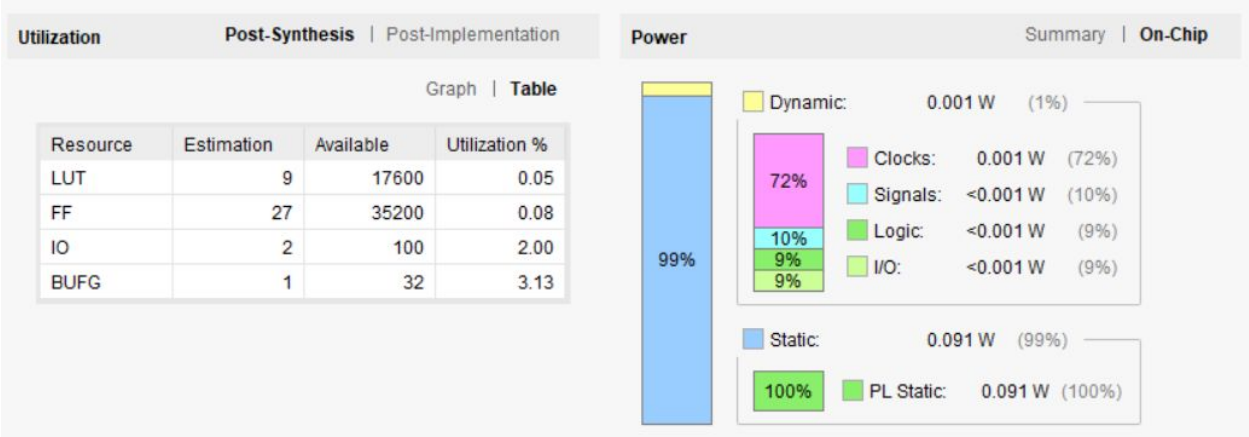


d. Implementation

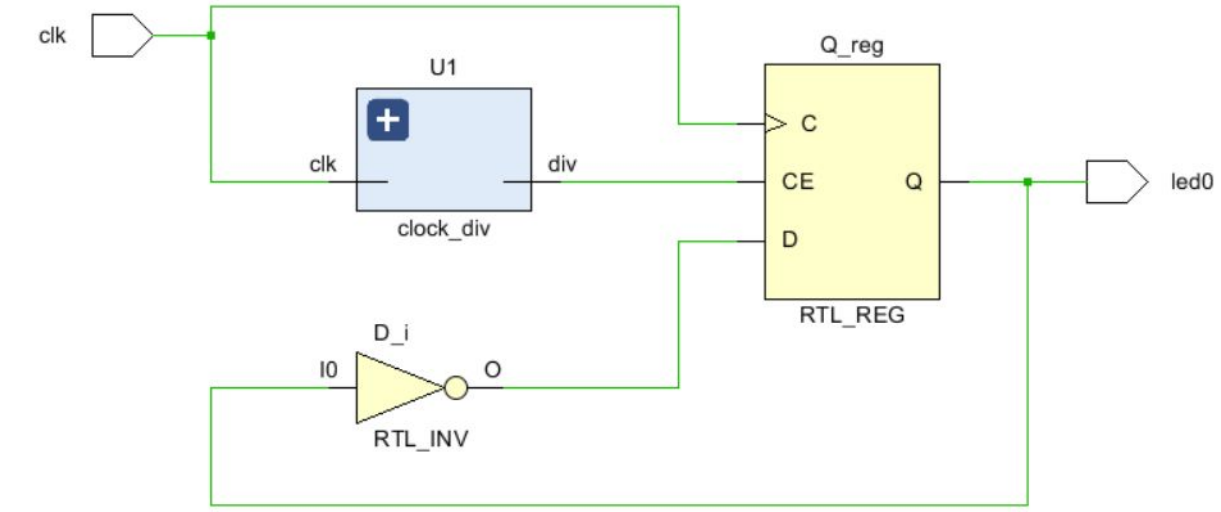
Clock_div

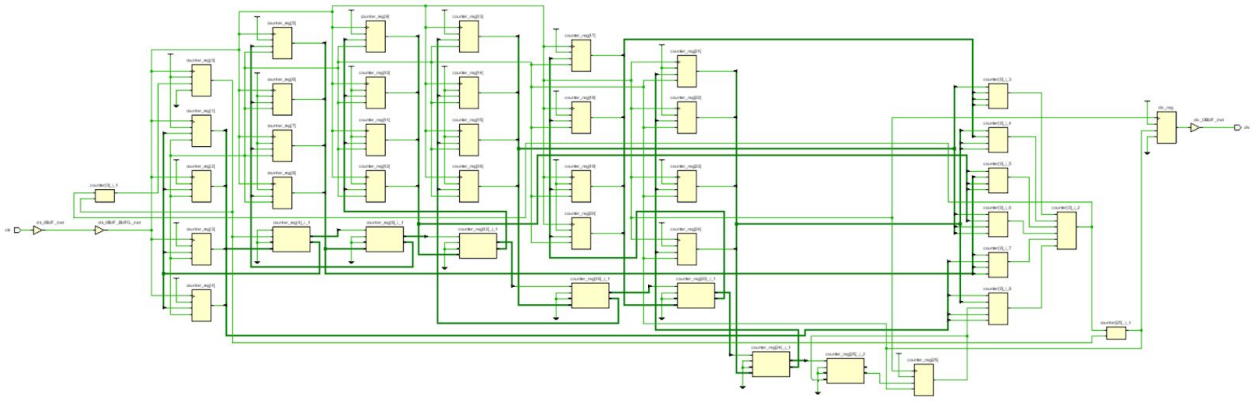






divider_top



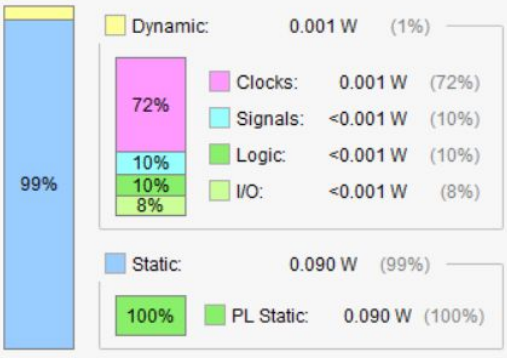


Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	10	17600	0.06
FF	28	35200	0.08
IO	2	100	2.00
BUFG	1	32	3.13

Power Summary | **On-Chip**



On the zybo_master.xdc, I only uncommented the lines for the clock and led0, and changed led[0] to led0.

3. Part 2

a. Theory

Push buttons can be shaky when pushed so we need to implement a digital debouncing circuit. We will sample the button with frequency equal to the clock and store it into a 2 bit shift register. Shift register[1] will hold the current value of Shift register[0], and Shift register[0] will get the new sampled value. We will implement a counter that checks Shift register[1] and if it's a 1, increment, else reset. The counter will count up to a designated number which indicates that that many consecutive samples were 1. When the counter is below that number, the debounce circuit will output 0. Once it hits the number, the output of the debounce circuit will be 1 and the counter stops incrementing. This will cause a button press to not immediately register but instead wait a few clock cycles for the button to stabilize and before registering that it was pressed.

b. Design

debounce

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity debounce is
   port(   clk, btn0    : in    std_logic;
           dbnc         : out   std_logic);
end debounce;

architecture debounce_arch of debounce is
   signal count    : std_logic_vector(2 downto 0) := (others => '0'); -- 3 bit counter signal
   signal shift    : std_logic_vector(1 downto 0) := (others => '0'); -- 2 shift register

begin

   debounce: process(clk)
   begin
      if rising_edge(clk) then
         shift <= shift(0) & btn0;    -- Shift left 1, put btn0 at shift(0)

         if ((shift(1) = '1') AND (unsigned(count) /= 4)) then
            count <= std_logic_vector(unsigned(count) + 1);
         elsif ((shift(1) = '1') AND (unsigned(count) = 4)) then
            dbnc <= '1';
```

```
            else
               dbnc    <= '0';
               count <= (others => '0');
            end if;
         end if;
      end process debounce;

   end debounce_arch;
```

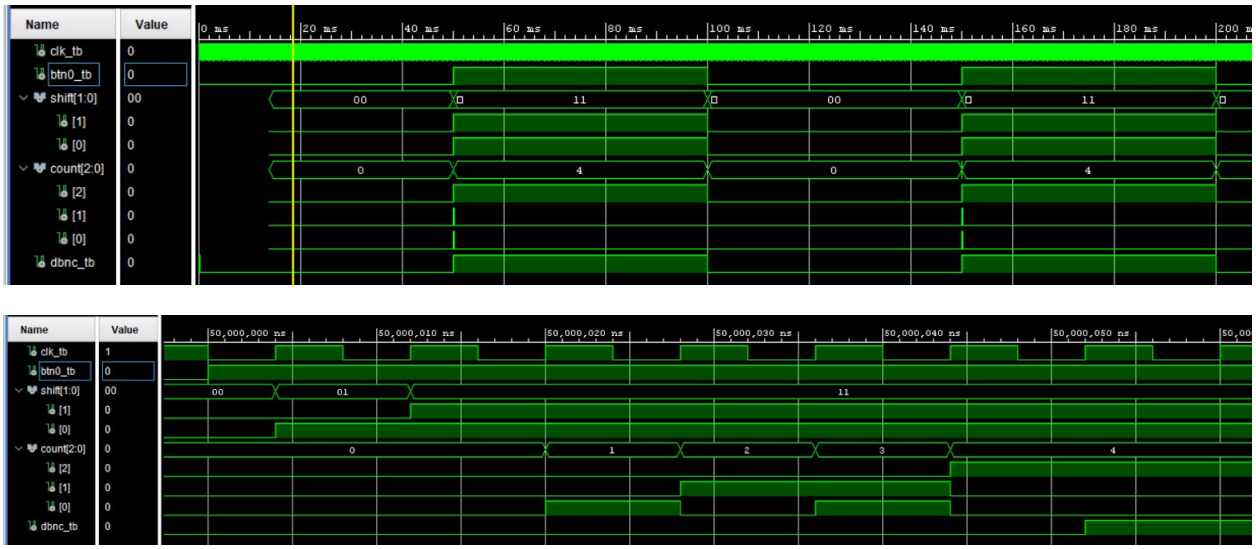
c. Test

debounce_tb

```
-- Lab 1 Iftidar Miah
-- Test bench for Lab1_part2_debounce.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity debounce_tb is
end debounce_tb;

architecture debounce_arch_tb of debounce_tb is
   signal clk_tb, btn0_tb, dbnc_tb : std_logic := '0';

   component debounce is
      port(   clk, btn0    : in    std_logic;
              dbnc       :  out   std_logic);
   end component;
begin

   clk_proc: process
   begin
      wait for 4ns;
      clk_tb <= '1';
      wait for 4ns;
      clk_tb <= '0';
   end process clk_proc;

   btn0_proc: process
   begin
      wait for 50 ms;
      btn0_tb <= '1';
      wait for 50 ms;
      btn0_tb <= '0';
   end process btn0_proc;

   dut: debounce
   port map(   clk     => clk_tb,
               btn0    => btn0_tb,
               dbnc    => dbnc_tb
```

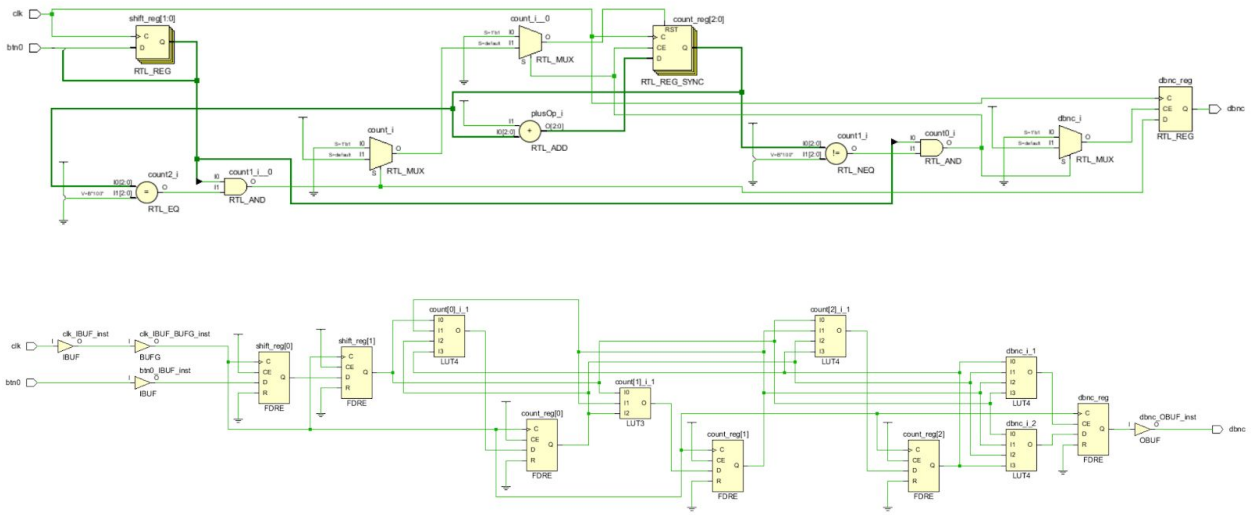
```
);

end debounce_arch_tb;
```

Debounce simulation



d. Implementation

debounce



Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	4	17600	0.02
FF	6	35200	0.02
IO	3	100	3.00
BUFG	1	32	3.13

Couldn't run implementation due to errors that I can't understand. It seems to have something to do with the IOs. Maybe I have to tie the output dbnc to an LED.

[Place 30-58] IO placement is infeasible. Number of unplaced terminals (1) is greater than number of available sites (0).
The following are banks with available pins:
IO Group: 0 with : SioStd: LVCMOS18 VCCO = 1.8 Termination: 0 TermDir: Out RangeId: 1 Drv: 12 has only 0 sites available on device, but needs 1 sites.
Term: dbnc

[Place 30-374] IO placer failed to find a solution
Below is the partial placement that can be analyzed to see if any constraint modifications will make the IO placement problem easier to solve.

IO Placement : Bank Stats							
Id	Pins	Terms	Standards	IDelayCtrls	VREF	VCCO	VR	DCI
0	0	0						
13	0	0						
34	50	1	LVCMOS33(1)			+3.30	YES	
35	50	1	LVCMOS33(1)			+3.30	YES	
	100	2						

IO Placement:

BankId	Terminal	Standard	Site	Pin	Attributes
34	btn0	LVCMOS33	IOB_X0Y9	R18	
35	clk	LVCMOS33	IOB_X0Y78	L16	

[Place 30-99] Placer failed with error: 'IO Clock Placer failed'
Please review all ERROR, CRITICAL WARNING, and WARNING messages during placement to understand the cause for failure.

[Common 17-69] Command failed: Placer could not place all instances

On the Zybo_Master.xdc, the clk and btn0 was uncommented and btn[0] was made into btn0.

4. Part 3
    a. Theory

    A synchronous bidirectional counter will only work if enabled. If enabled, but clock enable isn't on, it's only capable of resetting or doing nothing. When both enable and clock enable are on, it will increment or decrement depending on the value of a direction register. By default, I set my direction register to 0 so it decrements. A register is used to hold the direction we want to go and is only updated when an input updn = 1. A register is used to hold the value that the counter counts up to and is only updated when the input ld = 1. The counter will count up to that value then overflow back into 0000. It will count down from that value to 0000 and then underflow back into that value.

    b. Design

Fancy_counter

```
-- Lab 1 Iftidar Miah
-- Fancy Counter with enable, clock enable, reset, load, counts up or down with updn
input set by dir.
-- Counts up to 4 bit number in value register then reset to 0000. Count down to 0000
then load val input.

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity fancy_counter is
   port(   clk, clk_en, en, rst, dir, updn, ld : in    std_logic;
           val                                 : in    std_logic_vector(3 downto 0);
           cnt                                 : out   std_logic_vector(3 downto 0));
end fancy_counter;

architecture fancy_counter_arch of fancy_counter is
   signal dir_reg  : std_logic := '0';                              -- Registers for dir and val
   signal val_reg  : std_logic_vector(3 downto 0) := "1111";          -- Initialize to 1111
```

```
    signal counter    : std_logic_vector(3 downto 0) := "0101"; -- Intermediate signal for cnt
begin

    cnt <= counter;

    cnt_proc: process(clk)
    begin
        if rising_edge(clk) then
            if (en = '1') then
                if (rst = '1') then
                    counter <= (others => '0');
                else
                    if (clk_en = '1') then


                        if (dir_reg = '1') then
                            if (counter >= val_reg) then
                                counter <= (others => '0');
                            else
                                counter <= std_logic_vector(unsigned(counter) + 1);
                            end if;
                        else
                            if (counter = "0000") then
                                counter <= val_reg;
                            else
                                counter <= std_logic_vector(unsigned(counter) - 1);
                            end if;
                        end if;
                    end if;
                end if;
            end if;
    end process cnt_proc;

    dir_proc: process(clk)
        if (rising_edge(clk) AND en = '1' AND clk_en = '1' AND updn = '1') then
            dir_reg <= dir;
        end if;
    end process dir_proc;

    val_proc: process(clk)
        if (rising_edge(clk) AND en = '1' AND clk_en = '1' AND ld = '1') then
            val_reg <= val;
        end if;
    end process val_proc;

end fancy_counter_arch;
```

c. Test

Used multiple different test benches. Comment/Uncomment sections to be used.

Fancy_counter_tb

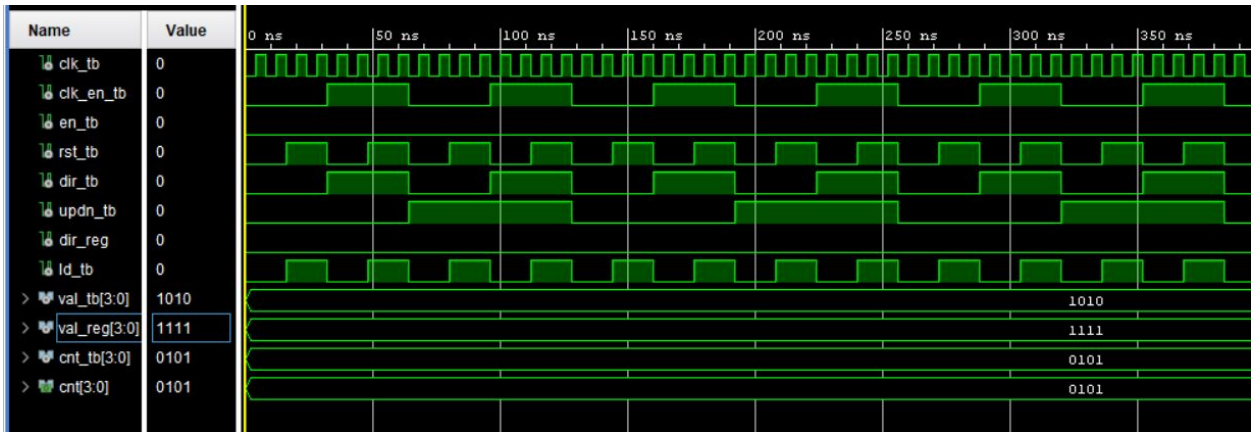
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity fancy_counter_tb is
end fancy_counter_tb;

architecture fancy_counter_arch_tb of fancy_counter_tb is
   signal clk_tb, clk_en_tb, en_tb, rst_tb, dir_tb, updn_tb, ld_tb :   std_logic := '0';
   signal val_tb  :    std_logic_vector(3 downto 0)    := "1010";
   signal cnt_tb  :    std_logic_vector(3 downto 0)    := "0000";

   component fancy_counter
         port(   clk, clk_en, en, rst, dir, updn, ld : in    std_logic;
                 val                                 : in    std_logic_vector(3 downto 0);
                 cnt                                 : out   std_logic_vector(3 downto 0));
   end component;
begin

   clk_proc: process   -- Generate 125 MHz clock
   begin
      wait for 4ns;
      clk_tb <= '1';
      wait for 4ns;
      clk_tb <= '0';
   end process clk_proc;

   ------- Multiple Test bench sections for different tests ------------
   ------- Uncomment sections to test different responses   ------------

   ---------------------------------------------------------------------------
   -- Test 1: Enable = 0, nothing should change
   -- Made default value of counter into 0101 to make sure reset works
   -- val_reg, dir_reg, and count should stay the same
   ---------------------------------------------------------------------------

--    clk_en_proc: process    -- Process for clock enable
--    begin
--       wait for 32ns;
--       clk_en_tb <= '1';
--       wait for 32ns;
--       clk_en_tb <= '0';
--    end process clk_en_proc;

--    rst_proc: process   -- Process for reset, affects counter and cnt
--    begin
--       wait for 16ns;
--       rst_tb <= '1';
--       wait for 16ns;
--       rst_tb <= '0';
--    end process rst_proc;

--    updn_proc: process  --Process for dupdn, affects when dir changes dir_reg
--    begin
```

```
--       wait for 64ns;
--       updn_tb <= '1';
--       wait for 64ns;
--       updn_tb <= '0';
--    end process updn_proc;

--    dir_proc: process   -- Process for dir, affects dir_reg and direction of counting
--    begin
--       wait for 32ns;
--       dir_tb <= '1';
--       wait for 32ns;
--       dir_tb <= '0';
--    end process dir_proc;

--    ld_proc: process    -- Process for ld and val, affects val_reg
--    begin
--       wait for 16ns;
--       ld_tb <= '1';
--       wait for 16ns;
--       ld_tb <= '0';
--    end process ld_proc;


    ---------------------------------------------------------------------------
    -- Test 2: clk_en = 0, only change will occur when En = 1 and rst = 1, cnt resets
    -- Made default value of counter into 0101 to make sure reset works
    -- val_reg, dir_reg, and count should stay the same
    ---------------------------------------------------------------------------

--    en_proc: process    -- Switches enable
--    begin
--       wait for 32ns;
--       en_tb <= '1';
--       wait for 32ns;
--       en_tb <= '0';
--    end process en_proc;

--    rst_proc: process   -- Process for reset
--    begin
--       wait for 16ns;
--       rst_tb <= '1';
--       wait for 16ns;
--       rst_tb <= '0';
--    end process rst_proc;

--    updn_proc: process  --Process for updn, affects when dir changes dir_reg
--    begin
--       wait for 64ns;
--       updn_tb <= '1';
--       wait for 64ns;
--       updn_tb <= '0';
--    end process updn_proc;

--    dir_proc: process   -- Process for dir, affects dir_reg and direction of counting
```

```
--    begin
--        wait for 32ns;
--        dir_tb <= '1';
--        wait for 32ns;
--        dir_tb <= '0';
--    end process dir_proc;

--    ld_proc: process    -- Process for ld and val, affects val_reg
--    begin
--        wait for 16ns;
--        ld_tb <= '1';
--        wait for 16ns;
--        ld_tb <= '0';
--    end process ld_proc;


    ---------------------------------------------------------------------------
    -- Test 3: clk_en = 1, en = 1, rst = 0, updn = 0 -> 1, dir = 0 -> 1, ld = 0
    -- val_reg stays at initial value of 1111
    -- dir_reg is 0 for half, 1 for other half of runtime
    -- Count will first decrement, then increment after dir_reg becomes 1
    -- Should underflow to 1111 and overflow to 0000
    ---------------------------------------------------------------------------

--        en_tb <= '1';
--        clk_en_tb <= '1';

--        updn_proc: process  --Process for updn, affects when dir changes dir_reg
--        begin
--            wait for 264ns;
--            updn_tb <= '1';
--            wait for 256ns;
--            updn_tb <= '0';
--            wait for 248ns;
--            updn_tb <= '1';
--            wait for 256ns;
--            updn_tb <= '0';
--        end process updn_proc;

--        dir_proc: process   -- Process for dir, affects dir_reg and direction of counting
--        begin
--            wait for 256ns;
--            dir_tb <= '1';
--            wait for 256ns;
--            dir_tb <= '0';
--        end process dir_proc;


---------------------------------------------------------------------------
    -- Test 4: clk_en = 1, en = 1, rst = 0, updn = 0 -> 1, dir = 0 -> 1, ld = 1
    -- val_reg goes from initial value of 1111 to val 1010
    -- dir_reg is 0 for half, 1 for other half of runtime
    -- Count will first decrement, then increment after dir_reg becomes 1
    -- Should underflow to 1010 and overflow at 1010 to 0000
```

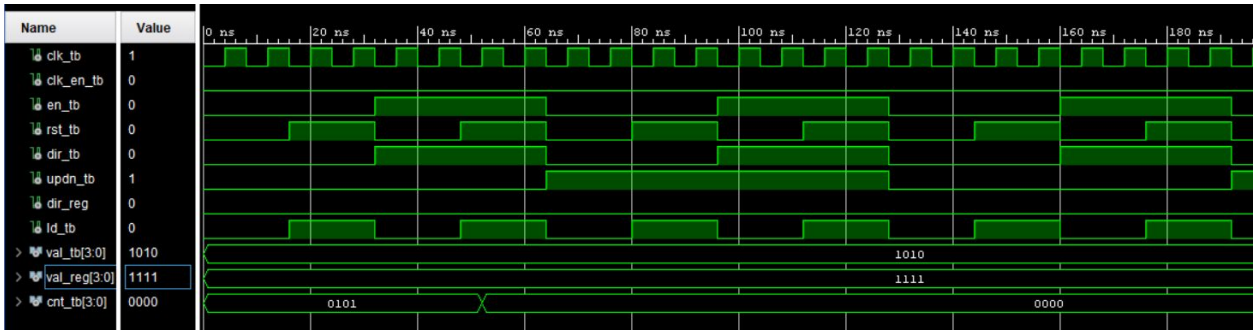
```
    ---------------------------------------------------------------------------

--      en_tb <= '1';
--      clk_en_tb <= '1';

--      updn_proc: process  --Process for updn, affects when dir changes dir_reg
--      begin
--         wait for 264ns;
--         updn_tb <= '1';
--         wait for 256ns;
--         updn_tb <= '0';
--         wait for 248ns;
--         updn_tb <= '1';
--         wait for 256ns;
--         updn_tb <= '0';
--      end process updn_proc;

--      dir_proc: process   -- Process for dir, affects dir_reg and direction of counting
--      begin
--         wait for 256ns;
--         dir_tb <= '1';
--         wait for 256ns;
--         dir_tb <= '0';
--      end process dir_proc;

--      ld_proc: process    -- Process for ld and val, affects val_reg
--      begin
--         wait for 8ns;
--         ld_tb <= '1';
--         wait for 8ns;
--         ld_tb <= '0';
--         wait for 496ns;
--      end process ld_proc;


---------------- Component Instantiation for test benching fancy_counter -----------------------
   dut: fancy_counter
   Port Map(   clk => clk_tb,
               clk_en => clk_en_tb,
               en => en_tb,
               rst => rst_tb,
               dir => dir_tb,
               updn => updn_tb,
               ld => ld_tb,
               val => val_tb,
               cnt => cnt_tb);

end fancy_counter_arch_tb;
```

Simulation, En = 0



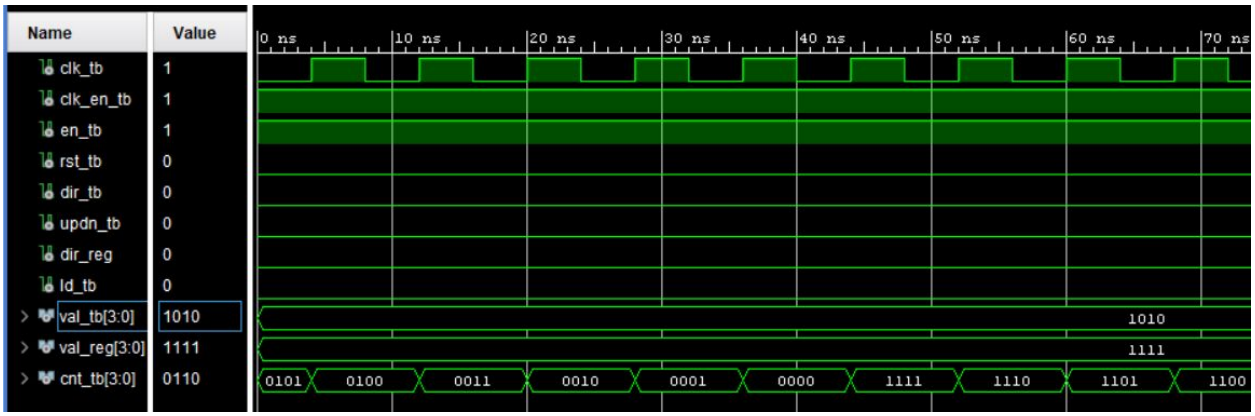
Enable is off so cnt retains initial value of 0101, the val_reg stays at 1111 and doesn't change to val = 1010 even when ld = 1. Dir_reg stays at 0 and doesn't change regardless of dir and updn.

Simulation, Clk_En = 0

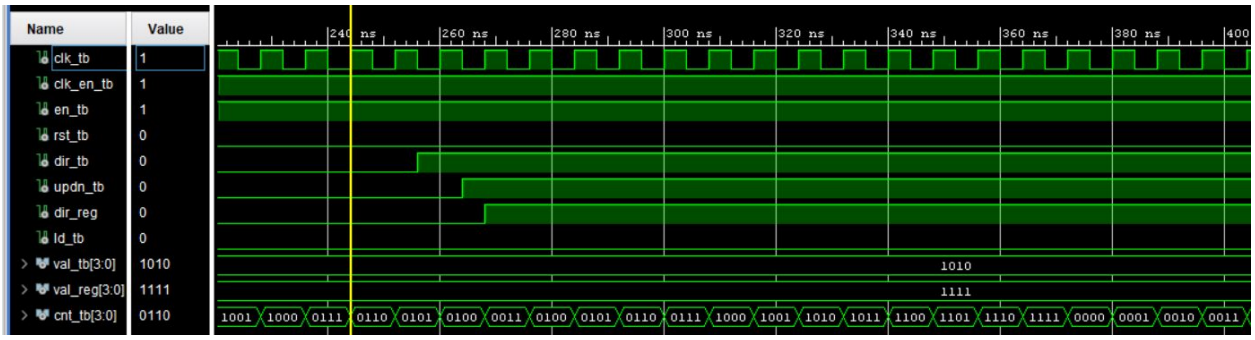


Clock enable is off so cnt resets from initial value of 0101 to 0000 when En=1 and RST=1, the val_reg stays at 1111 and doesn't change to val = 1010 even when ld = 1. Dir_reg stays at 0 and doesn't change regardless of dir and updn.

Simulation, Clk_En and En = 1, RST = 0, Alternating dir

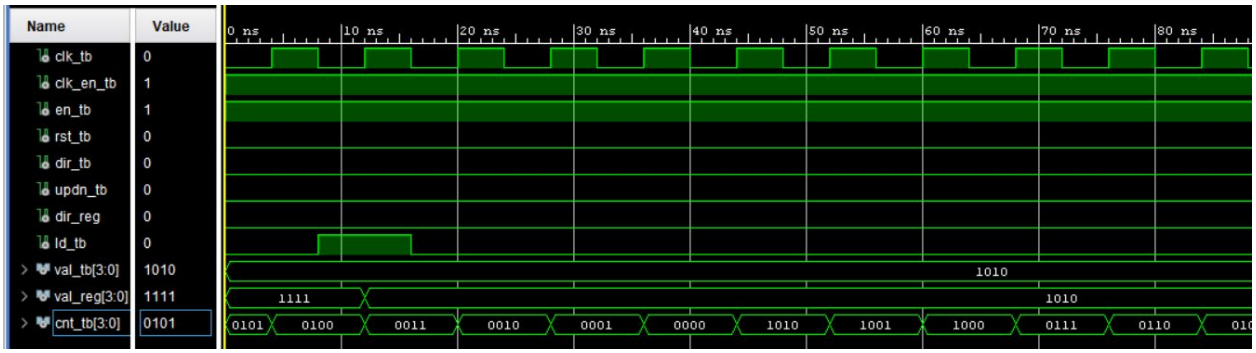


Starts at inital cnt = 0101, dir_reg = 0, val_reg = 1111 so counts down from 0101 to 0000 and underflows to 1111.

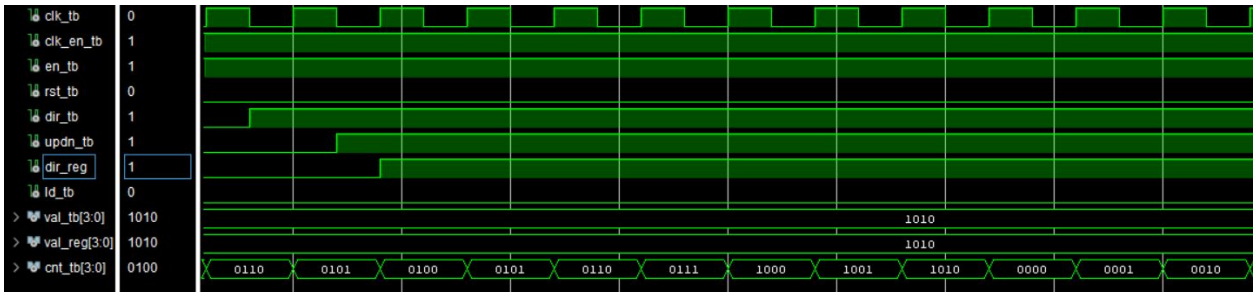


Dir becomes 1, then updn becomes 1, so at next rising edge, dir_reg becomes 1, and the rising edge after, the cnt starts to increment and overflows from 1111 to 0000.

Simulation, Load, alternating dir

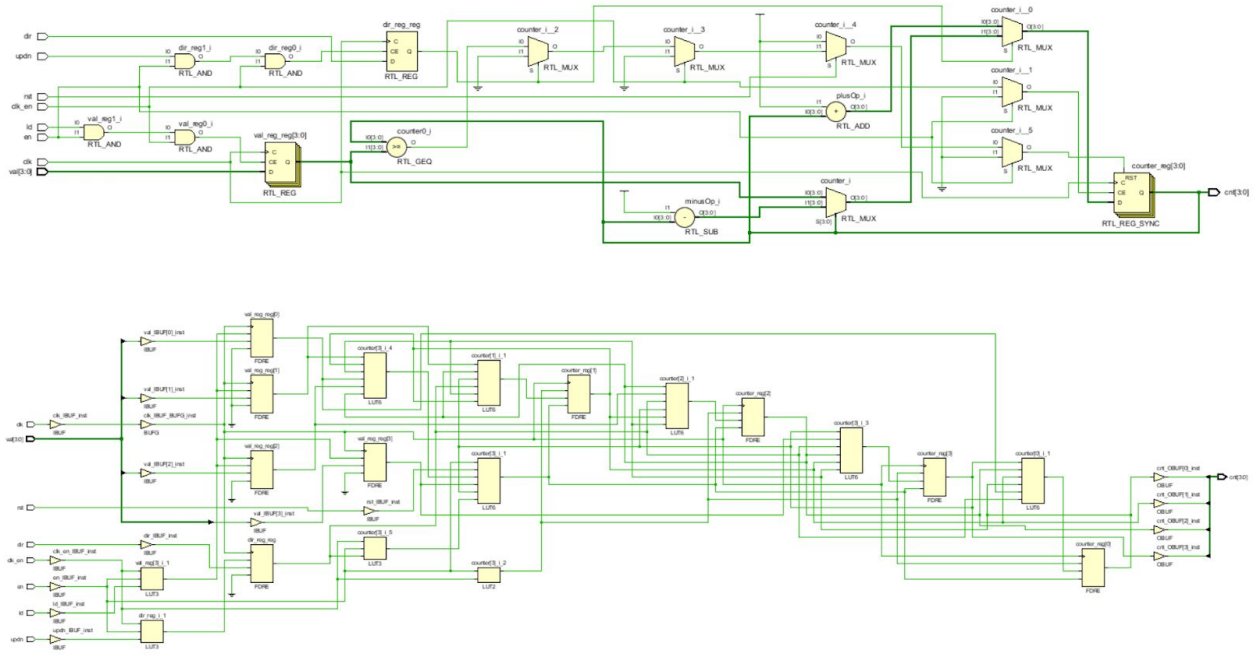


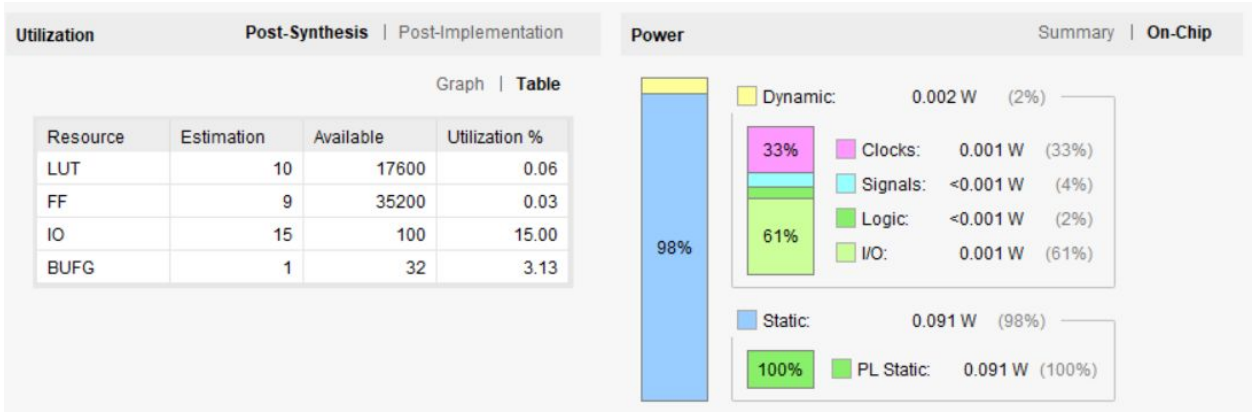
Val = 1010 and val_reg = 1111 initially. When ld = 1, changes val_reg = val = 1010. Since dir_reg = 0, we decrement and it counts down, underflows into 1010, and counts down again.



Once dir_reg = 1, starts incrementing. Because val_reg = 1010, we increment to 1010 then overflow to 0000.

d. Implementation





Only the clock was uncommented in the Zybo_Master.xdc.

5. Part 4
    a. Theory

        Use structural modeling to create 4 instances of debounce circuits for 4 separate button inputs that feed into the Reset, Enable, UPDN, and Load ports of a fancy counter. Make 1 clock divider instance to turn the 125 MHz clock into a 2 Hz divided clock to be pushed into the clock enable of the fancy counter. The clock port of the fancy counter takes the 125 MHz clock. The Dir takes switch 0 and value port is determined by all 4 switches. Button 1 will enable the fancy counter. The counter will count with every 8ns clock cycle but because the clock enable is 2 Hz (0.5s), it will be able to perform 62500000 instructions (increment, decrement) in 0.5 s, then wait 0.5 s before it can count again. Button 0 will reset the counter at any time as long as button 1 = en is pressed. Switch 0 will determine the direction while button 2 will tell when to change the direction register so we can change the actual counting direction. Button 3 turns on load which will load the value determined by all 4 switches into the value register so the counter will up to that value and underflow into that value.

    b. Design

        counter_top

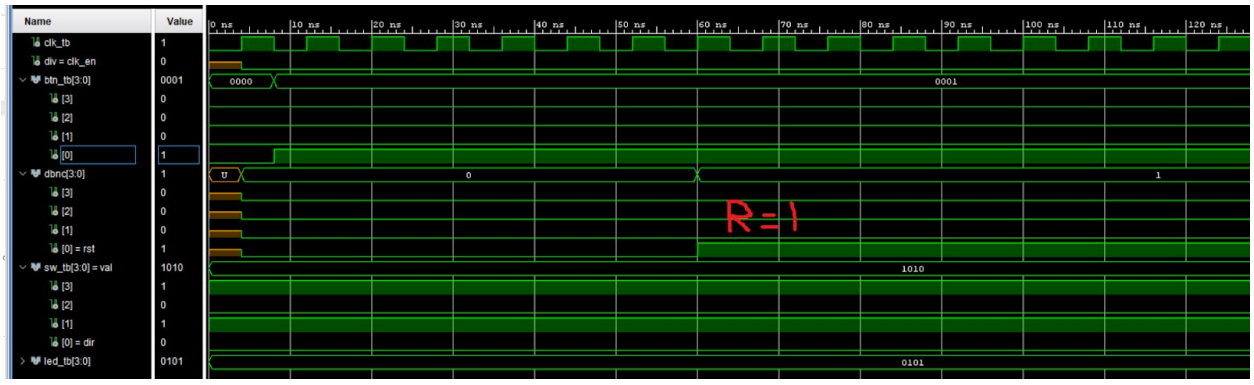
```
-- The entities and architectures for clock_div, debounce, and fancy_counter are the
same as above

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

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

architecture counter_top_arch of counter_top is
   signal dbnc     : std_logic_vector(3 downto 0);
```

```
    signal div      : std_logic;

    component clock_div
       port(   clk    : in    std_logic;
               div    : out   std_logic);
    end component;

     component debounce
       port(   clk, btn    : in    std_logic;
               dbnc        : out   std_logic);
    end component;

     component fancy_counter
       port(   clk, clk_en, en, rst, dir, updn, ld : in    std_logic;
               val                                 : in    std_logic_vector(3 downto 0);
               cnt                                 : out   std_logic_vector(3 downto 0));
    end component;

begin

    u1: debounce
       Port Map(   clk     => clk,
                   btn     => btn(0),
                   dbnc    => dbnc(0)
       );

    u2: debounce
       Port Map(   clk     => clk,
                   btn     => btn(1),
                   dbnc    => dbnc(1)
       );

    u3: debounce
       Port Map(   clk     => clk,
                   btn     => btn(2),
                   dbnc    => dbnc(2)
       );

    u4: debounce
       Port Map(   clk     => clk,
                   btn     => btn(3),
                   dbnc    => dbnc(3)
       );

    u5: clock_div
       Port Map(   clk     => clk,
                   div     => div
       );

    u6: fancy_counter
       Port Map(   clk     => clk,
                   clk_en  => div,
                   en      => dbnc(1),
                   rst     => dbnc(0),
```

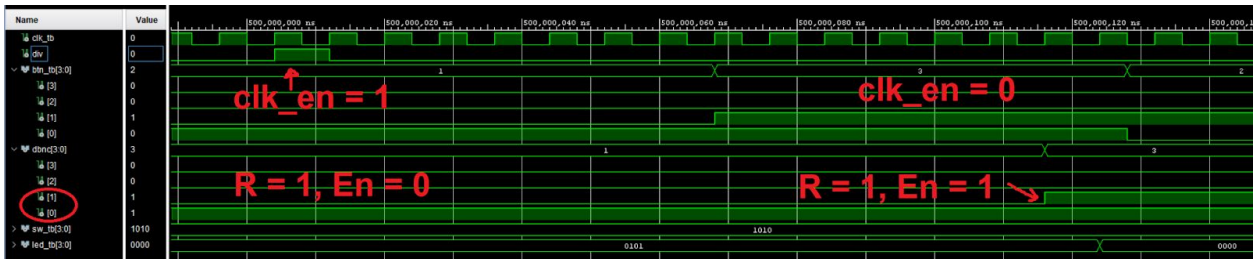
```
                dir     => sw(0),
                updn    => dbnc(2),
                ld      => dbnc(3),
                val     => sw,
                cnt     => led
        );

    end counter_top_arch;
```

c. Test

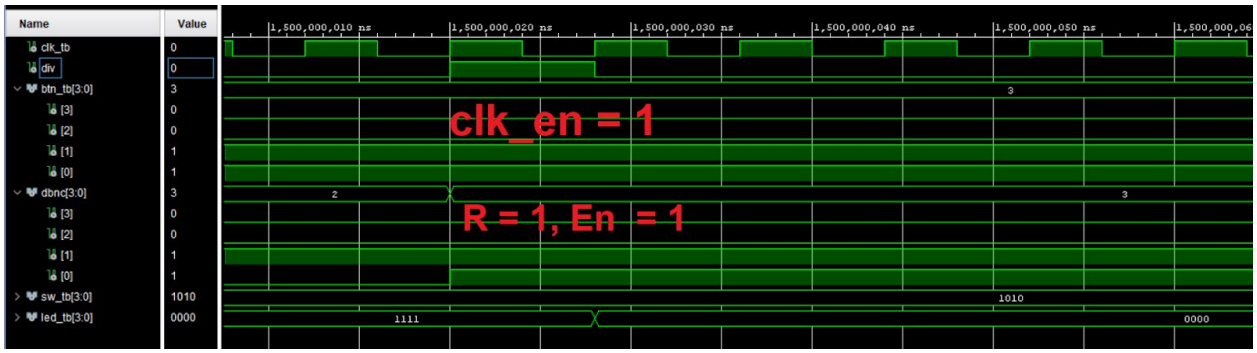
Simulation (Testing RST)



Btn(0) gets debounced and pushed to RST. RST = 1, CLK_EN = 0, EN = 0. Doesn't reset.

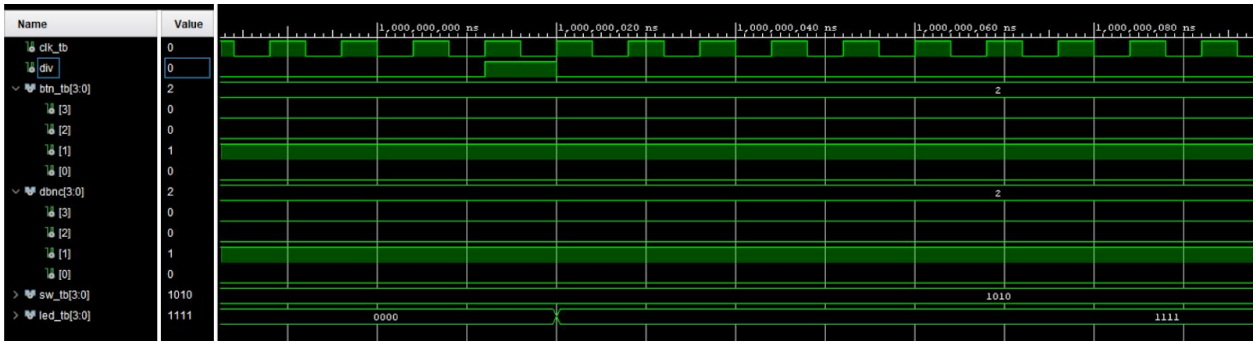


When R = 1, En = 0, clk_en = 1, doesn't reset. When R = 1, En = 1, and clk_en = 0, resets LED.

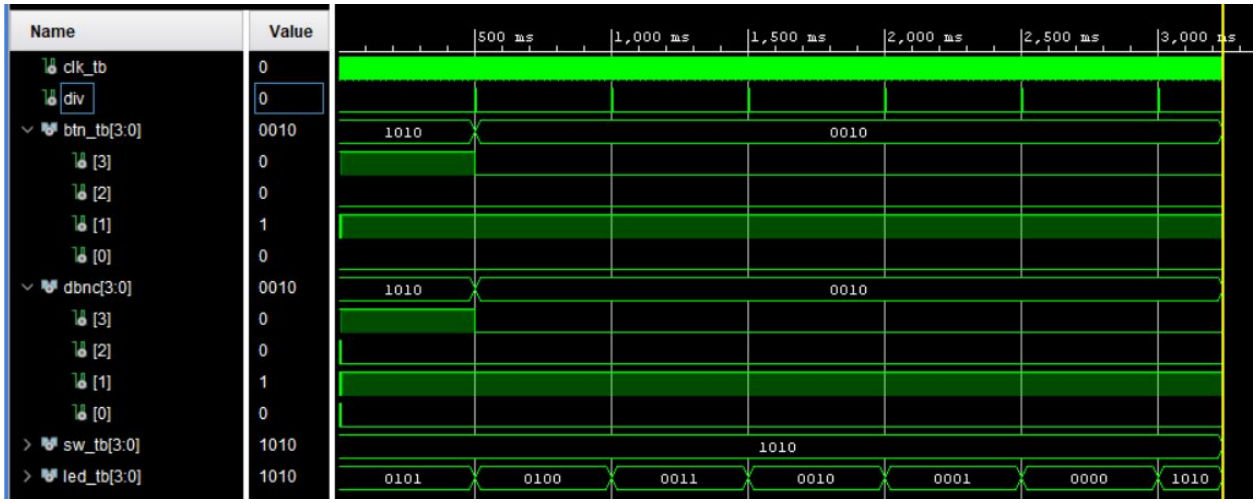


When R = 1, En = 1, clk_en = 1, resets.

Simulation (Counting)



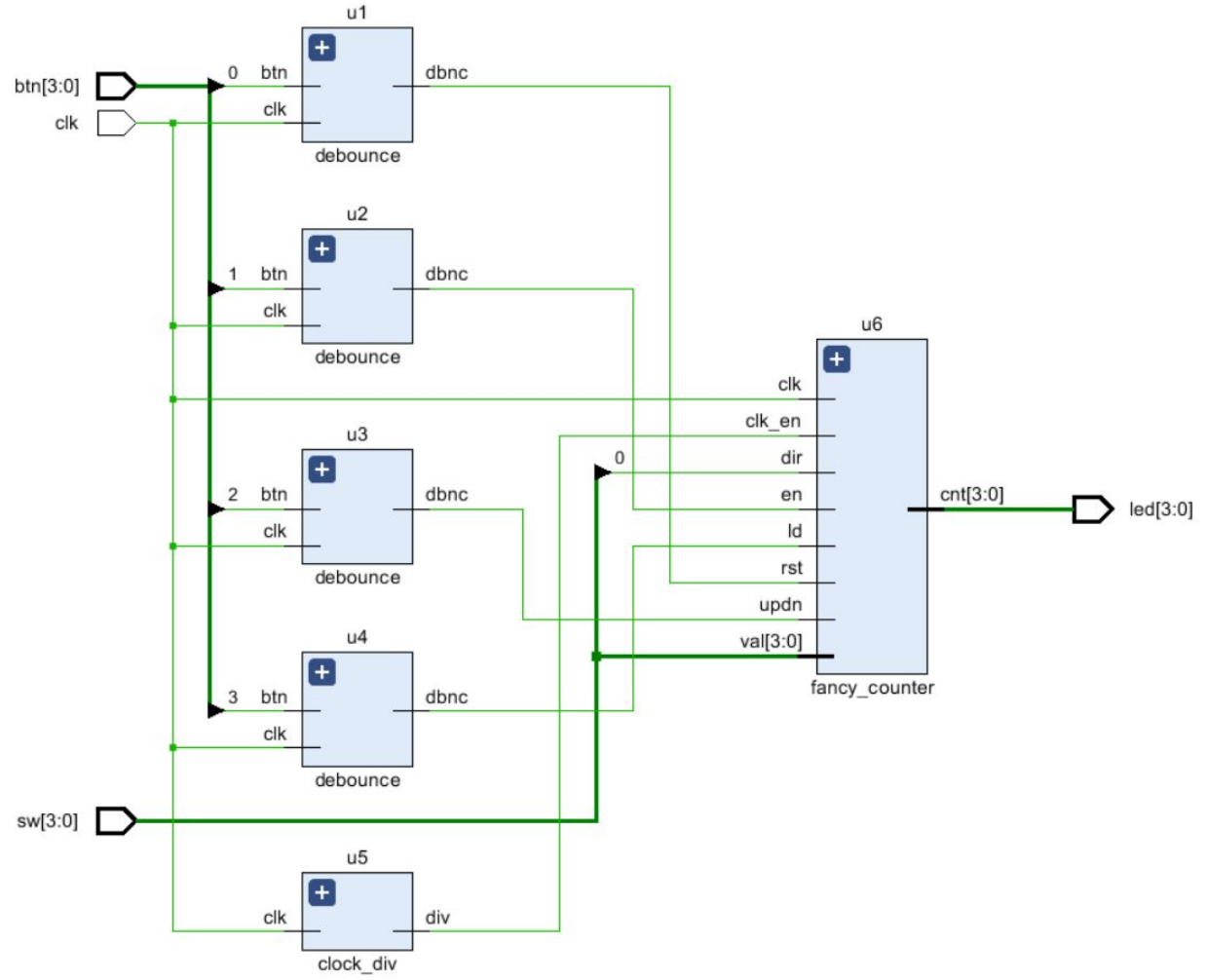
Will count down and underflow into value in value register (default is 1111).

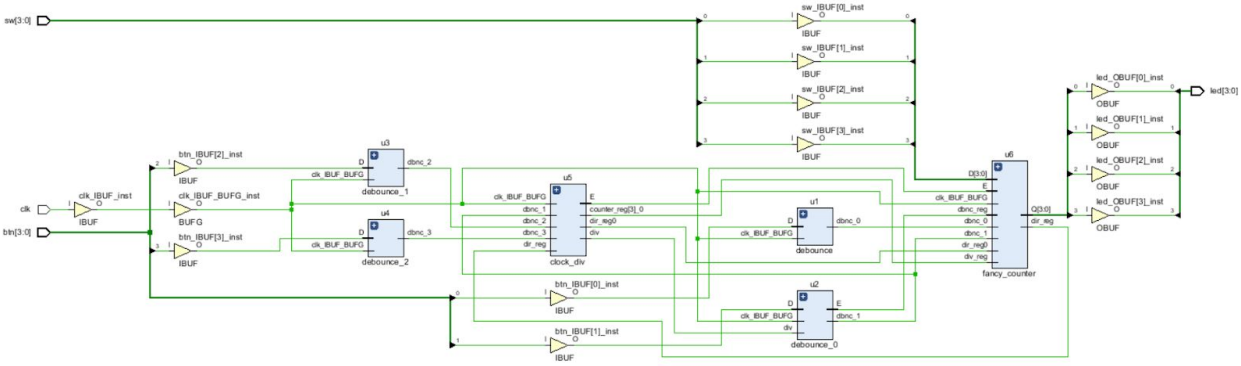


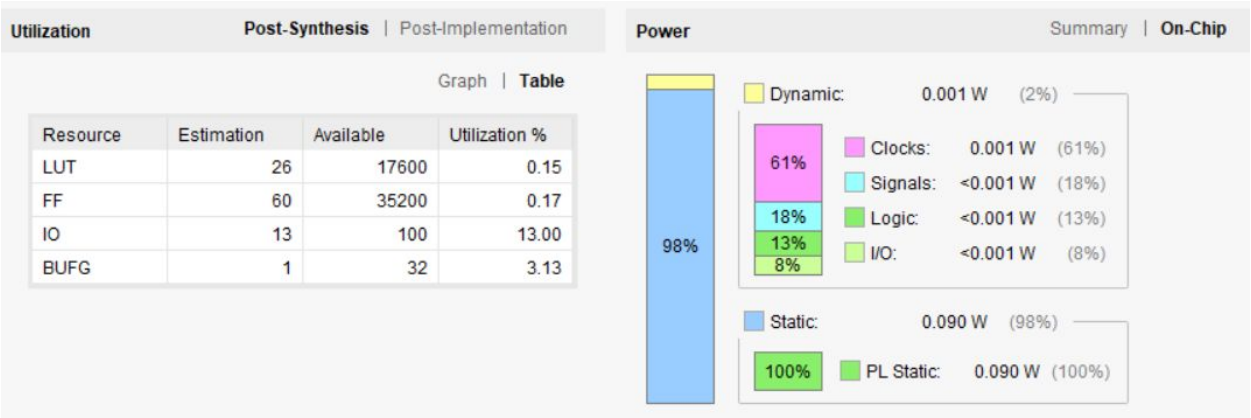
Successfully loaded 1010 into val_reg and counts down then underflows back to 1010.

Should also count up to 1010 then overflows back to 0000 but simulating takes too long because it only performs an action every 0.5s with the clk_en. Didn't have time to test this.

d. Implementation


u1
btn[3:0]
clk
0
btn
clk
dbnc
debounce
u2
1
btn
clk
dbnc
debounce
u3
2
btn
clk
dbnc
debounce
u4
3
btn
clk
dbnc
debounce
sw[3:0]
u5
clk
div
clock_div
u6
clk
clk_en
dir
0
en
ld
rst
updn
val[3:0]
cnt[3:0]
led[3:0]
fancy_counter


sw[3:0]
clk
btn[3:0]
IBUF
BUFG
OBUF
debounce
debounce_0
debounce_1
debounce_2
clock_div
fancy_counter
led[3:0]



On Zybo_Master.xdc, I uncommented the clock, the 4 switches, the 4 buttons, and the 4 leds.

6. Discussion
    a. Lab Questions

Question 1.1: How much do we need to divide our input by to get from 125 MHz to 2 Hz?

62500000

Question 1.2: How many bits are required to store a counter that can count up to the value obtained in Q1.1?

26 bits to count up to 67108863.

Question 2.1: What is the value of the button when it is pressed for the Zybo?

Didn't get to demo the board, and can't generate bitstream because implementation doesn't work.

Question 2.3: 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'?

Button is sampled every 8 ns with rising edge of clock, so 2500000 cycles.

Question 2.4: How many bits are required for a counter that can go that high?

22 bits

b. Observations/Discoveries

Processes run sequentially. Most processes should be synchronous with the clock, and a counter should be used to divide the clock into smaller frequencies instead of implementing multiple clocks. Structural models let me re-use the same code easily by just designing a component and creating as many instances as I please, then mapping ports to wires/signals. Debouncing circuits can use a counter to stabilize button presses. The amount of time a button has to be pressed to be registered as pushed in can be dictated by the max value of the counter before it resets. The fancy counter has multiple inputs which assert different conditions and increase the complexity of the code. Making multiple synchronous processes instead of cramming it all into one simplifies it a little bit and might help with timing. The debounce circuit also seems to take more cycles than I wanted to output dbnc = 1 when I was testing the counter_top. Slow clocks are painful to simulate.

c. Questions/Follow up

The way a counter can be used to divide clocks for other purposes makes sense. I understand the concept behind structural modeling, and even noticed how test benching works with it. I just need to practice to get the hang of it and get a better feel for the overall picture. Debouncing by using a counter makes sense since we want to make sure the button is stable.

Should I always give test bench signals an initial value or can I leave it as unknown? I also noticed that during my debounce simulation, I set up a synchronous process, and it was running the lines sequentially, but it looked like it was waiting until the next clock rising edge to execute the next sequential statement. Shouldn’t sequential statements try to complete every statement before the clock period is over?