ENGIN 341 – Advanced Digital Design

Engineering Department

University of Massachusetts Boston

Semester – Spring 2021

Instructor – Dr. Michael Rahaim

Lab Report for

Lab #3: Sequential BCD Counter

By

Name: Tyler McKean

Student ID: 01098154

Date: 4/11/2021

I pledge to uphold the governing principles of the Code of Student Conduct of the University of Massachusetts Boston. I will refrain from any form of academic dishonesty or deception, cheating, and plagiarism. I pledge that all the work submitted here is my own, and that I have clearly acknowledged and referenced other people's work. I am aware that it is my responsibility to turn in other students who have committed an act of academic dishonesty; and if I do not, then I am in violation of the Code. I will report to formal proceedings if summoned.

Signed:

CONTENTS

NOTE: Right click and select “Update Field” above to automatically update page numbers.

NOTE: Remove all highlighted text prior to submissio

OVERVIEW

For Lab 3, I was tasked with designing and implementing a BCD Sequential Counter that utilizes the onboard pushbuttons, switches, and the PMOD 7-Segment attachment to display the counting sequence. The counting sequence would occur at a rate of 1Hz with the PMOD 7-Segment displaying decimal values ranging from 00 to 99 and would have the ability to count up until 99 then start counting again back at zero. The process should also work by counting down from 99 to 00 and then start counting down again back at 99. The four onboard pushbuttons of the Zybo board are used for functions: Enable, Reset, Load, and Up/Down. The Enable button would start or stop the counting process when pushed. The Reset button would clear the values of the 7-Segment back to 00 and stop the counting process. The Load button was assigned to the onboard and additional PMOD switches for the 4-bit input vectors, *A* and *B*, such that when the Load button was pressed the 7-Segment would display the binary value between 0 to 9 of each digit. For this lab, the onboard switches represented by the input *A* were assigned to the MSD position of the 7-Segment, with input *B* assigned to the LSD of the display. Both a toggler and counter module were provided at the start of Lab and I was tasked with creating three Clock Divider modules and a Persistence-of-Vision Driver that would program the PMOD 7-Segment to display both the MSD and LSD to the user. The following sections outline the design process, simulation, and synthesized results.

DESIGN DESCRIPTION

The first step in my design process was to create a generic Clock divider module that could be used to slow down the internal 125MHz clock of the Zybo board for three different modules within the architecture of my Lab 3 structure. The design utilized generic statements to make the clock adjustable when used as a modular block later in the design process. The architecture of the Clock Divider used a variable that would count the number for clock ticks, and a VHDL process with the clock input port within its sensitivity list. If statements were used within the process to check when a clock event occurred and when the clock input was high. When this happened the variable *count* increased its value with each clock strike. Since I used generic statements, I wrote a nested If statement to check when the variable reached my predefined *N* value, a temporary signal was set HIGH at that moment with the *count* variable reset back to zero. The clock output port was then instantiated to equal the temporary variable, which achieved the clock division functionality of this design block. The value of *N* had to be an integer value to scale down the 125MHz internal clock of the Zybo. With the generic Clock divider module built, I moved onto creating the Persistence-of-Vision Driver for the Seven Segment display.

The design of the PoV Driver needed to output the 7-bit vector that would translate to the decimal BCD value on the Seven Segment, but also output a cathode signal that would rapidly turn both digits of the Seven Segment on and off. The 7-bit output would be assigned to the pins of both the J1 and J2 Connector ports for inputs AA to AG. In the previous lab, we utilized the same component and ports, but did not assign a value to the Cathode port represented by C as seen in Figure 1 below. When the cathode port is LOW, only the right digit of the Seven Segment will be powered on, but when the same port is HIGH, the left digit will be powered on. Thus, due to the NOT gates attached to the Cathode port of the Seven Segment, we could apply a frequency that is fast enough that it will rapidly turn both displays on and trick the human eye to appear like both displays are constantly powered. Referencing the SS Driver

Manuel, the documentation requires to have a frequency of at least 50 Hz to create the illusion that both digits are powered on.

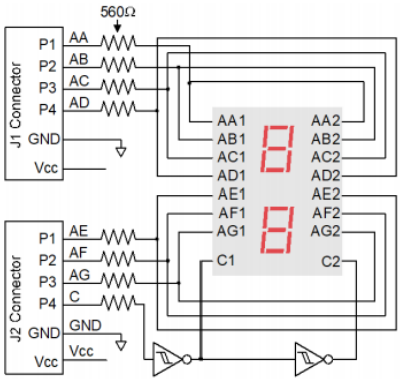

Figure 1 - Internal connections of PMOD Seven Segment Display.

The inputs of the PoV Driver would be connected to both the MSD and LSD counter modules, which required two multiplexers to be designed within the PoV Driver module. Figure 2 below shows the internal connections of the PoV Driver displaying the input/output relationship I have described.

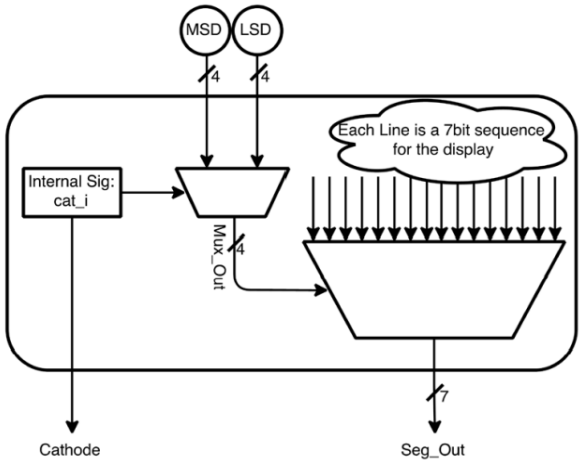

Figure 2 - Internal connections of the Seven Segment Persistence of Vision Driver.

The first multiplexer chooses whether to output the value of the MSD or LSD based on the logic value of the internal cathode signal. Thus, the cathode signal is used to drive the Seven Segment to display both digits and determine which counter input value to output to the 7-bit Segment output. When the cathode signal is LOW the LSD input would be sent to the select input of the second multiplexer. Likewise, when the signal is HIGH, the MSD will be sent to the select input of the following multiplexer. So, in real-time, the cathode signal will be displaying and selecting which digit value appears on the Seven Segment. To realize this design, I chose to write a VHDL function that achieves the functionality of the required process, instead of creating two multiplexer modules to incorporate into the PoV Driver module.

The function essentially uses both a 4-bit input and 7-bit output variable that checks when the internal cathode signal is HIGH or LOW and outputs the appropriate 7-bit vector for a value between 0 to 9 on the Seven Segment display. The PoV Driver also uses another VHDL process that is dependent on the change

to the *clk* input and checks when the signal is HIGH. When this occurs the internal cathode signal will oscillate, and the predefined function will output the appropriate 7-bit vector.

The toggler module that was given to us was to be designed for both the Up_Dn Toggler and Enable Toggler within the structural architecture of the BCD Sequential Counter. The module itself contained inputs for the divided clock signal, the reset pushbutton, an input *A* for either the up_dn pushbutton or Enable pushbutton, and a single output *Z* that would connect to both the MSD and LSD counters. The architecture of the toggler module was programmed with a VHDL process that contained both the *clk* and *rst* inputs on its sensitivity list. The process would check when the reset button was pressed, which would clear internal signals to zero. It would also check when a change in the clock has occurred and if the input *A* was pressed, it would toggle the output *Z*.

The counter module would then be designed for their respective MSD and LSD module blocks within the Lab 3 structural architecture. These design blocks contained inputs: *rst* for the reset pushbutton, *clk* for the divided clock signal, *ld* for the Load pushbutton, *en* for the Enable pushbutton, *up_dn* for the Up/Down pushbutton, input *D* which would tie to either input switches *A* or *B*, an *overflow* output to tie the LSD counter to the MSD Enable input, and *Q* the 4-bit output vector that would tie to the input of the SS PoV Driver. The counter module included a VHDL process that was dependent on changes to the *clk* and *rst* inputs. The process would clear the output if the reset pushbutton was pressed. During a clock event, if the Load pushbutton was pressed, the input *D* would be sent to the output *Q*, if the value was between 0 to 9 and allowed the counter to restart if it was counting up to 9 then back to zero or counting down from 9 to 0 then back up to 9. A second VHDL process is within the architecture of the module and it includes the inputs *en*, *up_dn*, and an internal signal *Qi*. The process checks when *en* is equal to one and sets the *overflow* output to one if *Qi* and *up_dn* are both zero or if *Qi* is binary 9 and *up_dn* is one. If neither are the case then *overflow* will be set to zero, concluding the end of the architecture for the design block.

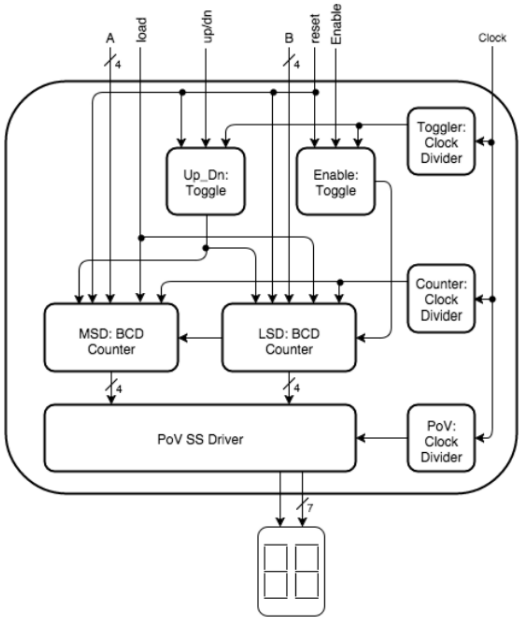

Figure 3 - Lab 3 Internal Connections and Modules.

To tie the entire project together, I created a structural architecture, like the design diagram shown in Figure 3 above. The Lab 3 design included inputs: *A* and *B* from the set of switches, *clk* from the internal 125MHz clock, *load* from the Load pushbutton, *up_dn* from the Up/Down pushbutton, *reset* for the reset pushbutton, *Enable* for Enable pushbutton, and outputs: *cathode* that would drive the Seven Segment to flip between digits, and *SegmentOut* that would display the appropriate decimal value to the PMOD Seven Segment display. The architecture included components such as the Clock divider module and SS PoV Driver that I designed and included the toggler and counter modules given to us. Internal signals were created to interconnect all the modules together and finalize the design.

DESIGN ENTRY

Designing the Clock divider module required the use of generic statements so that three modules could be implemented into the Lab 3 architecture with three different clock frequencies. The code for the Clock divider module can be found in the Appendix as [1]. A generic definition was placed before the port definitions in the entity of the Clock divider. Within the architecture, I used a shared variable named *count*, which was made *natural* rather than the usual *std_logic_vector* and an internal signal called *tmp*. Both were initialized to zero. The VHDL process used *clk_in* as its only signal within the sensitivity list. An If statement was made that checked if (*clk_in' EVENT) AND (clk_in = '1') then count := count + 1;* which would increment the variable *count* during the rising edge of the clock, effectively counting each clock strike. Another If statement is used to check when (*count = N)* then would set *tmp* to one and our variable *count* back to zero. This process is what creates our clock division because it creates a slower periodic signal as a result. The *clk_out* output was instantiated to equal the *tmp* signal before the architecture concludes.

The next step in the design process was creating the SS PoV Driver [2]. The port definitions contained inputs for *clk, MSD* and *LSD*, which were 4-bit input vectors and outputs for *cathode* and *Seg_Out* that was the 7-bit output vector. Within the architecture I defined an internal signal *cat_i* and a function named *loadSegment*. The function contained components *cat_s, LSD_sel*, and *MSD_sel* and would return variables *SO* and *input*. In the function declaration, if the *cat_s* signal was zero then *input* would equal *LSD_sel* and if *cat_s* equaled one then *input* would equal *MSD_sel*. This declaration is essentially the first multiplexer within the SS PoV Driver. The declaration continues using a case statement when *input* is equal to binary values between 0 to 9; the output variable *SO* would then become the corresponding 7-bit vector that represents the same decimal value. This declaration thus instantiates the second multiplexer needed for the internal architecture. The function declaration concludes and then a VHDL process depending on the change of the *clk* signal is used. Within the process, the internal signal *cat_i* is performed with a NOT operation and the *Seg_Out* is instantiated to *loadSegment(cat_i, LSD, MSD)*. The output *cathode* is then assigned to the internal signal *cat_i* before the process and architecture concludes for the SS PoV Driver module.

Within the toggle module there are internal signals *Zi, onecount,* and *twocount* and a VHDL process that is dependent upon the signals *clk* and *rst* [3]. The process starts by checking if *rst* is equal to one and would then set *Zi, onecount,* and *twocount* all to zero. The next steps observe a clock event on the rising edge to check if *A* is equal to one. If true, both *onecount* and *twocount* would increase in value from zero to one. If *A* is equal to zero, then *twocount* would become one and *Zi* is performed with a NOT operation.

The architecture ends with output port *Z* equaling internal signal *Zi*, which achieves the toggling requirement when the input *A* pushbutton is pressed.

The counter module code contains two VHDL processes to achieve the sequential counting needed for the Lab 3 design [4]. The architecture contains an internal signal *Qi*, which serves as a temporary signal of the output *Q*. A VHDL process called *Q_proc* is implemented and is dependent on the *clk* and *rst* signals. It starts by checking if *rst* is equal to one, in which case *Qi* would become reset to zero. Next, there is a clock event that checks when *ld* is equal to one. If true, and if the unsigned value of the input *D* is between binary values 0000 to 1001, then *Qi* would become equal to the value of *D*. Else, the signal *Qi* would be all zeros. This reassures that only values between 0 to 9 of both input switches *A* and *B* can be loaded into the Seven Segment driver. This avoids any unwanted hexadecimal values and exclusively targets decimal values from 0 to 9. If the *en* signal is equal to one, the process checks what the value of *up_dn* is at that moment. If *up_dn* is zero, then the values for *Qi* either decrement or, if *Qi* is 0000, then *Qi* would become 1001 and resume counting downward. These statements allow the digits of the Seven Segment to continuously count down in both the LSD and MSD place and even jump back to the upper limit and resume counting downwards. Else, if *up_dn* is equal to one, then the value of *Qi* increments or, if *Qi* is equal to 1001, it becomes 0000 then resumes counting upwards. This allows for the process to continually increase in decimal value and would reset the count back to 0 if it were to count past 9. The *Q_proc* ends with output *Q* instantiated to equal internal signal *Qi*. A second process called *overflow_proc* is included and depends on inputs *en*, *up_dn*, and internal signal *Qi*. The process begins by checking if *en* is equal to one. If *Qi* is all zeros AND *up_dn* is zero, then *overflow* is set to one. Else, if *Qi* is equal to 9 AND *up_dn* is equal to one, then again, *overflow* is equal to one. These statements allow for the carrying over from the LSD to the MSD when the count increases or decreases in the MSD place of the Seven Segment display. If none of the previous conditional statements are true, then *overflow* is set to zero, and the *overflow_proc* ends. This concludes the module design all of the necessary components to the entire Lab 3 structural architecture, so the next steps were to tie everything together into one .vhd file.

The code for the Lab3 design can be found in the Appendix as [5]. As mentioned in the previous Design Description section, the design for Lab3 structural architecture includes all the previous modules mentioned to successfully build the BCD Sequential Counter. The architecture included three clock divider modules: Toggle clock, Counter clock, and PoV Driver clock, two BCD counters: MSD counter and LSD counter, and the PoV 7-Segment Driver. A dozen or so internal signals were created to interconnect all the modules to their appropriate inputs and outputs.

For the Toggle Clock, the generic map was defined with an *N* value being equal to 125,000. This value would divide the internal clock down to a 1kHz clock so that the toggle could occur once every millisecond. This was specified within the LAB WORK instructions given to us for this assignment. Within the port, the internal 125MHz clock is tied to the *clk_in* port with an internal signal *toggle_clk* tied to the output *clk_out* port.

For the Counter Clock, the generic map was defined with an *N* value being equal to 125,000,000, which yields a clock divided frequency of 1Hz. This divided clock would yield a change in the counter once per second, which would give the Seven Segment the same clock frequency we experience as seconds pass on a traditional clock. Both the documents we were tasked with following to build the BCD Sequential Counter suggested either a 1Hz or 2Hz frequency, so I chose the 1Hz to keeper the counter slower and at the same

rate as a traditional clock. Again, the input port *clk_in* is tied to the internal 125MHz clock with the output port *clk_out* tied to an internal signal called *counter_clk*.

For the PoV Clock, the generic map was defined with an *N* value being equal to 1,250,000. This would create a new clock frequency of 100Hz that would drive the *cathode* signal tied to the Seven Segment display. The SS Display Reference Manuel recommended a frequency of at least 50Hz to have both values appear on the Seven Segment. I chose to double that value after originally setting the frequency to 50Hz, which had some visibility that the *cathode* signal was changing. With the value doubled to 100Hz, the Seven Segment is oscillating fast enough to where my own eyes cannot see the oscillations, which gave it more of the Persistence-of-Vision I found acceptable. The *clk_in* port was tied to the internal 125MHz clock just like the other clock dividers, and its output *clk_out* was tied to an internal signal *PoV_clk*.

For the Enable Toggle, the *clk* input was tied to the internal signal *toggle_clk*, which was running at 1kHz, the *rst* input was tied to the *reset* pushbutton, input *A* was tied to the *Enable* pushbutton, and output *Z* was tied to internal signal *EN_toggle*.

For the Up/Down Toggle, the *clk* input was also tied to the internal signal *toggle_clk*, the *rst* input tied to the *reset* pushbutton, input *A* tied to the *up_dn* pushbutton, and output *Z* tied to internal signal *UpDn_toggle*.

For the counters, the LSD BCD Counter had input *rst* tied to the *reset* pushbutton, its *clk* input was tied to signal *counter_clk*, and the *ld* input connected to the *load* input pushbutton. Its *en* input tied to the signal *EN_toggle*, the *up_dn* input tied to the signal *UpDn_toggle*, and its *D* input connected to input switches *B*. The *overflow* output was connected to internal signal *EN_btw*, and output *Q* was tied to internal signal *LSD_out*. This counter module was specified in its design to only be tied to input switches *B*.

For the MSD BCD Counter, input *rst* tied to the *reset* pushbutton as well. Its *clk* input was tied to signal *counter_clk*. The *ld* input connected to the *load* input pushbutton. Its *en* input tied to the signal *EN_btw*, which tied together the overflow from the LSD Counter to the MSD Counter. The *up_dn* input was tied to the signal *UpDn_toggle*, its *D* input connected to input switches *A*. The *overflow* output was unused for this design, so it was tied to internal signal *overflow_s*, which was set to zero. Finally, the output *Q* was tied to internal signal *MSD_out*. Similarly, this counter module was specified in its design to only be tied to input switches *A*.

Lastly, for the PoV Driver, its *clk* input was tied to the signal *PoV_clk*. The *MSD* input was tied to the *MSD_out* from the MSD BCD Counter. The *LSD* input was similarly tied to the *LSD_out* from the LSD BCD Counter. The *cathode* output was tied to the *cathode* output of the Lab3 design, and the *Seg_Out* port was connected to the *SegmentOut* of the Lab3 design as well. This concludes all the modular design necessary for the BCD Sequential Counter. So, the next steps were to create a Lab 3 testbench to simulate the structural architecture and then synthesize the design to the Zybo board and observe the results for both.

RESULTS AND OBSERVATIONS

Simulation Results

After completing the designs of all the modules. I created a testbench for the Lab 3 structural design and simulated a list of functions given to us to verify the design works as expected [6]. The list of functions included:

- Reset to 0
- Load in the value 19
- Count up to 22
- Count down to 15
- Reset to 0
- Count down to 89
- Load in the value 67
- Count up to 72

Within my testbench, I executed and simulated all the functions above into one timing diagram and the following figures show my simulation results. I also scaled up my clock divider modules so that my simulation results could be condensed into a short amount of time.

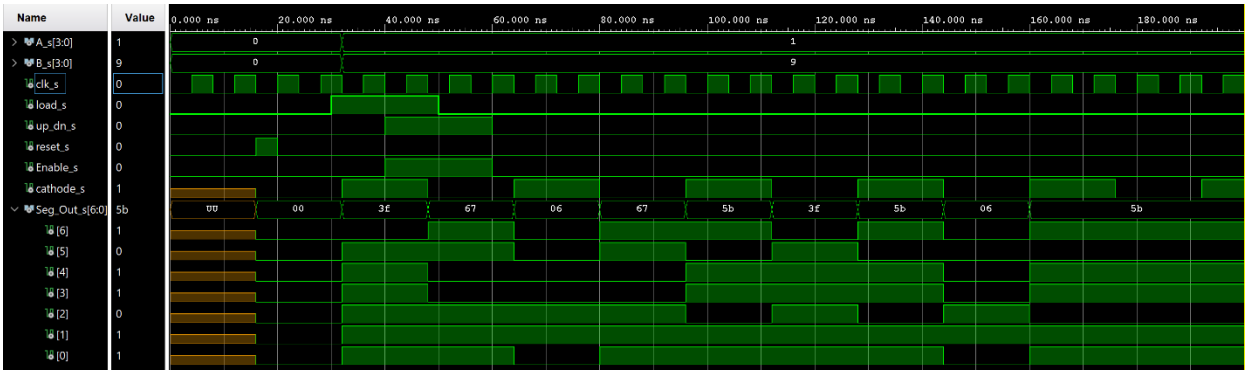

Figure 4 - Timing Diagram showing results for Reset to 0, Load value 19, and counting up to 22.

Figure 4 above shows the *reset_s* signal is HIGH at 16ns, which results in a 00 result for the *Seg_Out* bits. At 30ns, the *load_s* signal is high with *A_s* equal to decimal 1 and *B_s* equal to decimal 9. The value 19 is loaded into the *Seg_Out* and starts oscillating between the LSD and MSD starting at 48ns. Both the *up_dn_s* signal and *Enable_s* signals were executed as HIGH at 40ns and within one clock cycle of the *cathode_s* signal, the values of the *Seg_Out* start to increment from 19 to 22.

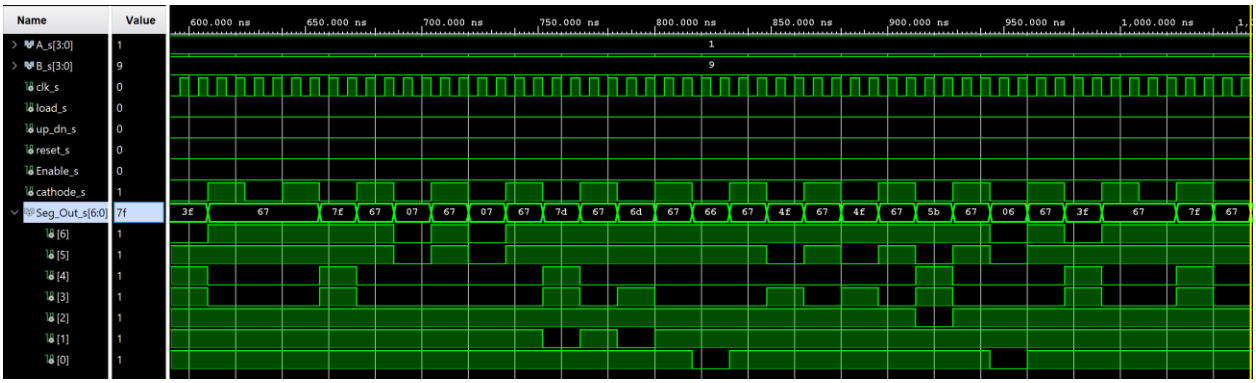

Figure 5 - Timing Diagram showing Reset to 0, Counting Down to 89.

Figure 5 above shows the *Seg_Out* bits become reset to zero shown by the values of 3f just before 600ns. The *Enable_s* signal was executed HIGH setting the counting down in motion. The *Seg_Out* bits can be seen decrementing from 00 to 99 and continuously counts down until reaching 89 at 1200ns.

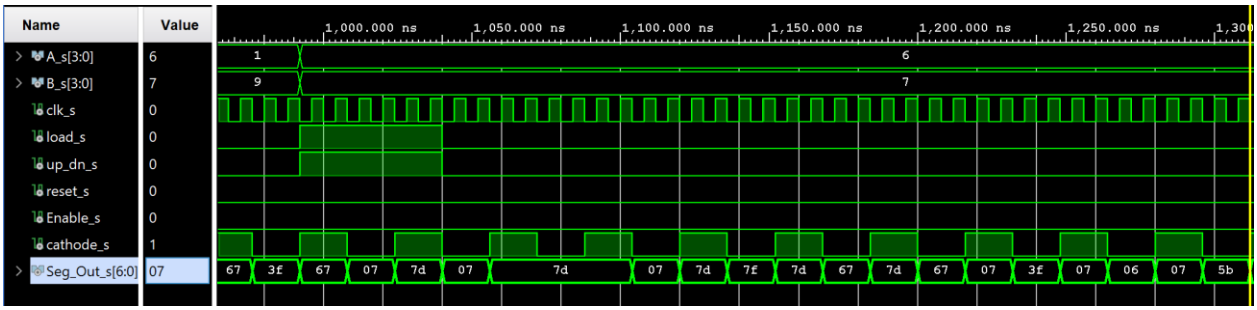

Figure 6 - Timing Diagram showing value 67 being loaded and counting up to 72.

Figure 6 above shows the value of 67 being loaded with the *load_s* and *up_dn_s* signals HIGH just before 1000ns, which loads the value and changes the down counting direction back to up. The counting up starts after the 1050ns mark and continues counting up from 67 to 72, which it reaches just after 1300ns. This concludes the simulation results, and my design appears to be working as intended. So the next steps were to synthesize the design and implement it onto the Zybo board to test.

Synthesized Results

To the synthesize the design, I used the I/O Planning tool to assign the correct pins to the onboard switches, pushbuttons, and additional PMOD switches and Seven Segment Display.

PMOD Switches	Onboard Switches	Onboard Buttons
SW1 = B(3)	SW1 = A(3)	BTN3 = Enable
SW2 = B(2)	SW1 = A(2)	BTN1 = Load
SW3 = B(1)	SW1 = A(1)	BTN0 = Up_Dn
SW4 = B(0)	SW1 = A(0)	BTN2 = Reset

Figure 7 - Recommended Pin Constraints from Lab Description.

Following the detail instructions pdf, I assigned my pins using the following Pin numbers of the Zybo board listed in Figure 8 below.

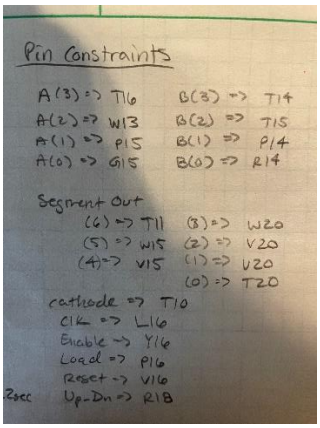

Figure 8 - Pin Constraints used for Lab 3

These pin assignments successfully assigned the necessary pins, to which I demoed the Zybo board with the simulated functions from before.

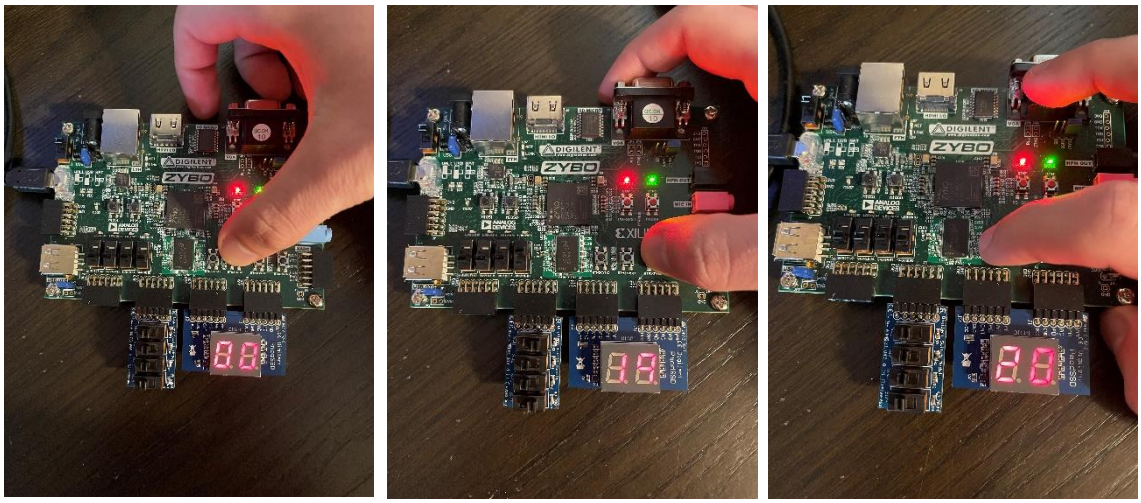

Figure 9 - Resetting the Zybo, Loading in value 19, and Enabling Count Up.

Figure 9 above shows the Reset pushbutton being pressed and resulting in the Seven Segment display clearing to zero. Setting the input switches *A* to binary 1 and PMOD switches *B* to binary 9 and then pressing the Load button displays the value of 19 onto the Seven Segment. Pressing the Up/Down button once and pressing the Enable button starts the counting up process resulting in a 20 on the Seven Segment display.

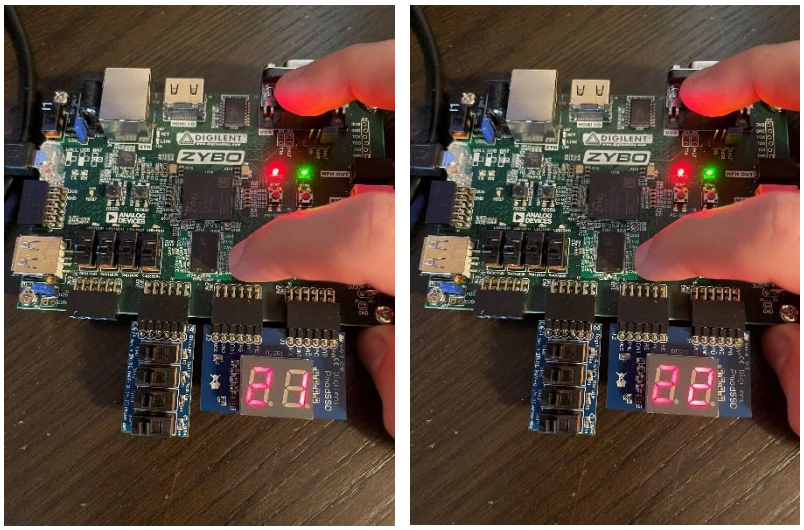

Figure 10 - Counting upwards continues and reaches 22 as expected.

The counting up sequence continues as seen in Figure 10 above as it reaches value 22 on the Seven Segment.

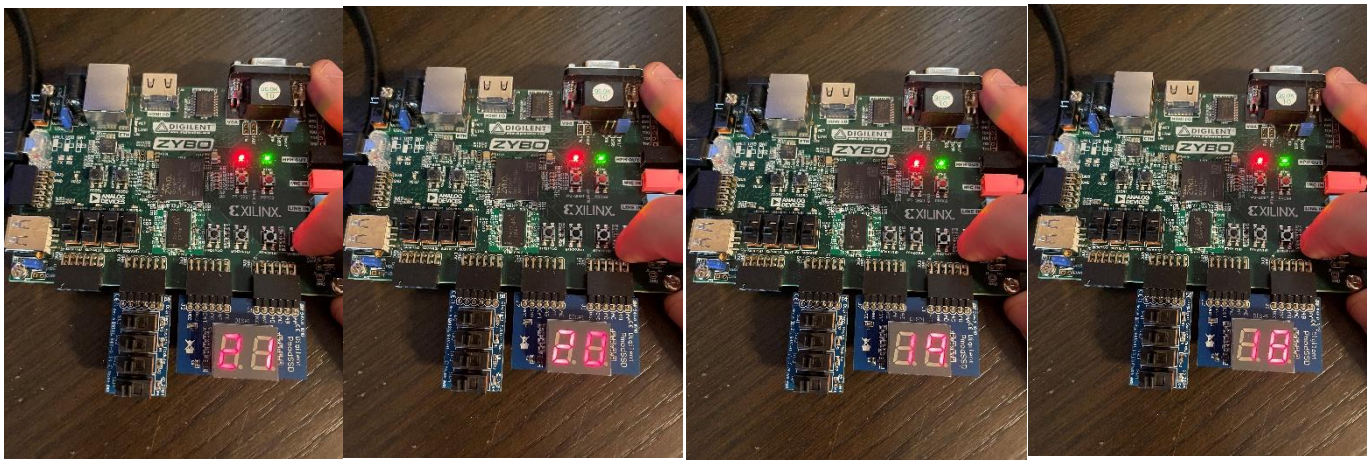

Figure 11 - Up/Down button is pressed initialing a count down sequence from 22 to 15.

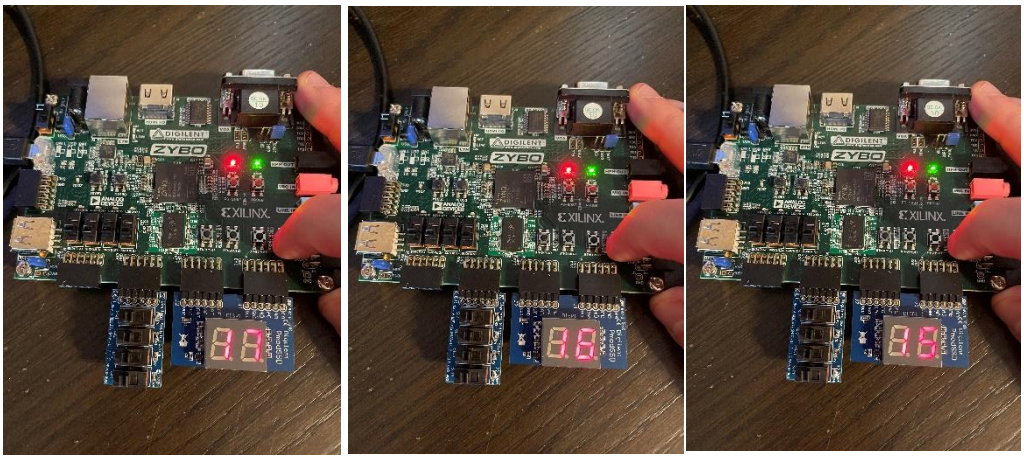

Figure 12 - Count Down sequence concludes as it reaches 15 as expected.

Figures 11 and 12 above display the counting down sequence that was initiated by pressing the Up/Down pushbutton. The sequence successfully counts down from 22 to 15 as expected.

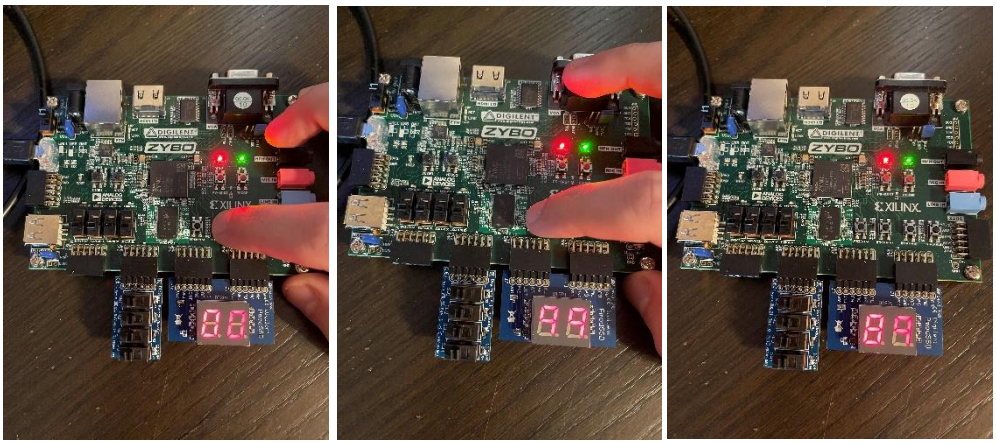

Figure 13 - Resetting Seven Segment to zero then Enabling a Count Down sequence to count from 99 to 89.

Figure 13 shows the Reset pushbutton being pressed and setting the values of the Seven Segment back to 00. Since the BCD Sequential Counter was last in the Count Down mode, once the Enable button is pushed, the value flips to 99 and continued to count down until reaching 89.

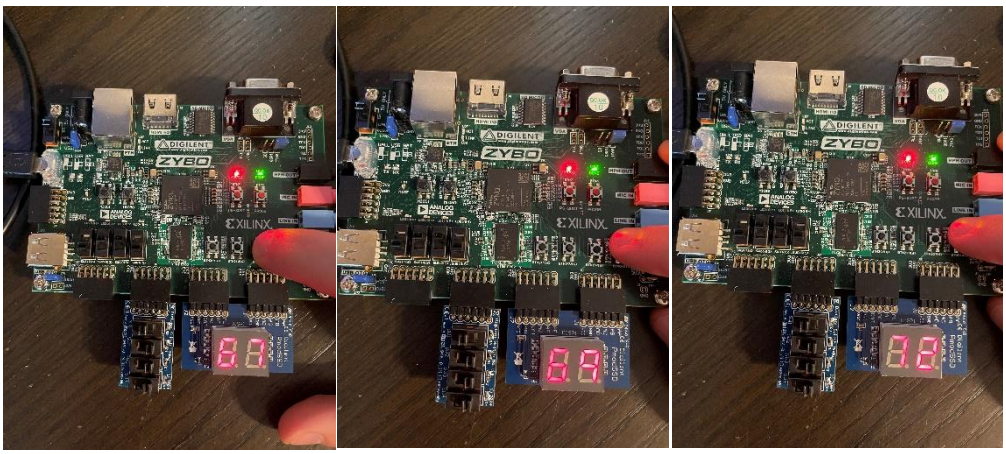

Figure 14 - Loading the value 67, then setting Up Counting, and Enabling to count up from 67 to 72.

Figure 14 shows the last of the simulated functions. I loaded in the value of 67 to the Seven Segment by setting the input switches *A* to binary 6 and PMOD switches *B* to binary 7, then pressed the Load pushbutton to have the value display. I pressed the Up/Down pushbutton to change from down-counting to up-counting and after pressing the Enable pushbutton, the BCD Sequential Counter was able to count up from 67 to 72.

These synthesized results also demonstrated that my design for the BCD Sequential Counter was a successful and the Zybo board was operating as intended.

SUMMARY

The overall purpose of this lab was to design a BCD Sequential Counter that could use the onboard switches, pushbuttons, and PMOD attachments, to count up or down from a decimal range of 00 to 99, and display the sequential counting to a Seven Segment display. My design process began by designing a clock divider module that would later be used to provide three difference clock frequencies to the internal structure of the BCD Sequential Counter. Next, I designed a SS PoV Driver that would designated a cathode output to oscillate the cathode port of the Seven Segment display, which displays both digits of the display and created a function that would serve as two multiplexers that chooses between the LSD and MSD based on the logic value of the cathode signal. The display received the appropriate 7-bit value based on the MSD and LSD being between a decimal range between 0 and 9. With the provided toggler and counter design blocks, I created a structural architecture that included three clock dividers, two counters, and the SS PoV Driver within it. Internal signals were then created to interconnect all the correct inputs/outputs to realize the design. With the design complete, a testbench was created to verify that the design works as intended and I was provided with a list of functions to test my design for. With the simulation concluded, I synthesized design to the Zybo board and used the I/O Planning tool to assign the Pin constraints for the appropriate connections. I verified the same simulation tests on the Zybo board and the BCD Sequential Counter was confirmed as a successful design.

APPENDIX

[1] library IEEE;

```
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;
entity clk_divider is
    generic (N: positive:= 62500000);
    Port ( clk_in : in STD_LOGIC;
           clk_out : out STD_LOGIC);
end clk_divider;
architecture Behavioral of clk_divider is
shared variable count: natural := 0;
signal tmp : std_logic := '0';
begin
    process(clk_in)
    begin
       if(clk_in'EVENT) AND (clk_in = '1') then
          count := count+1;
```

```
            if (count = N) then
             tmp <= not tmp;
             count := 0;
             else
             tmp <= '0';
             end if;
            end if;
        clk_out <= tmp;
        end process;
end Behavioral;
```

[2]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SS_PoV_Driver is
  Port (
        clk : in std_logic;
        MSD, LSD : in std_logic_vector (3 downto 0);
        cathode : out std_logic;
        Seg_Out : out std_logic_vector (6 downto 0));
end SS_PoV_Driver;
architecture Behavioral of SS_PoV_Driver is
signal cat_i : std_logic := '0';
function loadSegment (cat_s : std_logic; LSD_sel: std_logic_vector; MSD_sel: std_logic_vector)
    return std_logic_vector is
    variable SO: std_logic_vector(6 downto 0);
    variable input: std_logic_vector(3 downto  0);
begin
    if(cat_s = '0') then
        input := LSD_sel;
    elsif (cat_s = '1') then
        input := MSD_sel;
    end if;
```

```
    case input is
        when "0000" => SO := "0111111";
        when "0001" => SO := "0000110";
        when "0010" => SO := "1011011";
        when "0011" => SO := "1001111";
        when "0100" => SO := "1100110";
        when "0101" => SO := "1101101";
        when "0110" => SO := "1111101";
        when "0111" => SO := "0000111";
        when "1000" => SO := "1111111";
        when "1001" => SO := "1100111";
        when others => SO := "0000000";
    end case;
    return SO;
end function;
begin
    process(clk)
    begin
        if (clk' EVENT) and (clk = '1') then
        cat_i <= not cat_i;
        Seg_out <= loadSegment(cat_i,LSD,MSD);
        cathode <= cat_i;
    end if;
end process;
end Behavioral;
```

[3]

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity toggler is
                                port(
                                        clk : in  std_logic;
```

```
                rst : in  std_logic;
                A   : in  std_logic;
                Z   : out std_logic );
end entity toggler;
architecture Behavioral of toggler is
        signal Zi, onecount, twocount : std_logic;
begin
        process(clk, rst) is
        begin
                if rst = '1' then
                        Zi       <= '0';
                        onecount <= '0';
                        twocount <= '0';
                elsif clk = '1' and clk'event then
                        if A = '1' then
                                if onecount = '0' then
                                        onecount <= '1';
                                elsif onecount = '1' then
                                        twocount <= '1';
                                end if;
                        elsif A = '0' then
                                if twocount = '1' then
                                        Zi <= not Zi;
                                end if;
                                onecount <= '0';
                                twocount <= '0';
                        end if;
                end if;
        end process;
        Z <= Zi;
end architecture Behavioral;
```

[4]
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity counter is
                port(
                        rst     : in  STD_LOGIC;
                        clk     : in  STD_LOGIC;
                        ld      : in  STD_LOGIC;
                        en      : in  STD_LOGIC;
                        up_dn   : in  STD_LOGIC;
                        D       : in  STD_LOGIC_VECTOR(3 downto 0);
                        overflow : out STD_LOGIC;
                        Q       : out STD_LOGIC_VECTOR(3 downto 0));
end counter;
architecture Behavioral of counter is
                signal Qi : std_logic_vector(3 downto 0);
begin
                Q_proc : process(clk, rst) is
                begin
                        if rst = '1' then
                                Qi <= "0000";

                        elsif clk = '1' and clk'event then
                                if ld = '1' then
                                        if (unsigned(D) >= 0) and (unsigned(D) <= 9) then
                                                Qi <= D;
                                        else
                                                Qi <= "0000";
                                        end if;
                                elsif en = '1' then
                                        if up_dn = '0' then
```

```
                    if Qi = "0000" then
                        Qi <= "1001";
                    else
                        Qi <= std_logic_vector(unsigned(Qi) - 1);
                    end if;
                elsif up_dn = '1' then
                    if Qi = "1001" then
                        Qi <= "0000";
                    else
                        Qi <= std_logic_vector(unsigned(Qi) + 1);
                    end if;
                end if;
            end if;
        end if;
    end process Q_proc;
    Q <= Qi;
    overflow_proc : process(Qi, en, up_dn) is
    begin
        if en = '1' then
            if Qi = "0000" and up_dn = '0' then
                overflow <= '1';
            elsif Qi = "1001" and up_dn = '1' then
                overflow <= '1';
            else
                overflow <= '0';
            end if;
        else
            overflow <= '0';
        end if;
    end process overflow_proc;

end Behavioral;
```

[5] library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity Lab3 is

Port (

A, B : in std_logic_vector (3 downto 0);

clk, load, up_dn, reset, Enable : in std_logic;

cathode : out std_logic;

SegmentOut : out std_logic_vector (6 downto 0));

end Lab3;

architecture Structural of Lab3 is

component clk_divider is

generic (N: positive);

Port (clk_in : in STD_LOGIC;

clk_out : out STD_LOGIC);

end component;

component toggler is

port(

clk : in std_logic;

rst : in std_logic;

A : in std_logic;

Z : out std_logic);

end component;

component counter is

port(

rst : in STD_LOGIC;

clk : in STD_LOGIC;

ld : in STD_LOGIC;

en : in STD_LOGIC;

up_dn : in STD_LOGIC;

D : in STD_LOGIC_VECTOR(3 downto 0);

overflow : out STD_LOGIC;

```
                                    Q       : out STD_LOGIC_VECTOR(3 downto 0)
                                );
end component;
component SS_PoV_Driver is
  Port (
        clk : in std_logic;
        MSD, LSD : in std_logic_vector (3 downto 0);
        cathode : out std_logic;
        Seg_Out : out std_logic_vector (6 downto 0));
end component;
signal A_in, B_in, MSD_out, LSD_out : std_logic_vector (3 downto 0);
signal load_i, up_dn_i, reset_i, Enable_i, clk_i : std_logic;
signal toggle_clk, counter_clk, PoV_clk : std_logic;
signal UpDn_toggle, EN_toggle, EN_btw, cathode_s : std_logic;
signal overflow_s : std_logic := '0';
signal SS_out : std_logic_vector (6 downto 0);
begin
    Toggle_Clock: clk_divider
        generic map (N => 125000)
        port map(
            clk_in => clk,
            clk_out => toggle_clk );
    Counter_Clock: clk_divider
        generic map (N => 125000000)
        port map (
            clk_in => clk,
            clk_out => counter_clk );
    PoV_Clock: clk_divider
        generic map (N => 1250000)
        port map (
            clk_in => clk,
            clk_out => PoV_clk );
```

```
Enable_Toggle: toggler
   port map(
     clk => toggle_clk,
                                rst => reset,
                                     A => Enable,
                                     Z => EN_toggle );
Up_Dn_Toggle: toggler
   port map(
     clk => toggle_clk,
                                rst => reset,
                                     A => up_dn,
                                     Z => UpDn_toggle );
LSD_BCD_Counter: counter
   port map(
    rst => reset,
                                     clk => counter_clk,
                                     ld => load,
                                     en => EN_toggle,
                                     up_dn => UpDn_toggle,
                                     D => B,
                                     overflow => EN_btw,
                                     Q => LSD_out );
MSD_BCD_Counter: counter
   port map(
     rst => reset,
                                     clk => counter_clk,
                                     ld => load,
                                     en => EN_btw,
                                     up_dn => UpDn_toggle,
                                     D => A,
                                     overflow => overflow_s,
                                     Q => MSD_out );
```

```
    POV_DRIVER: SS_PoV_Driver
        port map(
            clk => PoV_clk,
            MSD => MSD_out,
            LSD => LSD_out,
            cathode => cathode,
            Seg_Out => SegmentOut );
end Structural;
```

[6]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Lab3_tb is
end Lab3_tb;
architecture tb_stimulus of Lab3_tb is
component Lab3 is
   Port (
        A, B : in std_logic_vector (3 downto 0);
        clk, load, up_dn, reset, Enable : in std_logic;
        cathode : out std_logic;
        SegmentOut : out std_logic_vector (6 downto 0));
end component;
signal A_s, B_s : std_logic_vector (3 downto 0):= "0000";
signal clk_s, load_s, up_dn_s, reset_s, Enable_s, cathode_s : std_logic := '0';
signal Seg_Out_s : std_logic_vector (6 downto 0);
begin
    UUT: Lab3
    port map(
        A => A_s,
        B => B_s,
        clk => clk_s,
        load => load_s,
        up_dn => up_dn_s,
```

```
        reset => reset_s,
        Enable => Enable_s,
        cathode => cathode_s,
        SegmentOut => Seg_Out_s );
    clk_s <= not clk_s after 4ns;
    A_s <= "0001" after 32ns, "0110" after 992ns;
    B_s <= "1001" after 32ns, "0111" after 992ns;
    load_s <= '1' after 30ns, '0' after 50ns, '1' after 992ns, '0' after 1040ns;
    reset_s <= '1' after 16ns, '0' after 20ns, '1' after 480ns, '0' after 496ns;
    up_dn_s <= '1' after 40ns, '0' after 60ns, '1' after 140ns, '0' after 180ns, '1' after 992ns, '0' after 1040ns;
    Enable_s <= '1' after 40ns, '0' after 60ns, '1' after 496ns, '0' after 560ns;
end tb_stimulus;
```