Lab2 Exercise3 Decimal Counter

1. Introduction

In this lab, we learned the basic design idea of sequential circuit and tried to design a decimal counter. Also we learned how to use the IP core in our design to reduce the stress and designing time to get everything done by ourselves (it takes a lot of time and most of the time it works worse than the IP core). Finally we also learned how to program our FPGA board and let the board runs as we expect.

Our goal in this lab is to design a three-digital decimal counter, which will plus one after every second. The additional function to this counter is that it can be reset to all zeros, and it can also be reset to arbitrary number we set. We use switches and buttons to control the inputs, and use LED to show the outputs.

We use a general idea of sequential circuit to design. We put the sequential part only update the value of registers, and use combination part get the value that should be assigned in the next clock period.

2. Pre-lab Preparation

2.1. Behavior of the three digital decimal counter

A three-digital decimal counter should have three digitals in decimal, ranging from 0 to 999. But in our digital system, the number must be represented by the bit numbers. Thus we use the 8421 BCD code to represent, which means each digital in decimal represented by four digital in binary. For three digital in decimal, we need totally 12 digital in binary. We name them d1, d10, d100 in decimal and they are also array in binary with length four bit.

Also, because the clock issue, we have to get a trigger to trigger the counter when it should plus one. The trigger should be a pulse with period 10MHz and pulse length 100ns (one clock cycle).

The part of the ideal waveform of this counter is shown in **figure 2.1**

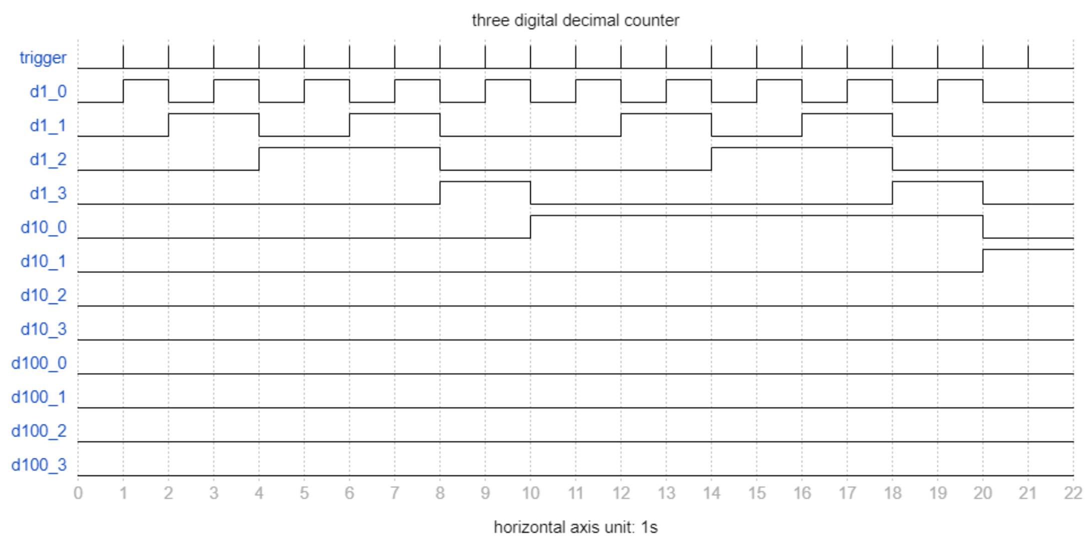


Figure 2.1 Part of ideal waveform of a three digital decimal counter

2.2. Design of the three digital decimal counter

To realize the three digital decimal counter, we must have three parts done.

The first part is the frequency (clock) divisor. We have a 100MHz crystal oscillator installed in the PL part on our FPGA board, but it is too fast for our counter which counts for every single second. Thus we need frequency divisor to get a much lower frequency.

The second part is the 10MHz counter. In the previous part we have a frequency divisor to slow down the original frequency but it is not enough. The requirement of a clock is very high and any extra delay is not acceptable, which means the layout and routing of the devisor should be elaborating. We may be not able to realize such design, thus we use the IP core 'clock wizard' to help us to get a stable, precise but slower clock. However, the IP core does not provide quite slow output frequency. The minimum supported frequency of the IP core is 5MHz, much faster than we need. Thus in this part, we design another counter which will count 10MHz (our frequency designed in the last part is 10MHz though it can provide 5MHz as minimum) and pull up a trigger signal to enable the main three digital decimal counter. Thus the trigger signal will be pulled up for 1 clock period (100ns), and for the rest 9,999,999 periods (1s - 100ns) it stays pulled down.

The third part is the main counter. The input clock of this part should still be the clock generated in the first part, but only counts when trigger is high. And when d1 reaches 9, in the next count, d1 should be assigned to 0 and d10 should be assigned to its next value. This is same for d100. For additional function, except we add set all zero when 'reset' is enabled, we have add another option to judge. When 'reset' is enabled, we check the 'load'. If 'load' is low, then we simply set all digitals to zero, while it is high, then we load the data of our switches to the digitals.

2.3. Design diagram

The design diagram is shown in **figure 2.2**

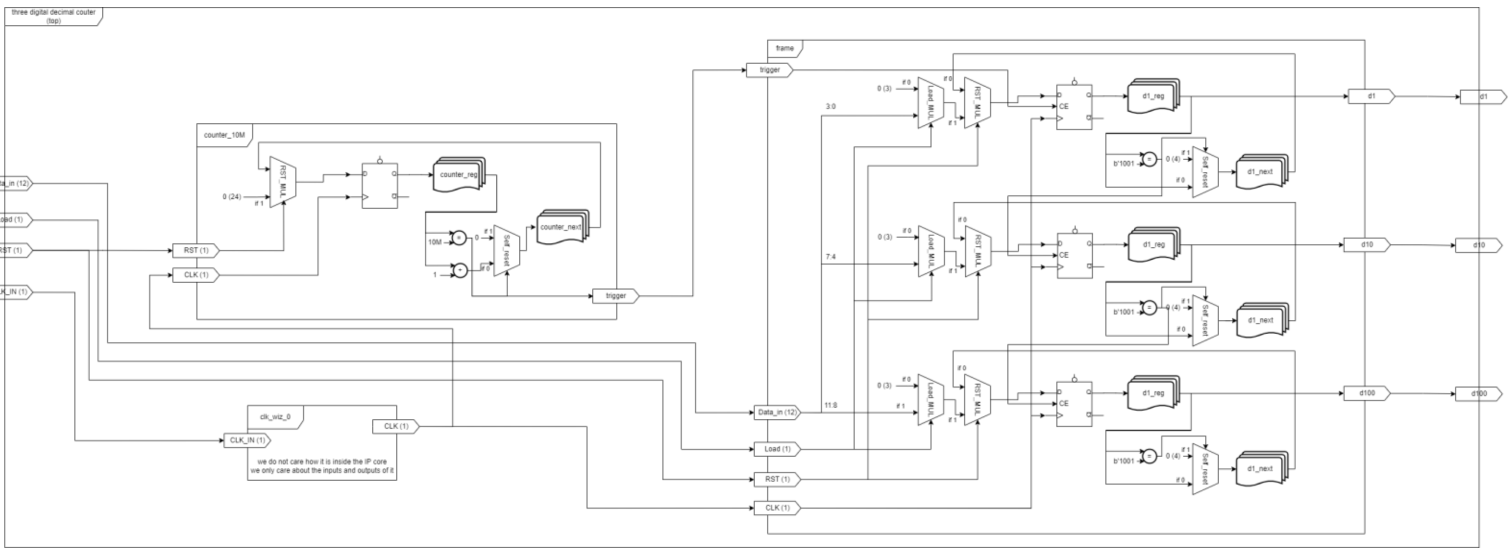


Figure 2.2 Design diagram

2.4. State diagram

The state diagram is shown in **figure 2.3**

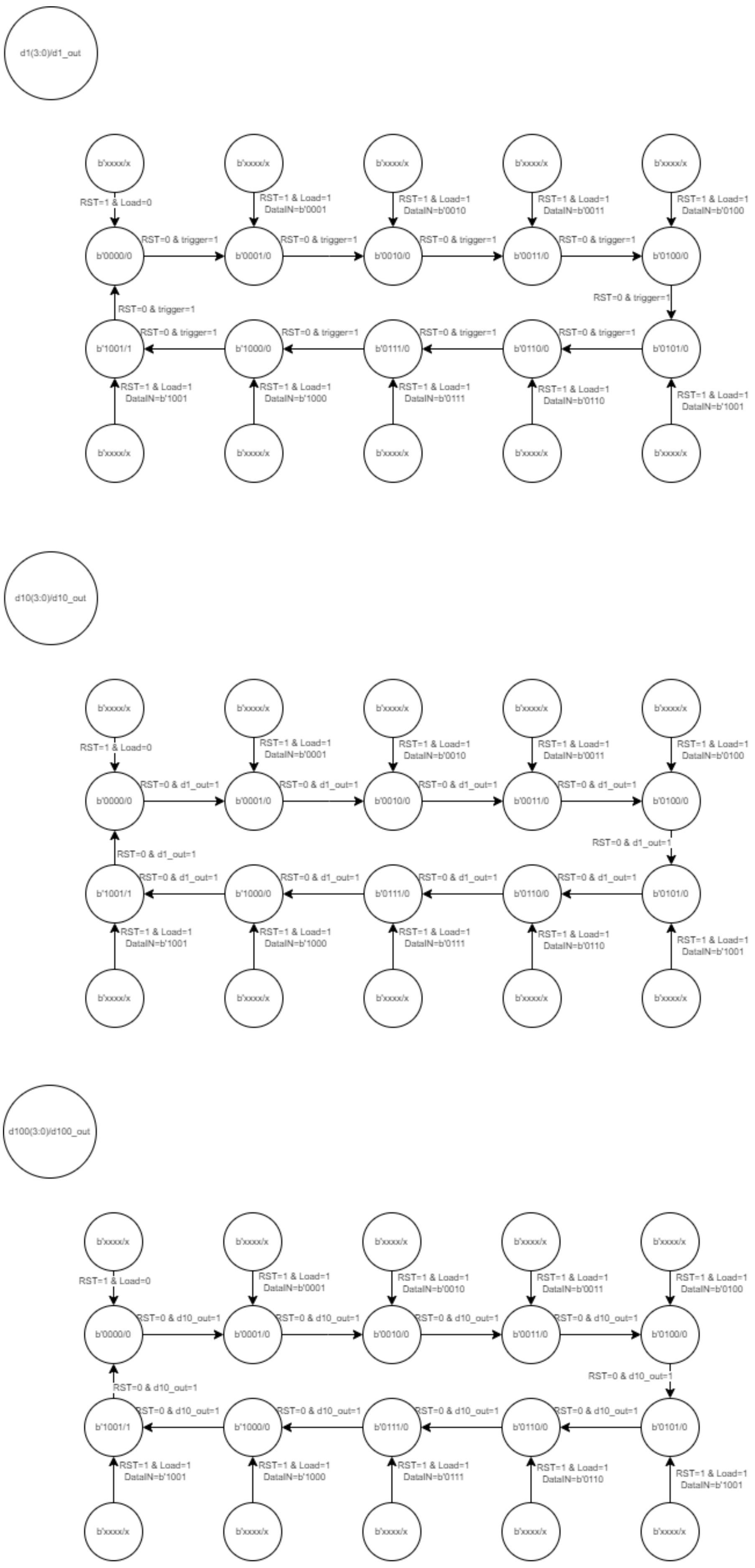

d1(3:0)/d1_out
b'xxxx/x
RST=1 & Load=0
RST=1 & Load=1 DataIN=b'0001
RST=1 & Load=1 DataIN=b'0010
RST=1 & Load=1 DataIN=b'0011
RST=1 & Load=1 DataIN=b'0100
b'0000/0
b'0001/0
b'0010/0
b'0011/0
b'0100/0
RST=0 & trigger=1
b'1001/1
b'1000/0
b'0111/0
b'0110/0
b'0101/0
RST=1 & Load=1 DataIN=b'1001
RST=1 & Load=1 DataIN=b'1000
RST=1 & Load=1 DataIN=b'0111
RST=1 & Load=1 DataIN=b'0110
RST=1 & Load=1 DataIN=b'1001
d10(3:0)/d10_out
RST=0 & d1_out=1
d100(3:0)/d100_out
RST=0 & d10_out=1

Figure 2.3 State diagram

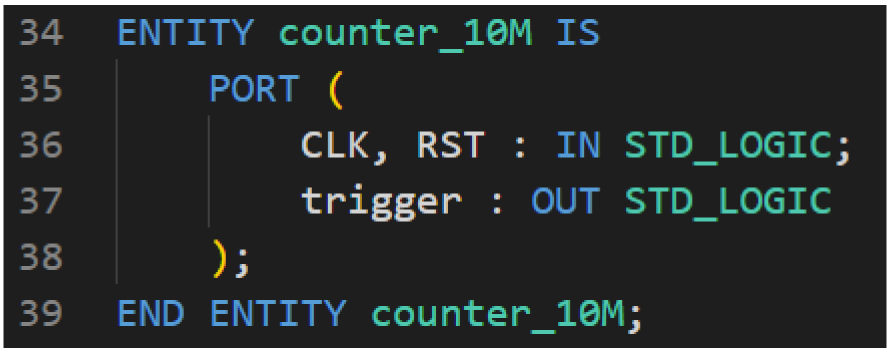
3. Source Design

3.1. 'Clock wizard' IP core setup

We use the IP catalog to find the proper IP core, and set up a few steps to finish customizing IP procedure. Now in this part, we only introduce some of the important setups in our design.
Our design is highly clock frequency dependent, but it is not sophisticated to use a clock monitor to supervise the clock. And for the usage to implement the clock, either the mixed-mode clock manager (MMCM) or phase lock loop (PLL) would work. And the utilization in this design is the same.
Also, the phase alignment is enabled in our design. This makes sure that the phase of the output clock is the same as the input one.
For our onboard oscillator provides a stable clock signal at 100MHz, thus our input for the IP core is 100MHz named 'clk_in1'.
For we only need one output clock, we set the output frequency 10MHz and duty cycle 50%.
After this, click 'OK' and wait for Vivado to generate the IP core itself.
These are all we done for the IP core customization.

3.2. Source file 'counter_10M' design

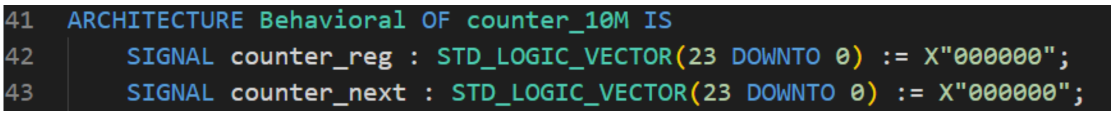
This entity only needs to count 10M and gives out a trigger signal, thus it has two inputs, for clock and reset, and one output, for trigger the main counter. They are named as 'CLK', 'RST', 'trigger' respectively.
The port design is shown in the **figure 3.2.1**



```
ENTITY counter_10M IS
    PORT (
        CLK, RST : IN STD_LOGIC;
        trigger : OUT STD_LOGIC
    );
END ENTITY counter_10M;
```

Figure 3.2.1 Port design of 'counter_10M'

Then we design the signal in this entity. We need two signals. The one that store the current counting number and the one that store the next counting number. We named them 'counter_reg' and 'counter_next' respectively. For we need it count up to 10M (0x989680 in hexadecimal), we need at least 24 bits to represent it. Thus the two signals are all signal array with length 24.
The signal design is shown in **figure 3.2.2**

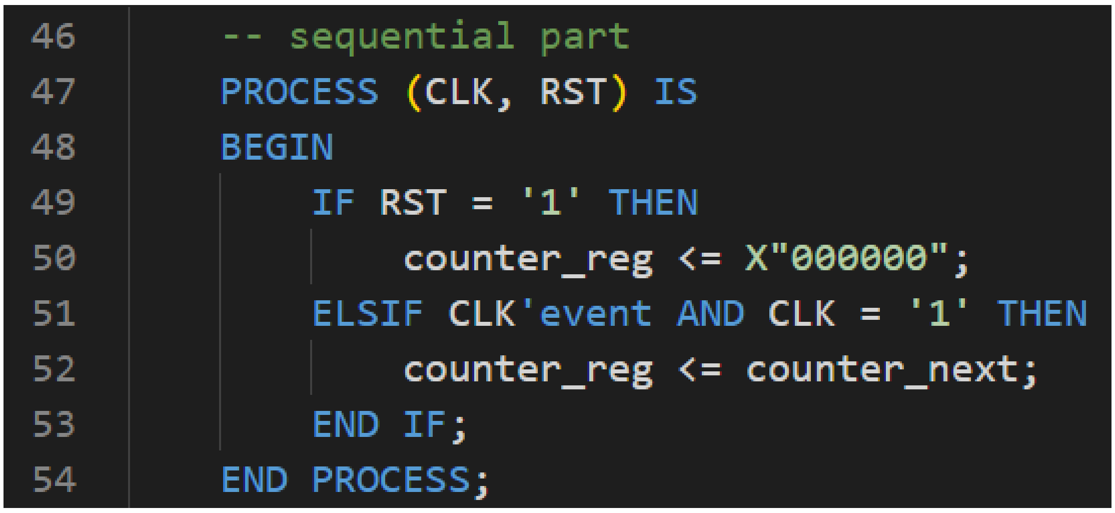


```
ARCHITECTURE Behavioral OF counter_10M IS
    SIGNAL counter_reg : STD_LOGIC_VECTOR(23 DOWNTO 0) := X"000000";
    SIGNAL counter_next : STD_LOGIC_VECTOR(23 DOWNTO 0) := X"000000";
```

Figure 3.2.2 Signal design of 'counter_10M'

For a good sequential circuits design, we should set the sequential part and combinational part apart.

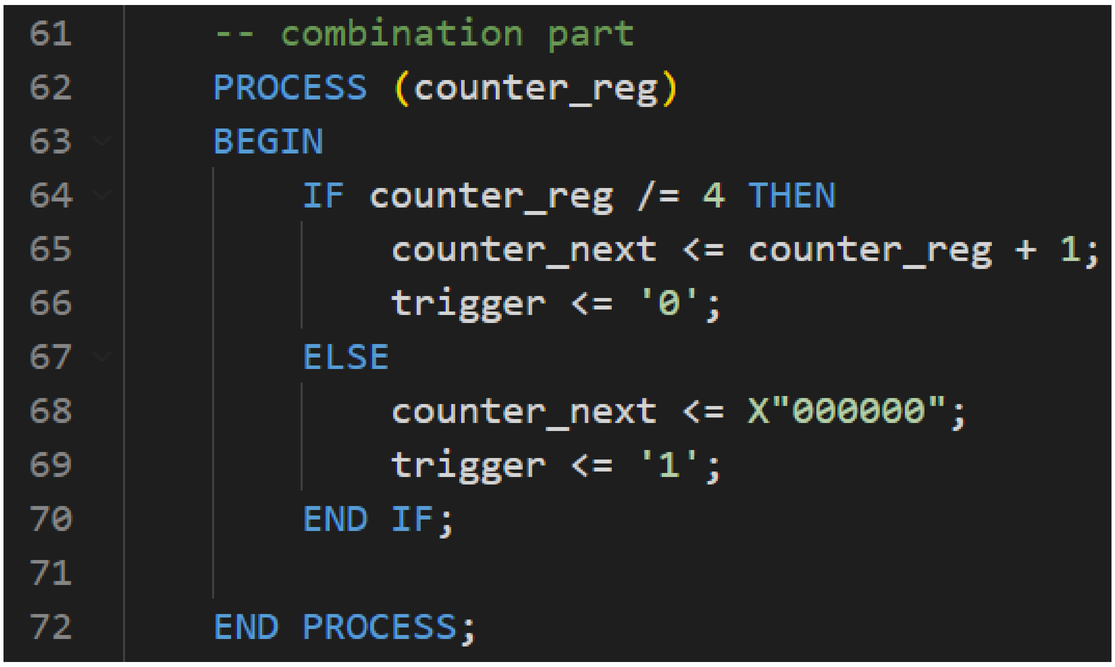
In the sequential part, we have to update the 'counter_reg' whether to 0 or to 'counter_next', depending by whether 'RST' is 0 or 1. And it also has to decide when to update this signal. In this design, updates is at the rising edge of the clock signal.
The sequential part design is shown in **figure 3.2.3**



```vhdl
    -- sequential part
    PROCESS (CLK, RST) IS
    BEGIN
        IF RST = '1' THEN
            counter_reg <= X"000000";
        ELSIF CLK'event AND CLK = '1' THEN
            counter_reg <= counter_next;
        END IF;
    END PROCESS;
```

Figure 3.2.3 Sequential part design of 'counter_10M'

Our reset here is asynchronous, thus whenever 'RST' is high, the 'counter_reg' is set to 0. And when 'RST' is 0 and the next rising edge of the clock coming, 'counter_reg' is updated to 'counter_next'.
In the combination part, we have to describe the behavior of the signal 'counter_next' (next state logic). In logic, it should be one larger than the 'counter_reg', for we count how many times the rising edge of the clock has come. But every time 'counter_reg' reaches 10M, we have to count from 0 again. Thus there is a multiplexer with input 'counter_reg' to decide what value will update to 'counter_next'.
The combination part design is shown in **figure 3.2.4**



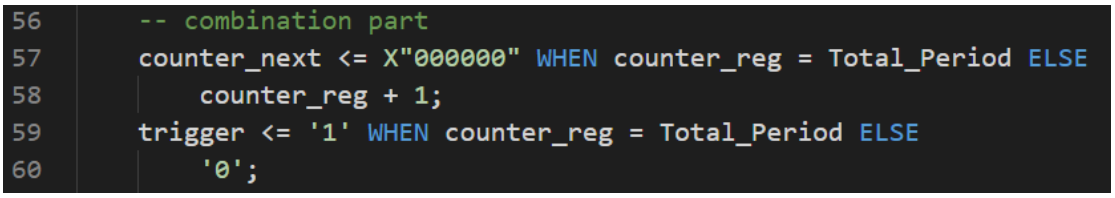
```vhdl
    -- combination part
    PROCESS (counter_reg)
    BEGIN
        IF counter_reg /= 4 THEN
            counter_next <= counter_reg + 1;
            trigger <= '0';
        ELSE
            counter_next <= X"000000";
            trigger <= '1';
        END IF;

    END PROCESS;
```

Figure 3.2.4 Combination design of 'counter_10M' (if arch)

In the first if arch, the judgement is whether 'counter_reg' is 4. This is because we use 4 as

simulation. Our simulation scale is in microsecond, thus wait the simulation to second is reasonless. Later when our design is proved acceptable in the simulations, we will change this to 10M and re-synthesis and re-implement.
Also we have to describe the behavior of the signal ‘trigger’. When it counts to 10M, the ‘trigger’ should give a high signal out, otherwise it should stay 0.

Also, we can write this combinational part in concurrent arguments. This is shown in **figure 3.2.5**



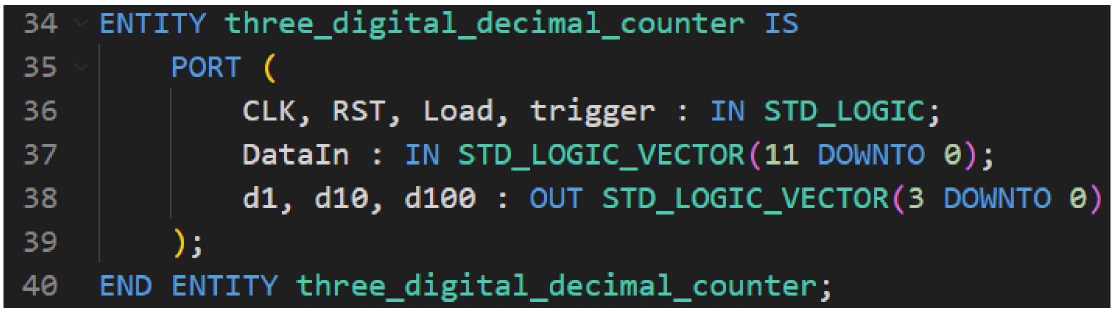
```
    -- combination part
    counter_next <= X"000000" WHEN counter_reg = Total_Period ELSE
        counter_reg + 1;
    trigger <= '1' WHEN counter_reg = Total_Period ELSE
        '0';
```

Figure 3.2.5 Combination design of ‘counter_10M’ (concurrent)

Where ‘Total_Period’ is a constant which represents how many clock the counter should count.
This looks much shorter in code, but they all describe the same thing in RTL level. Later I compared the two elaboration schematics generated by this concurrent and if-arch arguments. Vivado gives out almost the same RTL level design regardless which type of coding we use.
In our later design, we use the if-arch arguments because we need to frequently change the counting number (named ‘Total_Period’), this change could be done just modify one number in the if-arch but two modifications in the concurrent arguments if we do not use the constant. However the constant is defined in the beginning of the architecture, and the scrolling up to check the constant and down to check the logics sometimes makes me annoyed.

3.3. Source file ‘three_digital_decimal_counter’ design

This entity contains the main function module in counting by seconds.
The port of this entity should include clock, reset, and trigger. Also for additional function it should read the load signal to verify whether to read the data in. Thus of course it should have the data in and data out ports.
The port design is shown in **figure 3.3.1**



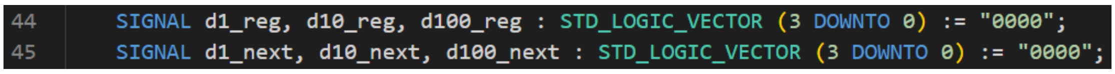
```
ENTITY three_digital_decimal_counter IS
    PORT (
        CLK, RST, Load, trigger : IN STD_LOGIC;
        DataIn : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        d1, d10, d100 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
    );
END ENTITY three_digital_decimal_counter;
```

Figure 3.3.1 Port design of ‘three_digital_decimal_counter’

‘CLK’ is the input clock (10MHz), ‘RST’ is the reset, ‘Load’ decides whether we preset the counter to 0 or the data we put in, ‘trigger’ is the trigger signal to get the counter plus one.
‘DataIn’ is the preset data input port with length 12. ‘d1’, ‘d10’, ‘d100’ is the output counter number with length 4.

Then we design the signal in this entity. Similar to ‘counter_10M’. We only need two kind of signals to store the current counting number and the next counting number. And we name them ‘dx_reg’ and ‘dx_next’ where ‘x’ could be 1, 10, 100, represents to the three digital unit in decimal. They are all signals with length 4.

The signal design is shown in **figure 3.3.2**



```
    SIGNAL d1_reg, d10_reg, d100_reg : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000";
    SIGNAL d1_next, d10_next, d100_next : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000";
```

Figure 3.3.2 Signal design of ‘three_digital_decimal_counter’

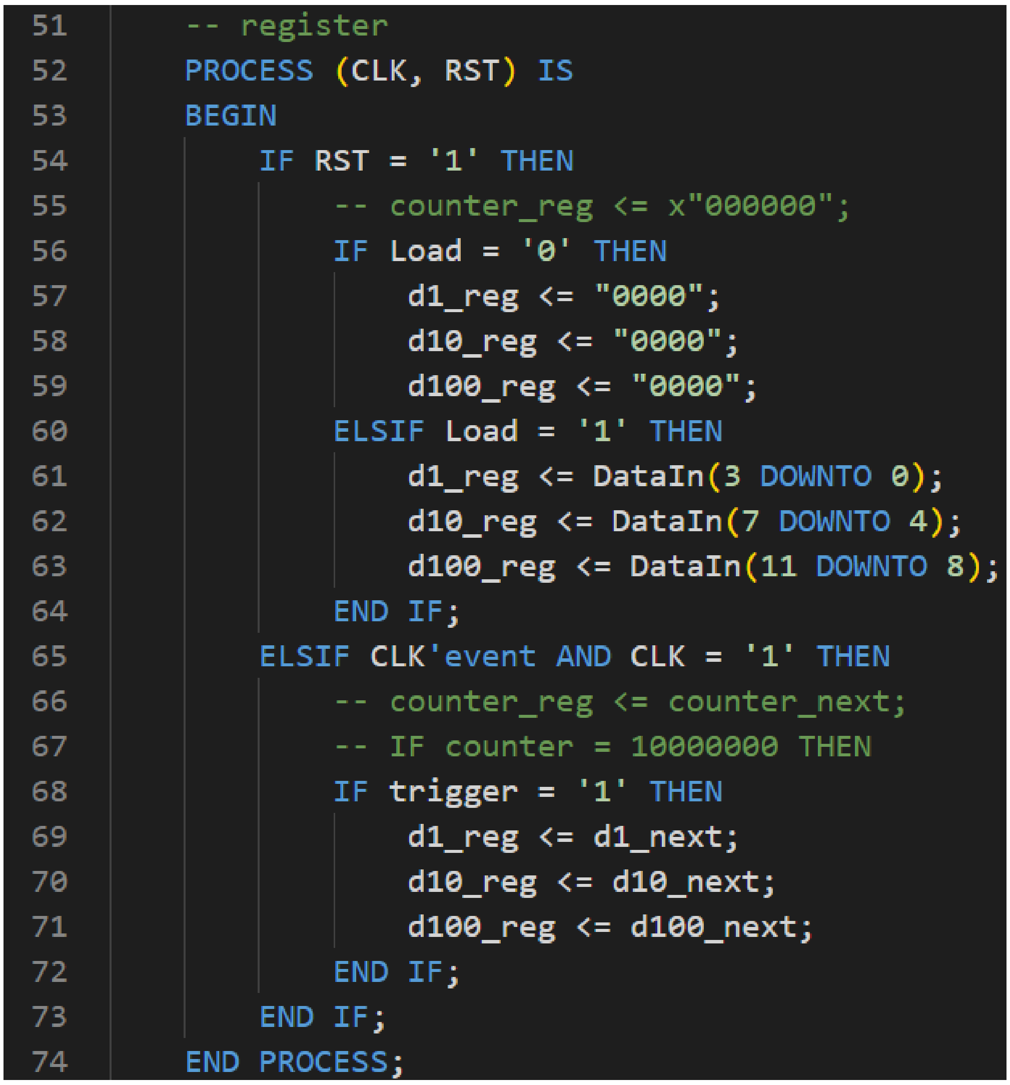
Similarly we need to design the sequential part and combinational part for this entity.

In the sequential part, we have to update the register ‘dx_reg’. And we also have to decide when to update.

For we use an asynchronous reset signal ‘RST’, thus it has the priority than ‘CLK’. The logic is if ‘RST’ is high, whatever ‘CLK’ is, the whole program is in resetting mode. And we check ‘Load’, if it is 0, then we set all the registers to 0, otherwise we set all register as the input data.

For the clock part, when the next rising edge of the clock come, we check the trigger. If ‘trigger’ is 1, which means ‘counter_10M’ already count for 10M times and it has passed 1second. The register ‘dx_reg’ will be updated by ‘dx_next’.

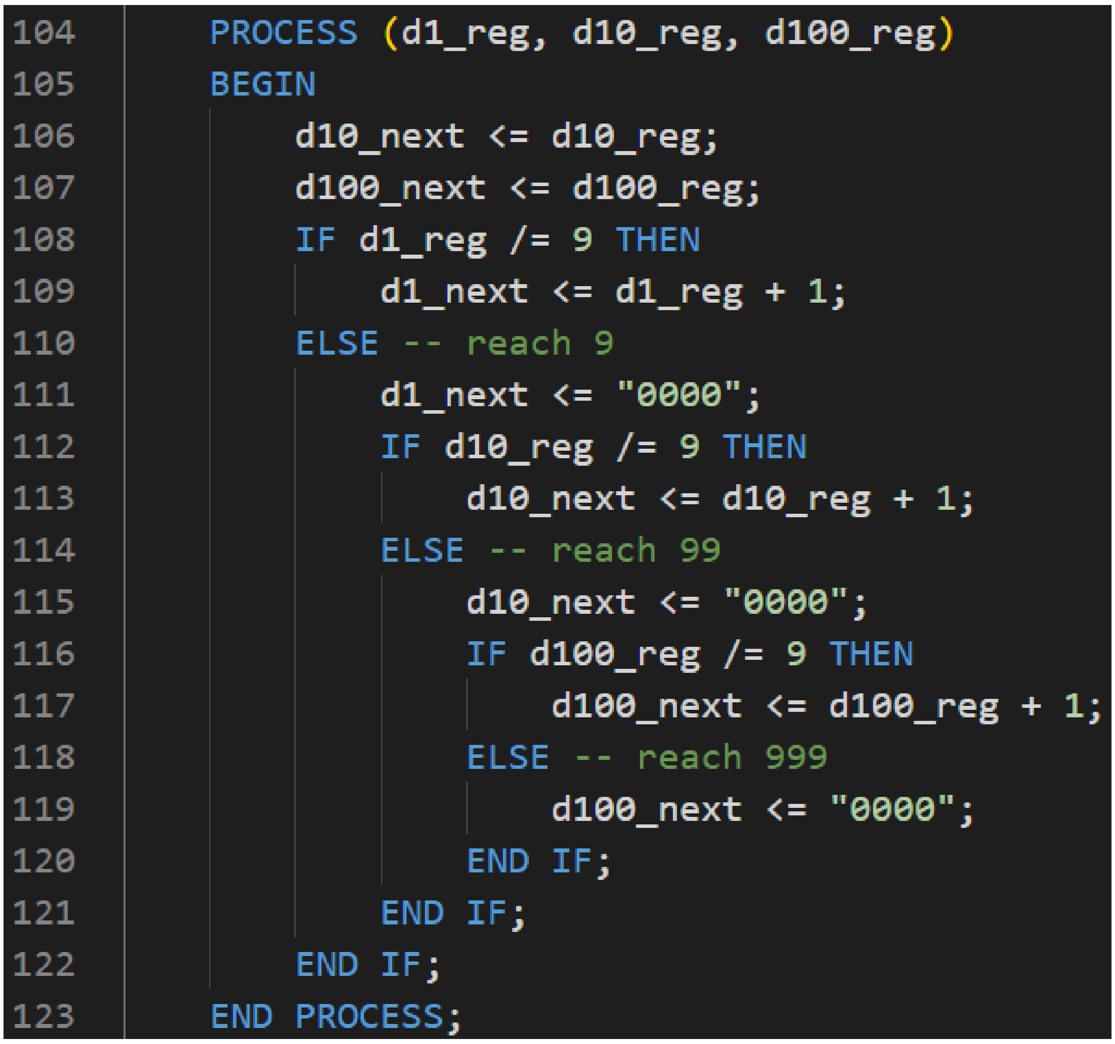
The sequential part is shown in **figure 3.3.3**



```
    -- register
    PROCESS (CLK, RST) IS
    BEGIN
        IF RST = '1' THEN
            -- counter_reg <= x"000000";
            IF Load = '0' THEN
                d1_reg <= "0000";
                d10_reg <= "0000";
                d100_reg <= "0000";
            ELSIF Load = '1' THEN
                d1_reg <= DataIn(3 DOWNTO 0);
                d10_reg <= DataIn(7 DOWNTO 4);
                d100_reg <= DataIn(11 DOWNTO 8);
            END IF;
        ELSIF CLK'event AND CLK = '1' THEN
            -- counter_reg <= counter_next;
            -- IF counter = 10000000 THEN
            IF trigger = '1' THEN
                d1_reg <= d1_next;
                d10_reg <= d10_next;
                d100_reg <= d100_next;
            END IF;
        END IF;
    END PROCESS;
```

Figure 3.3.3 Sequential part design of ‘three_digital_decimal_counter’

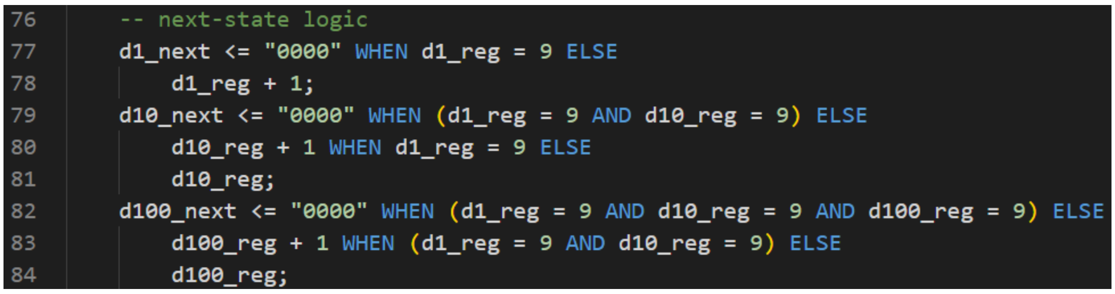
For the combinational part, we should describe the behavior of ‘dx_next’.
‘dx_next’ should change whenever ‘dx_reg’ changes. Thus the sensitive list should be ‘dx_reg’.
And we check each register. If ‘d1_reg’ is 9, then it should be set to 0 in the next period in logic, thus ‘d1_next’ is 0, otherwise ‘d1_next’ is ‘d1_reg’ + 1.
Normally, ‘d10_reg’ and ‘d100_reg’ should stay still. When ‘d1_reg’ reach 9, which means there will be a carry to decimal place, thus in this case, ‘d10_next’ should plus one. Also when ‘d10_reg’ reaches 9 and ‘d1_reg’ reach 9, which means we reach 99, there will be a carry to hundred place, thus in this case, ‘d100_next’ should plus one. Also when ‘dx_reg’ all reaches 9, which means we reach 999, all the register should set to 0 and we count again. Thus in this case, ‘dx_next’ should be 0.
The combinational part design is shown in **figure 3.3.4**



```vhdl
    PROCESS (d1_reg, d10_reg, d100_reg)
    BEGIN
        d10_next <= d10_reg;
        d100_next <= d100_reg;
        IF d1_reg /= 9 THEN
            d1_next <= d1_reg + 1;
        ELSE -- reach 9
            d1_next <= "0000";
            IF d10_reg /= 9 THEN
                d10_next <= d10_reg + 1;
            ELSE -- reach 99
                d10_next <= "0000";
                IF d100_reg /= 9 THEN
                    d100_next <= d100_reg + 1;
                ELSE -- reach 999
                    d100_next <= "0000";
                END IF;
            END IF;
        END IF;
    END PROCESS;
```

Figure 3.3.4 Combination design of ‘three_digital_decimal_counter’ (if arch)

Also, we can this combinational part in concurrent arguments. This is shown in **figure 3.3.5**

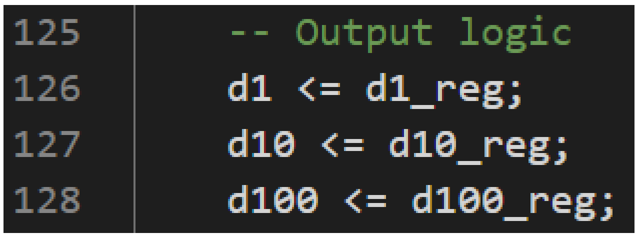


```vhdl
    -- next-state logic
    d1_next <= "0000" WHEN d1_reg = 9 ELSE
        d1_reg + 1;
    d10_next <= "0000" WHEN (d1_reg = 9 AND d10_reg = 9) ELSE
        d10_reg + 1 WHEN d1_reg = 9 ELSE
        d10_reg;
    d100_next <= "0000" WHEN (d1_reg = 9 AND d10_reg = 9 AND d100_reg = 9) ELSE
        d100_reg + 1 WHEN (d1_reg = 9 AND d10_reg = 9) ELSE
        d100_reg;
```

Figure 3.3.5 Combination design of 'three_digital_decimal_counter' (concurrent)

We find that it is always the case that concurrent arguments are much shorter than the if-arch. This may because concurrent focus on how to describe each signal independently. While if-arch arguments is case dependent, which means it will assign a signal in different cases.
In our design, we still use the if-arch arguments for uniformity.

Finally, we design the output logic. This part is quite simple, we just need connect the output of the register to the output port.
The output logic design is shown in **figure 3.3.6**

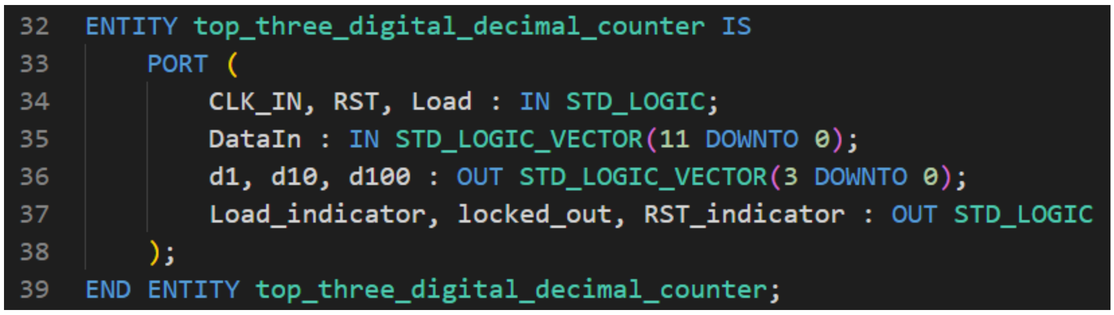


```vhdl
    -- Output logic
    d1 <= d1_reg;
    d10 <= d10_reg;
    d100 <= d100_reg;
```

Figure 3.3.6 Output design of 'three_digital_decimal_counter'

3.4. Source file 'top_three_digital_decimal_counter' design

Now we have all the three entities needed to realize our design. The top file now needs to connect them up and define the final inputs and outputs.
The ports in this entity includes 'CLK_IN', this is the clock signal from the on board oscillator. 'RST' for reset, connected to a switch. 'Load' for load mode, connected to a switch. 'DataIn' for preset data, connected to 12 switches. 'dx' where x is 1, 10, 100, connected to 12 LEDs, which shows the current counting number. And 'Load_indicator', 'locked_out', 'RST_indicator' for output to show whether these function is set up.
The port design is shown in **figure 3.4.1**

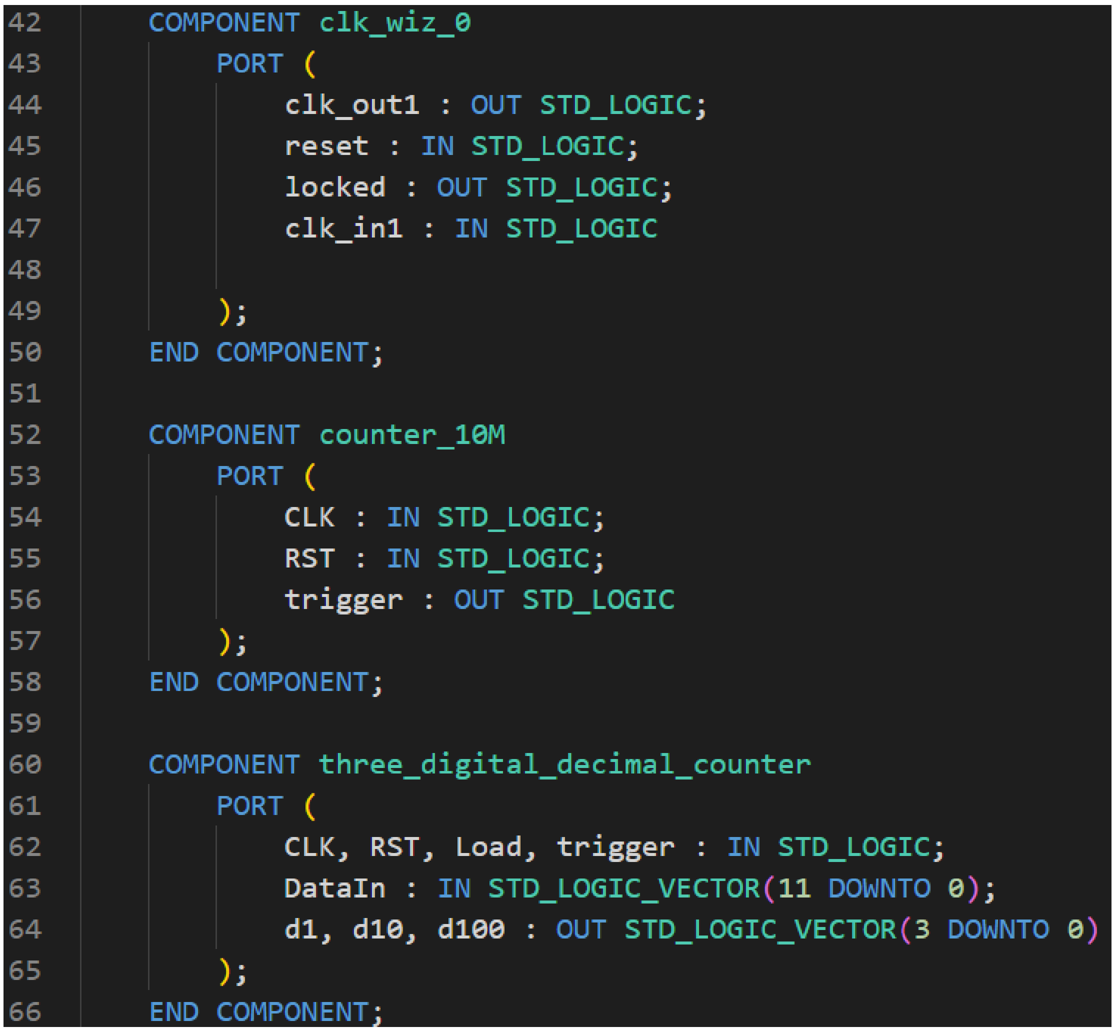


```vhdl
ENTITY top_three_digital_decimal_counter IS
    PORT (
        CLK_IN, RST, Load : IN STD_LOGIC;
        DataIn : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        d1, d10, d100 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        Load_indicator, locked_out, RST_indicator : OUT STD_LOGIC
    );
END ENTITY top_three_digital_decimal_counter;
```

Figure 3.4.1 Port design of 'top_three_digital_decimal_counter'

Then we define those three components, for further instantiation.

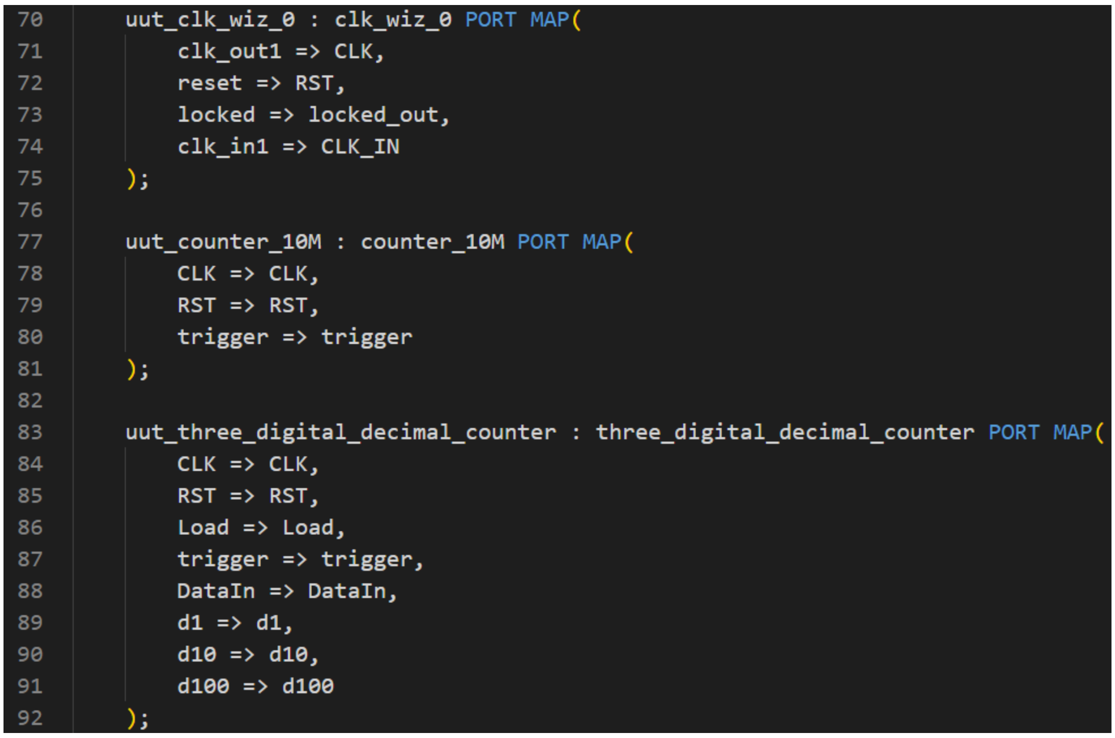
The component definition is shown in the **figure 3.4.2**



```
    COMPONENT clk_wiz_0
        PORT (
            clk_out1 : OUT STD_LOGIC;
            reset : IN STD_LOGIC;
            locked : OUT STD_LOGIC;
            clk_in1 : IN STD_LOGIC

        );
    END COMPONENT;

    COMPONENT counter_10M
        PORT (
            CLK : IN STD_LOGIC;
            RST : IN STD_LOGIC;
            trigger : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT three_digital_decimal_counter
        PORT (
            CLK, RST, Load, trigger : IN STD_LOGIC;
            DataIn : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
            d1, d10, d100 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;
```

Figure 3.4.2 component defination of 'top_three_digital_decimal_counter'

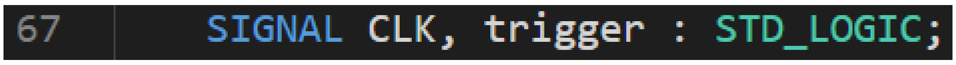
And their instantiation. This is shown in **figure 3.4.3**



```
    uut_clk_wiz_0 : clk_wiz_0 PORT MAP(
        clk_out1 => CLK,
        reset => RST,
        locked => locked_out,
        clk_in1 => CLK_IN
    );

    uut_counter_10M : counter_10M PORT MAP(
        CLK => CLK,
        RST => RST,
        trigger => trigger
    );

    uut_three_digital_decimal_counter : three_digital_decimal_counter PORT MAP(
        CLK => CLK,
        RST => RST,
        Load => Load,
        trigger => trigger,
        DataIn => DataIn,
        d1 => d1,
        d10 => d10,
        d100 => d100
    );
```

Figure 3.4.3 Instantiation part of 'top_three_digital_decimal_counter'

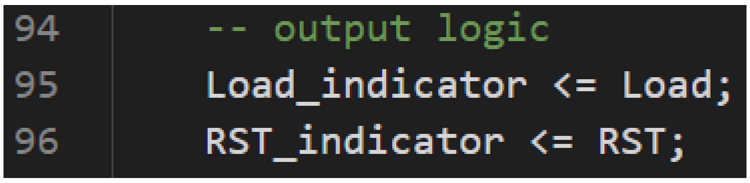
Note that we need intermedium signals 'CLK' and 'trigger'. 'CLK' is provided by instance uut_'clk_wiz_0' and need by 'uut_counter_10' and 'uut_three_digital_decimal_counter'. And 'tirgger' is generated by 'uut_counter_10 ' and needed by 'uut_three_digital_decimal_counter'. Thus we have to define these two signals. The definition of the signals is shown in **figure 3.4.4**



```
    SIGNAL CLK, trigger : STD_LOGIC;
```

Figure 3.4.4 Signal definition of 'top_three_digital_decimal_counter'

And finally define the two indicators



```
    -- output logic
    Load_indicator <= Load;
    RST_indicator <= RST;
```

Figure 3.4.5 Indicator output definition of 'top_three_digital_decimal_counter'

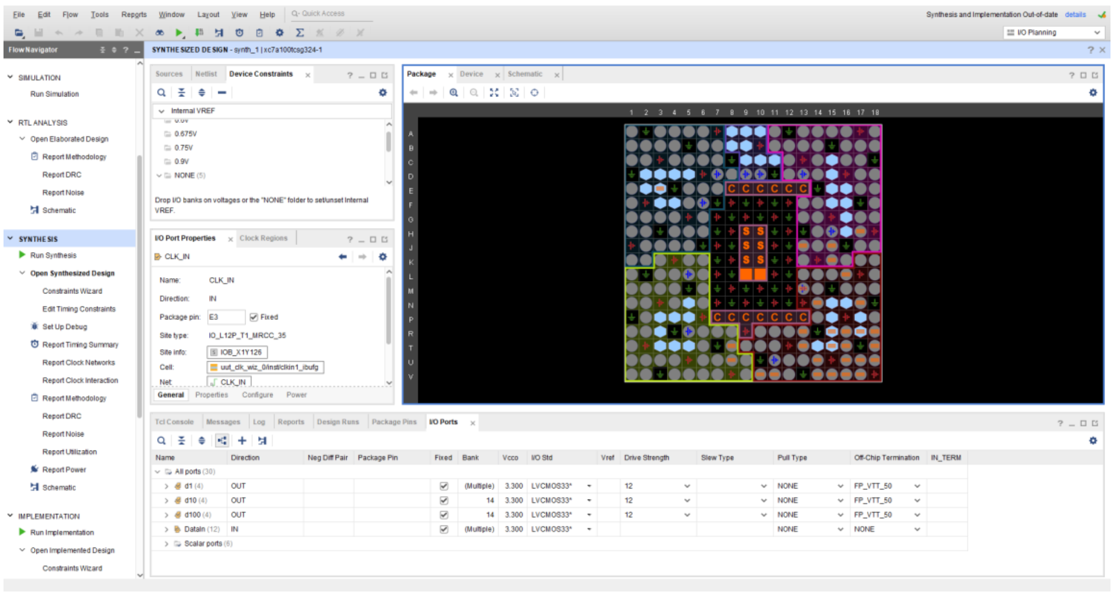
4. Constrain Design

Constrain includes many parts, including place constrains, routing constrains, IO port constrains and timing constrains. In this part, we only design the IO port constrains. Because it is essential to generate bit stream and program device.

Note that for a normal project, we may first exam the function of the design by simulation first, then bounding the IO ports. But I have examined it before and for better organization of this report, we introduce the constrain design first.

When we need to program our design into the hardware, we need associate the IO port we defined in the top file and the pins provided by the hardware. And the normal association is described by the constrain file.
There are generally three methods to generate the constrain file. The first one is using the constrains wizard in the synthesis and implement, this includes all the constrain we introduced in the beginning of this section. The second method is using the IO planning. This is a GUI window provided by Vivado and we can use this to match the IO ports easily. The third part is to create an empty constrain file and all written in hand. Here we focus on the second method because it is the most practical when our projects are small.

The overall interface of the IO planning layout is shown in **figure 4.1**



Internal VREF: 0.675V, 0.75V, 0.9V, NONE (5)

Drop I/O banks on voltages or the "NONE" folder to set/unset internal VREF.

I/O Port Properties: CLK_IN

Name: CLK_IN
Direction: IN
Package pin: E3 ✓ Fixed
Site type: IO_L12P_T1_MRCC_35
Site info: IOB_X1Y126
Cell: uut_clk_wiz_0/inst/clkin1_ibufg
Net: CLK_IN

I/O Ports

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	Vcco	I/O Std	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
All ports (30)													
d1 (4)	OUT			✓	(Multiple)	3.300	LVCMOS33*		12		NONE	FP_VTT_50	
d10 (4)	OUT			✓	14	3.300	LVCMOS33*		12		NONE	FP_VTT_50	
d100 (4)	OUT			✓	14	3.300	LVCMOS33*		12		NONE	FP_VTT_50	
DataIn (12)	IN			✓	(Multiple)	3.300	LVCMOS33*				NONE	NONE	
Scalar ports (6)													

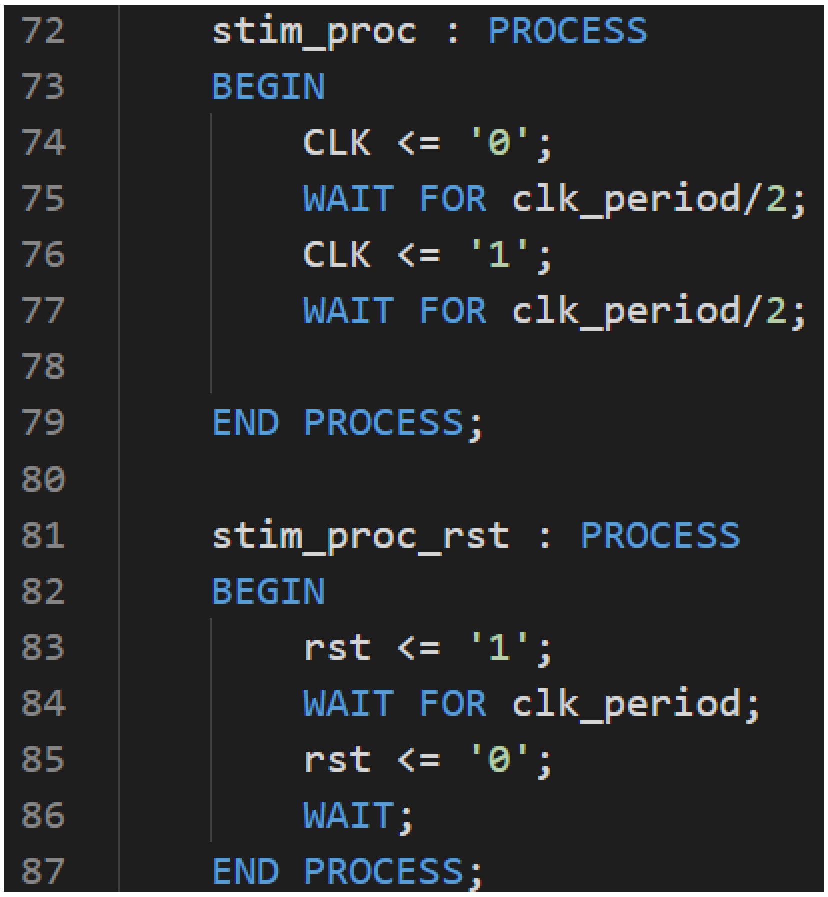
Figure 4.1 Layout of IO planning

The right package view shows the pins in the back of the chip. Each pin is connected to an outer peripheral and an inner bank port. The outer part is unchangeable because this is connected by PCB. What we can do is to associate our entity ports to the bank ports.
The window in the bottom is the IO ports planning window, it shows the ports we defined in the top file. We can define their corresponding bank ports and usage voltage here. After that we just simply save the planning to a constrain file and the constraining process is done.

5. Simulation Design

The stimulation is quite easy. We only need to create the clock signal with 100MHz (note that other frequency is not allowed because our IP core has strict input clock frequency requirement. Other clock frequency would work but there may be unexpected errors). And a reset signal. And we only set ‘RST’ to 1 for just one clock period.

Figure 5.1 illustrates the stimulation part in the simulation file.



```
    stim_proc : PROCESS
    BEGIN
        CLK <= '0';
        WAIT FOR clk_period/2;
        CLK <= '1';
        WAIT FOR clk_period/2;

    END PROCESS;

    stim_proc_rst : PROCESS
    BEGIN
        rst <= '1';
        WAIT FOR clk_period;
        rst <= '0';
        WAIT;
    END PROCESS;
```

Figure 5.1 Stimulation part in the test bench

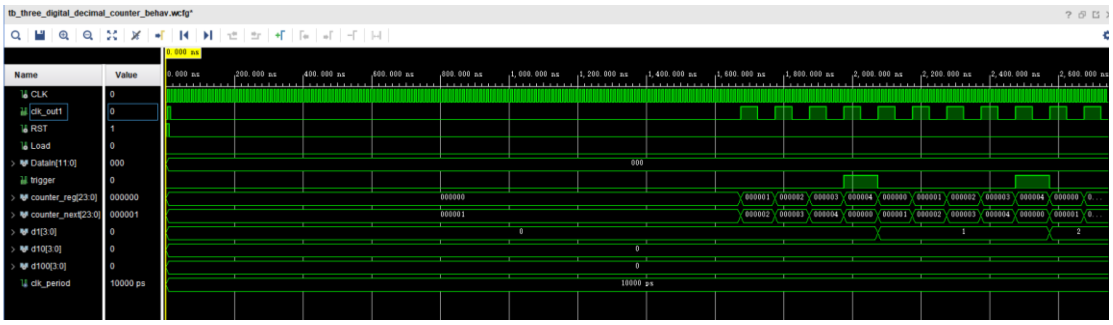
Where ‘clk_period’ = 10 ns.

6. Behavior Simulation and RTL Analysis

6.1. Simulation result

For we have to count from 0 to 999, it is unwisely to simulate all the 1000 cases. In this section we focus on some critical points: The first count when we are counting; the case when we reach 9; the case when we reach 99; the case when we reach 999; the case we set reset; the case when we use load mode and reset. Here are the results.

Figure 6.1.1 illustrates the case when we just start the program.



tb_three_digital_decimal_counter_behav.wcfg*

Name	Value
CLK	0
clk_out1	0
RST	1
Load	0
DataIn[11:0]	000
trigger	0
counter_reg[23:0]	000000
counter_next[23:0]	000001
d1[3:0]	0
d10[3:0]	0
d100[3:0]	0
clk_period	10000 ps

Figure 6.1.1 Start counting

For simulation, the counter reset when it reaches 4.
For the IP core needs some time to initiate, we can see that during the first 1650ns the output of the clk_wiz is low, and after that it works normally.
The 10M counter will count from 0 to 4 in this program, thus there are totally 5 numbers it has to count, this also explains why the main counter change to its next state after 5 clock period provided by the clk_wiz.

Figure 6.1.2 illustrates the case when we reach 9.

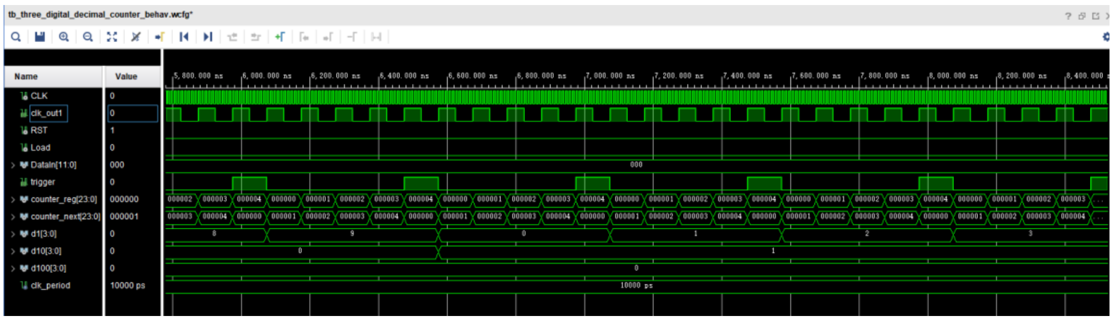


Figure 6.1.2 Case when we reach 9

We can see that when we reach 9, output ‘d1’ resets to 0 and ‘d10’ change to ‘1’ as expected.
When it counts to 11, the timestamp is 7,075 ns. And from the figure above, we can know that when we count to 1, the timestamp is 2,075 ns. This tells us the count need 5 us to count 10 numbers, which equals to 500ns to count one number. 500ns is 5 times clock period provided by the clk_wiz. This behavior meets our design.

Figure 6.1.3 illustrates the case when we reach 99.

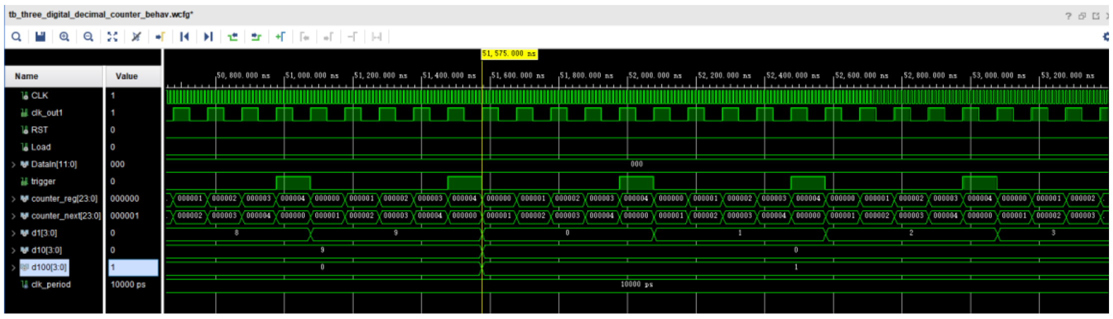


Figure 6.1.3 Case when we reach 99

We can see that when we reach 99, output ‘d1’ and ‘d10’ reset to 0 and ‘d100’ change to ‘1’ as expected.
When it counts to 101, the timestamp is 52,075 ns. And from the figure above, we can know that when we count to 1, the timestamp is 2,075 ns. This tells us the count need 50 us to count 100 numbers, which equals to 500ns to count one number. This behavior meets our design.

Figure 6.1.4 illustrates the case when we reach 999.

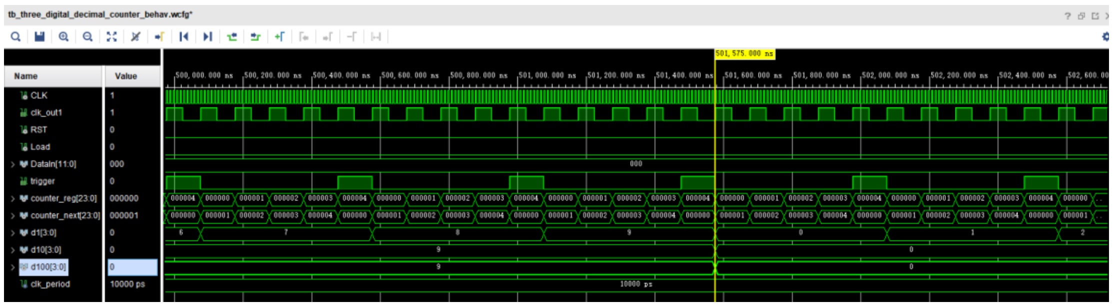


Figure 6.1.4 Case when we reach 999

We can see that when we reach 999, all the outputs reset to 0

When it counts to 1001, the timestamp is 502,075 ns. And from the figure above, we can know that when we count to 1, the timestamp is 2,075 ns. This tells us the count need 500 us to count 1000 numbers, which equals to 500ns to count one number. This indicates that our logic in counting is right.

Figure 6.1.5 illustrates the case when we push the reset button for one clock period provided by the oscillator.

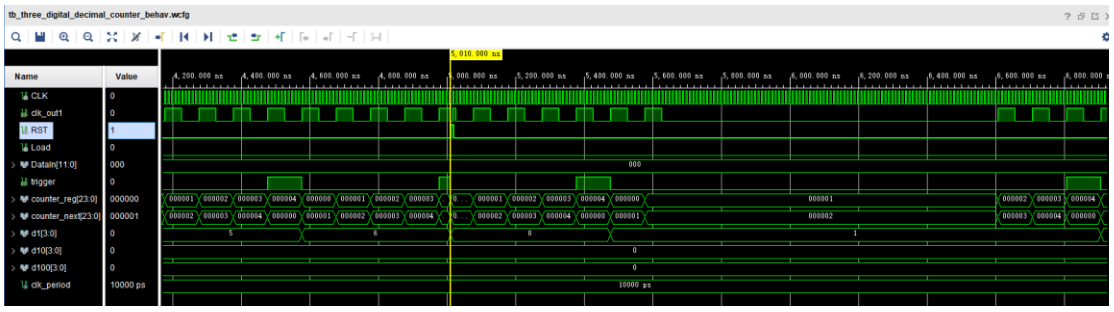


Figure 6.1.5 Case when we do a short reset

We can find that after the reset, our counters all been reset to its initial state immediately. However the clock IP core needs some extra time to be reset, and this tells us why the output clock continues for about 500 ns. And these extra clock signal makes the counter continues to count for 1. Also because the IP core needs time to initiate, there is time it stay low and our counters stay in the state before the IP core totally reset.

This is not we hope to see, this reset delay in the IP core makes the counting inaccurate. But fortunately we can not get a reset signal so short. The reset signal is normally given by real button and most of the time the reset signal stays high in a time period in scale of millisecond.

Figure 6.1.6 illustrates the case when we change the reset period to 1000 ns

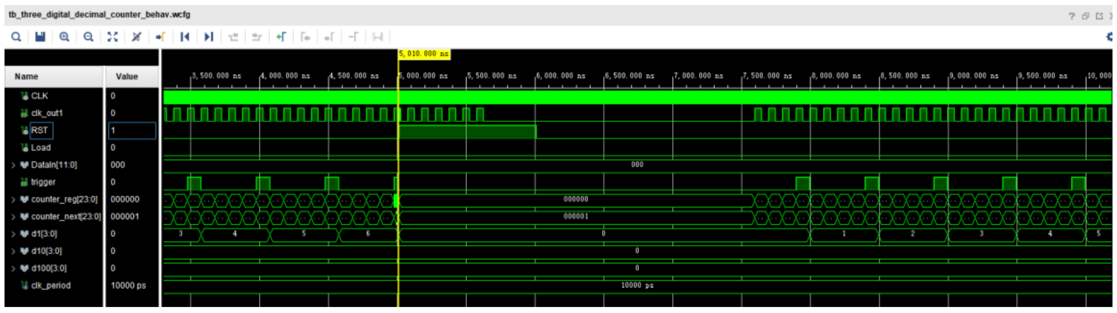


Figure 6.1.6 Case when we increase the reset period

We can find in this case, the reset works fine and there is no unexpected count before the IP core gets initiated.
This is because the IP core will start reset after reset signal is trigger and it needs a relatively constant time to be totally reset. However our counters are immediately reset. The reset delay causes the extra count. Longer reset time will keep the counters stay in the reset state after the IP been totally reset.
Figure 6.1.7 illustrates the case when we use load mode to reset. The inputs for date is 996

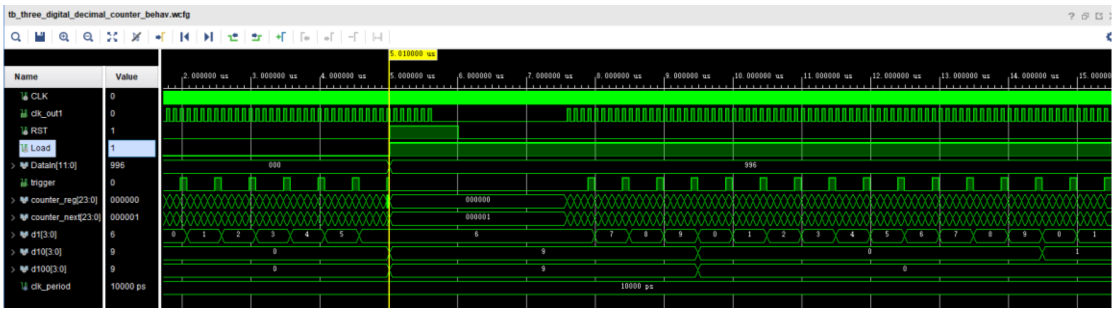


Figure 6.1.7 Case for load mode

After reset, the output data is reset to the data we give in. And it count from then on. Our design is correct.

6.2. RTL analysis

Figure 6.2.1 illustrates the RTL schematic in top view.

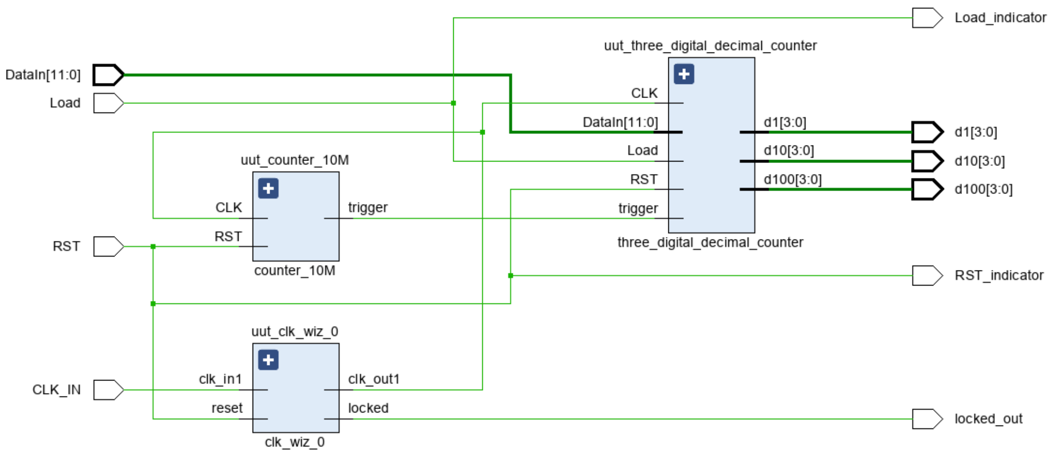


Figure 6.2.1 RTL schematic in top view

It shows the connection of the instances we create and those input and output ports. We can use schematics to check whether those instances and ports connected as our expectation.
Figure 6.2.2 illustrates the schematic inside the instance 'uut_counter_10M'.

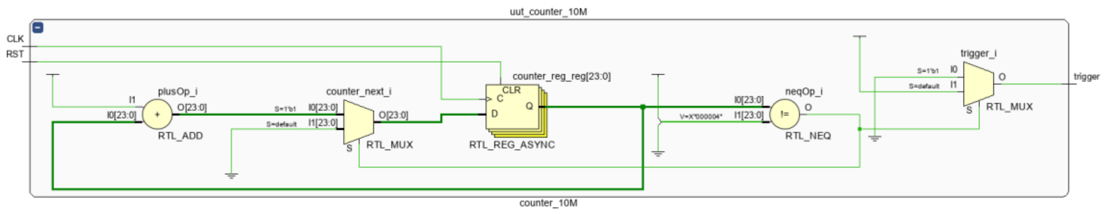


Figure 6.2.2 Schematic for 'uut_counter_10M'

We can see that there is a register named ‘counter_reg_reg’, which means it is the register for signal ‘counter_reg’. Its clock signal and reset signal are all from the inputs.
The output of the register connected to a comparator and an adder. The comparator is used for compare whether the output of the register is equal to the value we set (in simulation it is 4 and in real it is 9,999,999). The output signal is connected to a multiplexer named ‘trigger_i’ and another multiplexer named ‘counter_next_i’. This indicates that the comparator control the behavior of signal ‘counter_next’ and ‘trigger_i’. When they are not the same, the comparator gives out zero and the trigger gives out 0 and the ‘counter_next’ is the value given by the adder. And when the ‘counter_reg’ is the same with our preset value, the comparator gives out 1. This makes the trigger gives 1 to the output to trigger the main counter to count for 1 time. And also the ‘counter_next’ becomes one because we select the other date in the inputs of the multiplexer.

Figure 6.2.3 illustrates the schematic for ‘uut_three_digital_decimal_counter’

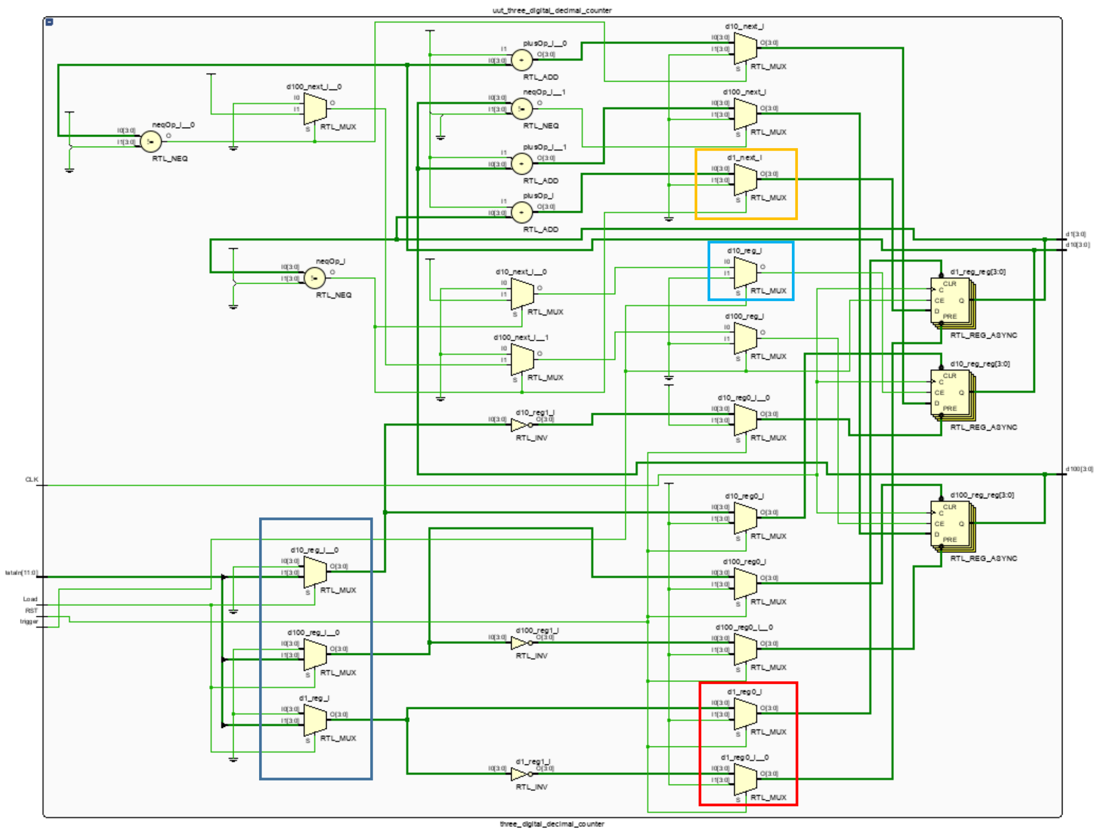


Figure 6.2.3 Schematic for ‘uut_three_digital_decimal_counter’

This schematics seems more complex than the previous one. This because it has three independent counter and it has an additional function: load data if is indicated.
In the dark blue rectangle is the three multiplexers used for decide whether use the input date as the reset date. We can see that if the ‘Load’ is low, then they select the low as output. And when the ‘Load’ is high, then they select the input date as their outputs.
Now we focus on the least significant digit for it is quite similar in the two other digits. In the red rectangle is the reset multiplexers. The upper multiplexer decides whether we reset for each bit digit in the register and the lower multiplexer decides what value will be preset to the register. There is an inverter ahead the lower multiplexer is because the input preset of the register is low active.

In the orange rectangle, the multiplexer decides what to give to the signal ‘d1_next’. It receive signal from the comparator and to choose whether the result from the adder or 0.
It is similar when we focus on ‘d10’. But there is something more. In the light blue rectangle, there is the multiplexer that control the clock enable signal (when this is high, the clock is activated and the register will update its value in the next clock edge, otherwise the register stays unchanged) for the register for ‘d10’. This is because ‘d10’ will get its next value when ‘d1’ reach 9. There needs a comparator that compare the output value of ‘d1’ and 9.
And for ‘d100’ there are two multiplexers to do this, a multiplexer for ‘d1’ and the other one for ‘d10’.

Similar with the case in the ‘counter_10M’ the outputs of the registers are given to an adder and a comparator respectively. The adder add one to the output value and as the pre-value for the next value for the registers. And the comparator gives output result whether the register counts to its limit and has to reset, and also gives a signal to its next digit.

7. Post-synthesis timing Simulation and Synthesis Analysis

7.1. Simulation result

The results for the cases discussed in the previous section are quite similar, thus we only focus on something different. Of course, there are delays for each signal when they change, but the behavior is the similar.
The most different part is when ‘Load’ signal is high, there are some intermediate states for the outputs of top file due to the timing hazard.
Figure 7.1.1 illustrates the intermediate state mentioned before

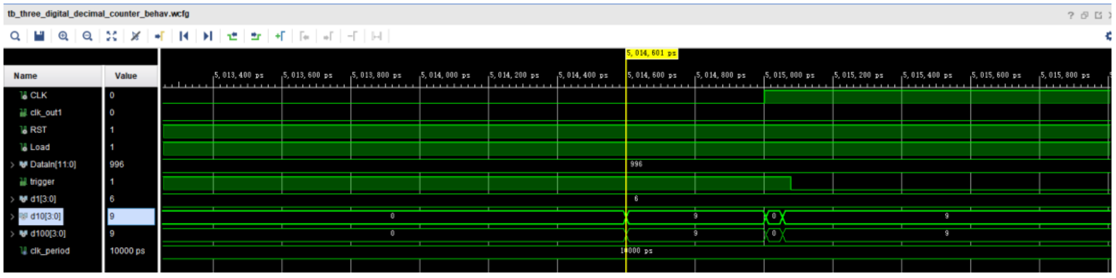


Figure 7.1.1 Post-synthesis with tb file the same as section 6.1.6 (timing hazard for ‘d10’, ‘d100’)
We can see clearly there is a very short state 0 after ‘d10’ and ‘d100’ change to 9.
This is also happened in ‘d1’, **figure 7.1.2** shows this.

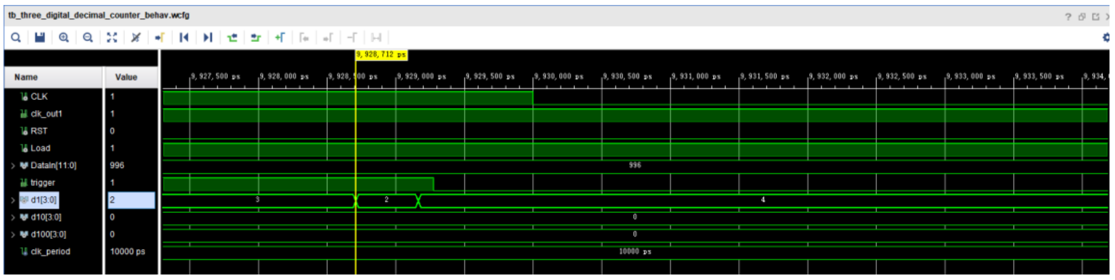


Figure 7.1.2 Post-synthesis with tb file the same as section 6.1.6 (timing hazard for ‘d1’)
There is a very short state 2 before ‘d1’ changes to its desired value.

The reason will be discussed in the later synthesis analysis.

7.2. Synthesis analysis

Figure 7.2.1 illustrates the synthesis schematic in top view.

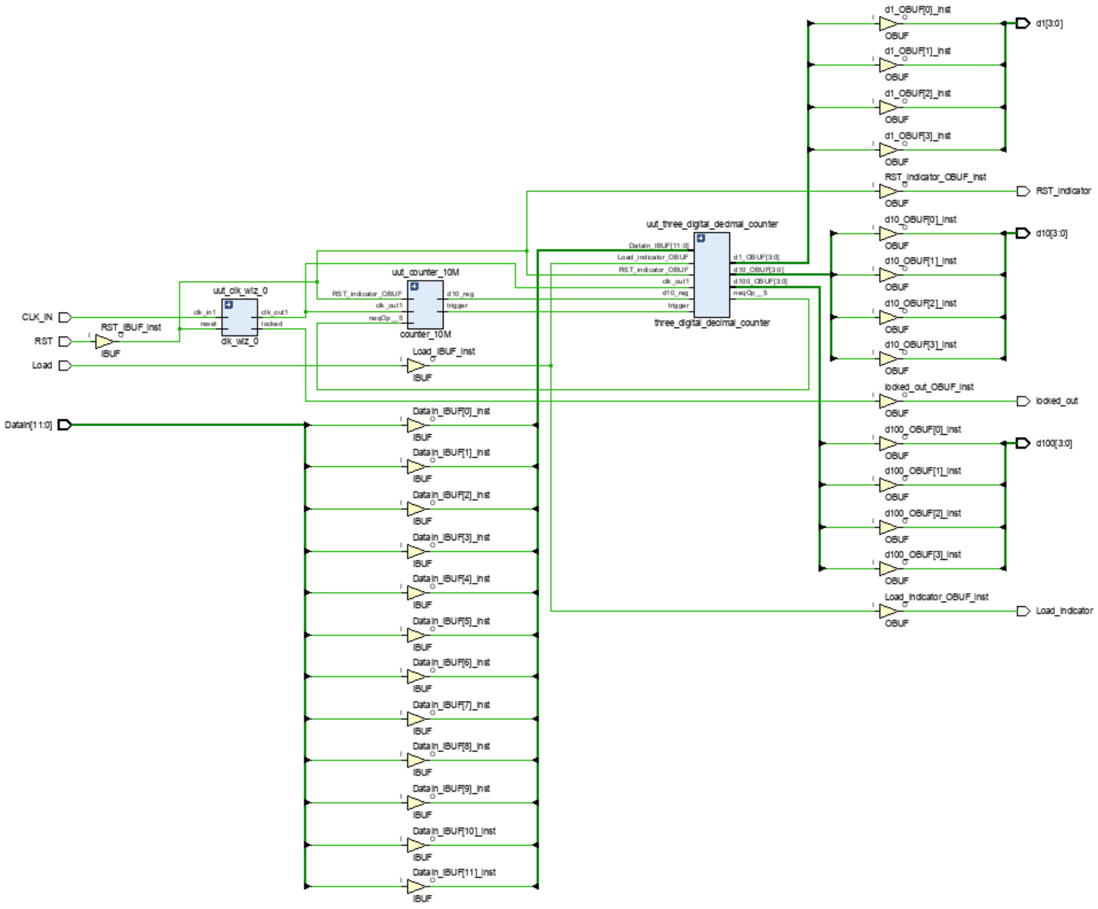


Figure 7.2.1 Synthesis schematic in top view

This schematic shows the instances connection relations and it adds input and output buffers.

Figure 7.2.2 illustrates the synthesis schematic of the instance ‘counter_10M’

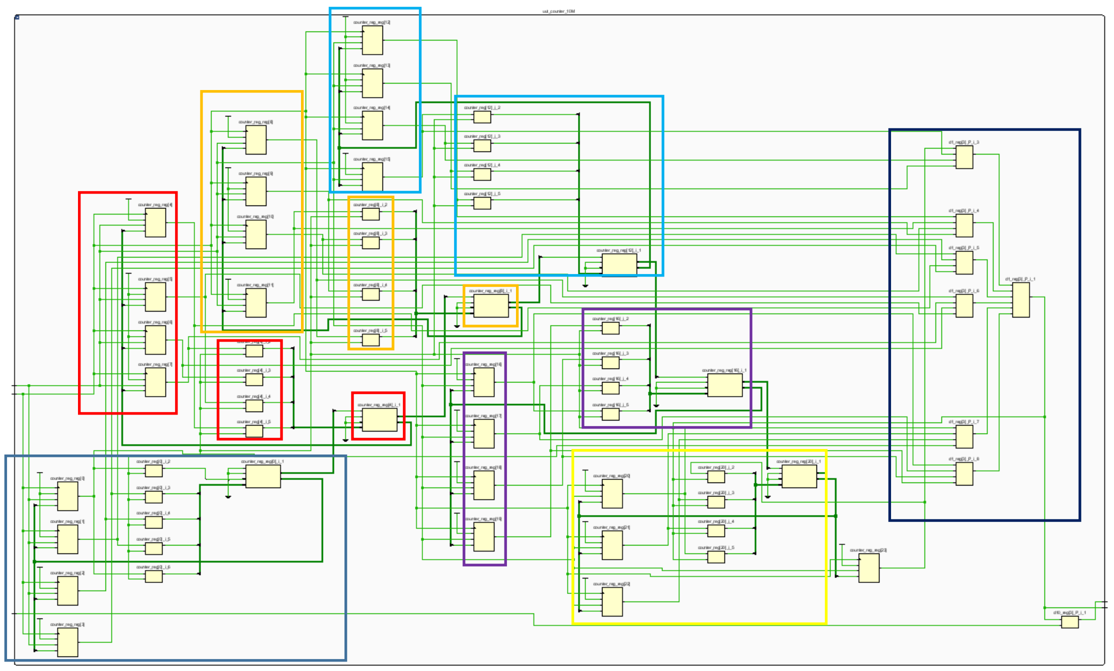


Figure 7.2.2 Synthesis schematic of ‘counter_10M’

The counter has 24 digits and thus has 24 registers. For better explanation, we divided the cells into six groups, which group contains 4 registers, 5 LUT2s (Look Up Table with 2 inputs) and 1 AFCL (Advanced Fast Carry Logic, this is used for arithmetical summation). Note that the part in the black rectangle are comparators, realized by LUTs.

The groups are surrounded by rectangles with different colors. Those group all has the similar function. Now we focus on of them to better understand how it works.

Figure 7.2.3 illustrates the low four digits in the ‘counter_10M’

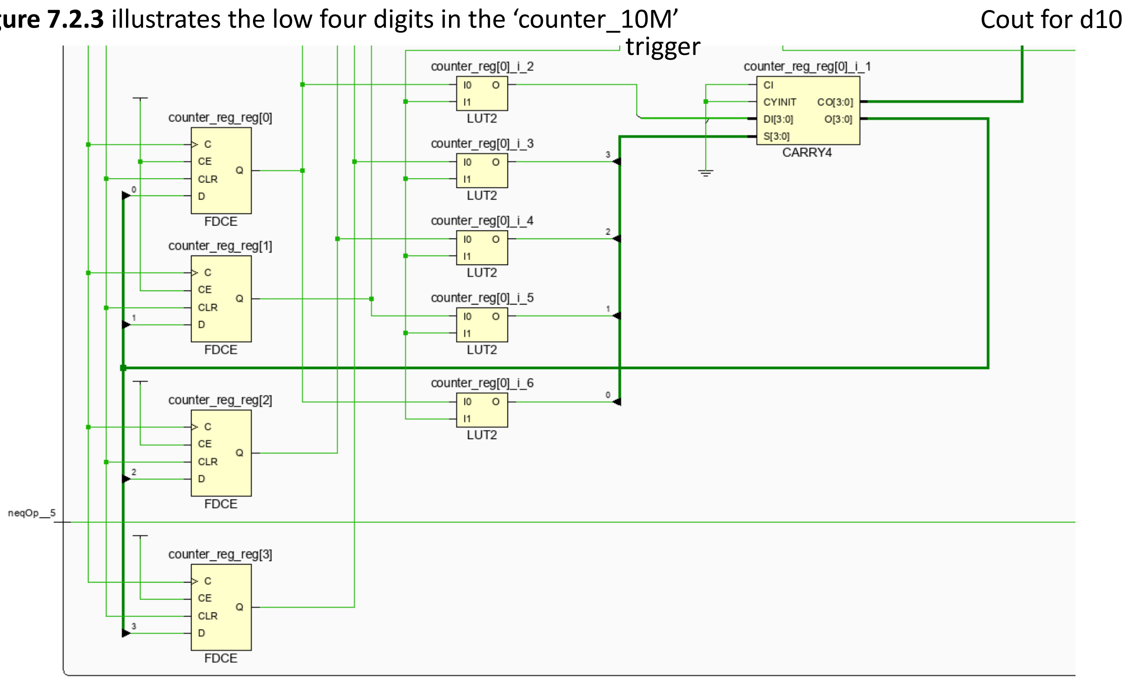


Figure 7.2.3 Lowest four digit of ‘counter_10M’

The FDCE is D-type Flip-Flop with Clock Enable and asynchronous reset. This is the basic register elements in the real hardware. The LUT2 are the multiplexers and the AFCL is the adder.

The logic here is quite simple, the output of FDCE are given to LUT2s and they decide what to give out to the AFCL. If the counting is going on, then the LUT2s outputs the register values directly. And the AFCL plus one to the value of the current registers. While the trigger signal is set, all the LUTs set to 0. And we also find that the AFCL has a carry digit to the next AFCL for ‘d10’, this is the signal that tells the next digit that this four register has reached 15 and needs to reset and gives out a carry.

Figure 7.2.4 illustrates the synthesis schematic of ‘three_digital_decimal_counter’

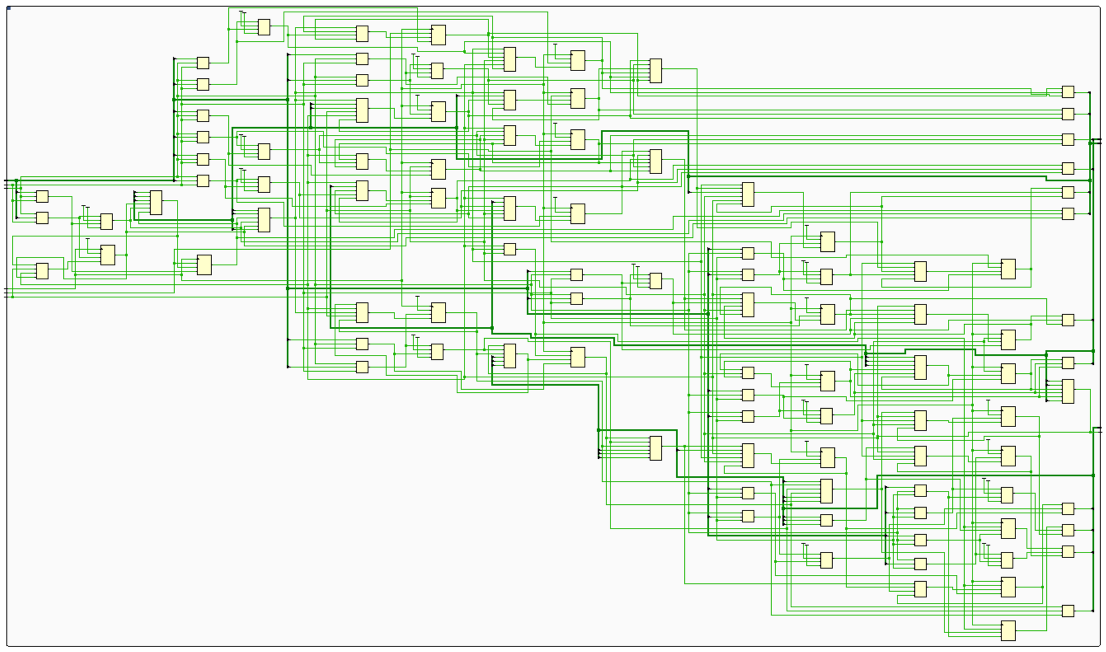
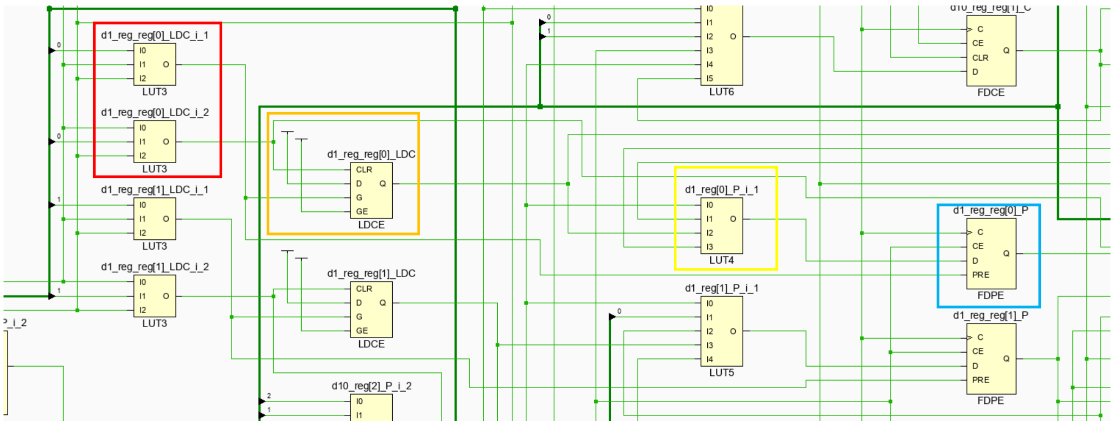


Figure 7.2.4 Synthesis schematic of ‘three_digital_decimal_counter’

The automatic placing and routing make the schematic quite in a mass and it is very hard for me to group them. Because there are 3 digitals in decimal and each of them has 4 registers to represents.

Here we only focus on one digital in binary, ‘d1[0]’

Figure 7.2.5 illustrates the 3lowest digit in the 12 bit long registers which represent the ‘dx’.



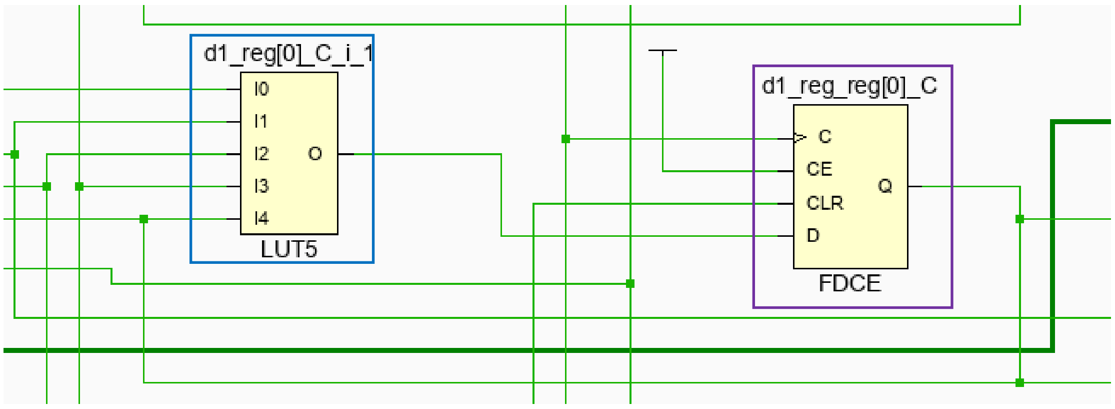


Figure 7.2.5 Lowest digit of 'three_digital_decimal_counter'

The LUT3s surrounded by a red rectangle are multiplexers used for 'RST' and 'Load' selection. It decides what value should be send to the register later with their inputs 'RST' and 'Load'.
The LDCE (D-type Latch with Clock Enable and asynchronous reset) in the organ rectangle is a buffer for the value given to the real signal 'd1[0]'.
The LUT4 in the yellow rectangle is the multiplexer which decide what value will really goes to the signal 'd1[0]'.
The FDPE (D Flip-Flop with asynchronous Preset and clock Enable) in the light blue rectangle is the flip-flop that really store value of 'd1[0]'.
The LUT5 in the dark blue rectangle is the multiplexer which deicide what value will be sent to the 'd1[0]_next'.
The FDCE in the purple rectangle is the flip-flop that store the value of 'd1[0]_next'.

Thus the core idea in the synthesis is use LUTs to replace the multiplexers and comparators in the RTL, and use different types of flip-flop to replace the registers in RTL.

Now back to our question mentioned in the simulation part of this section: why there is an intermedia state for the outputs when 'Load' signal is high.
We can see that the 'RST' and 'Load' are controlled by two different LUTs, and they have a sequential order. In logic, the circuit first check the 'Load' and then check 'RST'. If 'Load' is low, then it has only to check 'RST'. Thus in our previous simulation, when 'RST' and 'Load' are low, the counter does a good job, while when 'Load' gets high, though 'RST' is low, the circuit still has some time to check the signal of 'Load'. The delay caused by the delay of LUT and the LDCE that pass the signal from 'RST' and 'Load' to 'd1'.
There is another critical different to notice. The signal 'RST' and 'Load' are given to the two LUT3 in the red rectangle simultaneously. And their outputs will send to later flip-flops.
When 'Load' is high, the output of the LUTs in the red rectangle stay unchanged but their value to the later flip-flop indicates that they are ready to be preset. And when the trigger signal comes and trigger the next count, there is some LUTs has a delay signal, which will make the flip-flop think they are in a very short preset state. However when 'Load' is low, the flip-flop will never thinks they are ready for preset.
And this causes the intermedia state when we do the simulation.

8. Post-implementation timing Simulation

We do not do post implementation analysis because this kind of analysis in the hardware level are much quite tedious and it is totally unnecessary for such a simple project.
Similarly, the results in this section are generally the same as the previous sections, we only focus on the something different
Figure 8.1 illustrates the one of the timing hazard

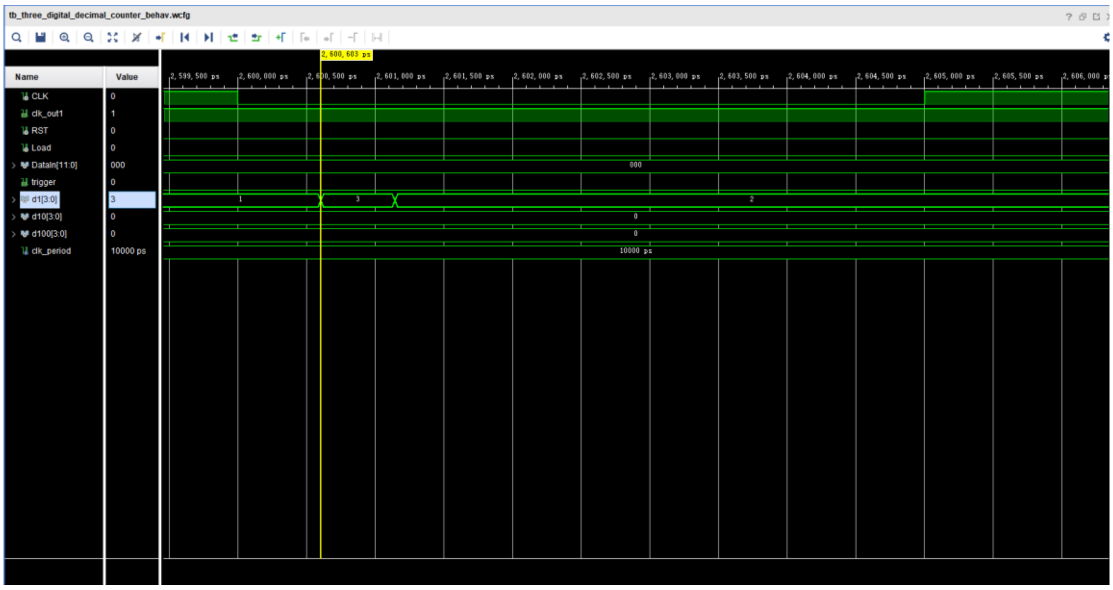


Figure 8.1 Post-implementation with tb file the same as section 6.1.6 (timing hazard for 'd1')

We notice that there are intermedia state even when 'Load' signal is low. This is because the delay of lines and LUTs themselves cause the LUTs give an intermedia result in a quite short period.
And when 'Load' signal is high, we can see one of the result in **figure 8.2**

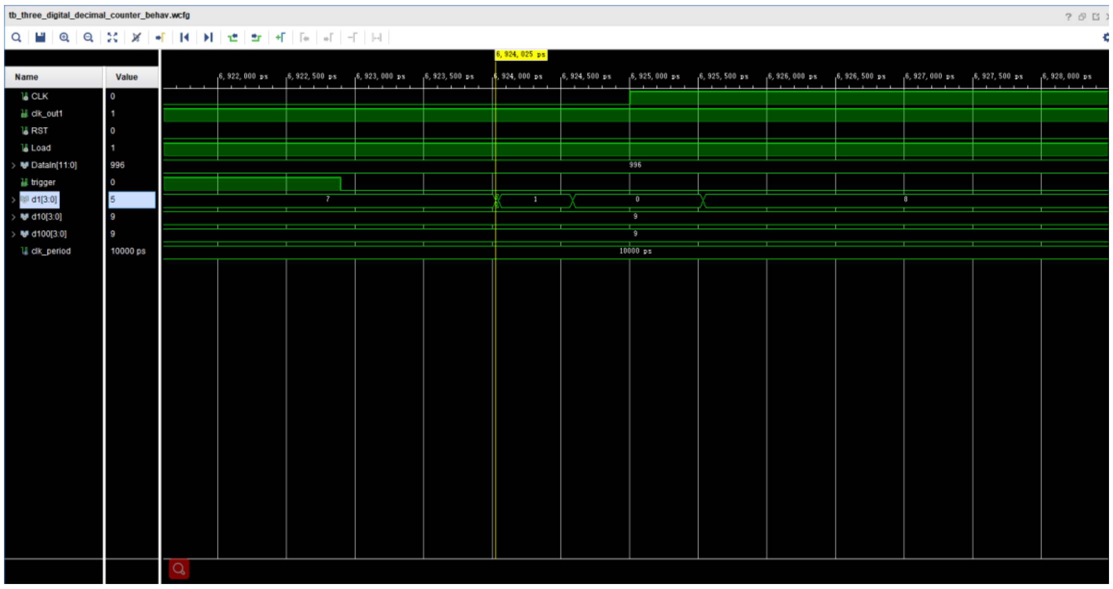


Figure 8.2 Post-implementation with tb file the same as section 6.1.6 (timing hazard for 'd1')

There are more intermedia states when 'd1' changes its value. However when we check more those intermedia states we found that when 'Load' is high, there are maximum two intermedia states, which means the delays behaves as our expectation.

9. Result of on-board Test

The result is shown in **figure 9**. The details are in the in-class checks by TAs.

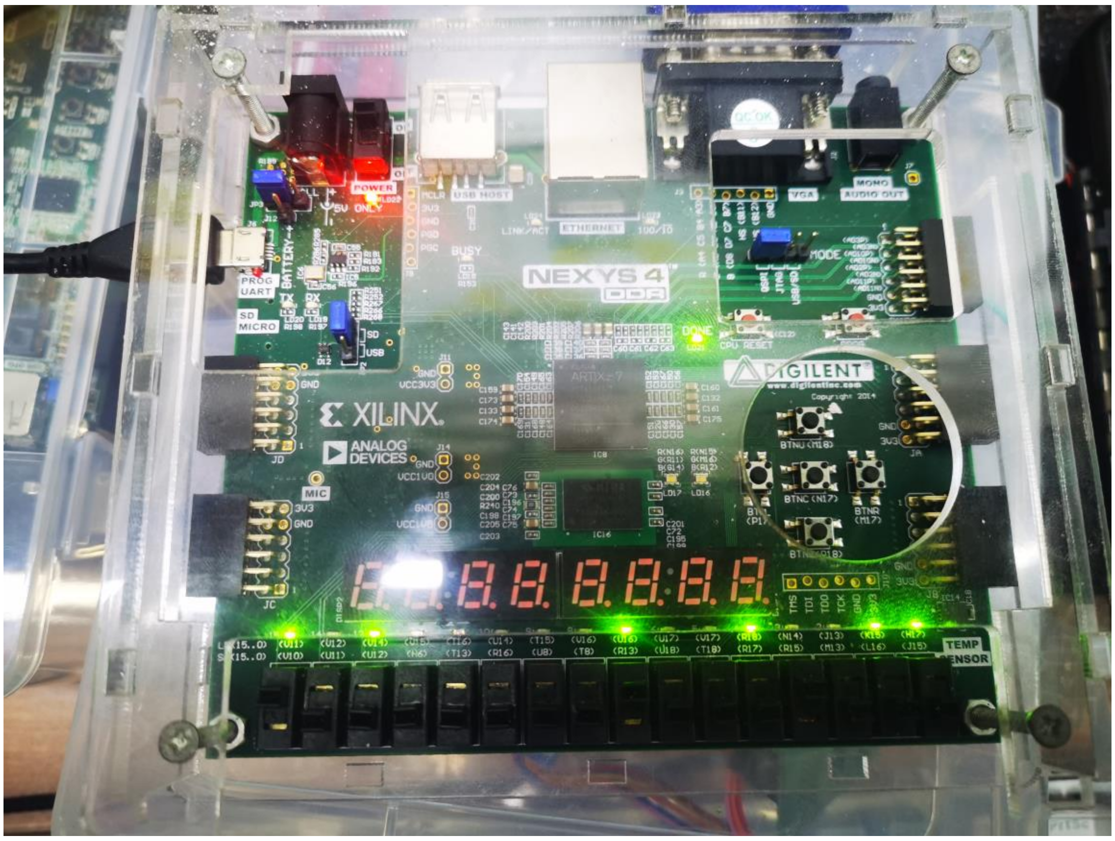
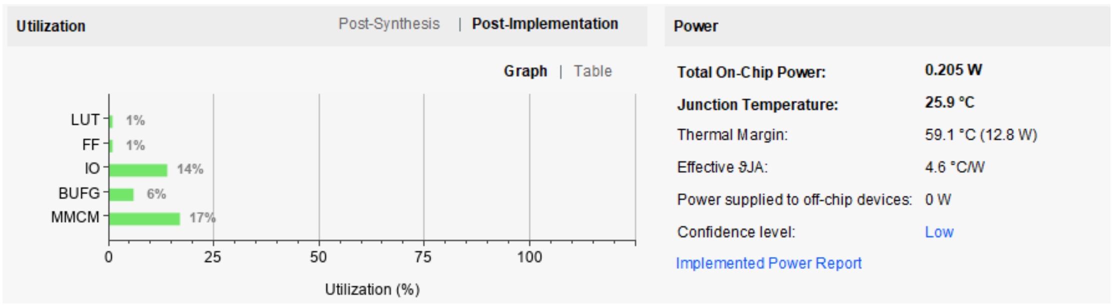


Figure 9 On-board test

10. Resource Utilization and Power States

Figure 10 illustrates the resource utilization and power analysis.



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

Graph | Table

	Utilization (%)
LUT	1%
FF	1%
IO	14%
BUFG	6%
MMCM	17%

Power

Total On-Chip Power:	**0.205 W**
Junction Temperature:	**25.9 °C**
Thermal Margin:	59.1 °C (12.8 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Figure 10 resource utilization and power analysis

Because this is a high clock dependent project and we use the IP core ‘clock_wizard’ to implement very accurate clocks, thus we have a relatively high usage of MMCM (Mixed-Mode Clock Manager) And we use a lot of switches and LEDs, thus the IO and buffer utilization is also relative high. For the counting is quite simple in logic, thus the usage of LUT and FF is very low.

We use a small scale of flip-flop thus the total power is low.

11. Conclusions

VHDL has its own logic in design. Though there are many coding method, but with standard coding method we can avoid lots of unexpected errors and mistakes and it can also makes the whole logic clear.

The idea to design top file is brilliant, we get all function done by each instance the and top file only do the connections between instances and ports. This reduce the mistakes and easier to debug.

Though this is a simple project, this lab needs us to go through a complete procedure in digital system design, from source file design to simulation file design and simulations analysis to find the bugs and to constrain file design to program the hardware.

The delays of LUTs and flip-flop are quite common and the line delay will intensify the delays. In this lab our flip-flop stage is short thus the delay can be neglected. But when we comes to a very large project the delay will accumulate larger than one clock period. This could probably cause the system stall.

Analysis of synthesis and implementation schematics and placing and routing in the device contains large workload. The compiling of the Vivado sometimes make the synthesis schematic unfriendly to human and sometimes even unreadable. Debug from the schematic can surely find bug but takes a lot of time. But this seems reasonable, people create this kind of software to help us generate the designs, we do not need to find the very basic design by ourselves.

PS: we do not check difference of the synthesis and implementation when we use concurrent method and if-arch method.