FPGA experiment report

Task: FPGA

Student 1 name: Zvi Gused ID: 207323247
Student 2 name: Yonatan Reches ID: 325105641

Part 1 - Status update

Fill in the following table according to your progress at the time of the submission.
Each row should refer to one of the modules or a simulation. For example, write FA under module name and refer to FA.v and FA_tb.v in that line.
The status column should be filled with one of the following:

- Done - If both are written and simulation passed.
- Partially done - If you started writing the module and simulation but it's not working properly.
- Not started - If you didn't get to that part of the task.

When the status is partially done, please use the notes/comments column to explain in a few more words what is the status of the module and what is the status of the simulation.

Fill here:

Module name/Step	status	notes/comments
FA	Done	
CSA	Done	
Limited Incrementor	Done	
Counter	Done	
Stash	Done	
Control	Done	
Debouncer	Done	
Seven Segment Display	Done	

Stopwatch	Done	
Setting up constraints	Done	
Implementing the design	Done	
FPGA programming and debugging	Done	

Issues:

In this section please describe the current issues you are facing.

- Per each module and simulation, if you receive any errors at the time of submission, please describe which errors and what they mean.

Fill here:

1. FA:
2. CSA:
3. Limited Incrementor:
4. Counter:
5. Stash:
6. Control:
7. Debouncer:
8. Seven Segment Display:
9. Stopwatch:
10. Setting up constraints:
11. Implementing the design:
12. FPGA programming and debugging:

Part 2 - Simulations and answers

In this section, please add the images for each simulation as described in the report. Remember to use annotations and explain in your own words what is shown in each simulation.
In some of the tasks in the report, there are also questions that you should answer here.
Make sure to include images of all simulations running at the time of submission, even if they are not correct.

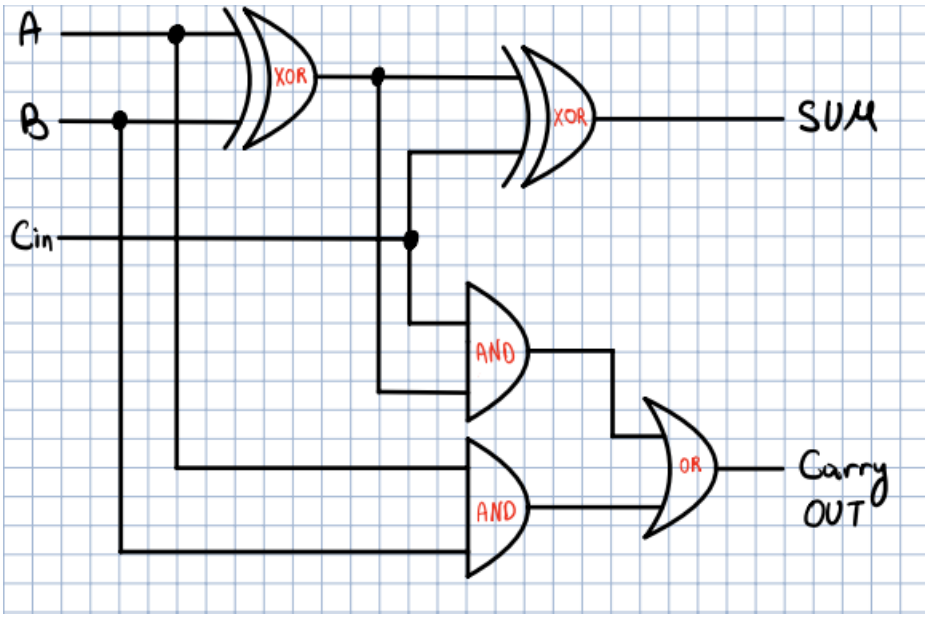
Fill here:

1. Full adder

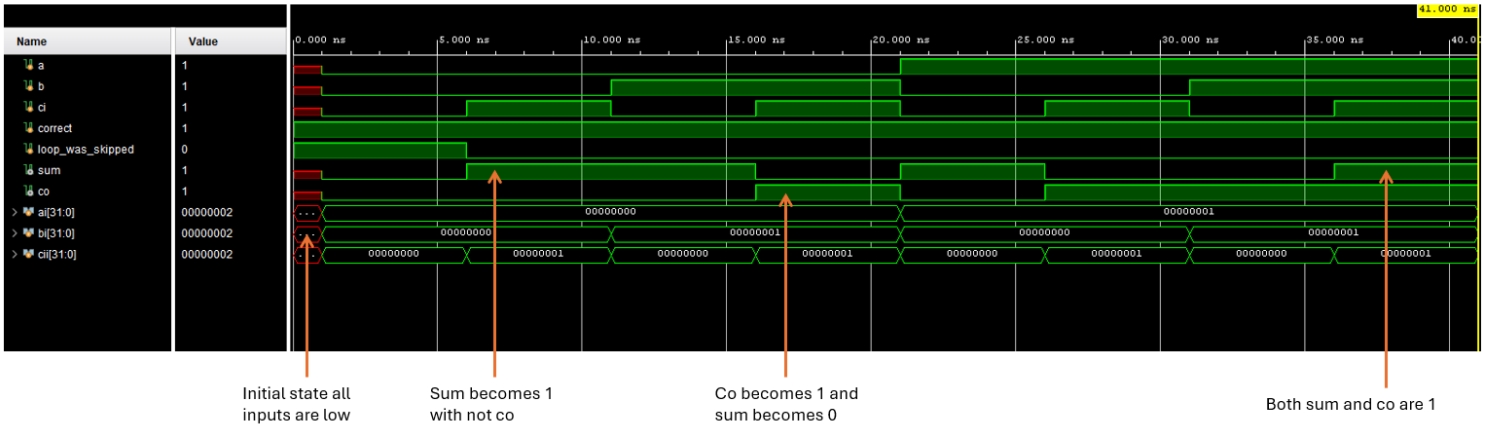
b.
The Full Adder functionality is a + b + ci = 2·co + sum.
The truth table shows all 8 possible input and output combinations:

a	b	ci	sum	co	Carry Out?
0	0	0	0	0	No
0	0	1	1	0	No
0	1	0	1	0	No
0	1	1	0	1	**YES**
1	0	0	1	0	No
1	0	1	0	1	**YES**
1	1	0	0	1	**YES**
1	1	1	1	1	**YES**

c.



f.

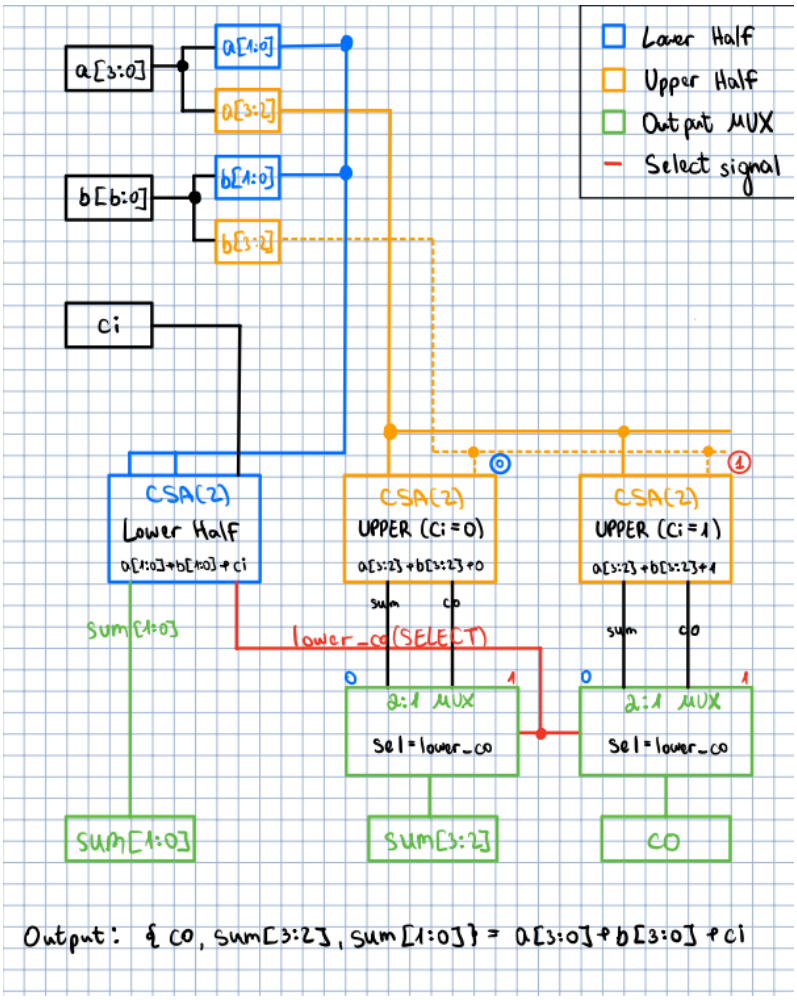


Annotated Waveform Analysis:

Time [ns]	a	b	ci	sum	co	Comment
0	0	0	0	0	0	Initial state, no inputs asserted
10	0	0	1	1	0	Carry-in only, sum = 1
15	0	1	0	1	0	b = 1, no carry-in
20	0	1	1	0	1	1 + 1 = 2 → sum wraps, carry generated
25	1	0	0	1	0	a = 1, simple addition
30	1	0	1	0	1	1 + 1 = 2 → carry generated
35	1	1	0	0	1	1 + 1 = 2 → carry generated
40	1	1	1	1	1	1 + 1 + 1 = 3 → sum=1, carry=1

2. Conditional sum adder

a.



The CSA(N) (Conditional Sum Adder) is implemented using a divide-and-conquer approach.
The input operands are split into lower and upper halves. The lower half is added using a CSA module with the real carry-in, producing the lower sum and a carry-out.
In parallel, the upper half is added twice using two CSA modules, once assuming carry-in = 0 and once assuming carry-in = 1.
When the carry-out from the lower half becomes available, multiplexers select the correct upper sum and carry-out.

c.

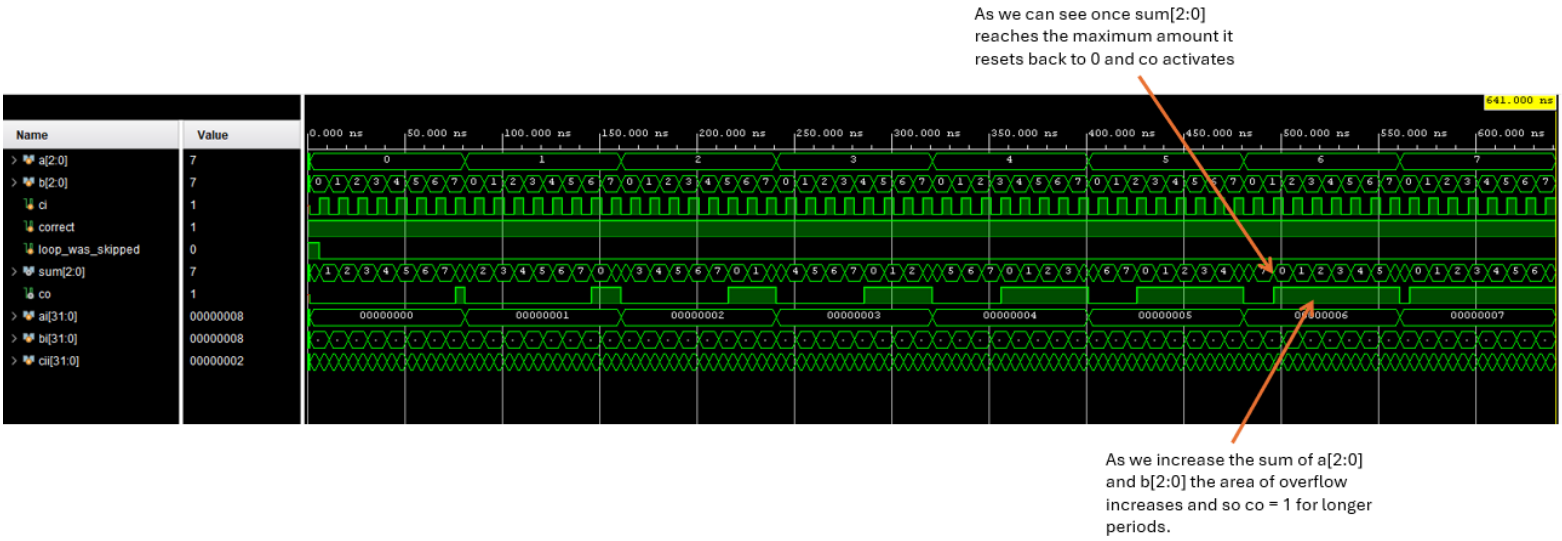
A ripple-carry adder has an O(N) delay because the carry signal must propagate sequentially through all N bit stages.
A conditional sum adder improves timing by computing both possible carry outcomes in parallel and selecting the correct result, resulting in an O(log N) delay.
This speed improvement comes at the cost of increased area due to duplicated adder blocks and multiplexers.

A ripple-carry adder is preferable for small bit widths, area-constrained designs, or low-frequency applications where timing is not critical.

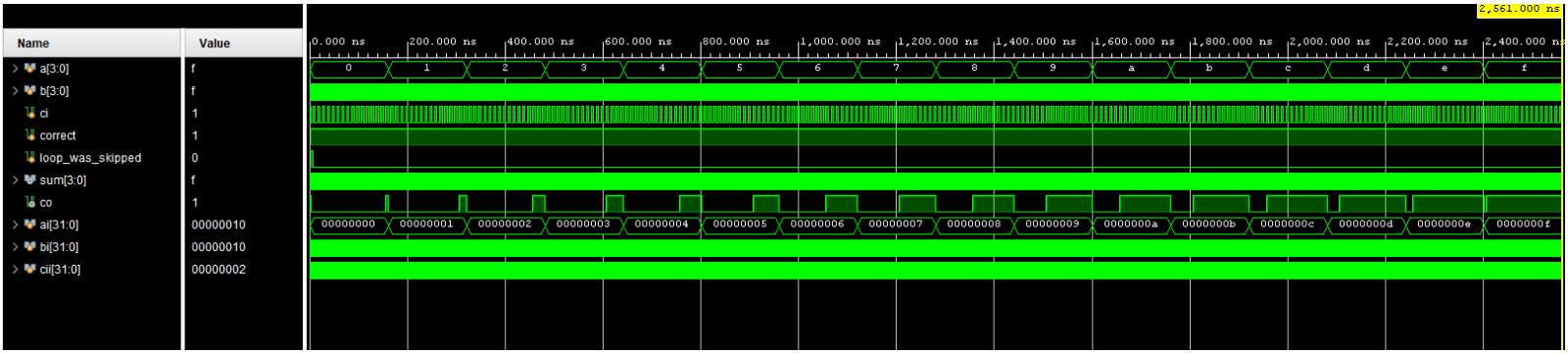
d.

A generate block is a Verilog construct that allows conditional and repetitive hardware instantiation at elaboration time based on parameters.
It enables different hardware structures to be created depending on parameter values and allows multiple module instances to be generated using loops.
In the CSA implementation, a generate block is required to support recursive construction where the base case handles small bit widths and larger widths are built by instantiating smaller CSA blocks.

g.

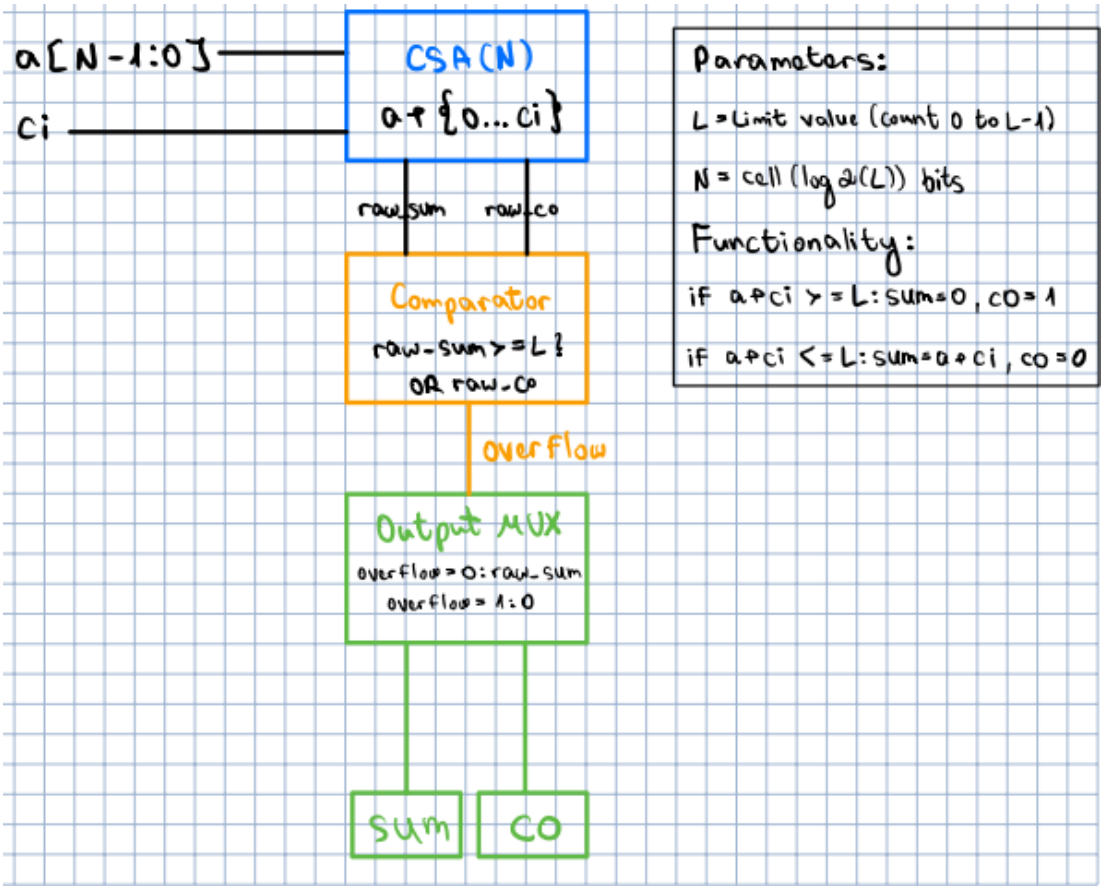


h.



3. Limited incrementor

a.



To implement the needed functionality, we first input a[N-1:0] and ci into a conditional sum adder. After that, using a comparator, we check if the output sum is smaller or equal/bigger than L. The output of this comparator was named overflow and is then fed into a MUX that chooses the value of sum. If overflow = 0 than sum = raw_sum and if overflow = 1 than sum = 0. In addition, co = overflow.

d.

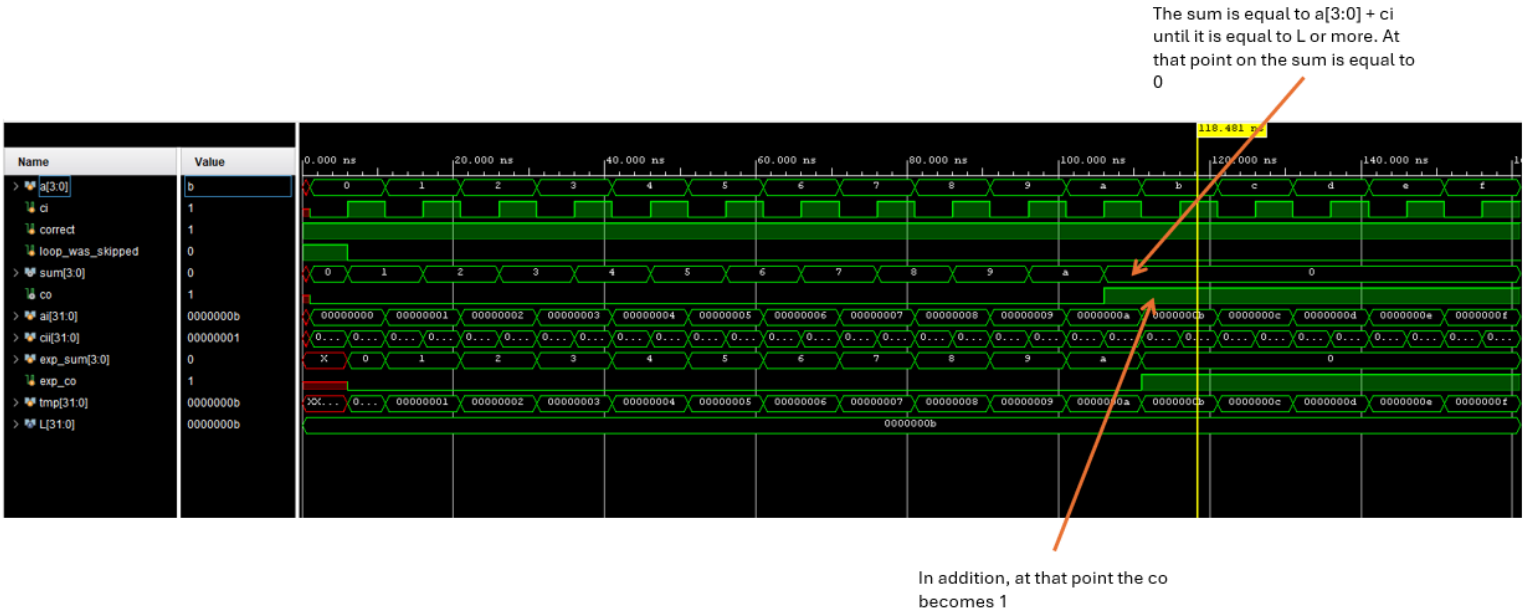
We ran the test bench where L=11, a=0,1,…,15 and ci = 0,1 and got a test passed. One case where the raw sum is different to the final sum is when a = 12 and ci = 1. The raw sum in this case is 13 but the final sum is 0. This is because L = 11 and once the raw sum is equal to or is bigger than L the overflow turns to 1 and the final sum is in turn equal to 0.

e.

Like we explained in the previous section, the raw sum is different from the output sum and co because the condition that raw sum is equal to or bigger than L was met. When this condition is met overflow turns to 1 and the MUX chooses the output sum to be 0. When L is equal to a power of two the output sum will always be equal to the raw sum. For example, if L=16 the size of a[N-1:0] is 4 bits, meaning that its maximum value is 1111 which is equal to 15. This value is

smaller than L and thus the raw sum is equal to the output sum. When we add a ci bit to this maximum value the sum will equal to 0 and the co will equal to 1. We can clearly see that when L is equal to a power of 2 the limited incrementor functions like a simple adder and can be implemented exactly like the conditional sum adder we implemented in the previous module.

f.



As we can see from the waveform the output sum is equal to the input value + ci until that sum is equal to or bigger than L (11 in this case). Once this condition is met the sum becomes 0 and co becomes 1.

4. Counter

a.
The time_reading output has 8 bits where the first 4 are the ones_seconds and the final 4 are the tens_seconds:

time_reading = {tens_seconds, ones_seconds};

Because each of these variables is made up of 4 bits, they can have values between 0 and 9 meaning that time_reading can have values between 0 and 99 seconds.

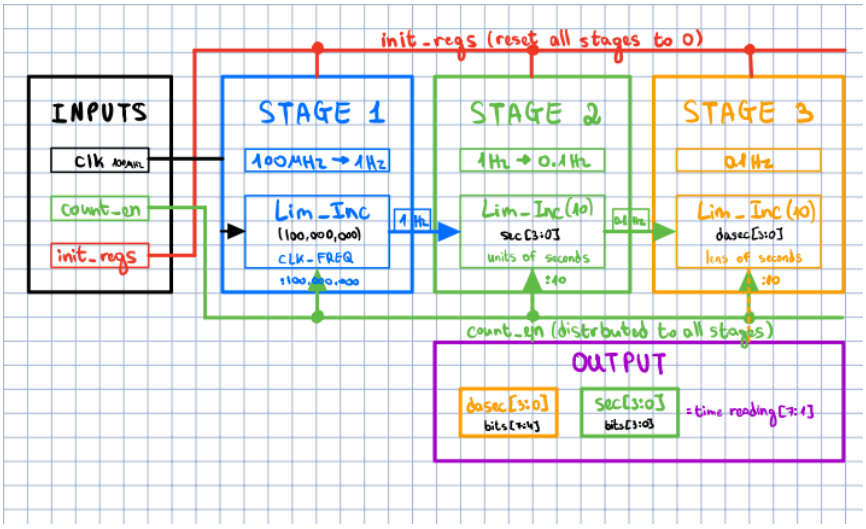
b.
A Lim_Inc(L) module is combined with a register to divide the input clock frequency.
The register holds the current count value and feeds it back to the Lim_Inc input.
When the count reaches the limit L, the Lim_Inc generates a carry-out (co) signal, which acts as a timing tick for the next stage.
This mechanism allows precise clock division without using arithmetic operators.

c.
The number of Lim_Inc modules is determined by the number of distinct counting stages required in the design.
Each time unit (clock divider, seconds digit, tens of seconds digit) requires its own Lim_Inc module with an appropriate limit value.

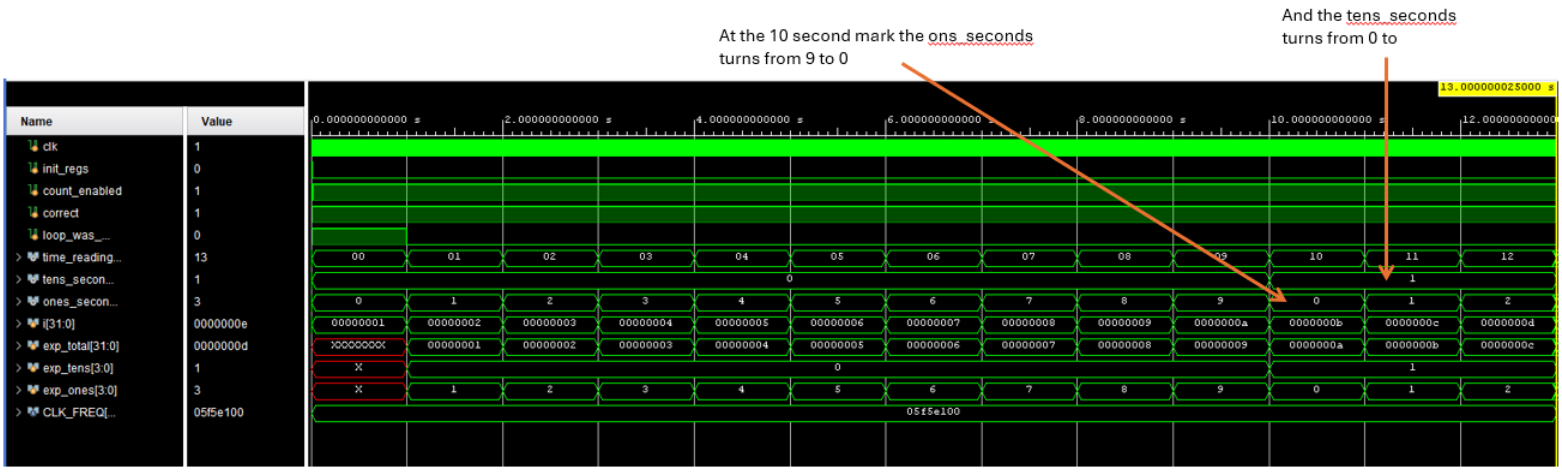
d.
To generate a 0.1 Hz tick from a 100 MHz clock, the design uses a hierarchical division approach.
First, a Lim_Inc module with L = 100,000,000 generates a 1 Hz tick corresponding to one second.
Then, a second Lim_Inc module with L = 10 counts these seconds to generate a tick every 10 seconds, corresponding to a 0.1 Hz frequency for the tens-of-seconds digit.

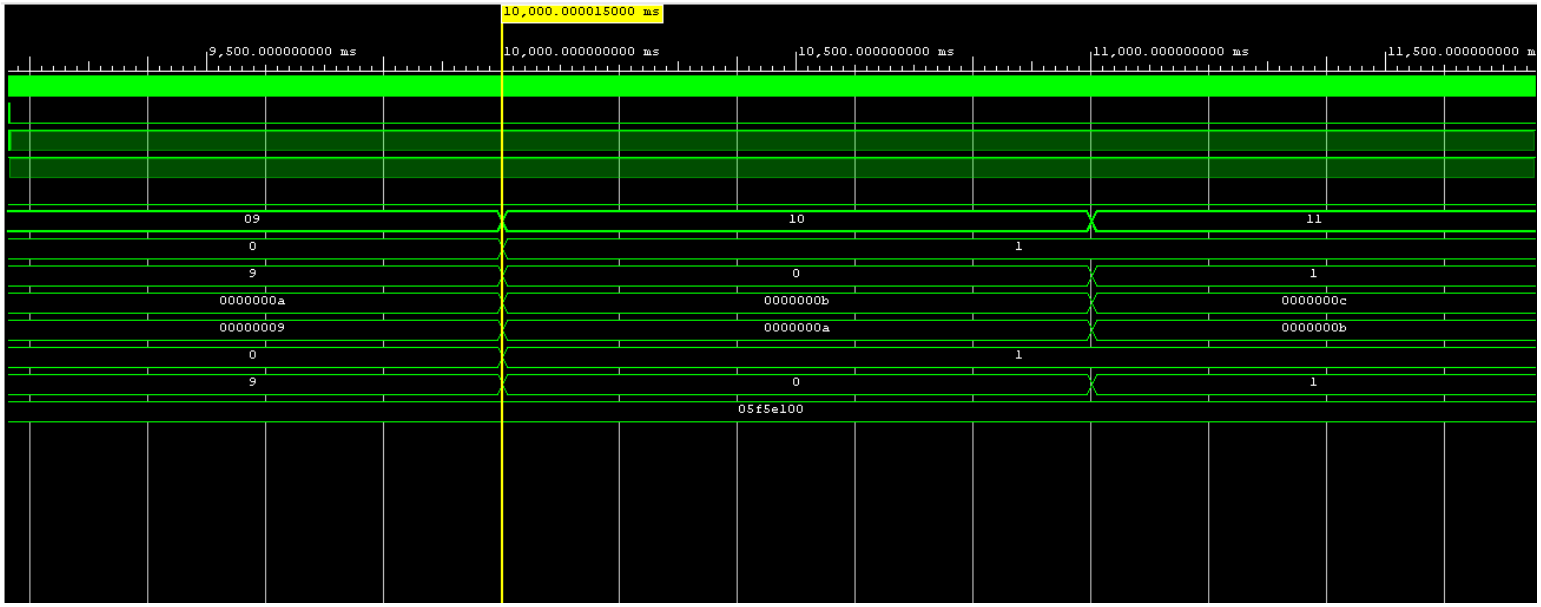
Stage	L Value	Input Freq	Cary out Freq	Output
1	100,000,000	100[MHz]	1[Hz]	sec_tick
2	10	1[Hz]	0.1[Hz]	ones_seconds [3:0]
3	10	0.1[Hz]	Unused	tens_seconds [3:0]

e.
Only the first divider parameter needs to be updated. The divider that generates the 1 Hz tick must use L = 50,000,000 instead of 100,000,000 while the Lim_Inc(10) modules used for ones_seconds [3:0] and Tens_seconds [3:0] remain unchanged. The design is fully parameterized using CLK_FREQ, so changing the clock frequency requires no structural changes, only a parameter update.
f.



The module is made up of three stages each made up of the previous Lim_inc module. As we explained before each module counts to a specific frequency that corresponds to a specific time unit (tick, seconds, tens of seconds). This time unit is then fed to the next stage as a carry in bit. The outputs of the second and third stage are than combined to the final output of the module.
j.





as we can see from the wave form the waveform at 10[sec] both the ones_seconds and tens_seconds change, from 9 to 0 and from 0 to 1 respectively.

k.
Lim_Inc is designed as a combinational module (no clock input). This design choice:

- Promotes reusability: Can be used in both combinational and sequential contexts
- Separates concerns: Arithmetic logic separate from storage logic

- Enables proper synchronous design: Registers are updated on clock edges in Counter module

Because of this design we must use external register.

l.

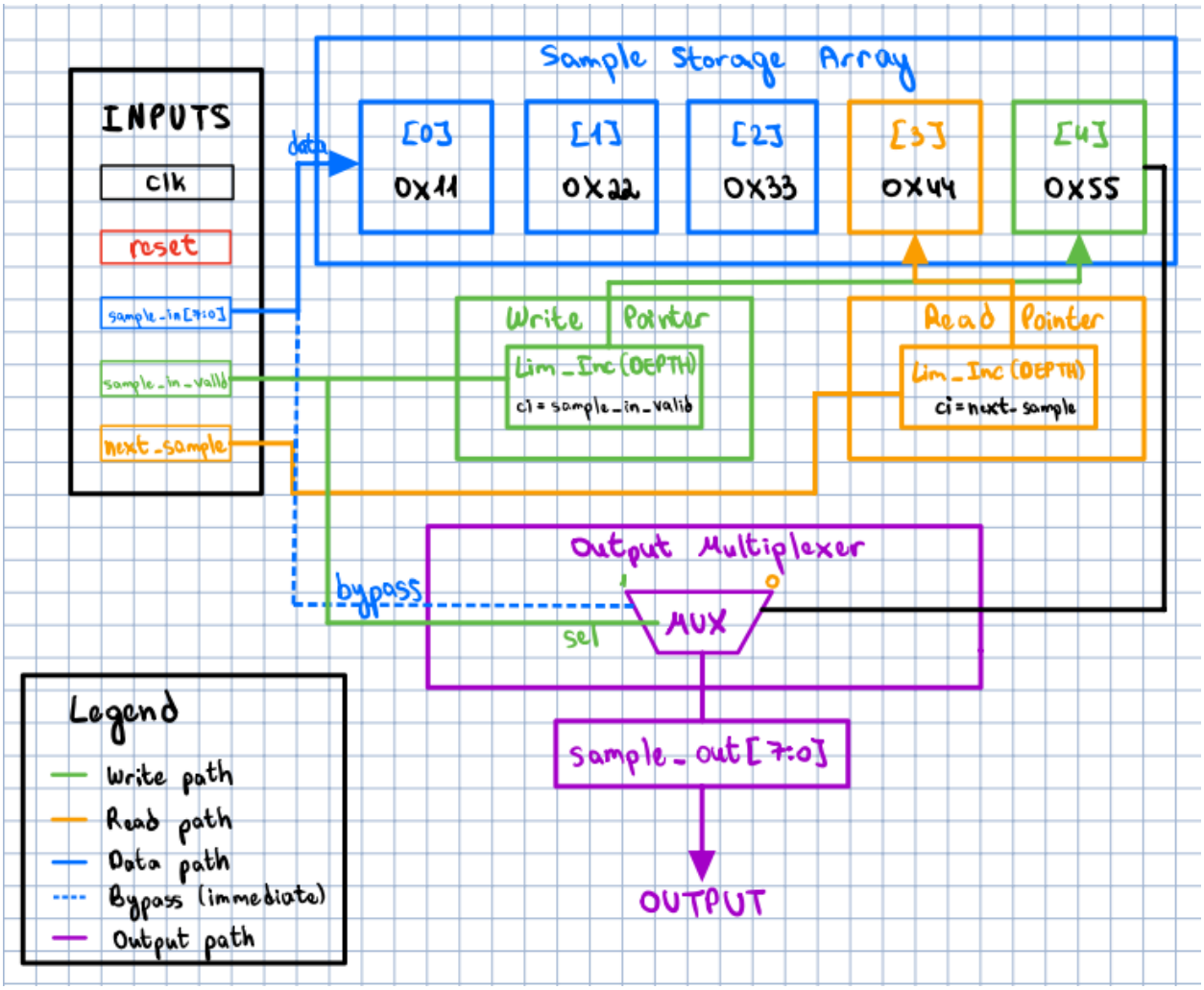
The maximum time that can be represented is 99 seconds. After reaching this time both the tens digit and the ones digit will go back to 0.

m.

To add minutes, we will add two more Lim_inc stages that will represent minutes and tens of minutes. The minutes stage will have L = 10 and the tens of minutes stage will have L = 6. We will also need to change the L of the tens of seconds stage from 10 to 6. All in all we will have seconds counting from 00 to 59 and minutes counting from 00 to 59. In total the counter will count from 00:00 to 59:59.

5. Stash:

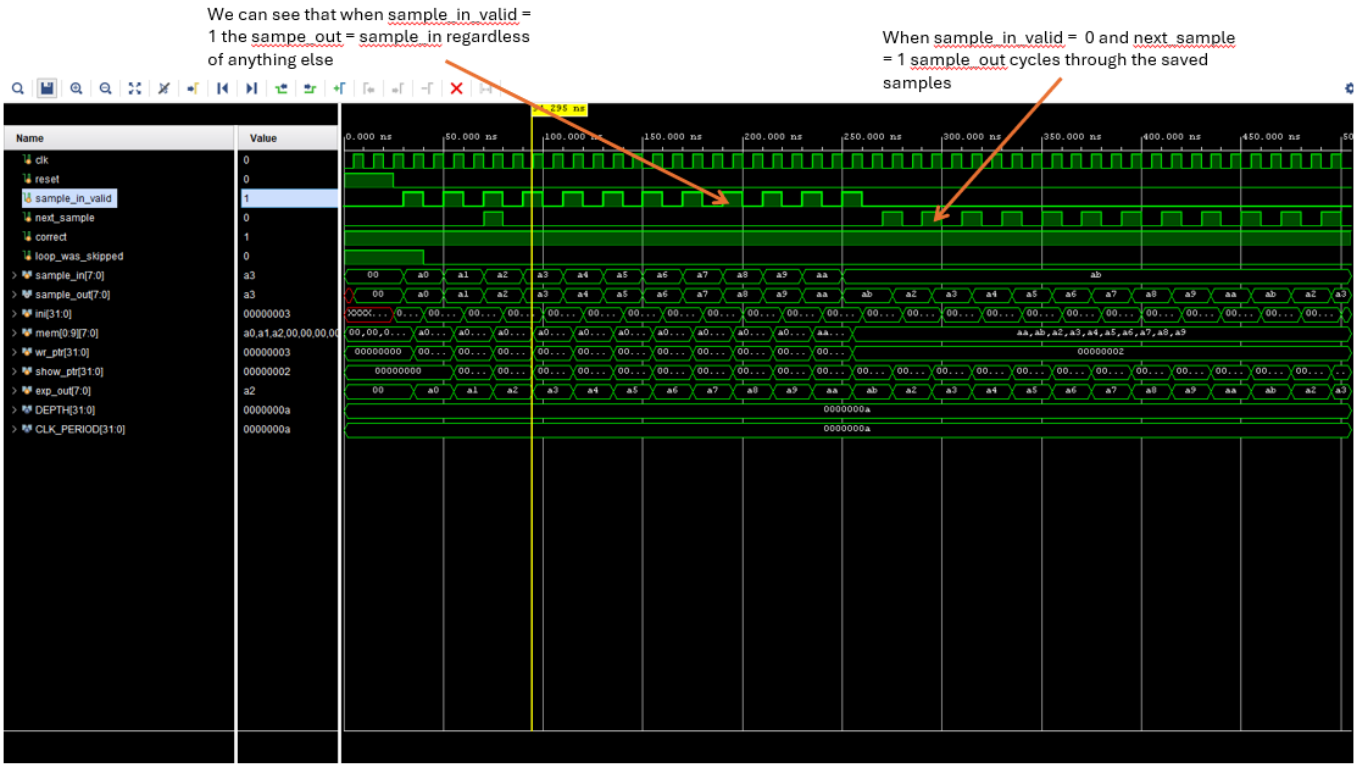
a.



In this block diagram we represented a stash of depth 5. Using two Lim_inc modules we created two pointers: the first one is the write pointer which advances every time sample_in_valid = 1. The new sample is written in the stash in the location of the write pointer. The second pointer is the read pointer; this pointer points to the location of the currently presented sample and can advance in two ways: firstly, when sample_in_valid = 1 the read pointer becomes equal to the write pointer. Secondly, when the sample_in_valid = 0 and next_sample = 1 the pointer advances to the next location in the stash. At the end of the module is a MUX that outputs the currently displayed module. When sample_in_valid = 1 it outputs sample_in and when sample_in_valid = 0 it displays the sample at the sample at the read pointer's location.

d.
We filed the given test bench and ran it for depth = 10 and depth+2 samples.

e.



As we can see from the waveform, the module works properly. We can also see that in the marked cycle sample_in_valid = 1 and next_sample = 0 and sample_out is equal to sample_in. This is because of the last line in this module:

assign sample_out = sample_in_valid ? sample_in : samples[read_ptr_reg];

sample_out is defined so that when sample_in_valide = 1 it automatically equals to sample_in. in addition once a new sample is stored the read pointer jumps to the write pointer thus ensuring consistency with the stash circulation.

f.
The sample_out selection logic operates like a mux with sample_in_valid acting as the selector. When smple_in_valid = 1 sample_out = sample_in regardless of anything else. When sample_in_valid = 0 sample_out is determined by read_ptr_reg. meaning that it is equal to the sample in location read_ptr_reg in the stash.

g.
An alternative method is to simply redirect the read pointer to the write pointer:

```
always @(posedge clk) begin
    if (sample_in_valid) begin
        read_ptr <= write_ptr;
    end
end
```

In this method the new sample is stored in memory, the read pointer jumps to the newly written location and finally the new sample becomes visible on sample_out in the next clock cycle.

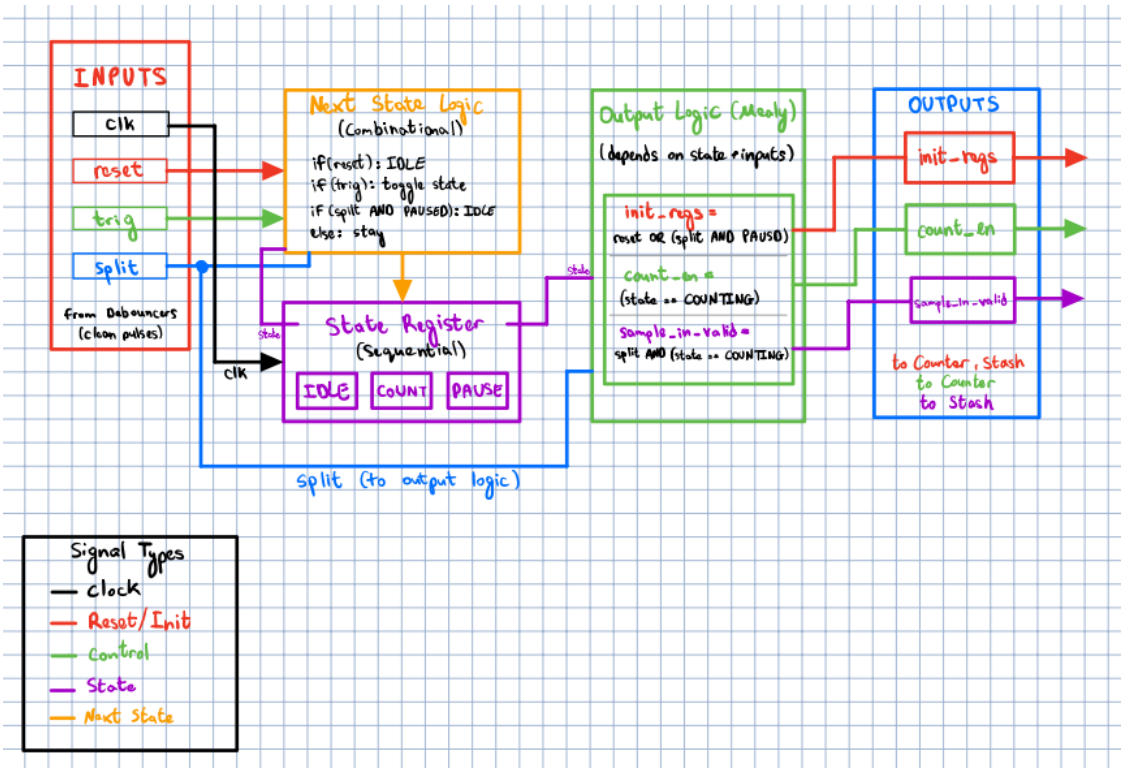
Aspect	MUX Bypass	Pointer Jump
Visibility latency	0 cycles (immediate)	1 clock cycle
Logic type	Combinational	Sequential
Hardware cost	Small multiplexer	No extra MUX
Timing	Combinational path	Fully registered
Spec compliance	Exact match	Slightly delayed

Although the pointer jump method is simpler structurally it introduces a one-cycle delay thus making it less suitable for our purpose.

6. Control

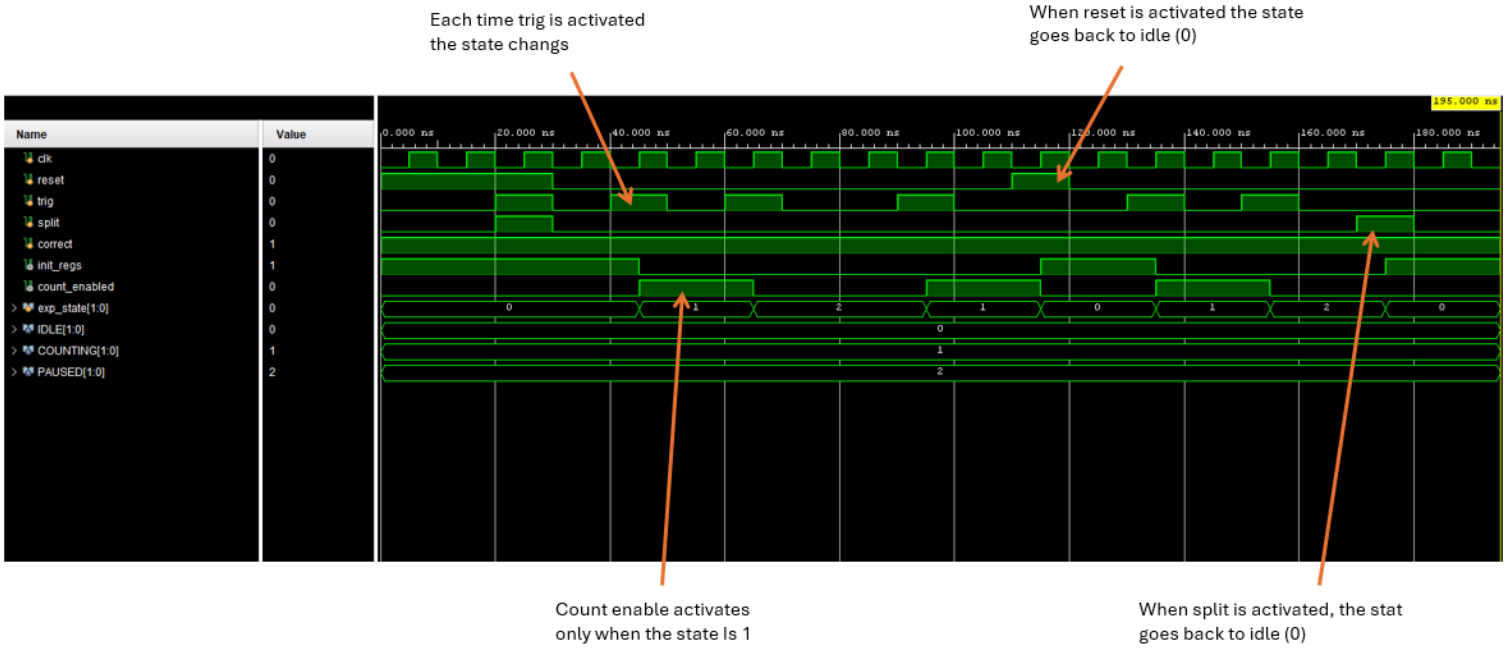
a.
The FSM is not a Moore FSM. A Moore FSMs output depend only on its current state while a Mealy FSMs output depend on its current state and its inputs. Our FSM output clearly depends on its inputs because using the trig button moves it from counting to paused and vice versa. Therefore, it is not Moore FSM.

b.



This module is divided into two main blocks: the first block is the next stage logic and the second is the output logic. The next stage logic determines what the state in the next clock cycle will be based on the current state and on the input command. The output logic determines what the value of count_enabled and init_regs will be based on the same parameters.

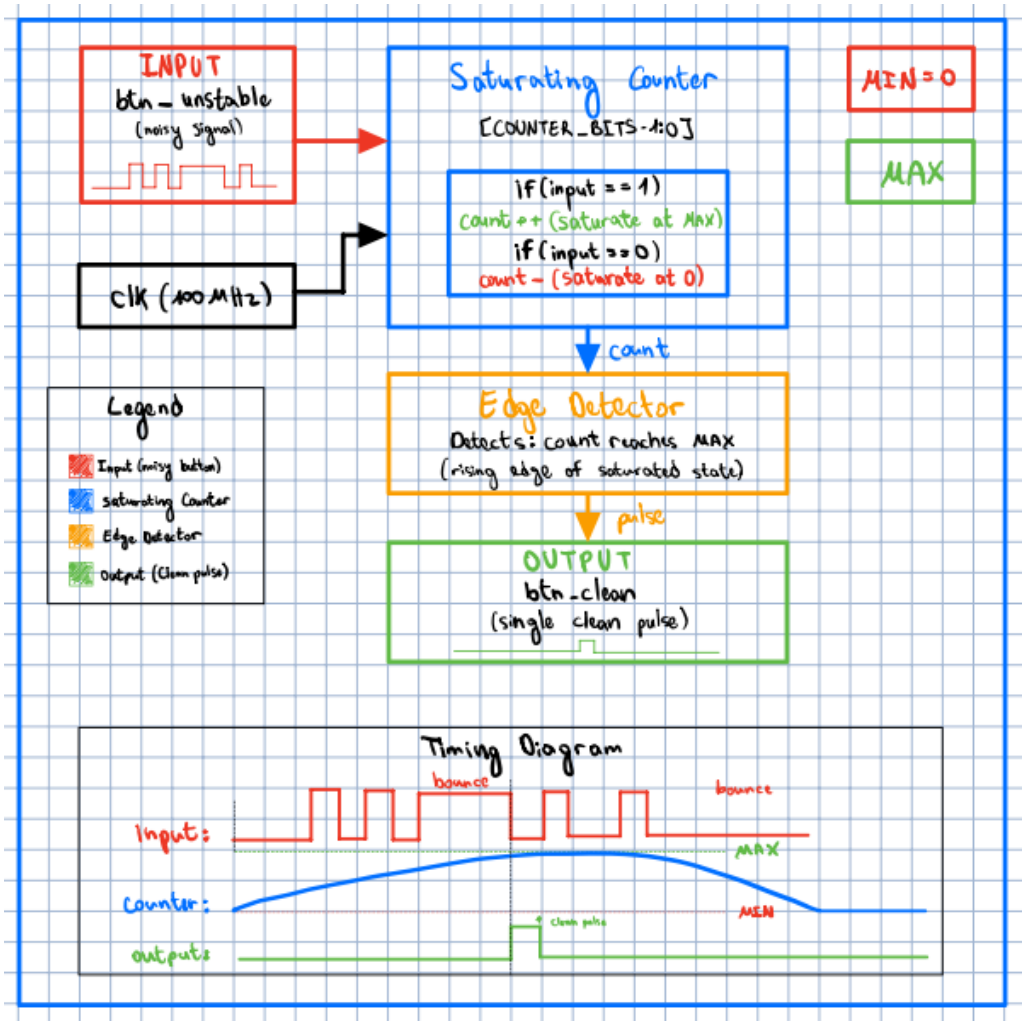
f.



As we can see from the waveform the test passed successfully. The state of the FSM changes each time a signal enters from the trig, reset or split like we wanted. The reset and split returns the system to state 0 and trig takes it to the next state in line.

7. Debouncer

a.



The debouncer gets an input of a clk and an unstable signal. The debouncer first checks if the input is 1 or 0 in this clock cycle and increases or decreases the counter accordingly. After that if the input is 1 using simple logic the debouncer checks if in the previous clock cycle the counter was at its maximal value. If it was then the output remains 0 but if it wasn't the output becomes 1. If the input is 0 the output naturally stays 0. Using this logic the debouncer ensures a stable and noise free output.

d.

Aspect	High Counter	Low Counter
Debounce time	Longer	Shorter
Noise immunity	Better filtering	Worse filtering
Response time	Slower response	Faster response
Area cost	More flip-flops	Fewer flip-flops

As we can clearly see from the table the tradeoff between a high and a low counter is that while the higher counter provides better filtering it is slower and more expensive.

8. Seven Segment Display

c.

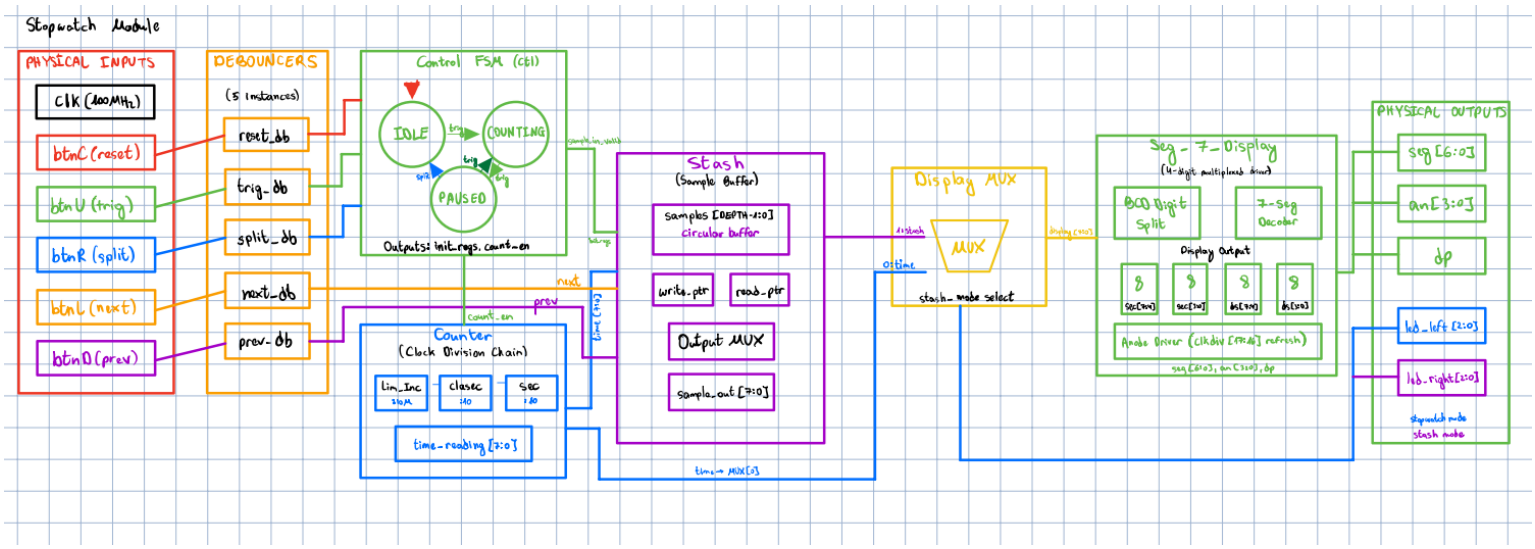
We were told that the seven segment display refresh time must be between 1[ms] and 16[ms]. The given clock rate is 100[MHz] meaning that each clock cycle takes 10[ns] to complete. We know that the time it takes each bit that is closer to the MSB to change is equal to the time it takes the previous bit times two. Meaning that if it took bit 0 10[ns] to change it would take bit 1 20[ns] to change and so on. Because we had to choose two bits, the time it would take both to complete a full cycle is 4 times that of the base clock. Meaning that to choose the stating bit K it would have to satisfy the following equation:

$$1[ms] < 10[ns] \cdot 4 \cdot 2^{K} < 16[ms]$$

we chose K=18 and got a refresh time of 10.49[ms]. Meaning that the selected bits were bit 18 and bit 19.

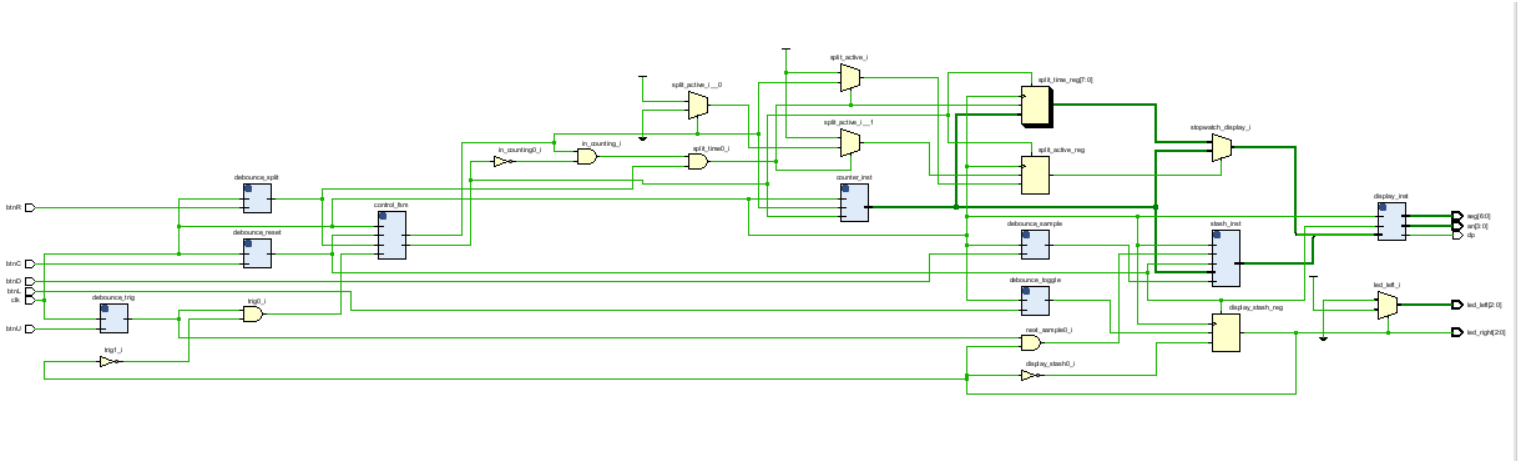
9. Stopwatch

a.



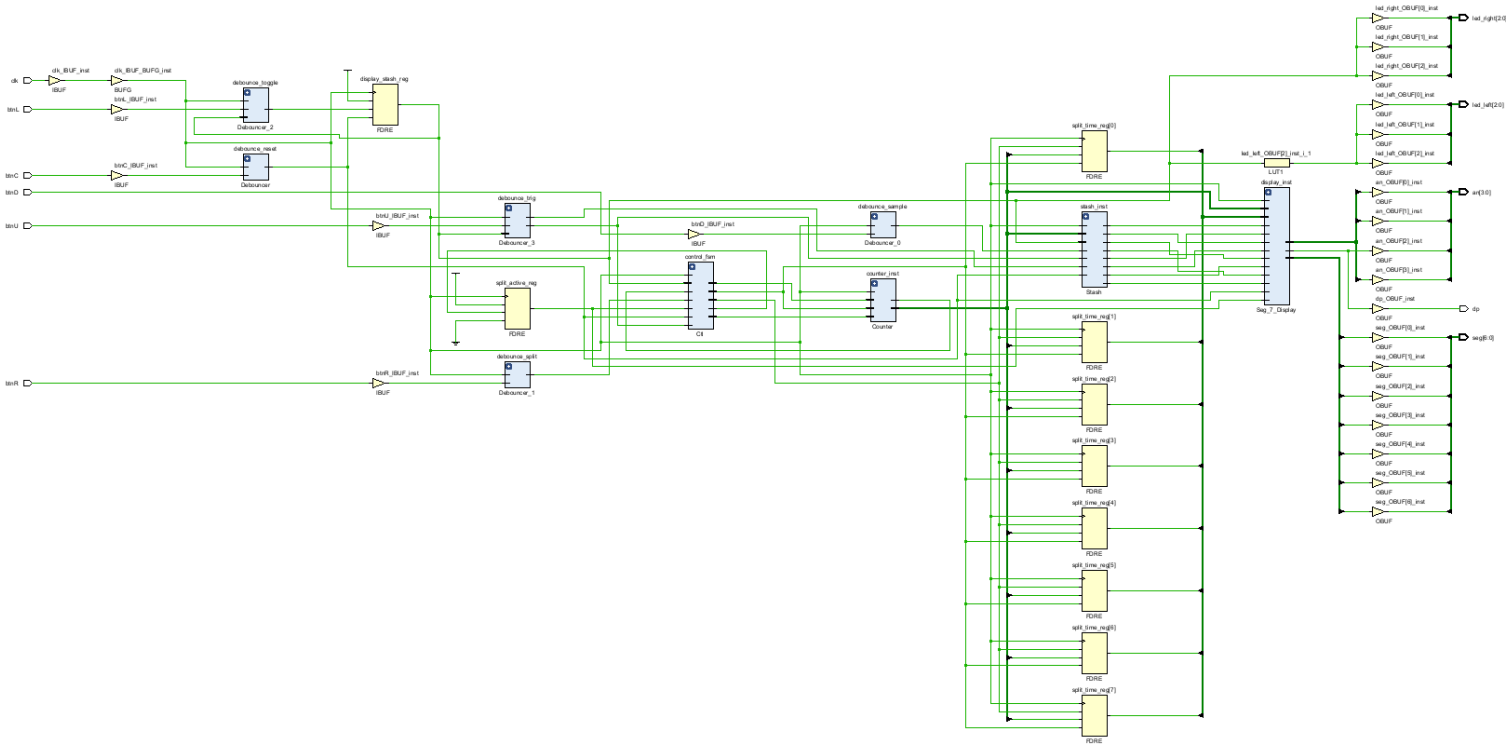
The stopwatch is the final module, and it uses all the previous ones we have implemented. The inputs of the stopwatch are the clock and all the buttons on the FPGA. All the signals from the buttons first go through the debouncer to ensure a stable signal. After the debouncer the reset, trig and split signals go into the control module to determine the state of the stopwatch. The output of the control module goes into the counter module to determine its operation while its other outputs alongside the signals from the final two buttons control the stash. The outputs of the counter and the stash are then fed into the display MUX and finaly to the 7-segment display and to the 6 LEDs.

e.



The elaborated design schematic shows the full hierarchy of the Stopwatch module, including Debouncer, Control FSM, Counter, Stash, and Seven Segment Display modules.
All inputs and outputs are correctly connected, and no missing or dangling signals were observed.

f.

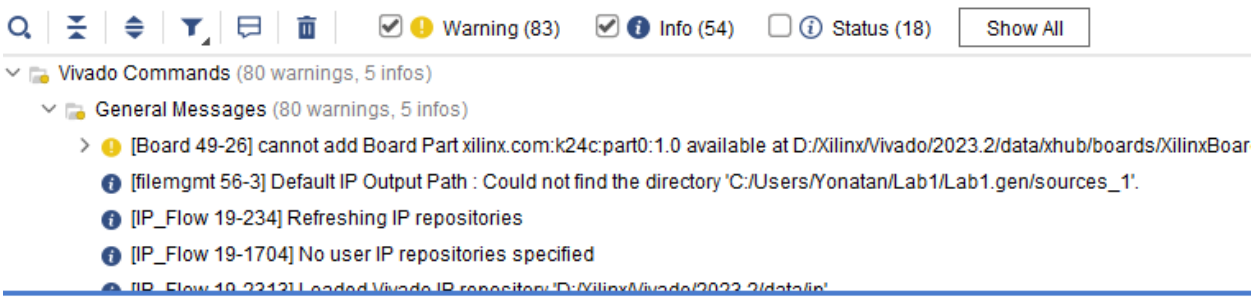


The RTL elaborated schematic shows the design exactly as written in Verilog. All modules (Stopwatch, Ctl, Counter, Stash, Debouncer, Seg_7_Display) are clearly visible, with registers and control logic represented at a high level. This view is useful to verify functional correctness and signal connectivity.

The synthesized schematic shows the FPGA implementation after optimization. Registers are mapped to flip-flops, logic is implemented using LUTs and multiplexers, and parts of the hierarchy may be flattened. Some logic is simplified or removed by the synthesis tool.

These differences are expected: the RTL view validates the design intent, while the synthesized view confirms correct and efficient hardware implementation on the FPGA.

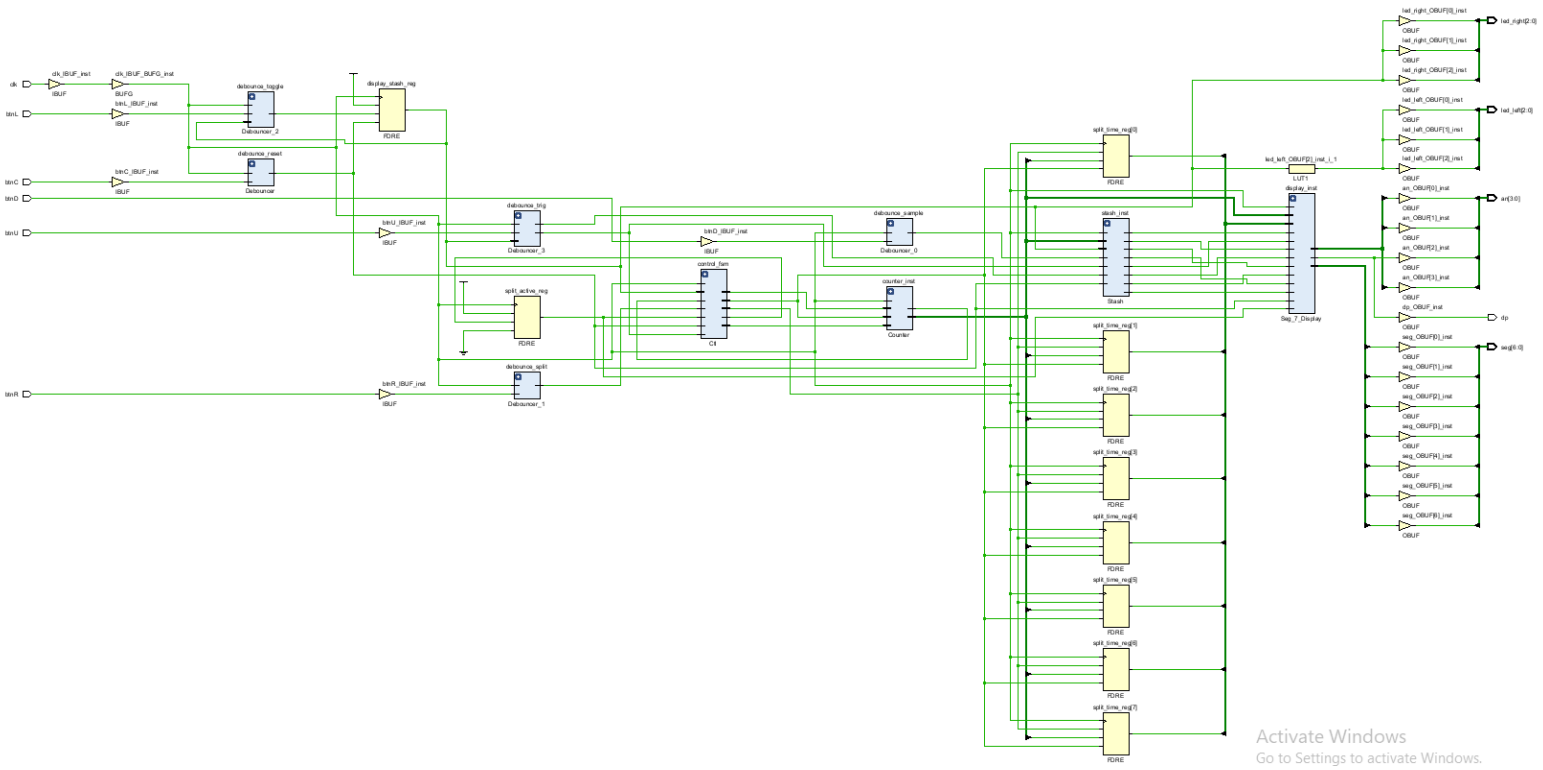
g.



We didn’t get any critical warnings

11. Implementing the design

c.



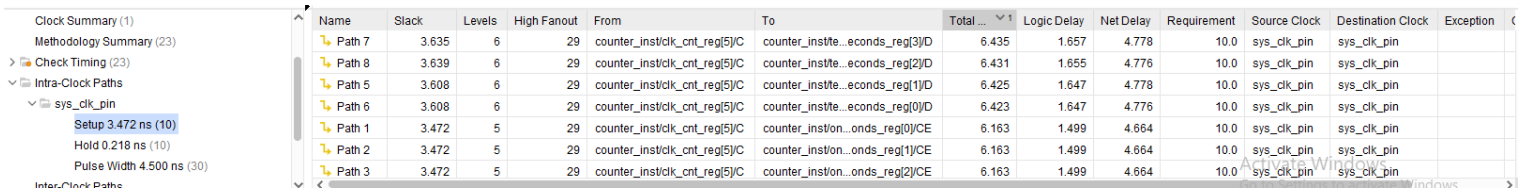
d.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.472 ns	Worst Hold Slack (WHS):	0.218 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	419	Total Number of Endpoints:	419	Total Number of Endpoints:	224

All user specified timing constraints are met.

As we can clearly see we got positive slack both for setup and for hold times like we wanted.

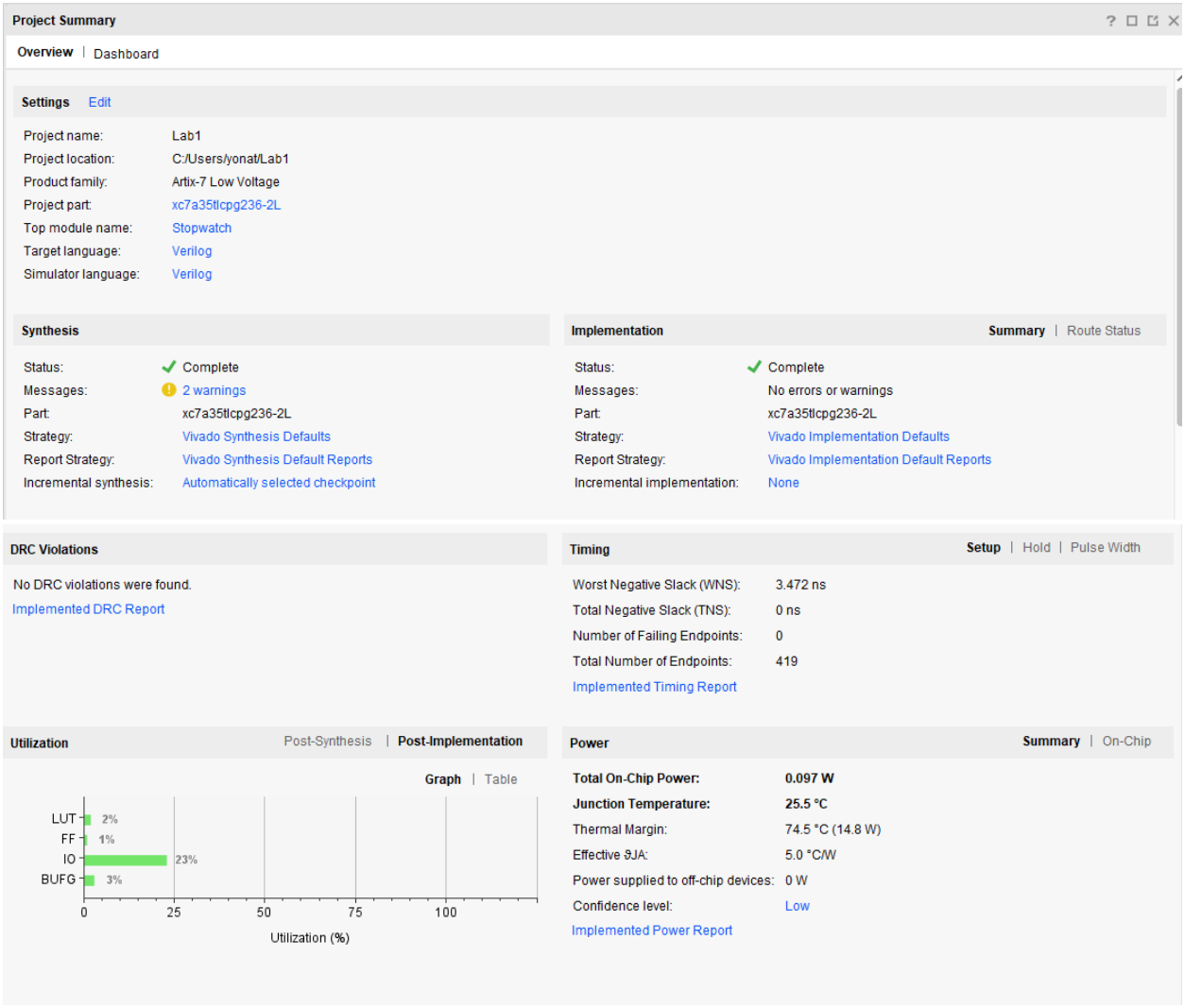
e.

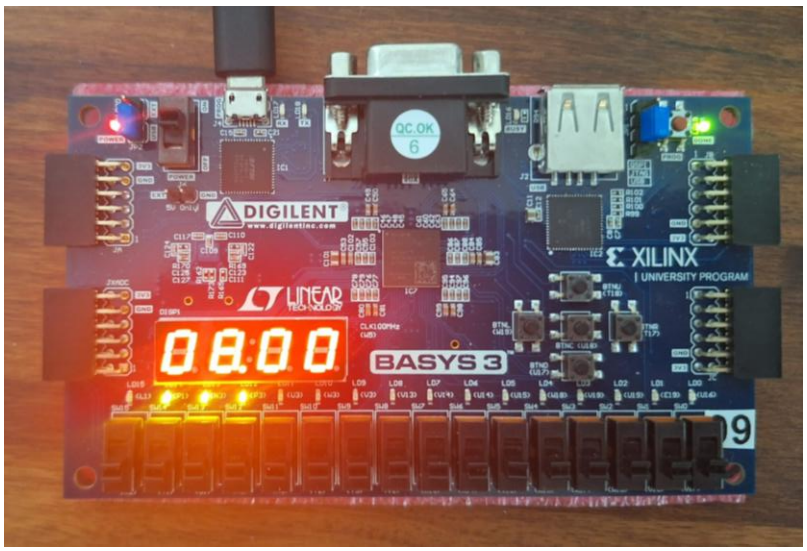


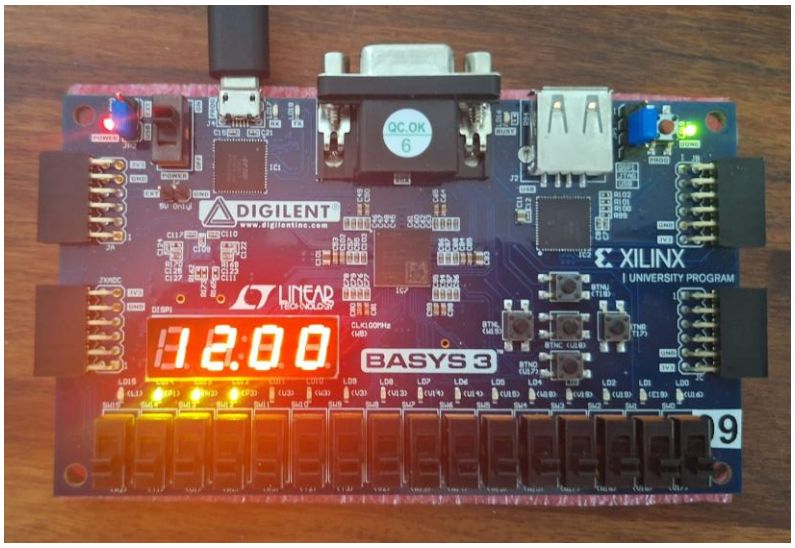
Name	Slack	Levels	High Fanout	From	To	Total ...	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 7	3.635	6	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/te...econds_reg[3]/D	6.435	1.657	4.778	10.0	sys_clk_pin	sys_clk_pin	
Path 8	3.639	6	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/te...econds_reg[2]/D	6.431	1.655	4.776	10.0	sys_clk_pin	sys_clk_pin	
Path 5	3.608	6	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/te...econds_reg[1]/D	6.425	1.647	4.778	10.0	sys_clk_pin	sys_clk_pin	
Path 6	3.608	6	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/te...econds_reg[0]/D	6.423	1.647	4.776	10.0	sys_clk_pin	sys_clk_pin	
Path 1	3.472	5	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/on...onds_reg[0]/CE	6.163	1.499	4.664	10.0	sys_clk_pin	sys_clk_pin	
Path 2	3.472	5	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/on...onds_reg[1]/CE	6.163	1.499	4.664	10.0	sys_clk_pin	sys_clk_pin	
Path 3	3.472	5	29	counter_inst/clk_cnt_reg[5]/C	counter_inst/on...onds_reg[2]/CE	6.163	1.499	4.664	10.0	sys_clk_pin	sys_clk_pin	

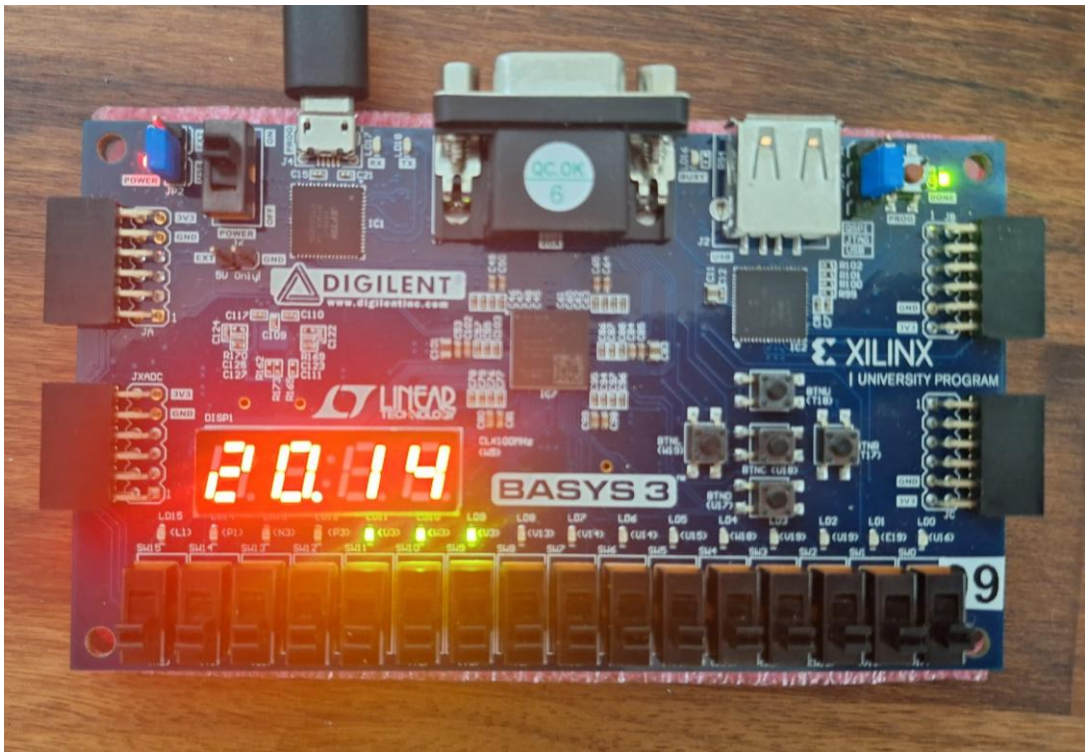
As we can see, the critical setup path was from counter_inst/clk_cnt_reg[5]/C to counter_ist/tens_seconds_reg[3]/D with a total delay time of 6.435 [ns] divided to 1.657 [ns] logic delay and 4.778 [ns] net delay.

12. FPGA programming and debugging




DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3
08:00


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3
12:00


DIGILENT
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3
20:14