FPGA-Based Digital Stopwatch

Project Report & Implementation Flow

Submitted by: Moshe Sofer (315038232) Rotem Vasa (322209529)
Tel Aviv University | Electrical and Electronics Engineering

Project Overview

This report details the bottom-up design, verification, and hardware implementation of a fully functional Digital Stopwatch with a memory stash feature on an Artix-7 FPGA.

Hardware Design Flow

The project was developed in four main phases, moving from basic RTL blocks to physical silicon implementation:

Phase 1: Low-Level Combinational Logic

- ◊ **FA (Full Adder):** Base addition block.
- ◊ **CSA (Conditional Sum Adder):** O(log N) optimized parallel adder.

Phase 2: Sequential Logic & Sub-Systems

- ◊ **Limited Incrementor:** Parameterized counting logic.
- ◊ **Counter:** Cascaded BCD timekeeping engine.
- ◊ **Stash:** Synchronous memory module for lap times.
- ◊ **Control (FSM):** Mealy state machine for system mode routing.
- ◊ **Debouncer:** Digital low-pass filter for stable mechanical button inputs.

Phase 3: System Integration & I/O

- ◊ **Seven Segment Display:** Time-Division Multiplexing (TDM) driver.
- ◊ **Stopwatch (Top Module):** Full system architecture and component routing.

Phase 4: Physical Implementation

- ◊ **Constraints Setup:** Mapping logical ports to physical FPGA pins.
- ◊ **Synthesis & Implementation:** Timing analysis and resource optimization.
- ◊ **Hardware Debugging:** Board bring-up on the Digilent Basys 3.

Part 1 – Full Adder:

Question b – the truth table for the full adder is the following, given inputs a, b, c_i (carry in) and outputs sum, c_o (carry out). The functionality, as expected will be $a + b + c_i = 2 \cdot c_0 + sum$.

a	b	c_i	Input in dec base	sum	c_0	Output in dec base
0	0	0	0	0	0	0
0	0	1	1	1	0	2
0	1	0	2	1	0	2
0	1	1	3	0	1	1
1	0	0	4	1	0	2
1	0	1	5	0	1	1
1	1	0	6	0	1	1
1	1	1	7	1	1	3

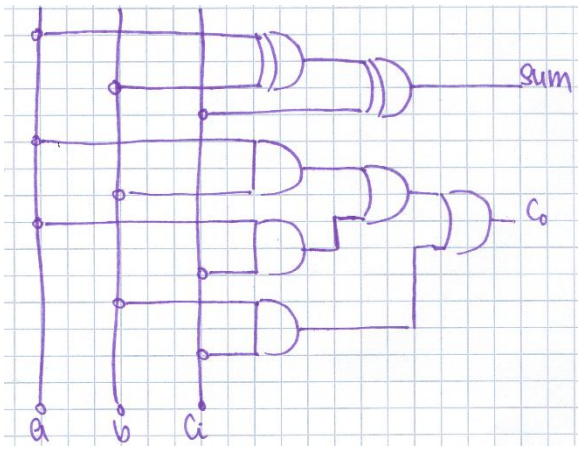
A carry out $c_o = 1$ is produced whenever at least two out of the three inputs are 1. We can formulate this as $c_o = ab + ac_i + bc_i$.

A note about the carry in and carry out bits, for later explanation: in a full adder, the carry in (ci) and carry out (c) pins are what enable multi-bit arithmetic by linking individual bit adders into a chain. The ci represents a carry digit generated from the addition of the previous, less significant bit position. Internally, the full adder combines this incoming carry with its two input bits (a and b). The co is then generated when the sum of the three bits (a + b + ci) is 2 or 3 in decimal, indicating a carry must be propagated to the next, more significant bit position.

Question c – from the truth table, the full adder outputs are:

$$sum = a \oplus b \oplus c_i$$
$$c_0 = ab + ac_i + bc_i$$

Using this, the FA schematic will be the following:

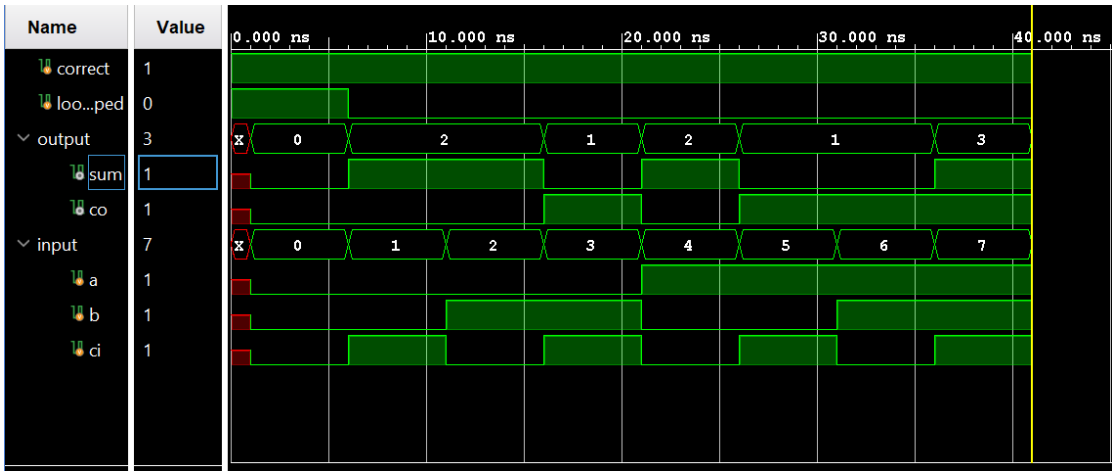


The Full Adder is composed of a parity network (XOR gates) for the sum output and a majority network (AND–OR gates) for the carry output, ensuring correct binary addition without the use of the '+' operator.

Question D – this code is following the schematics we've demonstrated earlier.

```
module FA(a, b, ci, sum, co);

  input   a, b, ci;
  output  sum, co;

    wire x1;
    wire t1, t2, t3;

    assign x1  = a ^ b;
    assign sum = x1 ^ ci;

    assign t1 = a & b;
    assign t2 = a & ci;
    assign t3 = b & ci;

    assign co = t1 | t2 | t3;

endmodule
```

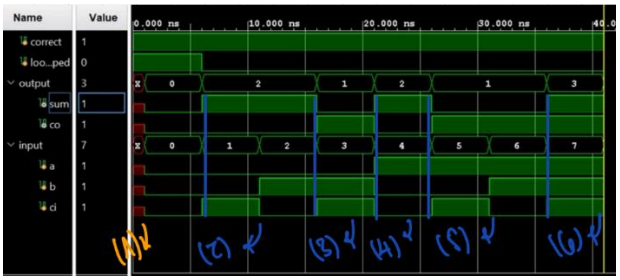
Question F – the following is the simulation results:



Looking at the buses we incorporated, we can see they match exactly the input\output numbers we presented in the 1st question in this section.

We can clearly see the response time of the circuit, which is the time passed between starting the clock and getting the 1st output value (the "red" zone, receiving a "x" label – number 1, following the notation of the image below) which is 1[ns].

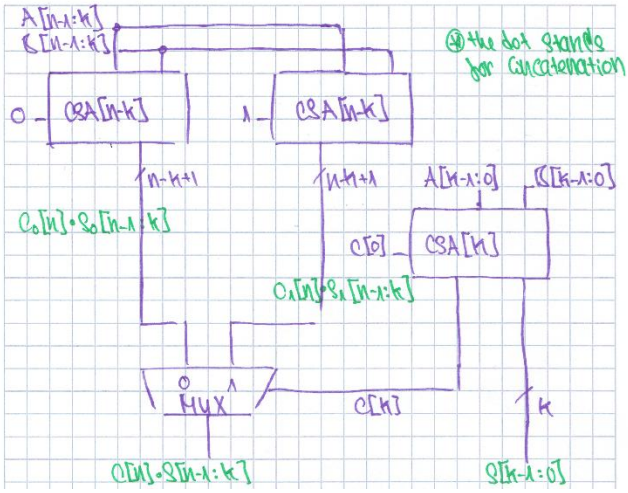
The logic we described earlier clearly works here – sum =1 only when an uneven number of inputs equals 1 (zones 2,4,6 – following the notations of the image below), and co =1 only when two out of the three inputs =1 (zones 3,5,6 – following the notations of the image below).



Part 2 - Conditional Sum Adder:

Question 1 – the conditional sum adder (CSA) is a fast adder architecture that speeds up binary addition by pre calculating results to avoid the delay of the carry propagation. Based on the functionality provided and the theoretical background, our design will follow the next logic –

(1) Split the adder into lower bits and upper bits. Using $k = n/2$ as the midpoint.
(2) Computing the upper bits part twice – once for every possible carry value.
(3) Selecting the correct upper result using a multiplexer, based on the actual carry produced by the lower part.
(4) The base case is a full adder, when adding three single bits. The FA design is as described in the previous part.



Question 3 - The Conditional Sum Adder improves timing by eliminating long carry-propagation chains through parallel computation and multiplexing, achieving logarithmic delay at the cost of increased hardware, whereas Ripple-Carry Adders remain preferable for small bit-width or area-constrained designs.

	RCA	CSA
Behavior	the carry output of each bit depends on the carry from the previous bit - the carry must propagate sequentially from the least significant bit (LSB) to the most significant bit (MSB).	improves timing by breaking the carry dependency, calculations are made without the consideration of the carry computation.
Worst path	The worst-case delay includes all N full adders in series - $O(N)$	The recursion (done by division 2), sets the delay as $O(\log N)$

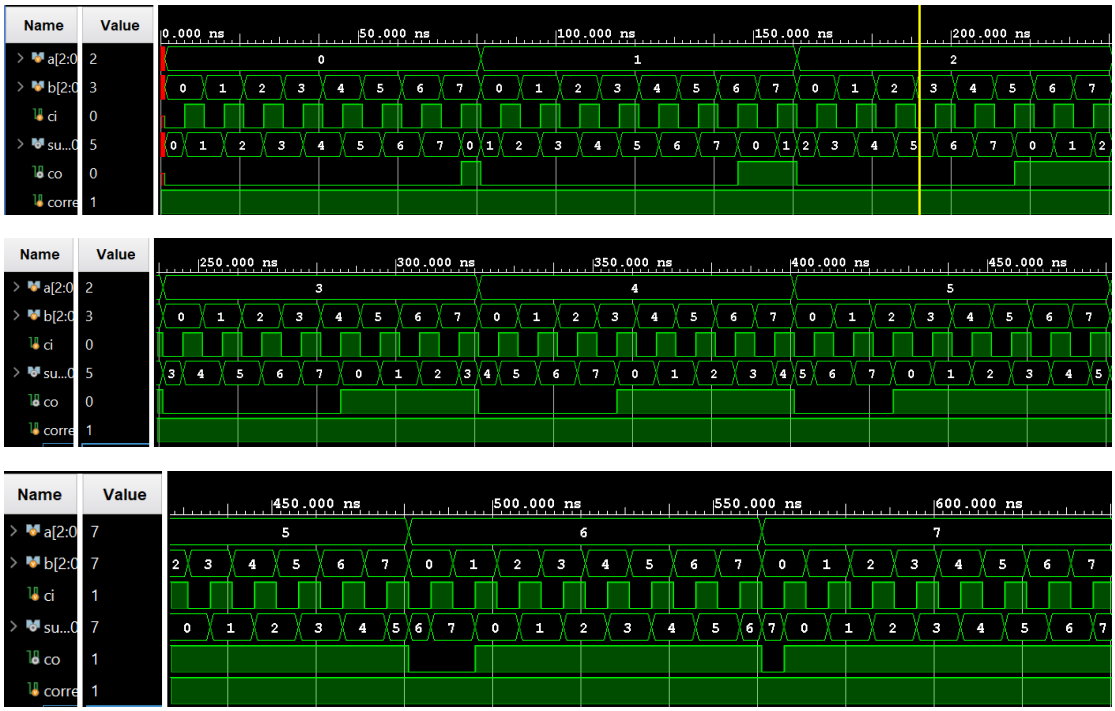
Despite being slower, RCA may still be preferred in the following cases:

- Small bit-width: RCA may be faster due to simpler logic.
- Area and resource constraints: CSA requires duplicate adders for upper bits and MUX, while RCA only requires N full adders.
- Low frequency designs: if timing constraints are loose, meaning no need to optimize critical path and simpler design is preferred, RCA offers a solution that is easier to debug, verify and maintain.
- Power sensitive designs - CSA performs redundant computations while RCA toggles fewer nodes, meaning RCA can have lower dynamic power consumption.

Question 4 - A generate block is a compile-time Verilog construct used to conditionally or repetitively create hardware based on parameters. It allows conditional hardware generation, repetitive hardware generation, as well as parameter-dependent structural descriptions. That means it creates actual hardware instances – it does not imply sequential behavior and is not a behavioral loop.

It is required in the CSA implementation to distinguish between the base case (N=1) and the recursive construction (N>1), enabling a parameterized and synthesizable recursive adder structure, since Verilog does not allow dynamic recursion and variable-size module instantiation at run time.

Question 7 – here are the input-output signals running CSA(3) –



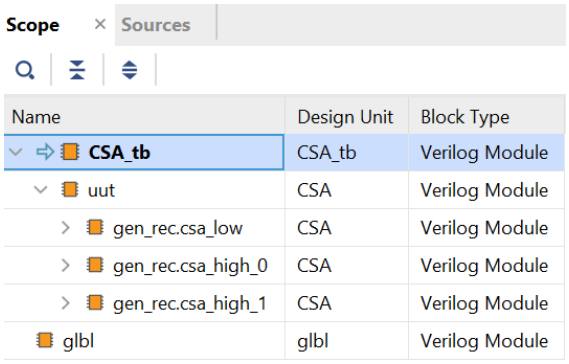
Showing a full truth table analysis $a = 0_{10}, 1_{10}, 2_{10}$:

a_{10}	b_{10}	c_i	sum_{10}	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	0
0	2	0	2	0
0	2	1	3	0
0	3	0	3	0
0	3	1	4	0
0	4	0	4	0
0	4	1	5	0
0	5	0	5	0
0	5	1	6	0
0	6	0	6	0
0	6	1	7	0
0	7	0	7	0

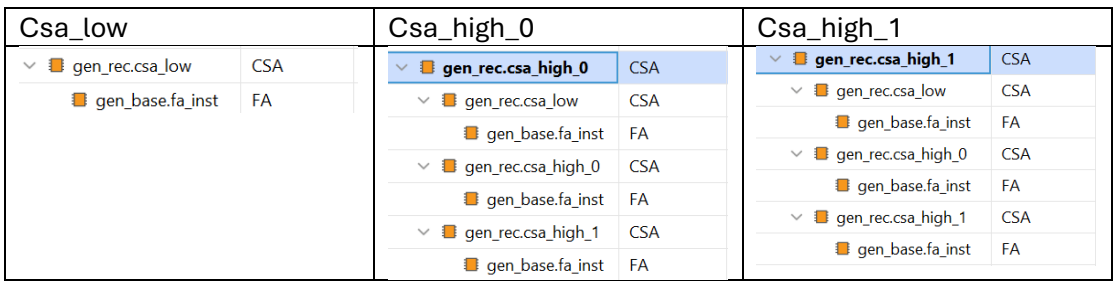
0	7	1	0	1
1	0	0	1	0
1	0	1	2	0
1	1	0	2	0
1	1	1	3	0
1	2	0	3	0
1	2	1	4	0
1	3	0	4	0
1	3	1	5	0
1	4	0	5	0
1	4	1	6	0
1	5	0	6	0
1	5	1	7	0
1	6	0	7	0
1	6	1	0	1
1	7	0	0	1
1	7	1	1	1

We can clearly see that the results matches the theory exactly – no unexpected value is shown – the bus representation makes it easier to calculate the expected sum, which is seen in explicit 10 base form.

Looking at the scope on the window next to the simulated waveforms, we can confirm that the recursion indeed worked – meaning that the conditional part of the code did generate physical multiplications of the CSA construction, up until the base case. The following print screen presents the first splitting:

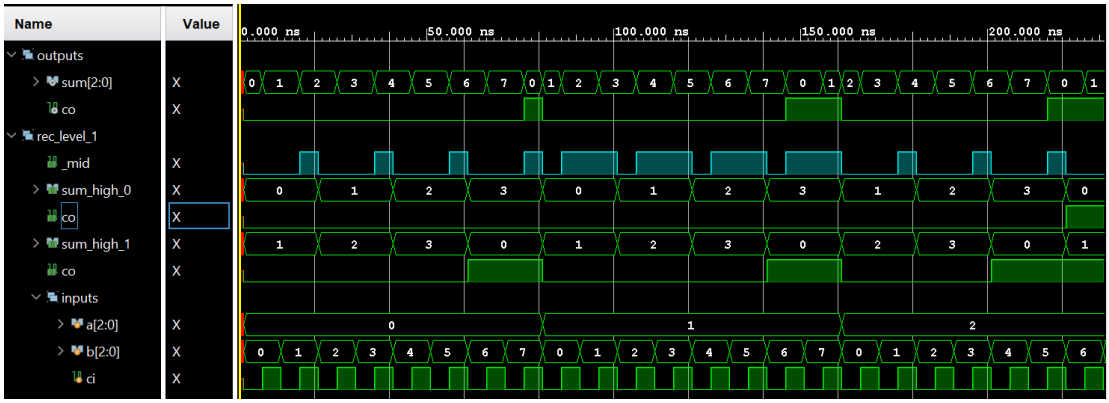


The second and third splitting, up until reaching the base case (using FA to compute):



As we confirmed, the recursion tree is 3 layers deep – which is expected given the inputs are 3 bits long. Now, all that's left to confirm is that the different parts of the recursion did their job correctly.

Rec level 1-



We will inspect the selected rows (using k=1):

a_{10}	b_{10}	c_i	c_mid	$sum\_high\_0_{10}$	$sum\_high\_1_{10}$
0	0	1	0	0	1

Since looking at the lower CSA as presented at the diagram, the addition will be 1 – meaning $s[0] = 1,\ c[1] = c\_mid = 0$. The mux will choose $sum\_high\_0 = 00$, and the full, concatenated output will be, as clearly seen in the sum bus:

$$c0\_high\_0 \cdot sum\_high\_0 \cdot s[0] = 0001$$

a_{10}	b_{10}	c_i	c_mid	$sum\_high\_0_{10}$	$sum\_high\_1_{10}$
1 [001]	6 [110]	0	0	3	0

Looking at the lower CSA, the addition will result in 1 - $s[0] = 1; c[1] = c\_mid = 0$. The mux w will choose $sum\_high\_0 = 3 = 11$, and the full, concatenated output will be, as clearly seen in the sum bus:

$$c0\_high\_0 \cdot sum\_high\_0 \cdot s[0] = 0111$$

a_{10}	b_{10}	c_i	c_mid	$sum\_high\_0_{10}$	$sum\_high\_1_{10}$
1 [001]	6 [110]	1	1	3	0

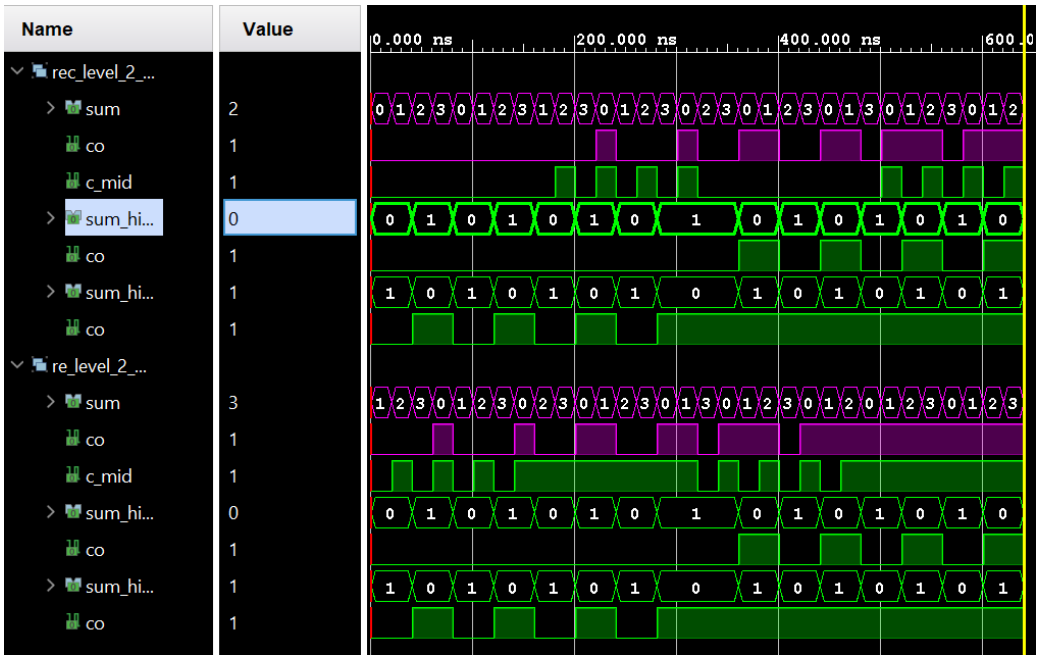
Looking at the lower CSA, the addition will result in 1 - $s[0] = 0; c[1] = c\_mid = 1$. The mux w will choose $sum\_high\_1 = 0 = 00$, and the full, concatenated output will be, as clearly seen in the sum bus:

$$c0\_high\_1 \cdot sum\_high\_1 \cdot s[0] = 1000$$

In summary, it seems that the 1st level of the recursion works well.

Proceed to show the 2nd level –

The inputs to these adders are the two input's MSBs. The following (the 1st group on top is sum_high_0, the2nd group at the bottom is sum_high_1):

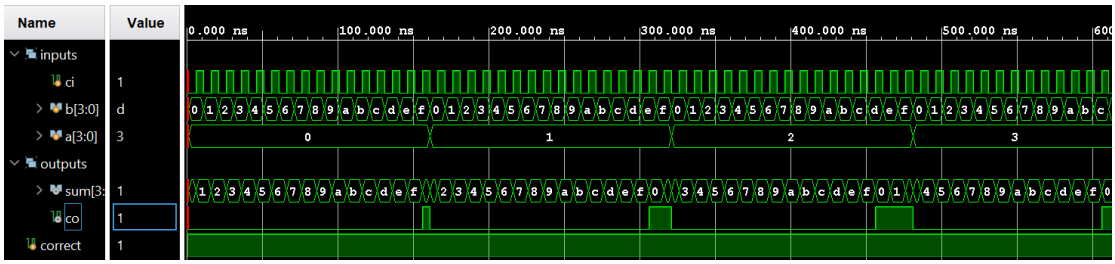


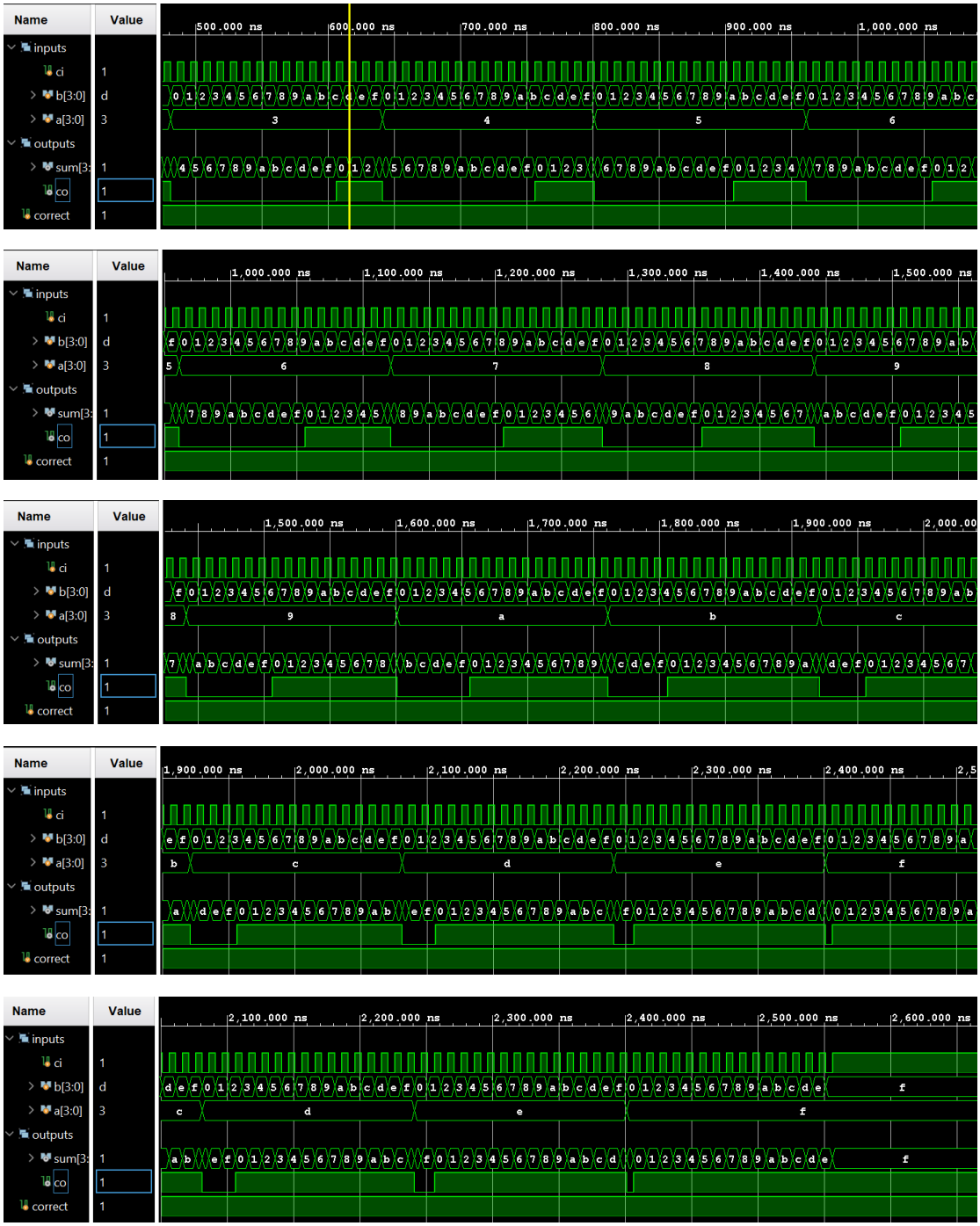
As expected, we can see that the sum_high_1 output is identical to the sum_high_0 output, with a shift. This is logical since the count at sum_high_1 is always larger by one. Checking individual cases:

a_{10}	b_{10}	c_i	$c\_mid = c[1]$	$s[0] = c[0]$	Coh	$sum\_high\_0_{10}$	Col	$sum\_high\_1_{10}$	Final
0	0	0	0	0	0	0	0	1	0/00
0	0	1	0	1	0	0	0	1	0/01
1	1	0	1	0	0	0	0	1	0/10
1	1	1	1	1	0	0	0	1	0/11
3[11]	1[01]	0	1	0	0	1	1	0	1/00
3[11]	1[01]	1	1	1	0	1	1	0	1/01

As seen in the examples given above, the 2nd layer of the recursion works as well. We won't be checking the correctness of the full adder, since we visited that in the 1st part of the report.

Question 8 – the following screenshots presents the inputs\ outputs of the entire design:





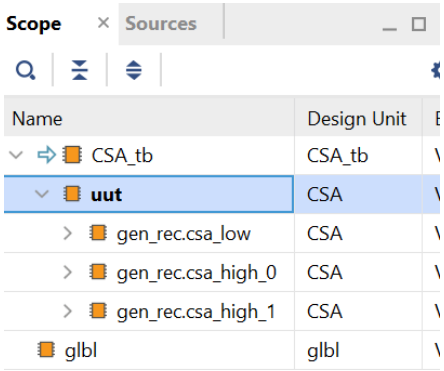
Now, looking at specific inputs and outputs to confirm the results are as expected:

a_{10}	b_{10}	c_i	sum_{10}	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	0
0	2	0	2	0
0	2	1	3	0
0	3	0	3	0
0	3	1	4	0
0	4	0	4	0

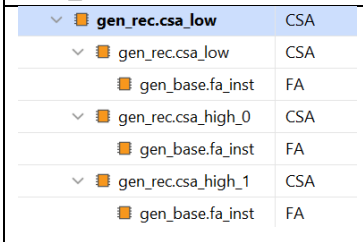
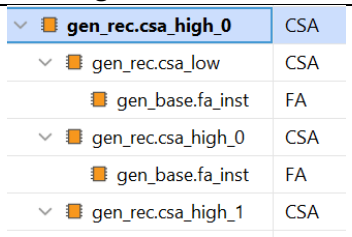
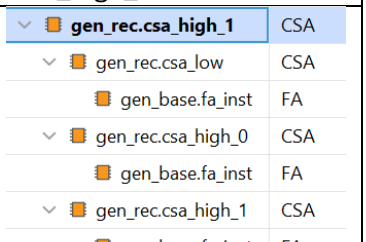
0	4	1	5	0
0	5	0	5	0
0	5	1	6	0
0	6	0	6	0
0	6	1	7	0
0	7	0	7	0
0	7	1	8	0
1	8	0	9	0
1	8	1	10[a]	0
1	9	0	10[a]	0
1	9	1	11[b]	0
1	10	0	11[b]	0
1	10	1	12[c]	0
1	11	0	12[c]	0
1	11	1	13[d]	0
1	12	0	13[d]	0
1	12	1	14[e]	0
1	13	0	14[e]	0
1	13	1	15[f]	0
1	14	0	15[f]	0
1	14	1	0	1
1	15	0	0	1
1	16	1	1	1

And indeed, as expected, the results match the theory. Now, looking at the recursive levels and confirming their functionality:

First layer (the same as the prev. run):



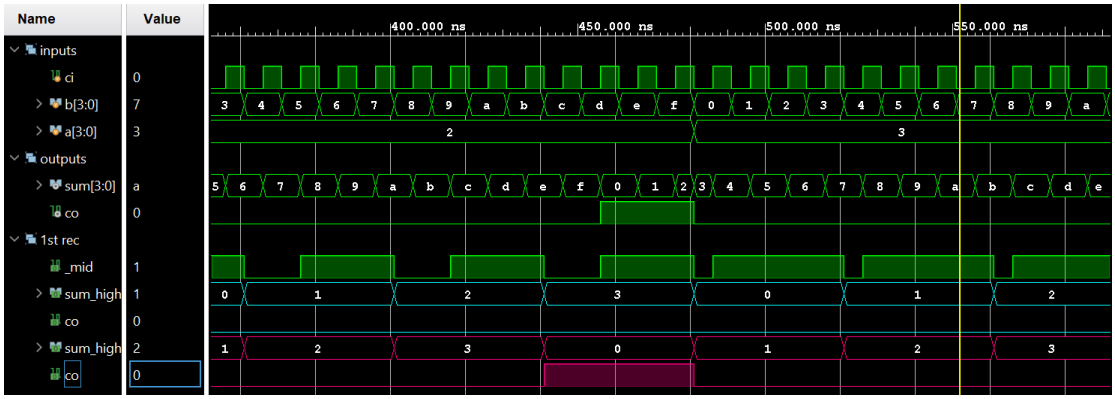
Second and third layers:

Csa_low	Csa_high_0	Csa_high_1
		

The difference between this run and the previous run is the additional recursive call at the low CSA, since now, the input bit division is to two pairs. We will confirm this circuit recursive calls perform as expected by –

- 1st layer -a detailed analysis.
- 2nd layer –
    - going over in details the new lower CSA recursive call (we expect a difference between this analysis and the previous rec analysis done in the last run with high_0 and high_1 calls, since ci is now a changing variable)
    - csa_high_0, csa_high_1, we will just make sure they are identical to the same parts of the last run, since their construction and recursive calls are the same.

1st rec layer analysis: below, the 1 rec blocks output signals are shown, plus the MUX selection signal, c_mid. In teal – csa_high_0 outputs, and in pink csa_high_1 outputs.



We will now inspect selected rows –

b_{10}	a_{10}	c_i	c_mid	$sum\_high\_0_{10}$	$sum\_high\_1_{10}$
2[0010]	7[0111]	0	1	1[01]	2[10]

The inputs of the low csa is 2 and 3, making its output 5. Meaning $c\_mid \cdot s[1:0] = 101$. the mux will "choose" the output of sum_high_1, meaning the result will be:

$$c0\_high\_1 \cdot sum\_high\_1 \cdot s[1:0] = 01001 = c0 \cdot sum[3:0]$$

b_{10}	a_{10}	c_i	c_mid	$sum\_high\_0_{10}$	$sum\_high\_1_{10}$
a[1010]	3[0011]	1	1	2[10]	3[11]

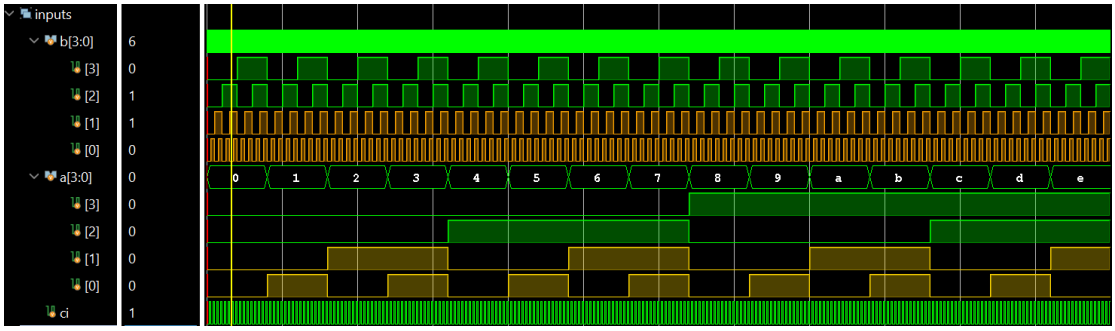
The inputs of the low csa is 2 and 3, making its output 5. Meaning $c\_mid \cdot s[1:0] = 101$. the mux will "choose" the output of sum_high_1, meaning the result will be:

$$c0\_high\_1 \cdot sum\_high\_1 \cdot s[1:0] = 01101 \rightarrow 0 \cdot e = c0 \cdot sum[3:0]$$

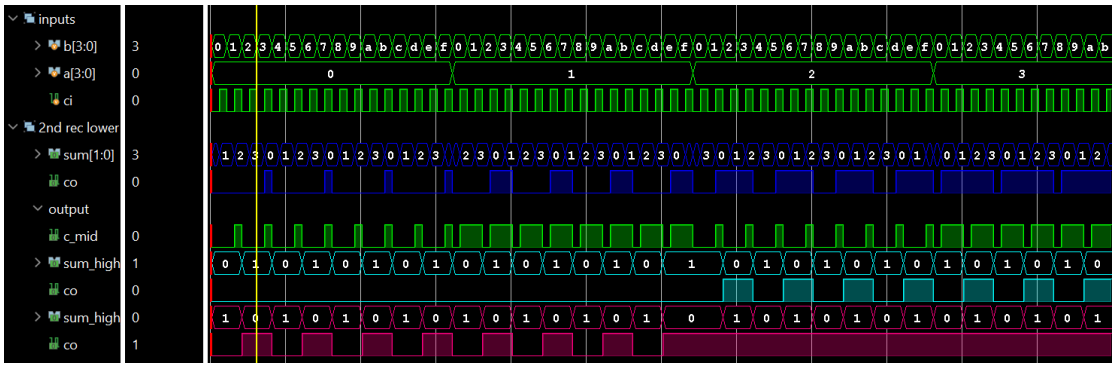
In summary, it seems that the 1st level of the recursion works well.

Proceed to show the 2nd level –

Lower CSA rec call analysis. Looking at this part of the recursion, by the block diag. the two input bits to this CSA are the two LSBs and ci –



The following is the inner recursive layer (the individual blocks in it are FAs).



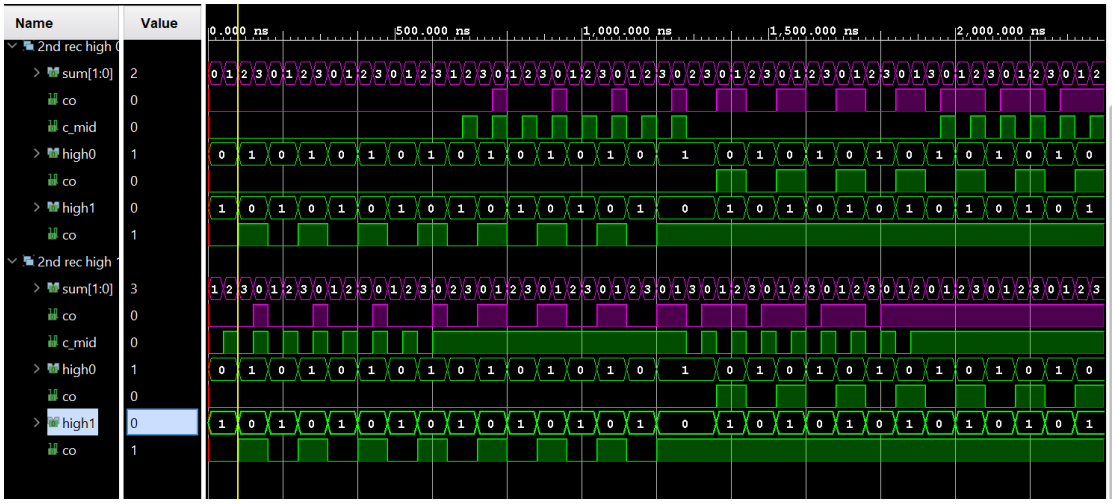
- in blue – lower CSA output (sum bus and co).
- in teal – lower CSA CSA_high_0 block output.
- in magenta – lower CSA CSA_high_1 block output.

As expected, the magenta signal is a shifted version of the teal signal, since the count starts at a higher value. Now, looking at specific values in the cycle –

a_{10}	b_{10}	c_i	$c\_mid = c[1]$	$s[0] = c[0]$	Coh	$sum\_high\_0_{10}$	Col	$sum\_high\_1_{10}$	Final
0	0	0	0	0	0	0	0	1	0/00
0	0	1	0	1	0	0	0	1	0/01
1	1	0	1	0	0	0	0	1	0/10
1	1	1	1	1	0	0	0	1	0/11
3[11]	1[01]	0	1	0	0	1	1	0	1/00
3[11]	1[01]	1	1	1	0	1	1	0	1/01

As expected, This rec block works as it should.

CSA_high_0/1: as noted before, in this part we just want to make sure that the blocks produce an identical outcome to the same blocks in the last run. The inputs of these blocks are still the two input's MSBs –



We can clearly see the blocks are identical – which means the recursion works as expected.

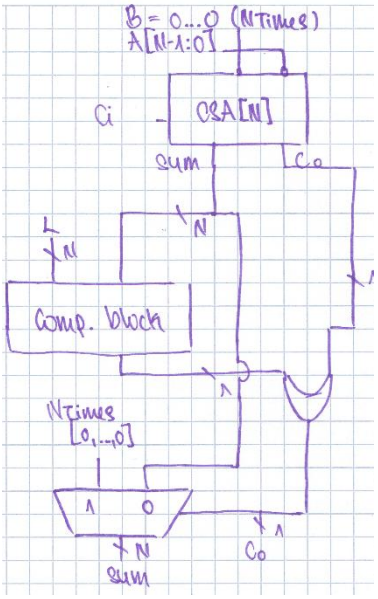
Part 3 - Limited Incrementor

Question 1 – The purpose of this module is to perform a "guarded" increment. It adds a carry-in (ci) to an input value (a) but prevents the result from exceeding a specified limit (L).

- if the result of the addition $a + ci$ is strictly less than the limit L, the module outputs the sum and sets the carry out co to 0.
- If the addition result is greater than or equal to the limit, the module "wraps around" or resets the output sum to 0 and sets the carry out co to 1.

Design approach:

- A CSA module that takes a and ci (b=0) as inputs to generate a "raw sum".
- A block that compares the raw sum against a constant value L.
- Selection logic (MUX) - A multiplexer that chooses between the "raw sum" and the value 0, controlled by the output of the comparison logic.

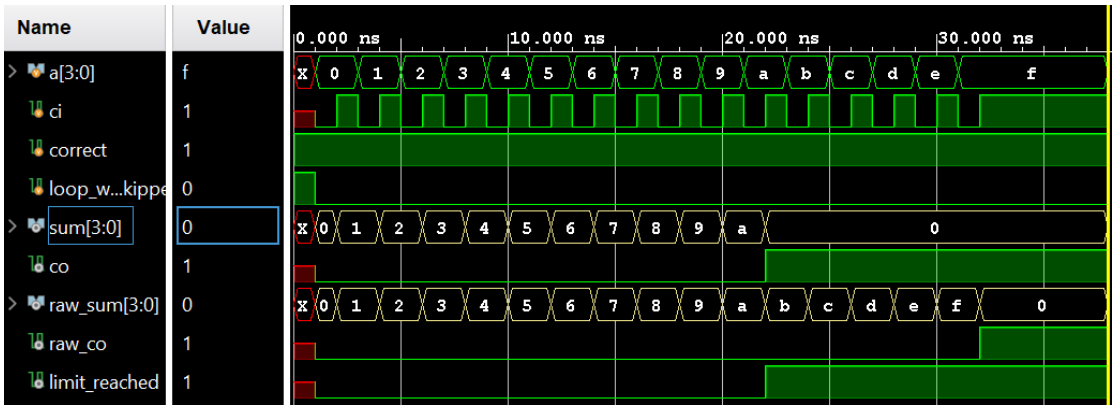


The CSA functionality is clear – by setting B=0 we make sure we only perform addition between a and ci, as requested. The MUX functionality is also clear – selecting between the outputs set by the functionality as defined in the report. The OR gate guarantees that the limit condition is detected both when the arithmetic sum exceeds the representable range (raw carry-out) and when it reaches the programmed limit L within range.

The comparator performs an unsigned magnitude comparison between raw sum and the constant limit L by evaluating the most significant differing bit, if raw sum is greater than or equal to L, the comparator asserts its output.

Question 5 – The raw sum differs from the outputted sum when the arithmetic result reaches or exceeds the predefined limit L. In this case, Lim_Inc enforces a rollover by forcing the output sum to zero and asserting the carry-out signal, thereby implementing a modulo-L incrementor rather than a plain adder. The difference is intentional and enables controlled counting behavior.

Question 6 – below is the asked for annotated waveform simulation:



As denoted in the follow-up email, we ran the simulation for L=11, meaning the count is to stop at 10.

Correctness – the correct signal is always = 1, meaning the test we defined always holds, the output matches the expect output exactly through the entire simulation, we can also see that past 10 (sum=a), sum =0 and co =1, as defined.

Sum and raw sum difference – when the limit is reached (when the limit_reached signal =1, or when co=1, we can see that the values of sum and raw sum are different. This is expected due to the reasons we detailed in the last question.

Part 4 – Counter

Question 1 – the counter will be the time keeping engine of the stopwatch. Its job is to count the clock cycles of the FPGA clock, convert those clock cycles into elapsed time in seconds, and output the time in a human readable decimal format – allowing a counting range between 0-99.

The output is an 8-bit bus, which represents a two-digit decimal number in seconds. Internally, the module maintains a conceptual counter $cc(t)$ – which is the number of clock cycles since the last reset. It is used to derive real time using the clock frequency. the FPGA clock runs at $100[MHz]$, which means that $100 \cdot 10^6$ clock cycles equal one second. The elapsed time in seconds is computed conceptually as:

$$\text{elapsed sec} = \left\lfloor \frac{cc(t)}{\text{CLK FREQ}} \right\rfloor mod\ 100$$

this ensures that only 2 decimal digits are needed, since the stopwatch rolls over every 100 seconds.

Question 2 – To utilize a Lim_Inc(L) module and a register to count seconds from a $100[MHz]$ clock, we must create a frequency divider circuit that generates a pulse every 10^8 clock cycles. The process involves treating the Lim_Inc as the combinational logic and the register as the state memory:

- Using a feedback loop – connecting the output of the register is connected to the input a of the Lim_Inc(L) module.
- The sum output of the Lim_Inc(L) is connected back to the input of the register.
- The register samples the sum value on the positive edge of the clk, effectively incrementing the stored value by 1 each cycle (assuming ci=1).
- By setting $L = 10^8$ the module will count from 0 to 99,999,999.
- When the count reaches its final value, the Lim_Inc(L) co signal will assert to 1 for exactly one clock cycle, and the register will reset to 0 on the next cycle.

To specifically "toggle" or trigger the counting of the seconds, the co from the $100\ [MHz]$ divider acts as the "enable" signal (connected to the ci input) for the next Lim_Inc module in the hierarchy. This second Lim_Inc would have a limit of $L = 10$ to track the actual seconds digits (0-9). Because the co of the first stage only stays high for one cycle every second, the second stage only increases its value once per second.

Question 3 - The number of Lim_Inc modules is determined by the total number of distinct frequency division stages and counting digits required for the stopwatch display.

Clock divider - At least one module is needed to divide the high-frequency master clock ($100\ [MHz]$) down to a usable $1\ [Hz]$ pulse.

Digit counters - Each decimal digit in the output requires its own incrementer. Since the output time_reading[7:0] consists of two 4-bit digits (sec (0–9) and deca-sec (0–9)) we need one module for the units place and another for the tens place.

So, using this design logic we will use 3 modules – one for the $1[Hz]$ pulse, one for the second digit, and one for the deca-sec digit.

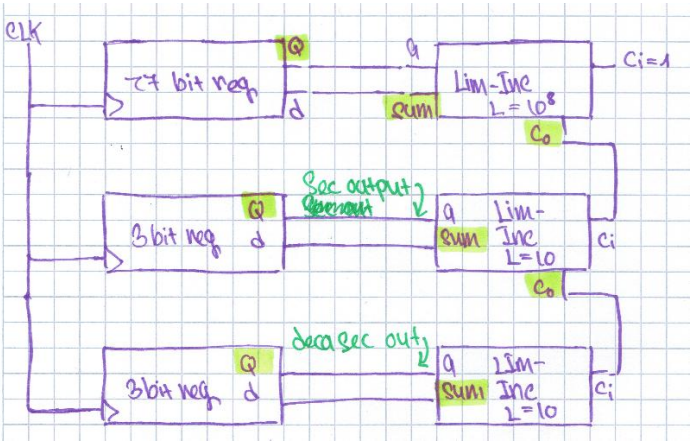
Question 4 – the parameter L is determined by the ration between the frequency of the input clock and the target frequency of the output pulse – $L = f_{in}/f_{out}$. Effectively counting the clock cycles needed to be counted as one unit, to produce the desired frequency.

Deca – sec, derived from the clk input	Deca -sec, derived from the sec pulse
$L = \frac{100 \cdot 10^6}{0.1} = 10^9$	$L = \frac{1}{0.1} = 10$

Question 5 - If the clock frequency changes from $100\ [MHz]$ to $50\ [MHz]$, the primary change occurs in the parameter L of the initial frequency divider (the clock divider). Since the input frequency is now half as fast, the divider needs to count half as many cycles to reach the same time interval. For a $1\ [Hz]$ tick, $L = 50 \cdot 10^6/1 = 50 \cdot 10^6$. The rest of the Lim_Inc L parameters can stay the same (using the deca-sec from the sec inc logic).

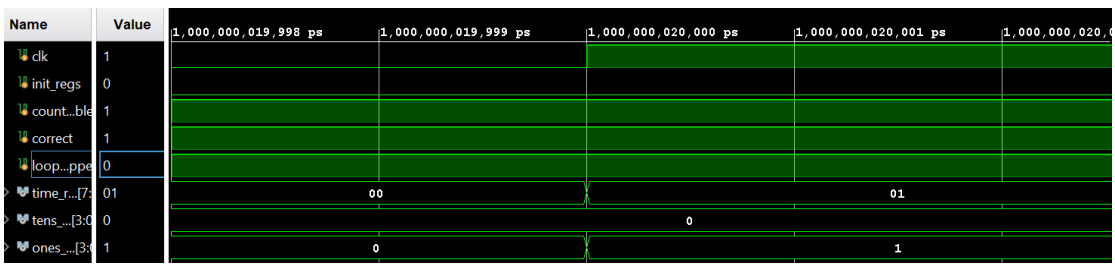
If we would've kept L in the same value, the stopwatch would run at half-speed (it would take 2 real seconds for the display to increment by 1).

Question 6 – as described in the previous sections, the block diagram is the following:



The outputs of the entire module is written in green, the outputs of each block used in the design is highlighted in green. Ci at the 1st Lim_Inc acts as the count enable. When it equals to 0, the count never progresses – therefore "jamming" the entire counting chain.

Question 10 – the simulation took a long time to run, so we ended up not running it through the whole range of possible representable seconds.



Timing accuracy and clock frequency – the last screen capture is on the smallest scale possible on the simulation window – we can clearly see the high precision in the second count. This also demonstrates that the Lim_Inc module used for frequency division is successfully reaching its limit (L = 100,000,000) before generating the carry-out (co) signal to trigger the next stage. The same precision is evident in the transition from 0 to 1 on the 10-secound count.

BCD output representation - The time_reading[7:0] signal shows values like 09, 10, 11, in hexadecimal format. This proves the logic in Counter.v is correctly concatenating the two four-bit BCD strings.

Synchronous behavior and carry propagation – All digit changes are perfectly aligned with the rising edge of the simulated clock. There is no visible "ripple" effect because the Counter is a synchronous module where registers are updated simultaneously using an always@(posedge clk) block.

When the ones_seconds counter reaches 9 and rolls over to 0 at 10 seconds, it simultaneously triggers the tens_seconds counter to increment to 1. This confirms that the carry-out (co) of the units Lim_Inc is properly connected to the carry-in (ci) of the tens Lim_Inc.

Testbench verification logic – The correct signal remains at high throughout the provided window. This indicates that the automated comparison logic in the testbench, which checks if ones_seconds_wire !== os || tens_seconds_wire !== ts, has not detected any mismatches.

The ts and os integer signals show the testbench is currently iterating through the simulation, satisfying the requirement to verify a correct count of at least one second.

Question 11 – The Lim_Inc module cannot hold its own count internally because it is designed as a purely combinational circuit, whereas storing a count requires state, which can only be maintained by registers (flip-flops) in a synchronous design.

By design, Lim_Inc contains no registers and therefore no memory. Its outputs depend only on its current inputs, which makes it a combinational module. In contrast, a counter must remember its value from one clock cycle to the next, update this value synchronously with the clock, and respect control signals such as reset and enable. This behavior requires state retention, which can only be implemented using flip-flops (registers) driven by a clock. If Lim_Inc were to contain its own internal register, it would no longer be a generic combinational building block.

Question 12 – the output of the counter module is time_reading = {dasec[3:0], sec[3:0]}, where each field's range is 0-9. The display time is $dasec \cdot 10 + sec$, meaning the max value that I representable is $9 \cdot 10 + 9 = 99sec$. After reaching 99 seconds, the next valid tick causes both digits to roll over to 00.

Question 13 – To display minutes and seconds (MM:SS), the counter hierarchy must be extended. To support minuets, we need to adjust the L value for tens of seconds, adjusting it's range to 0-5 (setting L=6).

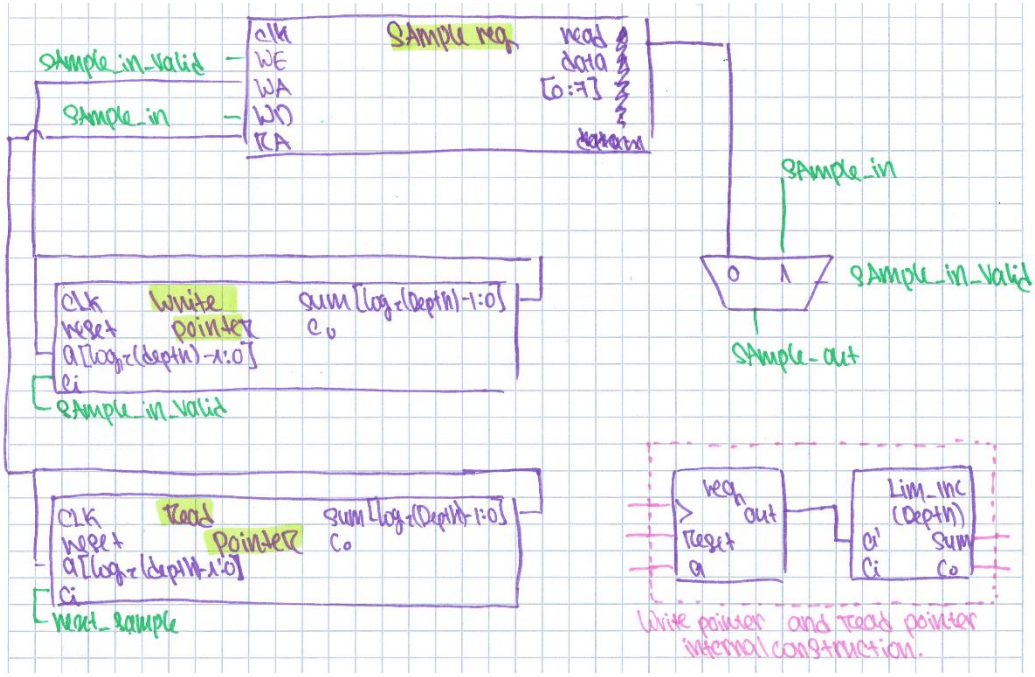
Then, we need to add two Lim_Inc(10) modules, allowing a count of 99 minutes, where each stage increments only when the previous stage rolls over (via its co output). We also

need to add two more registers - each new decimal digit requires a 4-bit register to store its value and a tick signal from the previous stage. The new output format will be time_reading = {min_tens, min_ones, sec_tens, sec_ones}.

Part 5 – Stash

Question 1 - The Stash module is a synchronous component designed to store time samples from the stopwatch upon a user's request. It needs to support the following logic and functions:

- maintains a set of registers to store up to DEPTH (default is 5) samples, where each sample is 8 bits wide
- If more than the allowed number of requests arrive, the module must overwrite the oldest samples
- A sample from the input (sample_in) is only stored if the qualifier signal sample_in_valid is high
- Users can cycle through stored samples using the next_sample signal. Crucially, the module must provide immediate visibility: when a new sample is captured, it must appear on sample_out immediately, and the internal pointer should "jump" to this new sample
- Asserting the reset signal clears the stash, initializing all stored samples to zero

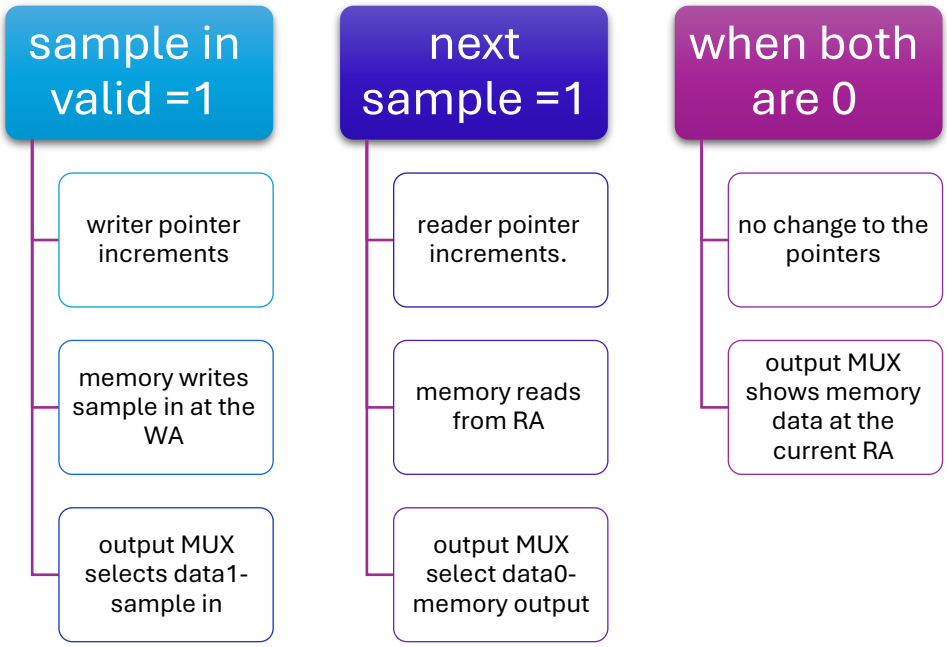


The sample reg is a group of DEPTH 8 bit registers, which is the core storage element. The write port is controlled by the Write Pointer. When sample_in_valid is high, the value on sample_in is written to the memory location indexed by the Write Pointer. The read port is controlled by the Read pointer. When next_sample is high, the read pointer increments it's value by one, setting the read data of the sample red to the "next" register in the array. That output is then connected to a mux – allowing the override functionality as the question demanded – when a new sample is written, the display will show the new sample.

The writer pointer - A Lim_Inc module with L = DEPTH. It acts as a counter that increments (wrapping around) each time a valid sample is written. Connecting it to a register, allows us to create a circular structure, feeding Lim_Inc with the previous value, to ensure proper handling of the memory spaces. The register also interduces a reset node, which is required for the specified reset property the question demanded.

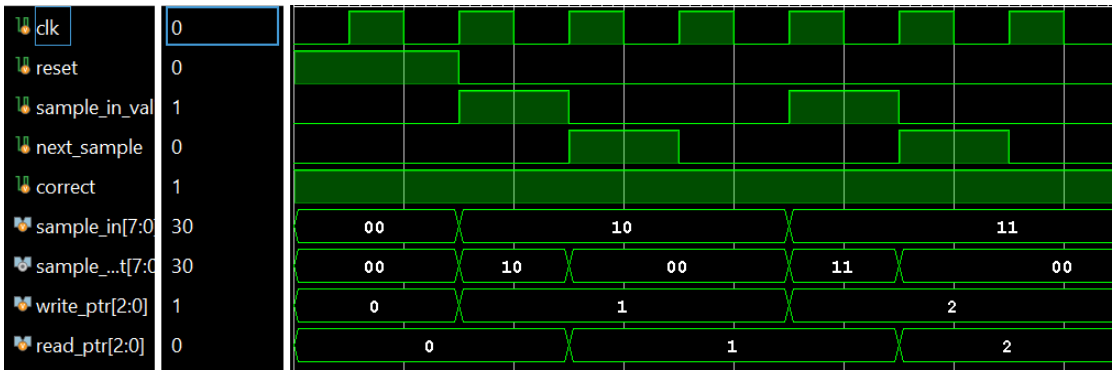
The read pointer – uses the same logic as the writer pointer, now incrementing when the user wants to see the next sample stored – next_sample ==1.

Signal flow summary (assuming valid signal combinations) –



Question 5 – the simulation included several parts, in order to make sure everything works as expected.

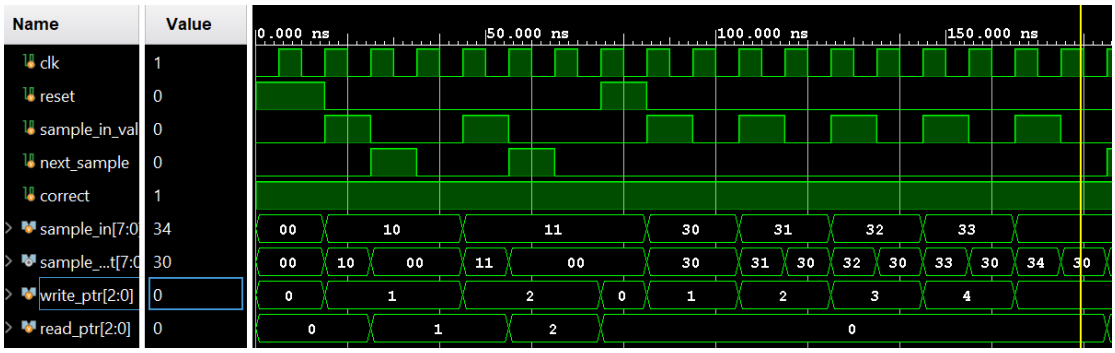
Part 1 – examining sample flow:



Sample in valid	Write ptr	Next sample	Read ptr	
1	+1=1	0	0	sample out = sample in =10 as required. 10 is stored in cell num. 0
0	1	1	+1=1	Sample out= what was stored in cell num. 1, which is 0 due to the restart
0	1	0	1	Sample out= what was stored in cell num. 1

1	+1=2	0	1	sample out = sample in =10 as required. 10 is stored in cell num. 1
0	2	1	+1=2	Sample out= what was stored in cell num. 2, which is 0 due to the restart
0	2	0	1	Sample out= what was stored in cell num. 2

Part 2 – testing write ptr rollover:

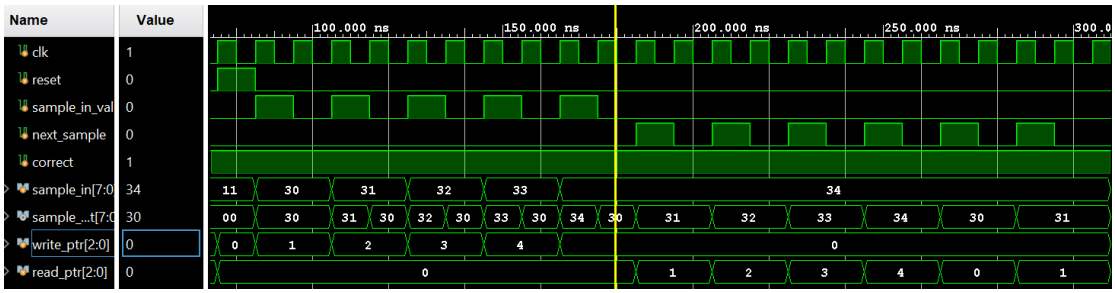


After the last test, we pulled the restart signal, restarting all the registers. Now, Appling pulses to sample in valid, we test the rollover of the write ptr block. For each cycle of the sample in valid signal, we can clearly see that when the signal is high, the output= the input, but as the signal drops the output= the value stored at cell num. 0. The memory now holds:

Cell num.	0	1	2	3	4
value	30	31	32	33	34

The rollover indeed works, the next value of write_ptr is 0, as shown in the next screen capture.

Part 3 – testing read ptr rollover:



Now we apply 6 pulses to the read ptr block. We can see that the stored values are as in the table above, and the rollout logic works as expected.

Question 6 – The sample_out selection logic was implemented using a combinational 2-to-1 multiplexer controlled by the sample_in_valid signal.

- When sample_in_valid is high (indicating a new sample is being written), the multiplexer directly routes the incoming sample_in to sample_out, providing immediate visibility of the new data in the same clock cycle.
- When sample_in_valid is low, the multiplexer selects the stored value from memory at the current read address.

This approach satisfies the "jump to new sample" requirement without complex pointer manipulation: the read pointer remains unchanged, but users immediately see new samples as they arrive, bypassing the memory read path entirely.

Question 7 – instead of using an output multiplexer, we could implement dynamic pointer redirection. When sample_in_valid = 1, we would temporarily override the read address to point to the write location, then restore it when the write completes. It would look something like this:

```
wire [N-1:0] effective_read_addr = (sample_in_valid) ?
write_addr : read_addr;
assign sample_out = memory[effective_read_addr];
```

Why is our approach better:

- It avoids unnecessary memory reads for new samples, saving power.
- The critical path in our design is shorter, just through the output mux rather than through both address selection and memory access.
- No address switching means no risk of reading wrong memory locations during transitions.

Why pointer approach might be better (in a system where):

- Memory has a registered output (pipelined)
- You want consistent timing for all reads (always through memory)
- Power consumption is not a concern

For this stopwatch design with small DEPTH values and simple memory, our multiplexer approach is clearly superior in simplicity, efficiency, and timing.

Part 6 – Control

Question 1 – the state diagram in Figure 1 is drawn as a Mealy machine because the output init_regs is shown on transitions and depends on both the current state (IDLE) and the input trig. However, the FSM can be converted into an equivalent Moore machine by splitting the IDLE state into two substates: one where init_regs=1 and another where init_regs=0. This would eliminate the input dependence in the output logic, making it purely state-based. Since such a transformation is always possible, the underlying behavior is fundamentally Moore-representable.

Implementation Choice – in the Verilog implementation, we retained the Mealy structure, with init_regs = (state == IDLE) && (trig == 0). This approach simplifies the design by avoiding extra states, reduces latency because the output responds immediately to input changes, and stays faithful to the given diagram. While a Moore implementation would be equally correct, the Mealy style offers a more direct and efficient realization for this control module.

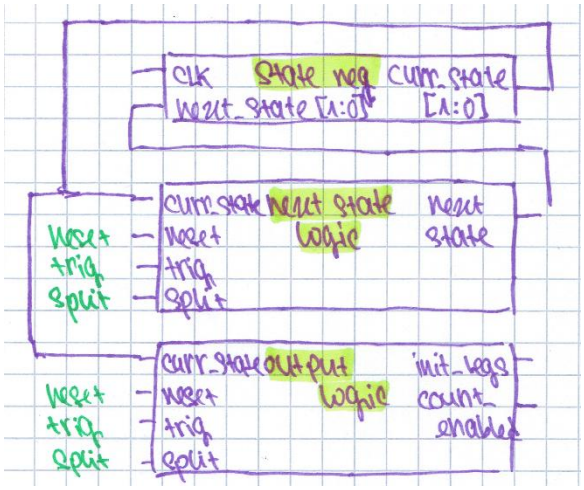
Question 2 – we will use combinational logic, as well as a state register to transition between the states. The combinational logic will assist us in the following cases:

- Next-state logic (δ): taking the current state and inputs, implementing the transition rules from the figure, and outputting the next state value.
- Output logic (λ): takes the current state and inputs, produces init_regs and count_enabled according to the transition labels in Figure 1

Transforming the FSM given into a table (using the following encoding for the states – IDLE=00, COUNTING= 01, PAUSE=10, SPLIT=11):

From state	Inputs (reset, trig, split)	Next state	Outputs
IDLE	1,*,*	IDLE	1,0
IDLE	0,0,*	IDLE	1,0
IDLE	0,1,*	COUNTING	1,0
COUNTING	0,0,*	COUNTING	0,1
COUNTING	1,*,*	IDLE	0,1
COUNTING	0,1,*	PAUSED	0,1
PAUSED	0,0,0	PAUSED	0,0
PAUSED	0,0,1	IDLE	0,0
PAUSED	0,1,*	COUNTING	0,0

This will help us code the exact logic needed in the combinational blocks in the following block diagram.



The next state logic code will be something like:

```
if (current_state == IDLE)
    next_state = (reset || !trig) ? IDLE : COUNTING
else if (current_state == COUNTING)
    next_state = reset ? IDLE : (trig ? PAUSED : COUNTING)
else if (current_state == PAUSED)
    next_state = reset ? IDLE : (split&&!trig ? IDLE : (trig
? COUNTING : PAUSED))
```

This code demonstration does not include cases where current state = next state – that will be included in the Verilog code.

The output logic code will be something like:

```
init_regs = (current_state == IDLE) && (reset || !trig)
count_enabled = (current_state == COUNTING)
```

Where this is also, not the final Verilog code.

Note – the explanation given above about the logic is only conceptual, since the Verilog code skeleton needed to be filled uses 1-hot encoding.

Question 6 – we ran the simulation, testing the following transitions, where the outputs are a tuple of (init_regs, count_enabled), and using the one-hot encoding (IDLE=1, COUNTING=2, PAUSED=4 in decimal basis)–

Test	Transition Being Tested	Inputs Sequence (reset, trig, split)	Expected Outputs
1	X → IDLE (X-1)	reset=1 → 0 (held)	(X,X) → (1,0)
2	IDLE → COUNTING (1-2)	trig=0 → 1 → 0 (pulse)	(1,0) → (0,1)
3	COUNTING → PAUSED (2-4)	trig=0 → 1 → 0 (pulse)	(0,1) → (0,0)
4	PAUSED → COUNTING (4-2)	trig=0 → 1 → 0 (pulse)	(0,0) → (0,1)
5	COUNTING → COUNTING (2-2)	trig=0 (held)	(0,1) → (0,1)
6	COUNTING → IDLE (2-1)	reset=0 → 1 → 0	(0,1) → (1,0)
7	PAUSED → IDLE (4-1)	split=0 → 1 → 0 (pulse, trig=0)	(0,0) → (1,0)
8	PAUSED → PAUSED (4-4)	trig=0, split=0 (both held)	(0,0) → (0,0)

And a more detailed action to result table –

Test #	Inputs (R,T,S)	State (before→after)	Outputs (before→after)	Action
1	(1,0,0)	X → IDLE (x-1)	(X,X) → (1,0)	Reset asserted
1	(0,0,0)	IDLE (1)	(1,0)	Reset released
2	(0,1,0)	IDLE → COUNTING (1-2)	(1,0) → (0,1)	trig pulsed
2	(0,0,0)	COUNTING (2)	(0,1)	trig released
3	(0,1,0)	COUNTING → PAUSED (2-4)	(0,1) → (0,0)	trig pulsed
3	(0,0,0)	PAUSED (4)	(0,0)	trig released
4	(0,1,0)	PAUSED → COUNTING (4-2)	(0,0) → (0,1)	trig pulsed
4	(0,0,0)	COUNTING (2)	(0,1)	trig released
5	(0,0,0)	COUNTING → COUNTING (2-2	(0,1) → (0,1)	No input change
6	(1,0,0)	COUNTING → IDLE (2-1)	(0,1) → (1,0)	Reset asserted
6	(0,0,0)	IDLE	(1,0)	Reset released
7	(0,1,0)	IDLE → COUNTING (1-2)	(1,0) → (0,1)	Setup: trig to COUNTING
7	(0,0,0)	COUNTING (2)	(0,1)	
7	(0,1,0)	COUNTING → PAUSED (2-4)	(0,1) → (0,0)	Setup: trig to PAUSED
7	(0,0,0)	PAUSED (4)	(0,0)	
7	(0,0,1)	PAUSED → IDLE (4-1)	(0,0) → (1,0)	split pulsed
7	(0,0,0)	IDLE (1)	(1,0)	split released
8	(0,1,0)	IDLE → COUNTING (1-2)	(1,0) → (0,1)	Setup: trig to COUNTING
8	(0,0,0)	COUNTING (2)	(0,1)	
8	(0,1,0)	COUNTING → PAUSED (2-4)	(0,1) → (0,0)	Setup: trig to PAUSED
8	(0,0,0)	PAUSED (4)	(0,0)	
8	(0,0,0)	PAUSED → PAUSED (4)	(0,0) → (0,0)	No inputs, should stay

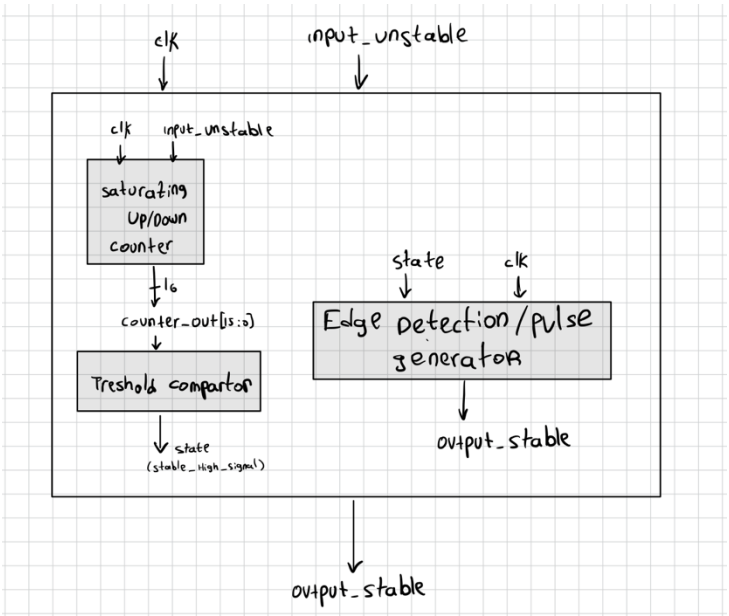
The following is the simulation output –

Comparing the output we got to the table, we can clearly see that up to some intermediate transitions (that were made to test the transitions as stated above, and to avoid racing conditions) we go the expected results.

7. Debouncer

Mechanical buttons on the Basys3 board do not create a clean electrical transition. Instead, they "bounce" between logic levels for several milliseconds. In a stopwatch application, these bounces would be interpreted as multiple rapid presses, causing the FSM to skip states or the counter to start/stop unpredictably. The debouncer ensures that only one stable transition is registered per physical press.

a.



Operation Description: The debouncer uses a **Saturating Up/Down Counter**. When the raw input from the button is high, the counter increments; when low, it decrements. The counter is "saturating," meaning it stays at its maximum or minimum value without rolling over. A comparator monitors the counter: only when the counter reaches its upper threshold is the signal considered "stable high." This effectively filters out high-frequency mechanical noise.

b. done

c. To ensure the system reacts only once per press, the stabilized signal is passed through an Edge Detector. By storing the previous cycle's state in a register, we apply the logic: assign output_stable = (current_state) && (!last_state).

This produces a pulse that is high for exactly one clock cycle (10ns) at the rising edge of the stabilized signal.

d. In debouncer design, a fundamental trade-off exists between noise immunity and response time. Using a longer counter effectively suppresses mechanical bounce artifacts, yet it increases the latency before a button press is registered, which may be noticeable to the user.

The saturated counter method functions as a digital equivalent of an analog low-pass filter (RC circuit) used in earlier experiments. In the analog LPF, a capacitor integrates the noisy signal over time, smoothing out rapid transitions. Similarly, the counter accumulates or decrements based on consecutive clock samples – the counter's "saturation level" corresponds to the RC time constant. A higher counter threshold (or larger RC constant) provides stronger noise rejection but also slower response. Thus, both methods share the same trade-off between filtering effectiveness and response speed, but the digital counter offers more precise control and easier integration into FPGA-based synchronous designs.

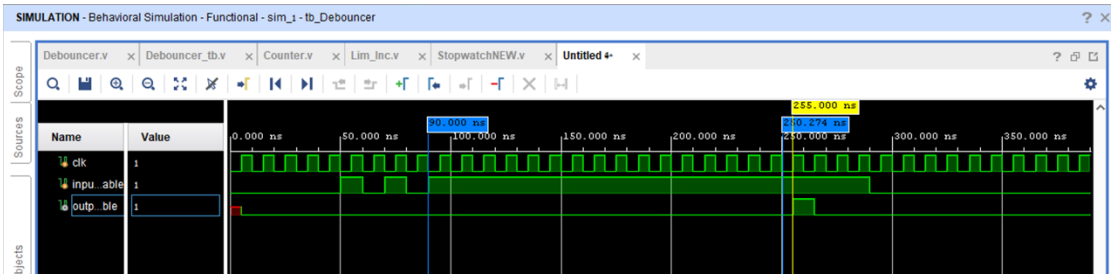
Counter Length Selection: For the final physical implementation on the Basys3 board, we selected a 16-bit counter. Operating at the system clock frequency of $f = 100MHz$ $(clock\ period\ T = 10ns)$, the analytical latency for the signal to reach the stable threshold is calculated as:

$$Latency = 2^{16} * 10ns = 65{,}536 * 10ns \approx 0.65ms.$$

Since typical mechanical bounces last between $1ms$ and $10ms$, this value effectively filters out high-frequency glitches while maintaining a reaction time that is virtually instantaneous to the human eye.

Simulation Strategy and Parameter Adjustment: To verify the logic within a reasonable simulation time frame, we modified the parameter to COUNTER_BITS = 4 and the stopwatch clock frequency to a lower value in the testbench. This allows the saturating counter to reach its threshold ($2^4 = 16$ cycles) within nanoseconds rather than milliseconds, making the internal transitions visible in the waveform.

Simulation Results Analysis:

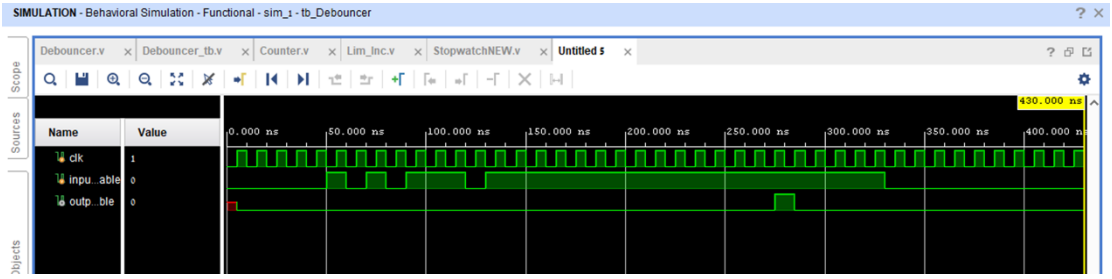


As shown in the simulation results, the debouncer operates exactly as designed:

1. **Noise Filtering:** Even when the raw input signal (btn) transitions, the system does not react immediately. The internal saturating counter begins its "up-count" only when the signal is high.

2. **Threshold Reached:** Once the counter reaches its saturation point (the stable threshold), the internal (state) signal transitions to high.
3. **Single-Cycle Pulse Generation:** The Edge Detector logic identifies this transition and triggers the (output_stable) pulse.
4. **Verification:** In the waveform, we can observe that the resulting pulse is exactly **one clock cycle wide** (10ns), regardless of how long the button remains pressed. This pulse is what successfully triggers the Control FSM and the Stash sampling logic, ensuring that a single physical press corresponds to exactly one system action.

Additional Functional Verification:



To ensure complete design robustness, an additional functional verification was performed during simulation. This 'stress test' was designed to confirm that the debouncer remains stable under non-ideal conditions, such as rapid successive glitches or exceptionally long button presses. By observing that the system ignores the initial jitter and produces only a single, precisely-timed pulse regardless of the input duration, we provided an extra layer of validation. This confirms that the logic is not only theoretically correct but also practically reliable for the final physical implementation on the FPGA board.

8. Seven-Segment Display

The Basys3 board features a 4-digit 7-segment display where all four digits share the same seven cathode lines (seg[6:0]). To display different values on each digit simultaneously, we use **Time-Division Multiplexing**. In this method, only one digit (anode) is activated at any given moment, and the corresponding data for that specific digit is sent to the cathodes. By switching between the four anodes at a high enough frequency, the human eye perceives all four digits as being illuminated constantly due to the phenomenon of **Persistence of Vision**. This method is necessary because it drastically reduces the number of FPGA I/O pins required - from 32 pins (8 pins per digit) down to only 12 pins (8 cathodes + 4 anodes).

a. Done
b.

Operation Description: The Seg_7_Display module manages the 4-digit output using the following logic:

1. **Clock Division (clkdiv):** A 18-bit counter, the **clkdiv register**, slows down the 100MHz system clock. By using its high-order bits (17 and 18), we create a stable enable signal for the display transistors, which cannot switch at the full system speed.
2. **Digit Selection:** A 2-bit signal cycles through four states (00 to 11), sequentially activating each of the four anodes (an) while selecting the corresponding 4-bit nibble from the 16-bit input bus (x).
3. **Decoding:** The selected 4-bit value is passed through a combinatorial BCD-to-7-segment decoder to drive the cathode lines (seg[6:0]), forming the visible digit.

This high-speed cycling (multiplexing) creates the illusion of four constantly lit, independent digits.

C.

To achieve a stable and flicker-free display, the refresh rate must be carefully selected. According to the design requirements, the total refresh cycle for all digits should ideally fall between **1ms and 16ms** (corresponding to a frequency range of 60Hz to 1000Hz).

Parameter Selection: In our implementation, we utilized bits 17 and 18 of a free-running counter driven by the 100MHz system clock as the selection lines for the multiplexer.

Mathematical Derivation: The most significant bit of the selection logic (bit 18) toggles every 2^{18} clock cycles. The total time required to complete a full cycle across all four digits is calculated as follows:

$$T_{total} = \frac{2^{18}}{Clock\ Frequency} = \frac{262{,}144}{100 * 10^6\ Hz} \approx 2.62[ms]$$

Compliance with Requirements:

1. **Timing:** The calculated total refresh period of **2.62ms** is well within the required **1ms to 16ms** range. This ensures that the switching is fast enough to be imperceptible to the human eye (Persistence of Vision) while allowing enough time for the hardware drivers to stabilize.
2. **Refresh Rate:** This period translates to a total refresh frequency of approximately **381Hz**. Since each digit is active for a quarter of this time ($\approx 0.65ms$), the individual digit refresh rate is roughly **95Hz**, which is significantly above the 60Hz threshold required to avoid flickering.

Simulation Verification: In our simulation waveforms, we can observe the signals switching exactly according to this timing logic. The sequential activation of the anodes confirms that the clock division and the 2-bit counter provide a consistent and reliable refresh cycle for the 4-digit display.

Simulation Waveform Analysis:

The simulation of the Seg_7_Display module was designed to verify the correct integration between the clock division logic and the hardware drivers. By observing the signals in the waveform, we can confirm that the system correctly translates a static 16-bit BCD input into a dynamic, time-multiplexed output. The following analysis demonstrates this functionality through two different perspectives: a wide-scale view of the cycling process and a detailed inspection of single-digit data accuracy.

Simulation Waveform – Zoom-Out (Full Multiplexing Cycle):

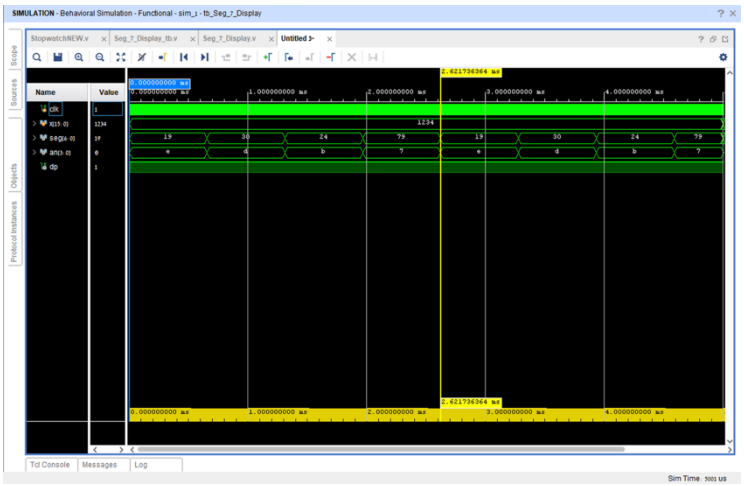
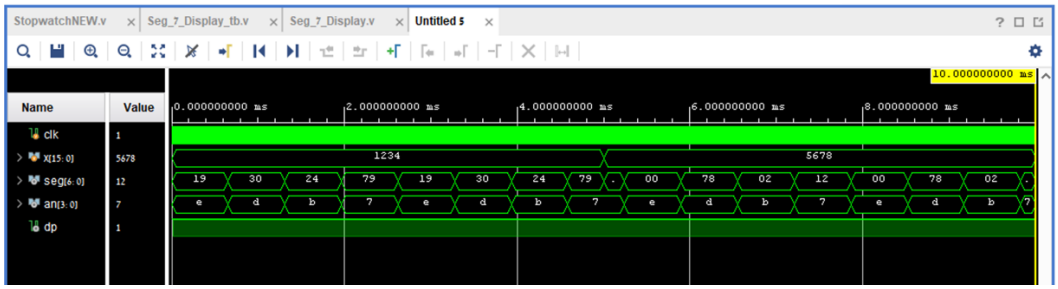


Figure 1- complete cycle of the Time-Division Multiplexing (TDM) logic.



- **Anode Cycling:** The an (anode) signal is seen switching rapidly between its four states: 1110, 1101, 1011, and 0111. Each state represents the activation of one specific digit on the display.
- **Refresh Frequency:** The waveform shows that all four digits are refreshed in a continuous loop. This periodic switching is what allows the human eye to perceive four independent digits simultaneously on the board.

- **Dynamic Data:** While the anodes cycle, the seg (cathode) bus updates its value to match the data intended for the currently active digit, proving the 4-to-1 Multiplexer is synchronized with the anode decoder.

Simulation Waveform – Zoom-In (Single Digit Detail):

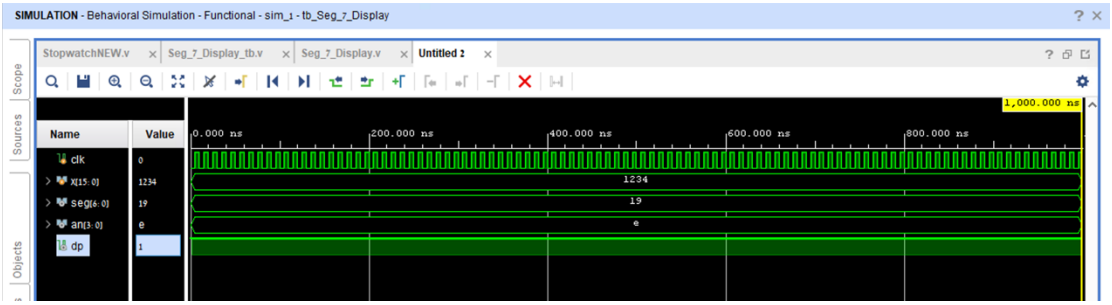


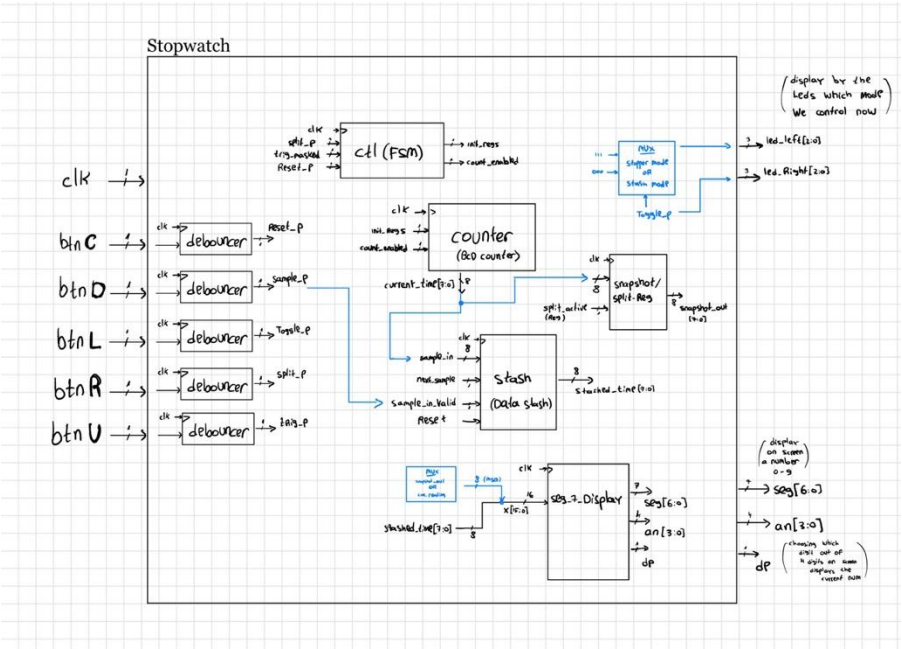
Figure 2- specific time window where only one anode is active

- **Segment Pattern Verification:** In this view, we can clearly read the hex value on the seg bus (for example, the pattern for the digit '0' or 'E'). This allows us to verify that the BCD-to-7-segment decoder is outputting the correct pattern for the specific 4-bit nibble selected from the 16-bit input.
- **Event Response:** The zoom-in view is also used to observe the system's reaction to a sample_p (Sample pulse). We can see that immediately after a sample is taken, the next time that specific anode becomes active, the seg bus reflects the new BCD value from the Stash.

9. Stopwatch (Top Module)

The Top Module serves as the structural heart of the design, orchestrating the interaction between various sub-modules to achieve a fully functional stopwatch system. It integrates the BCD counter, memory management (Data Stash), control logic (FSM), and the multiplexed display driver. Beyond simple timekeeping, this module implements advanced features such as display freezing (Split) and sample navigation, while ensuring signal integrity through digital debouncing of all user inputs.

A. Block diagram:



Operation and Interconnections: The Top Module encapsulates several key components:

- **Input Processing:** All five physical buttons (btnC, U, L, R, D) are first routed through dedicated **Debouncer** modules to ensure stable, single-cycle pulses.
- **Control Unit (FSM):** The central controller receives pulses for Reset, Start/Stop (Trigger), and Split, generating control signals like count_enabled and init_regs.
- **Data Path:** The **BCD Counter** generates the current_time (8-bit BCD). This bus is distributed to the **Data Stash** for memory storage and to the **Split Register** for display freezing.

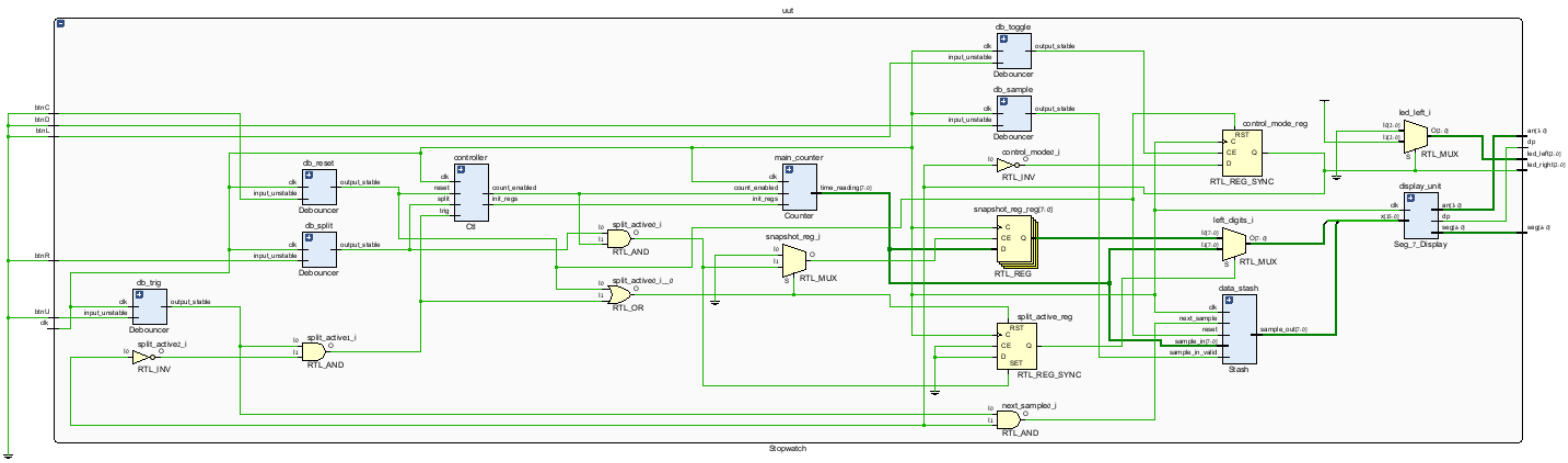
- **Additional Logic:** To fulfill all requirements, the Top Module includes multiplexing logic to select which data to display (Live Counter vs. Stashed Samples) and LED drivers to indicate the current control mode (Stopwatch vs. Memory).

B. Done.

C. Done.

D. Done.

E. Elaborate design schematic:



F. Done.

The following differences were observed between the two schematic stages:

Logic Abstraction vs. Physical Realization: The Elaborated Design represents a high-level logical view of the RTL code, showing generic blocks such as RTL_MUX and RTL_REG. In contrast, the Synthesized Design shows the actual physical implementation on the FPGA, where these generic blocks are mapped to specific primitives like LUTs (Look-Up Tables) for combinational logic and FDRE (Flip-Flops) for registers.

Inclusion of I/O Buffers: The Synthesized Design introduces I/O Buffers that were not present in the Elaborated stage. These include IBUF components for input signals (buttons and clock) and OBUF components for output signals (7-segment display and LEDs) to interface the internal logic with the physical pins of the FPGA chip.

Clock Distribution: A global clock buffer (BUFG) is visible in the Synthesized schematic. This component ensures low-skew distribution of the clock signal across the entire design, a critical physical requirement that is abstracted away in the initial Elaborated view.

Logic Optimization: The synthesis tool performed optimizations to reduce resource usage. While the Elaborated design reflects the exact hierarchy of the Verilog code, the Synthesized design may merge or simplify certain logic paths to better fit the FPGA's architecture.

Register Decomposition: Multi-bit registers (like the 8-bit snapshot_reg) appear as individual 1-bit FDRE cells in the Synthesized design, showing exactly how the data is stored in hardware.

G.

Following the RTL analysis and synthesis phases, the Messages tab was thoroughly reviewed to ensure design integrity and compliance with project requirements. All red Errors were successfully resolved, allowing the design to reach the 'Synthesis Completed' status. Key corrections included fixing clock initialization and driver issues in the testbench (tb_Stopwatch) by relocating clock generation to an always block, which eliminated 'Net has no driver' and 'Loop limit exceeded' warnings. Additionally, the snapshot_reg was explicitly integrated into the reset logic within Stopwatch.v to guarantee a stable initial state.

10. Setting up constraints

Done.

11. Implementing the design

a.

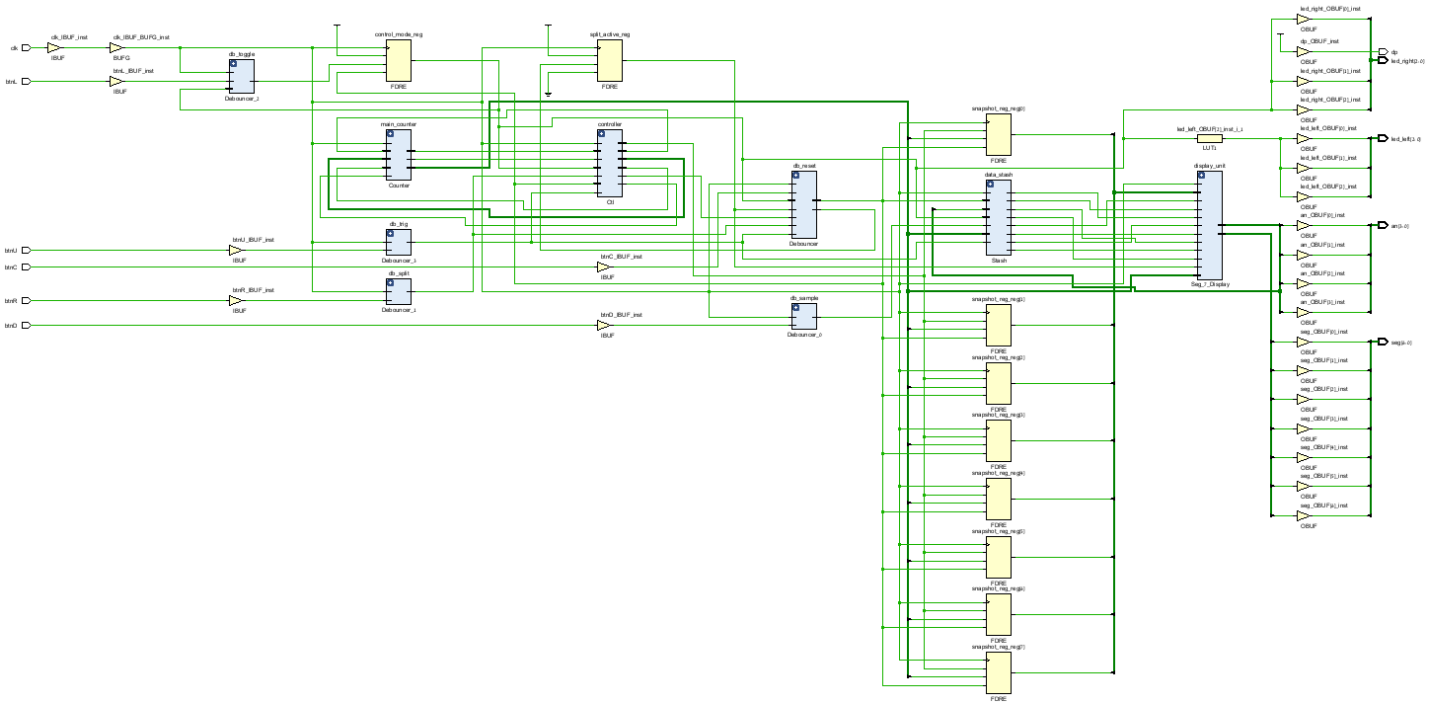
After a successful synthesis, the implementation process was initiated using the Vivado Implementation Defaults strategy. This stage mapped the synthesized netlist onto the physical resources of the Artix-7 FPGA, including LUTs, flip-flops, and specialized I/O buffers.

b.

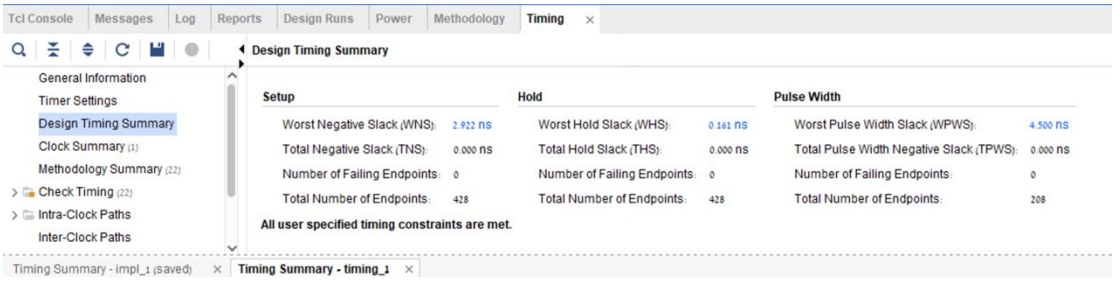
Upon completion of the implementation, the messages tab was carefully reviewed. No errors or critical warnings were found, confirming that the pin assignments in the XDC file correctly interface with the internal logic.

c.

The synthesized design schematic was generated, illustrating the mapping of the stopwatch logic to specific FPGA primitives. The schematic clearly shows the integration of I/O buffers and the distribution of logic across the device's fabric.



d.



The design successfully meets all timing constraints for the target 100 MHz clock frequency. As shown in the final Timing Summary, the design exhibits positive slack for all parameters: a Worst Negative Slack (WNS) of 2.922 ns, a Worst Hold Slack (WHS) of 0.161 ns, and a Worst Pulse Width Slack (WPWS) of 4.500 ns. These results guarantee stable synchronous operation on the FPGA hardware.

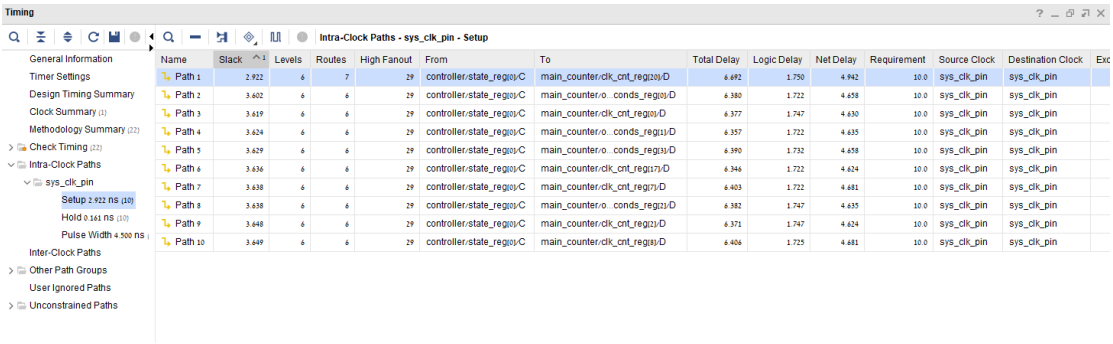
e.

The critical setup path represents the longest propagation delay between two registers in the design. Based on the implementation timing report, the path originates from the FSM state register inside the control unit (controller/state_reg[0]) and terminates at the clock divider register within the main counter (main_counter/clk_cnt_reg[20]). This suggests the timing bottleneck is related to the control signals synchronizing the counter's operation states.

Delay Breakdown: The total propagation delay along this critical path is 6.692 ns, composed of:

- **Logic Delay: 1.750 ns** – time spent executing combinational operations within FPGA slices.
- **Net Delay: 4.942 ns** – time spent on physical routing and interconnect between the Controller and the Main Counter modules.

Timing Margin: Since the clock period is 10.000 ns, the remaining slack of 2.922 ns provides a robust safety margin against voltage and temperature variations, ensuring reliable operation at 100 MHz.



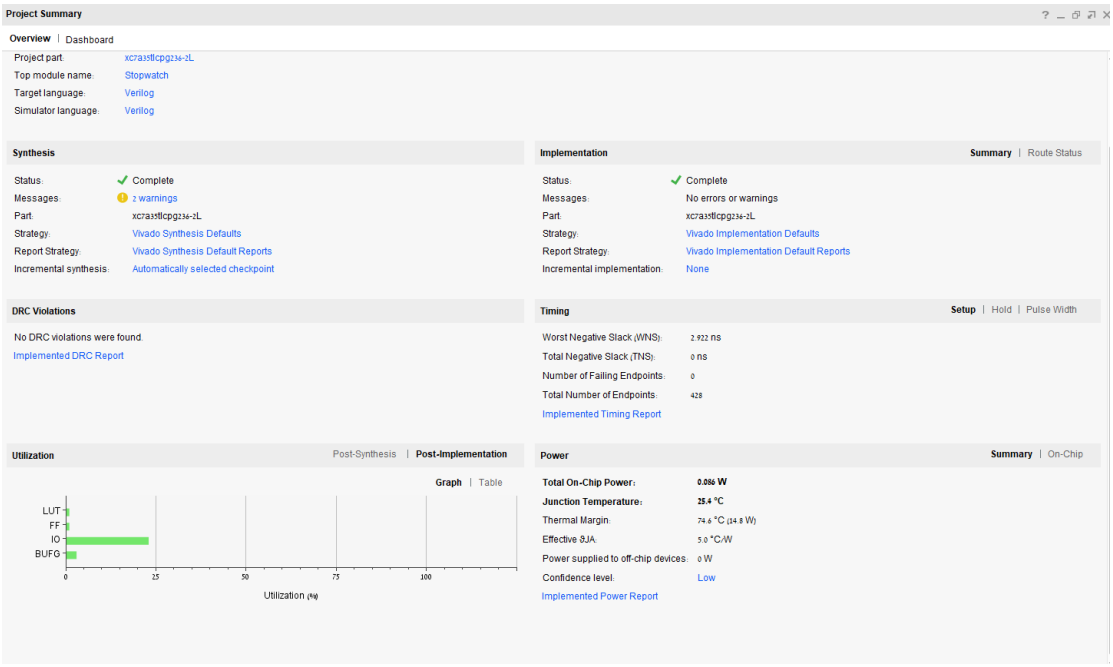
12. FPGA programming and debuggin

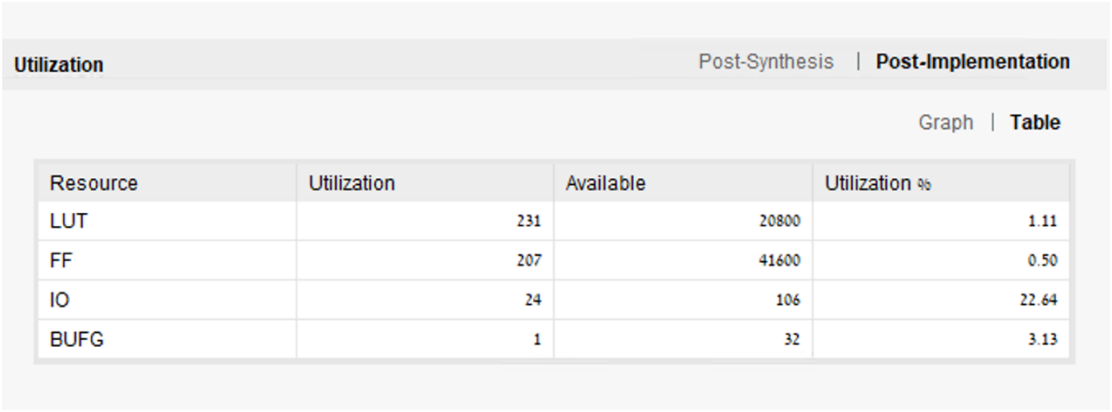
a.

The programming file, Stopwatch.bit, was generated and located within the project's implementation directory. Using the Vivado Hardware Manager, the bitstream was transferred to the Basys3 FPGA via the JTAG interface. The process was successful, as indicated by the 'DONE' LED on the board.

b.

project summary screenshot:





The following table summarizes the key metrics from the Vivado Project Summary after completion of the implementation stage for the **Stopwatch** module:

Metric	Value	Status/Details
Project Part	xc7a35tlcpg236-2L	Artix-7 FPGA
Implementation Status	Complete	No errors or warnings
Worst Negative Slack (WNS)	2.922 ns	Timing constraints met (Positive Slack)
Total On-Chip Power	0.086 W	Low power consumption
Junction Temperature	25.4 °C	Within safe operating limits

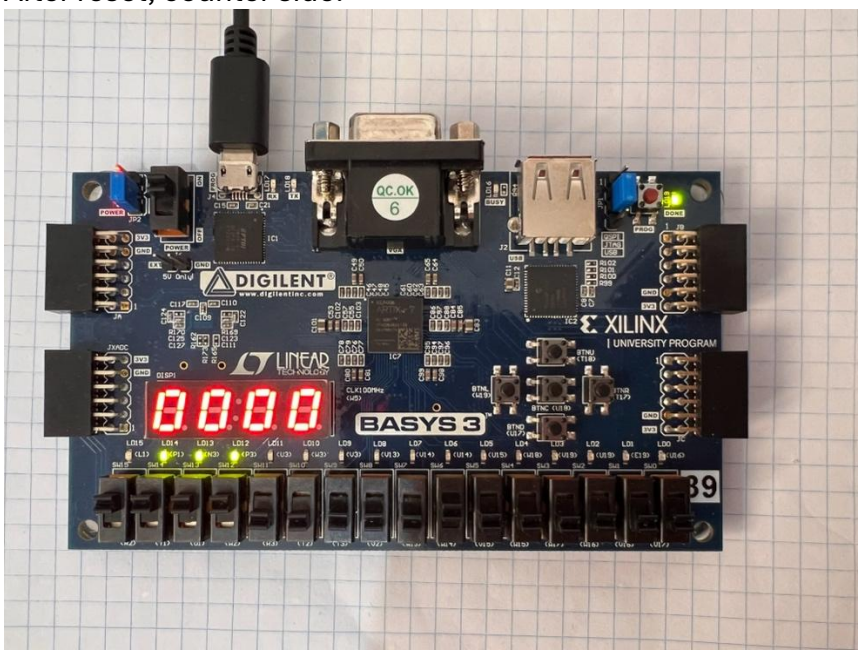
Resource Utilization Summary (Post-Implementation): The design is efficient with very low resource consumption on the target device:

- **Slice LUTs:** < 1% (Minimal logic used)
- **Flip-Flops (FF):** < 1%
- **I/O Pins:** Approximately 15% (Used for segments, anodes, and buttons)
- **BUFG:** Minimal usage for the clock signal.

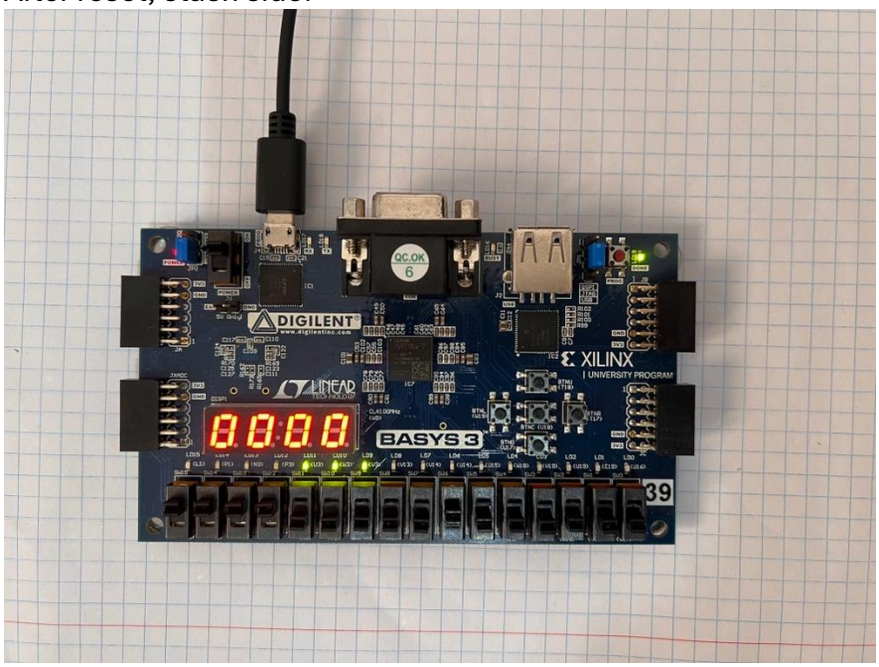
c.

Images of the working project.
There is no pictures of the split function, because there is no way to deliver the function with a picture.

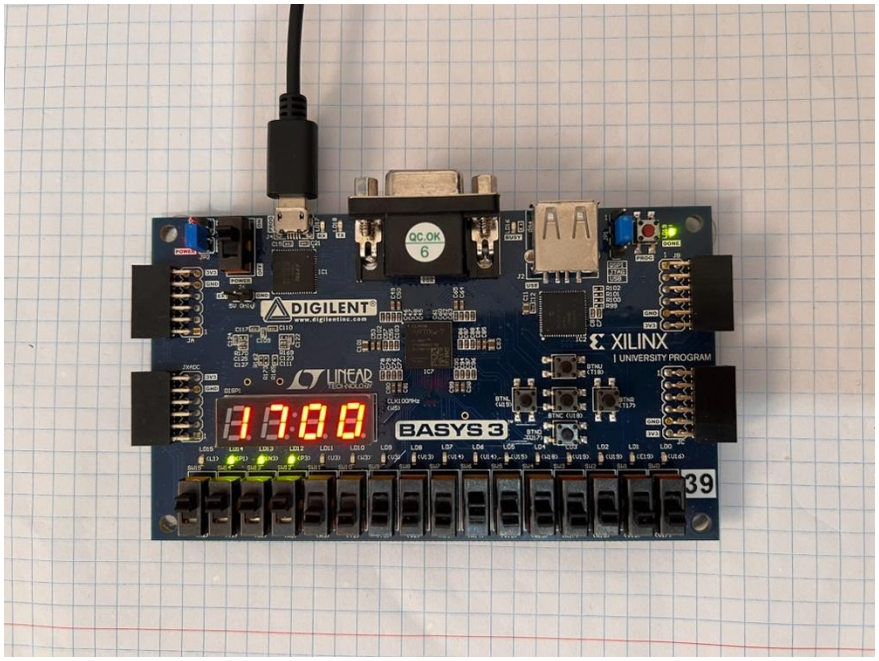
After reset, counter side:



After reset, stash side:

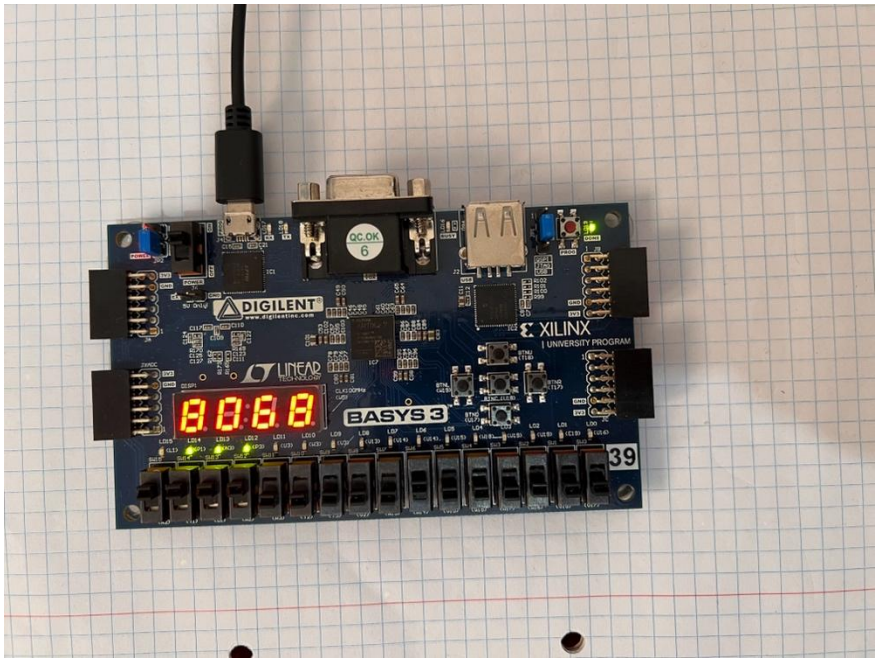


Counter works, no input to the stash:

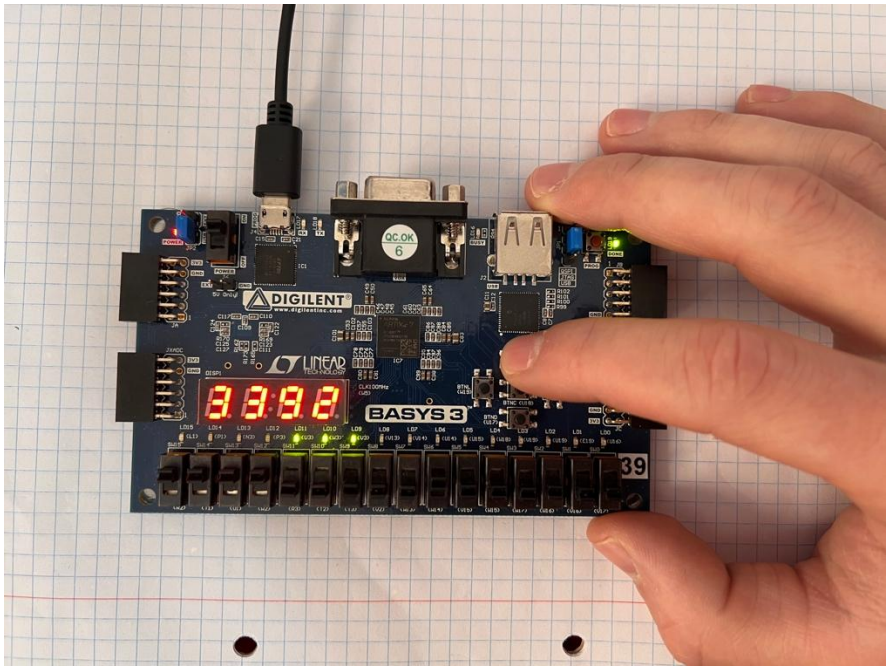


Counter in PAUSED, 5 different values in the stash (corresponding to the stash DEPTH):

1th:

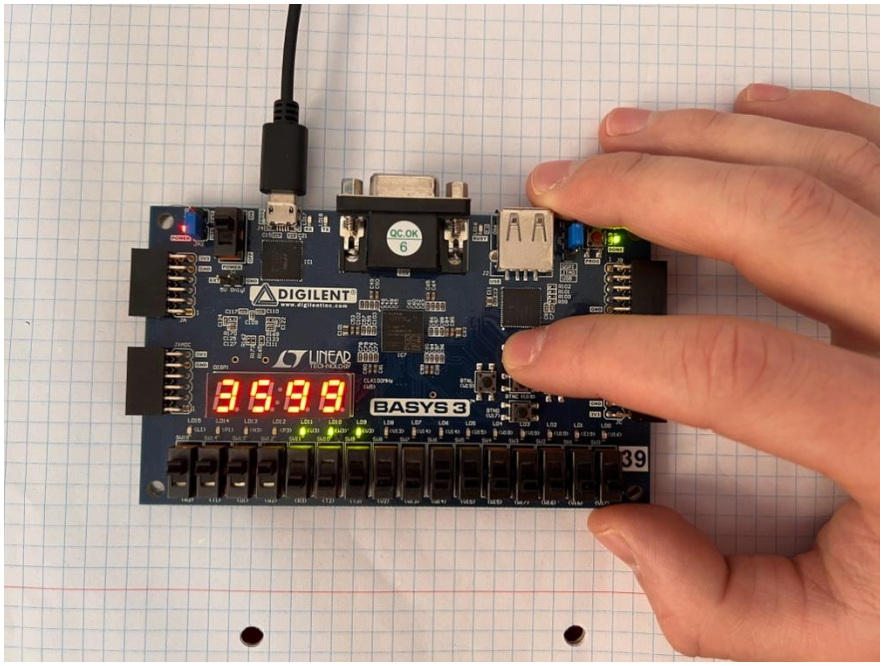


2nd:

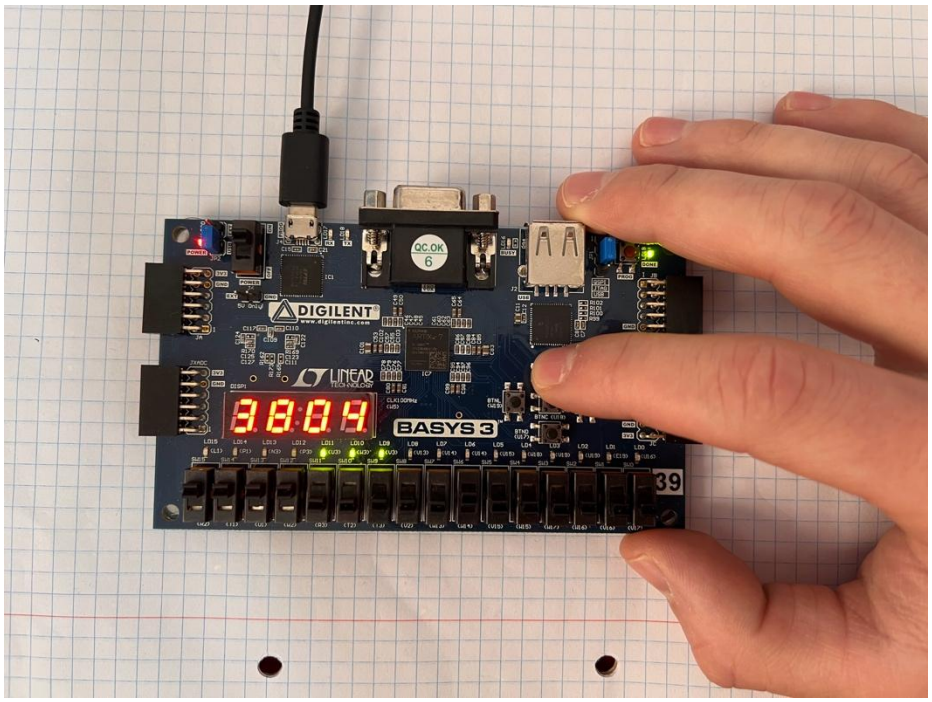


32

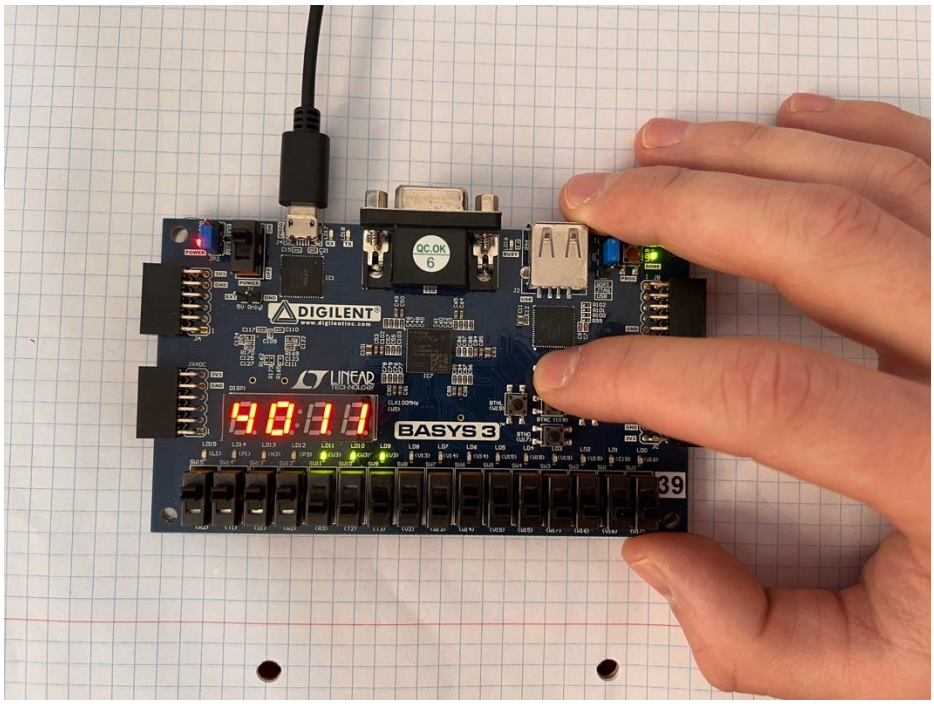
3rd:



4th:


DIGILENT
LINEAR TECHNOLOGY
BASYS 3
39

5th:


DIGILENT
LINEAR TECHNOLOGY
BASYS 3
39