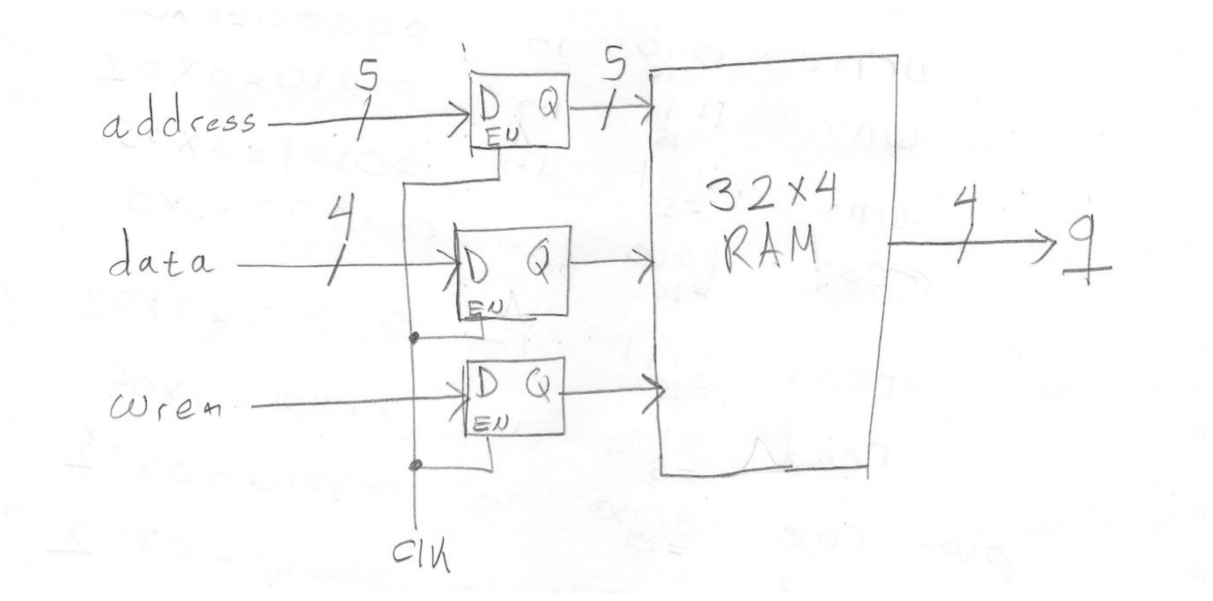
Student ID Number

EE/CSE 371: Laboratory #2, Memory Blocks

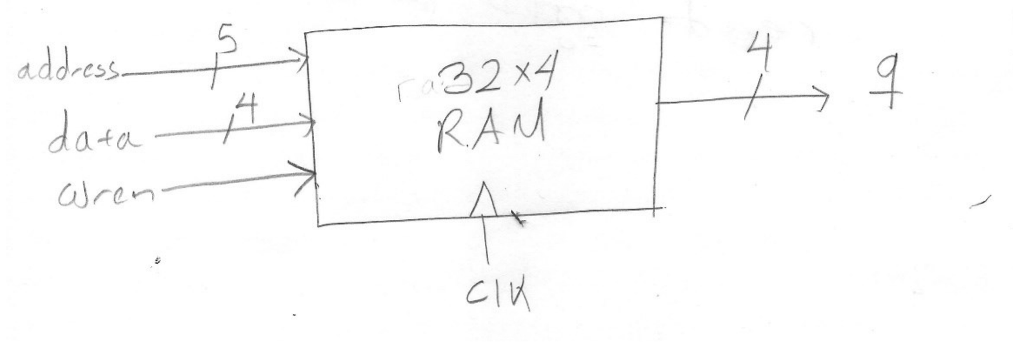
Design Procedure

Task 1

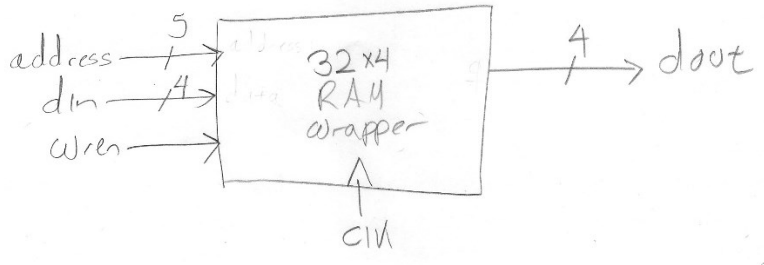
The laboratory guides us into using a 32x4 (32 words of 4 bits each) RAM from the IP catalog as shown below:



Using an internal implementation like the one above, the generated Verilog file is attached as *ram32x4.v* and exposes a module such as the following:



Therefore, creating a top-level module with the goal of testing this memory is as simple as creating a wrapper and then a testbench:



The associated module and its testbench has been attached as *ram32x4_wrapper.sv*. The testbench tests the memory by performing the following procedure:

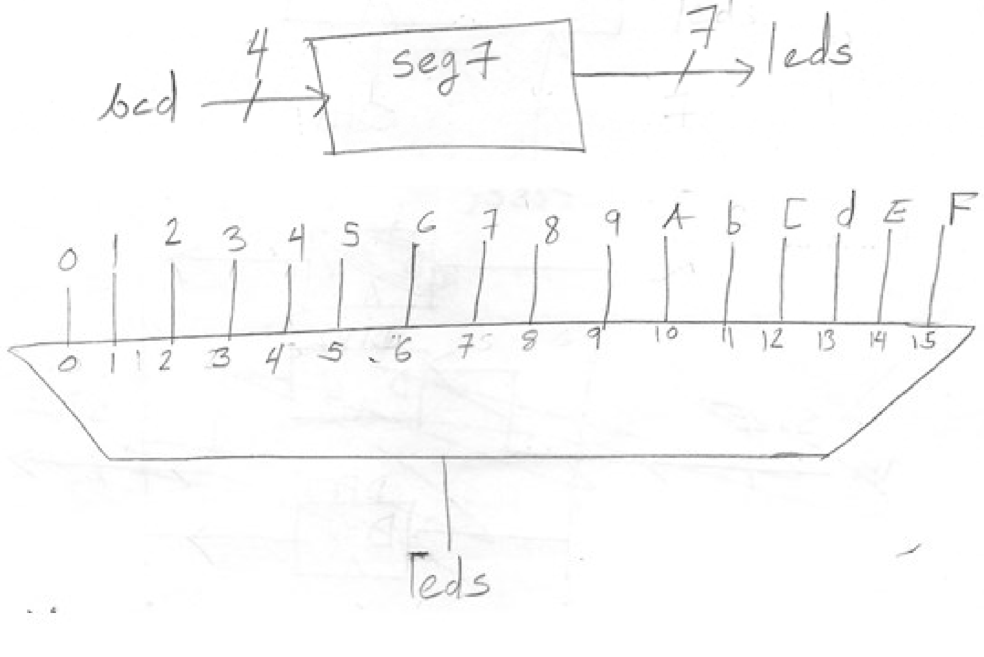
1- Writes consecutively into three arbitrary locations: 4’b1010 in 5’b00000, 4’b1111 in 5’b00010, and 4’b0101 in 5’b00101.
2- Then reads the locations above (5’b00000, 5’b00010, and 5’b00101).
3- Overwrites the old 4’b1111 in 5’b00010 by writing 4’b0000.
4- Checks that the write in step 3 took effect by reading 4’b0000.

Which covers all of the functions of the RAM (writing, reading, over-writing, and the fact that memory holds previous values unless overwritten).

Task 2

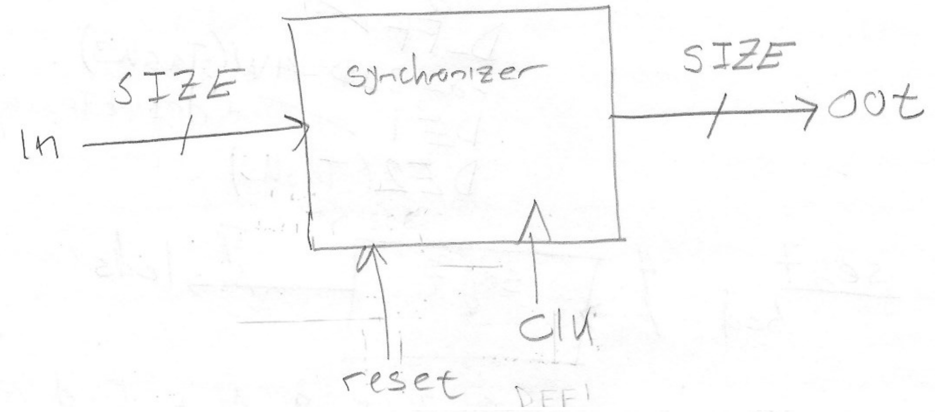
Seg7

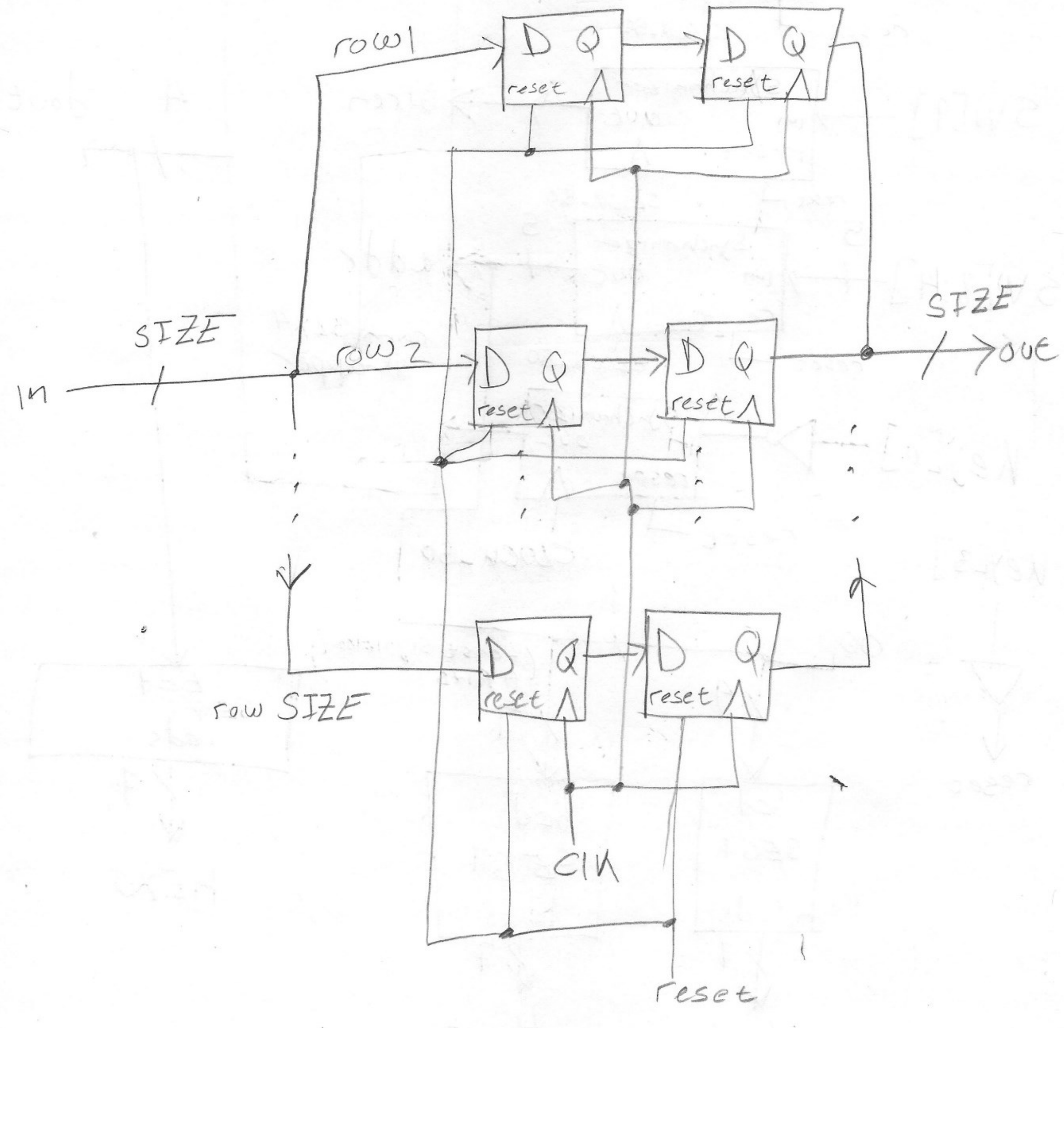
Since I needed to show hex numbers on the hexadecimal display, I had to extend my previous *seg7* implementation for one that displays the hexadecimal digits from A to F (attached as *seg7.sv)*. This is done by adding more *case* statements that provide the appropriate outputs. The block diagram of the module is exactly as the previous version, except that the multiplexer now has the digits from A-F instead of *Don’t Care* values. The hexadecimal representation is used in the diagram for readability instead of the format taken by the 7-segment display.



Synchronizer

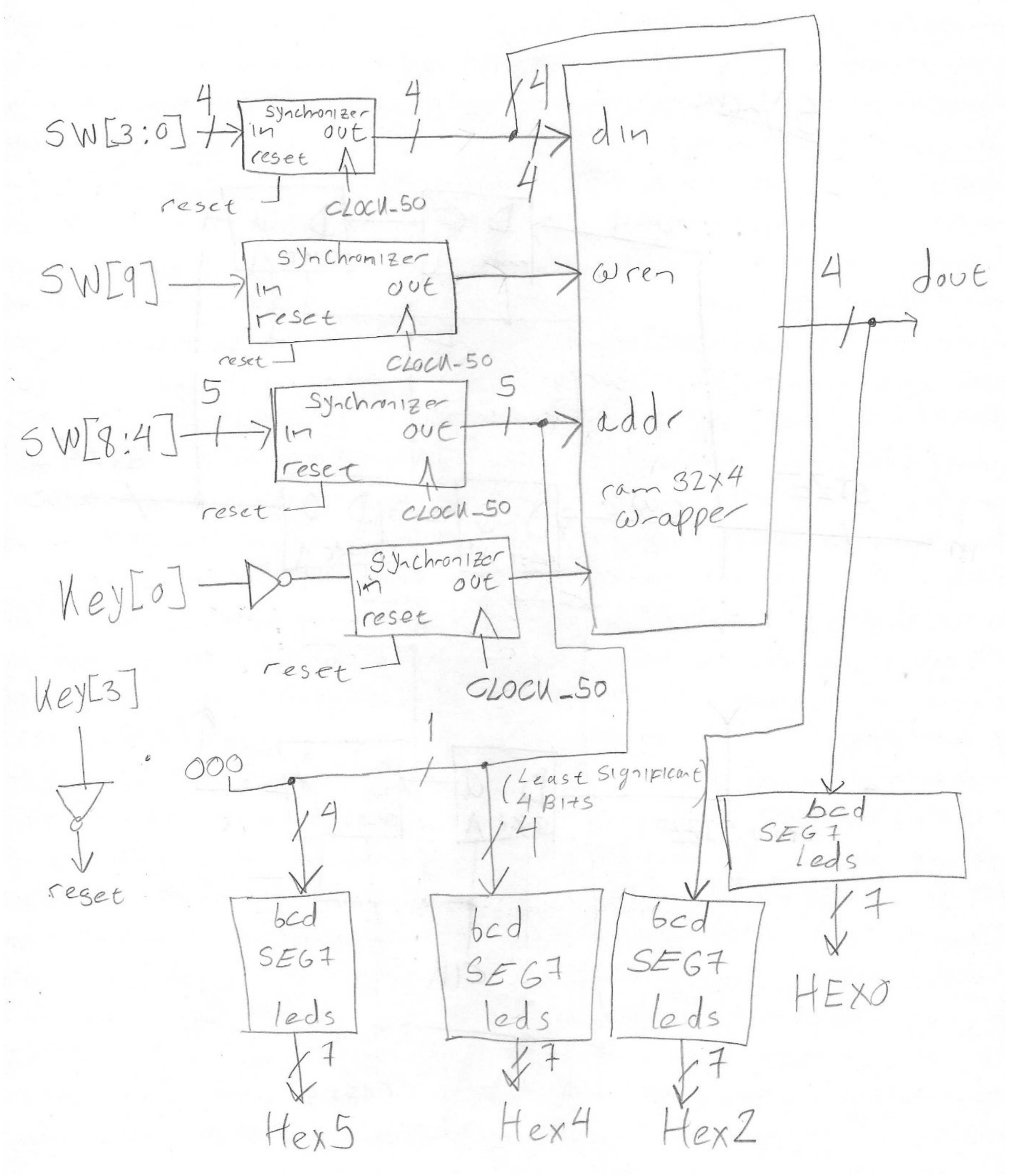
I also built a synchronizer module to make writing my code easier to write and read (previously I used to put two single DFFs for every bit of input from the external world). The module uses a generate statement to produce the necessary DFFs and is attached as *synchronizer.sv*. Its diagrams are shown below:





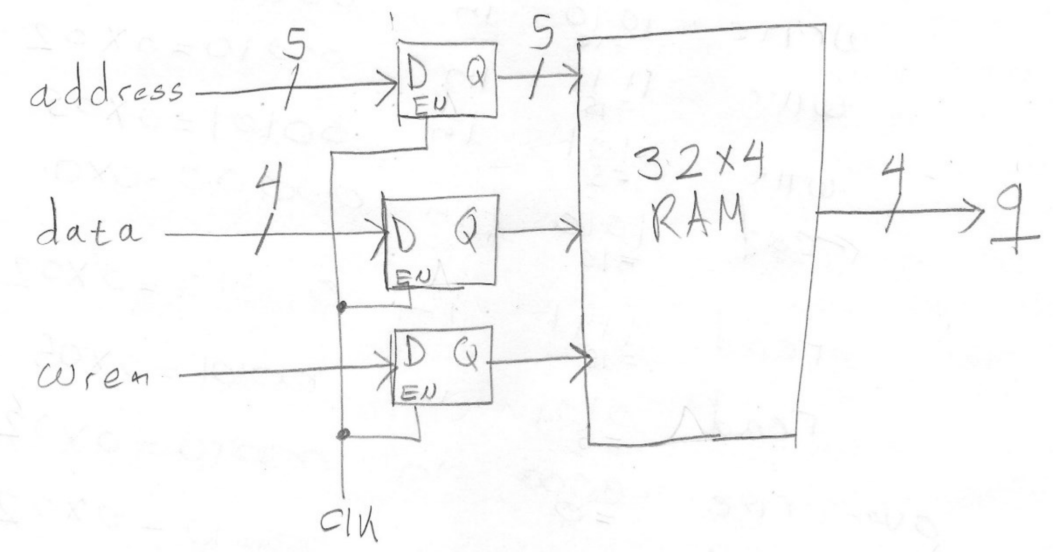
DE1_SoC

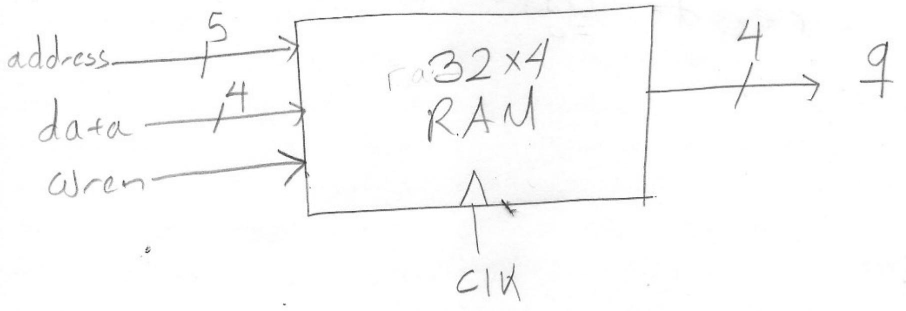
This is the top-level module (attached as *DE1_SoC_task2*) that connects all the pieces necessary to interface with the memory, including the KEY[0] which serves as the clock for the memory. The address (*addr)* input is easily converted into a format appropriate to the *seg7* module by splitting the four least significant bits (forms the least significant hexadecimal digit) and the most significant bit (which is the least significant bit of the most significant hex digit of the address)



Task 3

To create a SystemVerilog module that provides the same functionality as the RAM module from the IP catalog, it is necessary to start with the same block diagram of the internals:

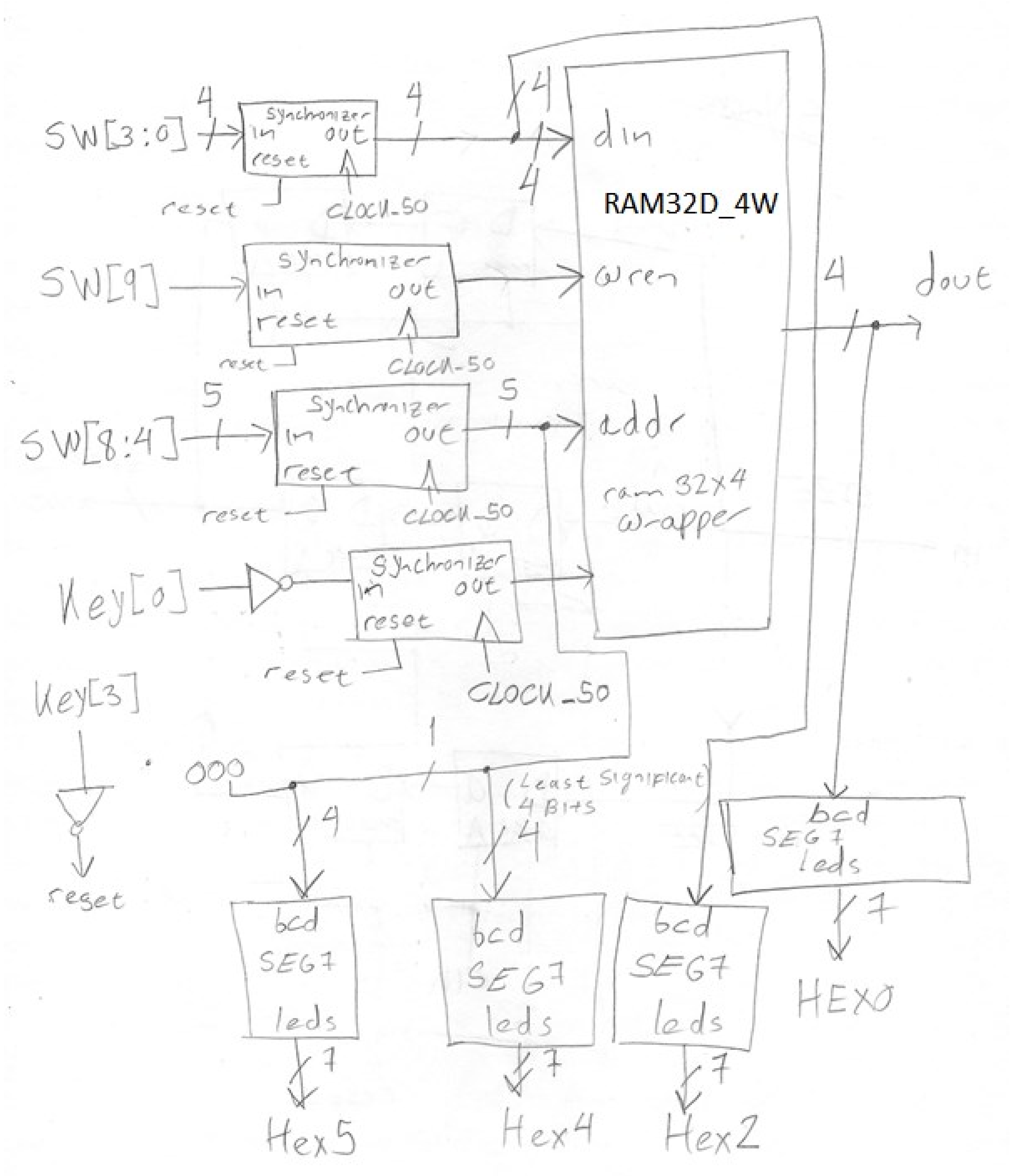




The 32x4 RAM block above can be produced with an array declaration:

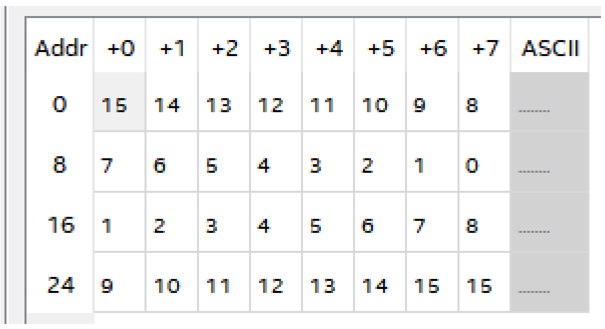
```
logic [3:0] RAM[31:0];
```

Which the laboratory document predicts to be implemented as memory blocks since these are registered (confirmed in the results section). Also, to match the behavior of the RAM from the IP Catalog, the memory will also read the data that is being written when wren is TRUE. The module and its testbench are attached as *ram32D_4W.sv.* The top-level module where this memory used is almost the same as *DE1_SoC_task2,* with the difference that this module uses *ram34D_4W* for memory instead of *ram32x4.v*. The module has been attached as *DE1_SoC_task3.sv*:



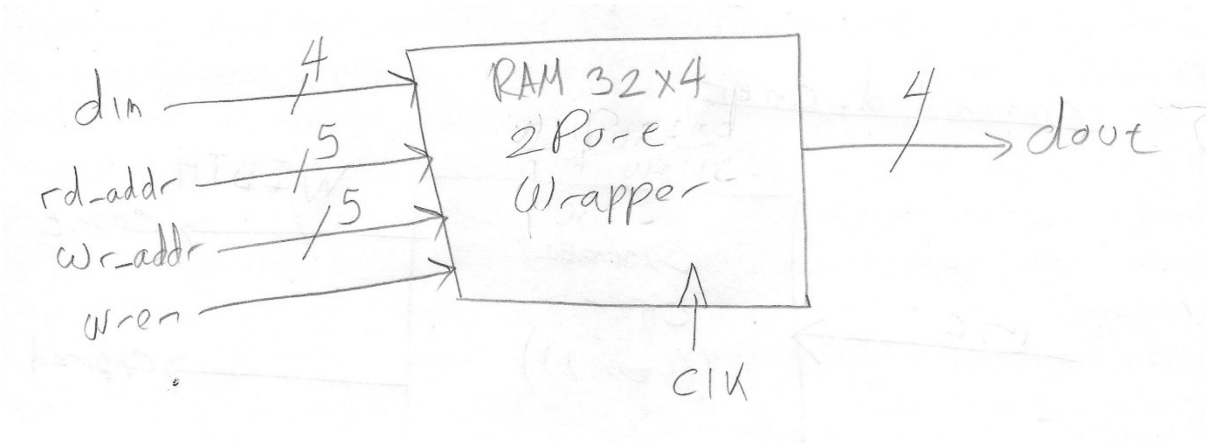
Task 4

First, the 2-Port RAM module was generated as per the laboratory instructions from the IP catalog (attached as *ram32x4port2.v)*. Then, a memory initialization file was created with the following contents (also attached as *ram32x4.mif*):



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	15	14	13	12	11	10	9	8	....
8	7	6	5	4	3	2	1	0	....
16	1	2	3	4	5	6	7	8	....
24	9	10	11	12	13	14	15	15	....

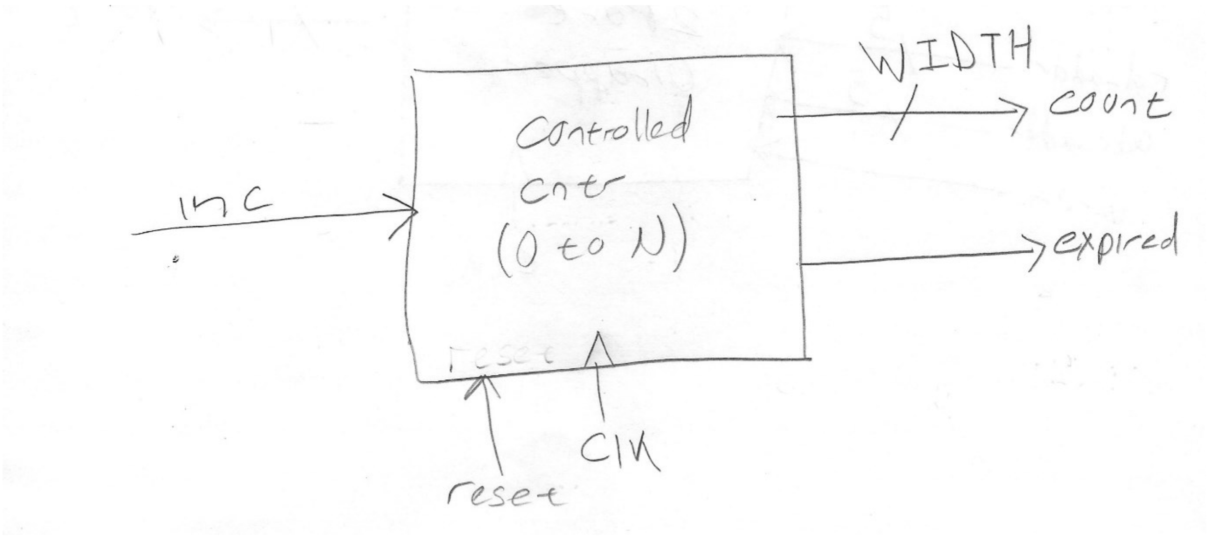
The wrapper module for *ram32x4port2.v* looks like the following and it is attached as *ram32x4port2_wrapper.sv*.



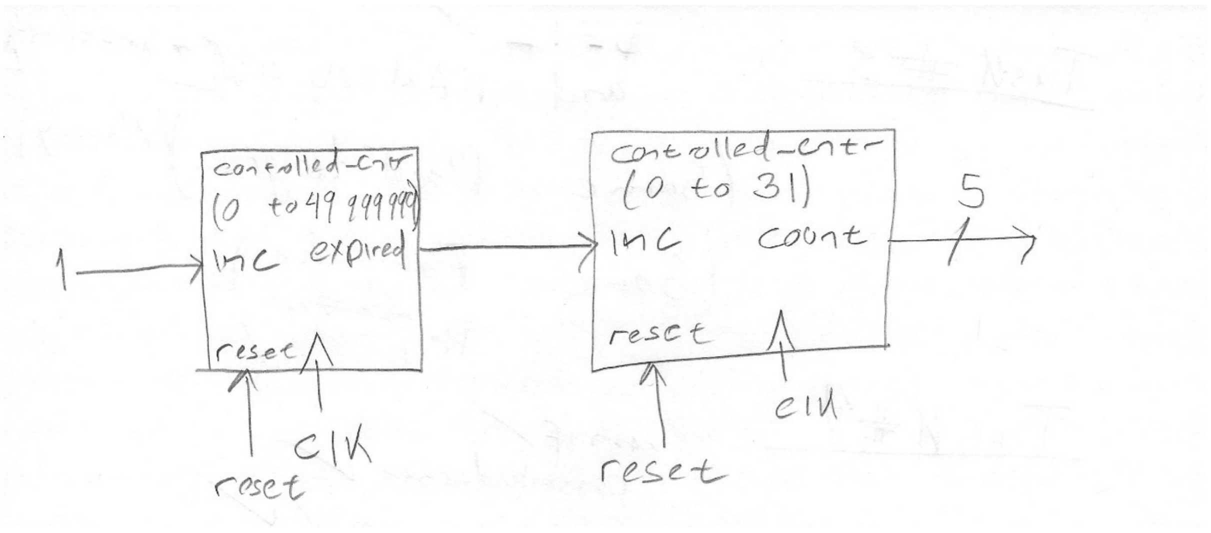
Controlled Counter

This module is attached as *controlled_cntr.sv* and works as follows:

- Given an increment signal *inc*, the counter increases (synchronized to the clock).
- It is an up-counter from 0 to a parameter N.
- It outputs the current count (user must make sure N fits into a binary unsigned number of width WIDTH), and whether the counter expired (this is held for only one clock cycle).



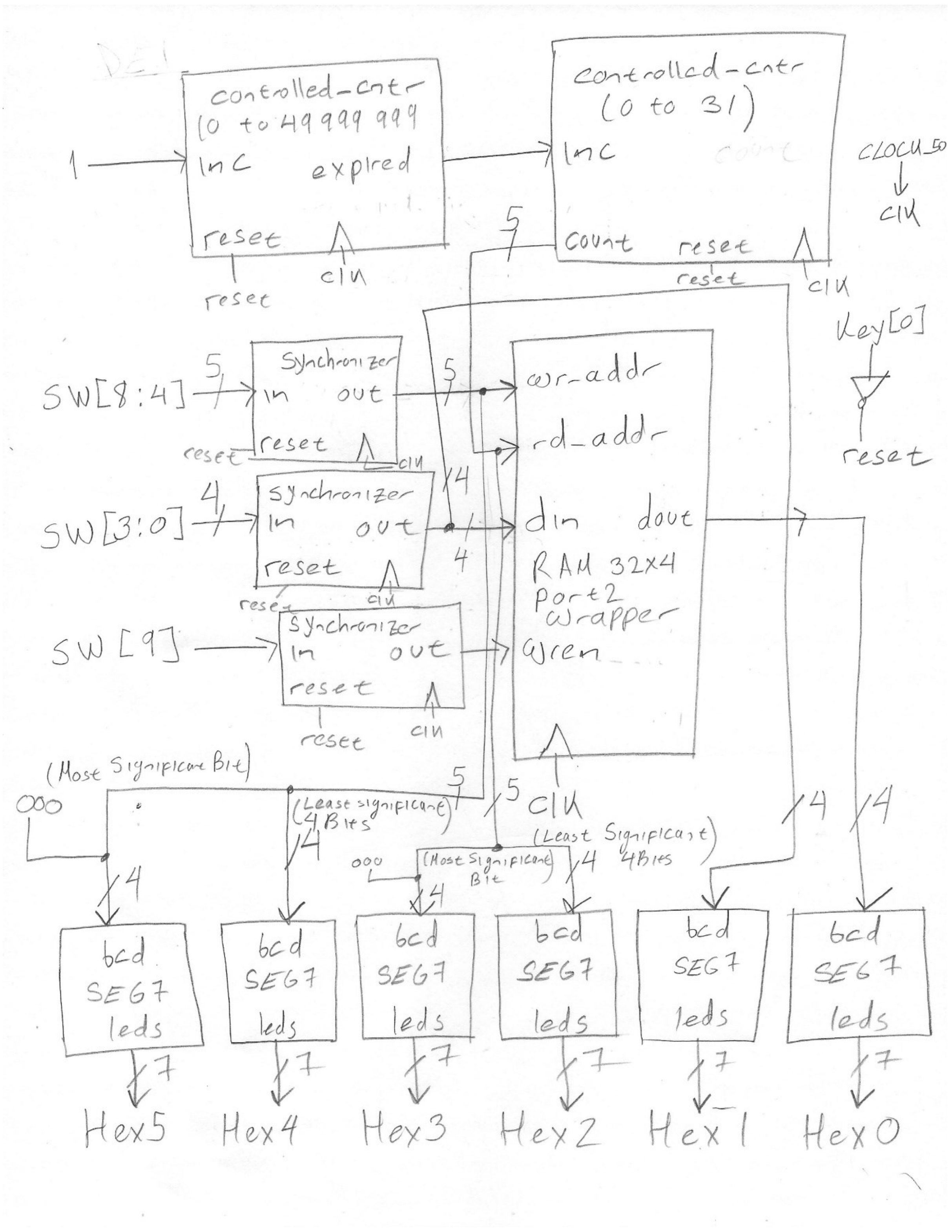
Then, using two controlled counters it is possible to produce a new reading address every 1 second, assuming that CLOCK_50 runs at 50MHz (the output count on the right is the address):



Each clock cycle the first controlled counter increases. After every 50 million clock cycles it outputs an expired signal which makes the next controlled counter increase. This way a new read address is computer every 50 million clock cycles. Since *CLOCK_50* runs at 50 million cycles per second, the address will be computed approximately once every second (within the accuracy of *CLOCK_50*).

DE1_SoC

Finally, this module (attached as *DE1_SoC_task4)* connects the external world inputs (using synchronizers) and displays the outputs on the hexadecimal displays. Using the connection shown above with the controlled counters, the read address *rd_addr* is changed every 1 second to the next value, while *CLOCK_50* remains being the clock as the laboratory document requires.

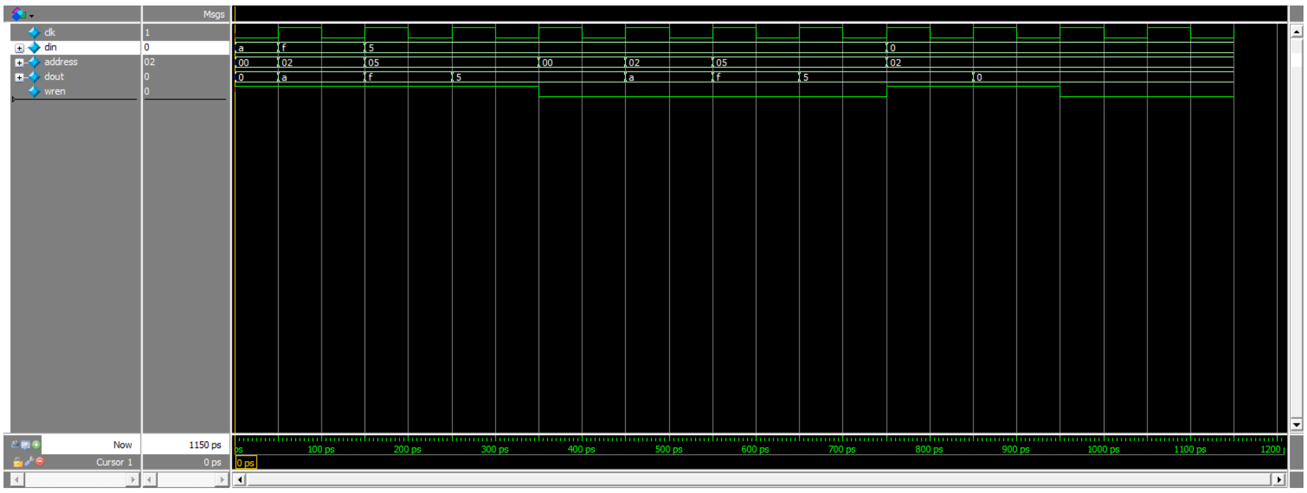


Results

Results for Task 1

The memory module does correctly retain data, reads data, and writes data. The testbench for the *ram32x4.v* module is written in *ram32x4_wrapper.sv*, and tests the following behavior which is confirmed in the simulation:

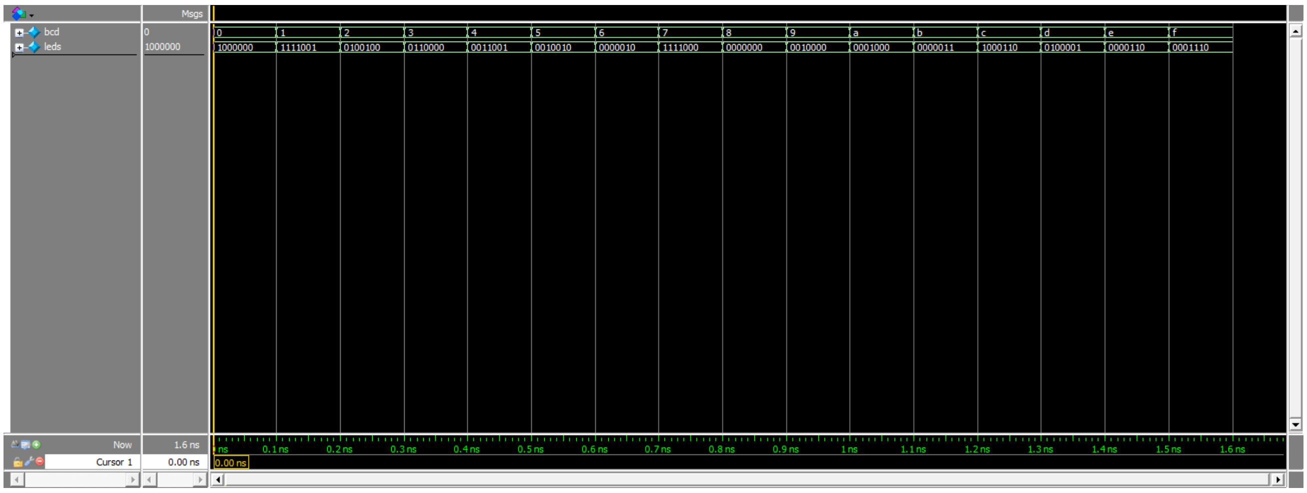
1- Writes 4’b1010 in 5’b00000 (0xA in 0x00).
2- Writes 4’b1111 in 5’b00010 (0xF in 0x02).
3- Writes 4’b0101 in 5’b00101 (0x5 in 0x05).
4- Reads 4’b1010 in 5’b00000 (0xA in 0x00).
5- Reads 4’b1111 in 5’b00010 (0xF in 0x02).
6- Reads 4’b0101 in 5’b00101 (0x5 in 0x05).
7- Over-writes the old 4’b1111 in 5’b00010 by 4’b0000 (0x0 in 0x02).



Results for Task 2

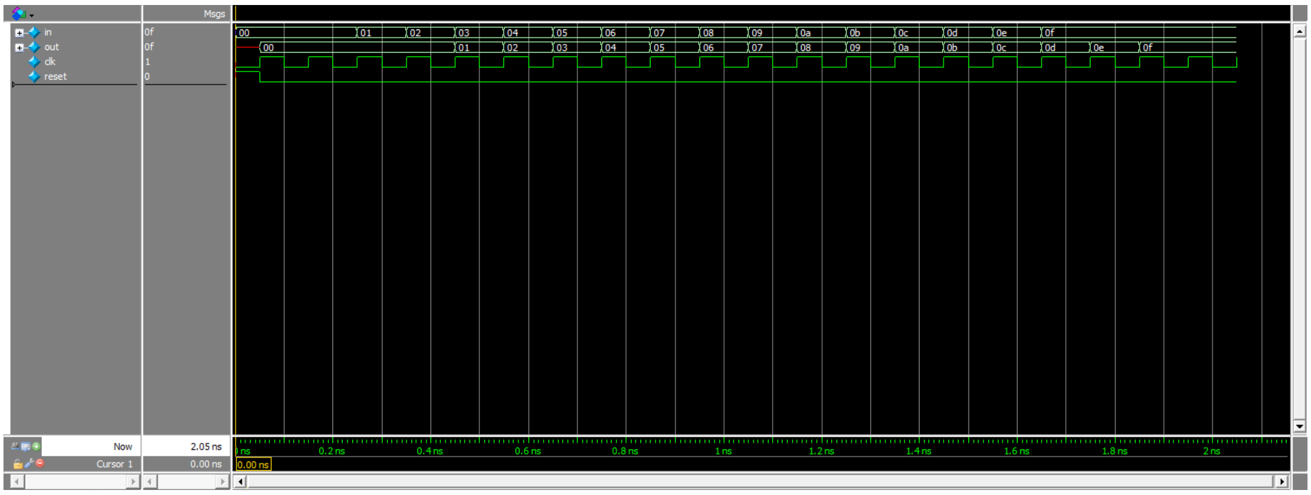
Seg7

The updated *seg7* module does correctly show a hexadecimal digit on a HEX display. It is easy to test since there are only 16 possible inputs and it is a combinational circuit. The testbench tests from 0x0 to 0xF. The equivalent of *leds* for every *bcd* matches the behaviour (*leds* represents the value in *bcd* in a format that the 7-segment display on the DE1_SoC board takes):



Synchronizer

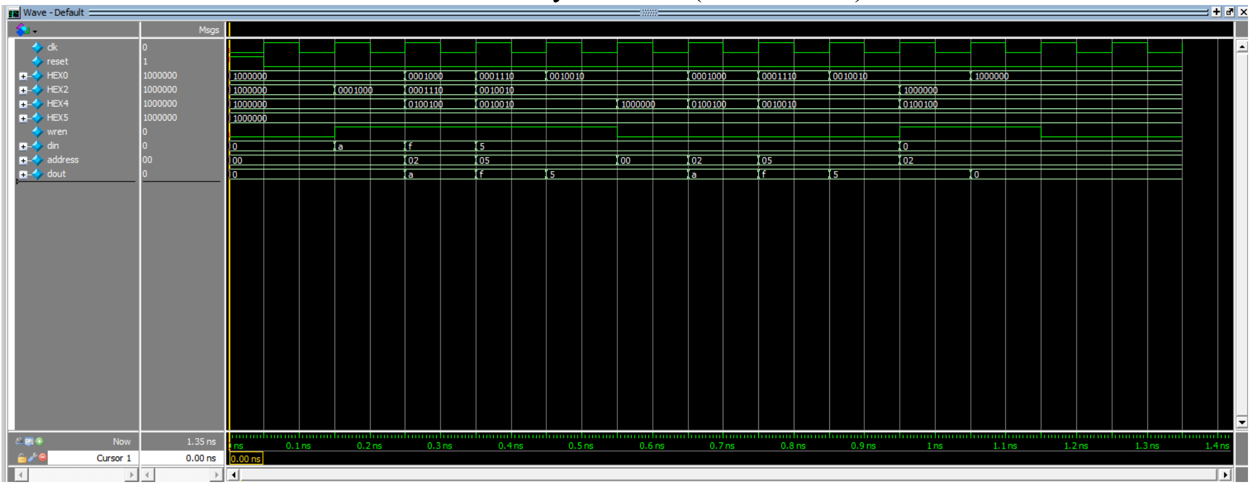
The *synchronizer* module is used to synchronize external inputs by passing every bit through two DFFs to avoid meta-stability. The testbench takes a few inputs from 0x0 to 0xF and outputs the value two clock cycles later as shown in the simulation:



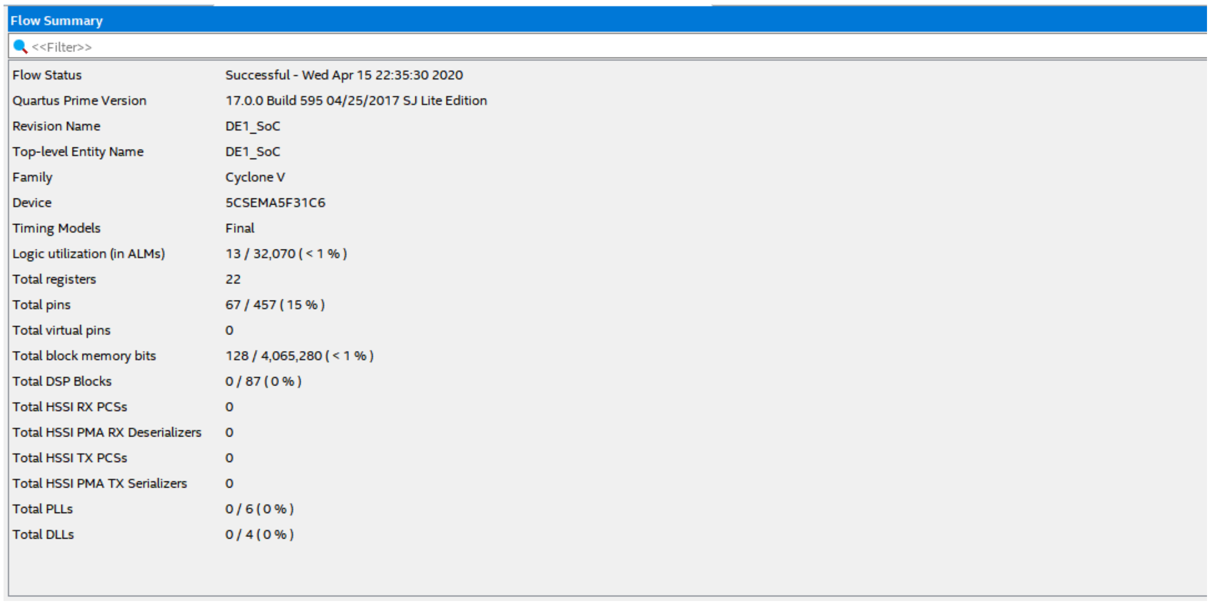
DE1_SoC

The *DE1_SoC* module for task 2 (*DE1_SoC_task2*) correctly manages the memory based on external inputs coming from the switches and the buttons on the board and displays output data, address, and input data on the 7-segment displays. The testbench performs the similar checks as task 1. This module also tests the outputs on the hexadecimal displays (*din* in HEX2, *dout* in HEX0, and *address* in *HEX5-HEX4).* Note that uncommenting `define TESTING disables the synchronizers which are unnecessary for simulation to make the simulation easier to read.

1- Writes 4’b1010 in 5’b00000 (0xA in 0x00).
2- Writes 4’b1111 in 5’b00010 (0xF in 0x02).
3- Writes 4’b0101 in 5’b00101 (0x5 in 0x05).
4- Reads 4’b1010 in 5’b00000 (0xA in 0x00).
5- Reads 4’b1111 in 5’b00010 (0xF in 0x02).
6- Reads 4’b0101 in 5’b00101 (0x5 in 0x05).
7- Over-writes the old 4’b1111 in 5’b00010 by 4’b0000 (0x0 in 0x02).



Furthermore, the Flow Summary provides information about the resources used: 13 ALMs, 22 Registers, and 128 Block Memory Bits (32 words at 4 bits per word is 32x4=128).



Flow Summary

<<Filter>>

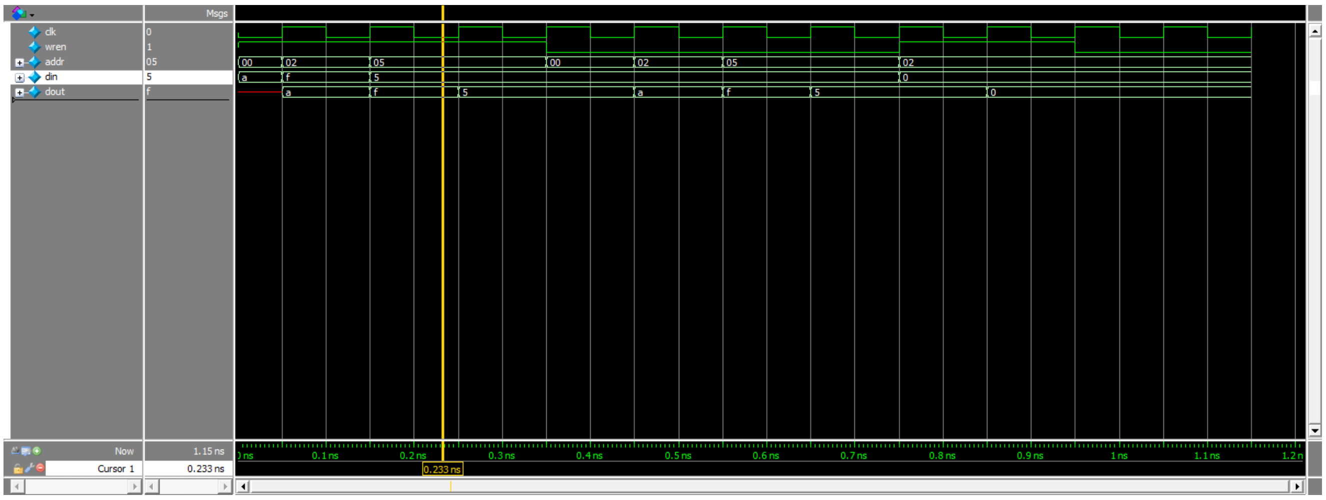
Flow Status	Successful - Wed Apr 15 22:35:30 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	13 / 32,070 (< 1 %)
Total registers	22
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	128 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Results for Task 3

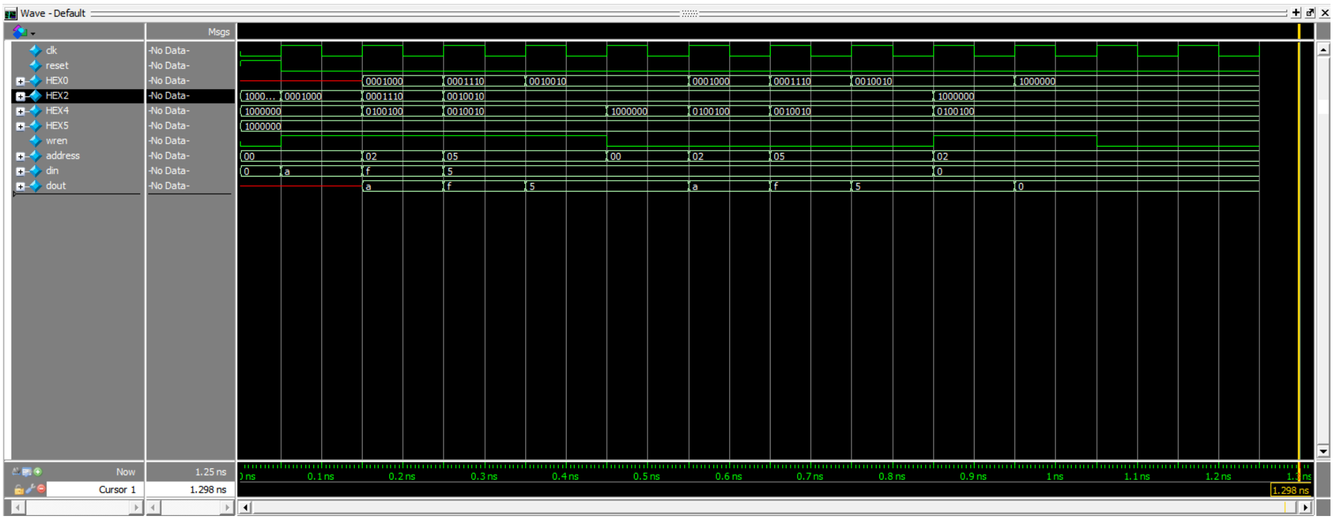
The following is the simulation for *ram32D_4W*, which performs the same memory tests and shows the same behavior as the memory from task 2 from the IP catalog:

1- Writes 4’b1010 in 5’b00000 (0xA in 0x00).
2- Writes 4’b1111 in 5’b00010 (0xF in 0x02).
3- Writes 4’b0101 in 5’b00101 (0x5 in 0x05).
4- Reads 4’b1010 in 5’b00000 (0xA in 0x00).
5- Reads 4’b1111 in 5’b00010 (0xF in 0x02).
6- Reads 4’b0101 in 5’b00101 (0x5 in 0x05).

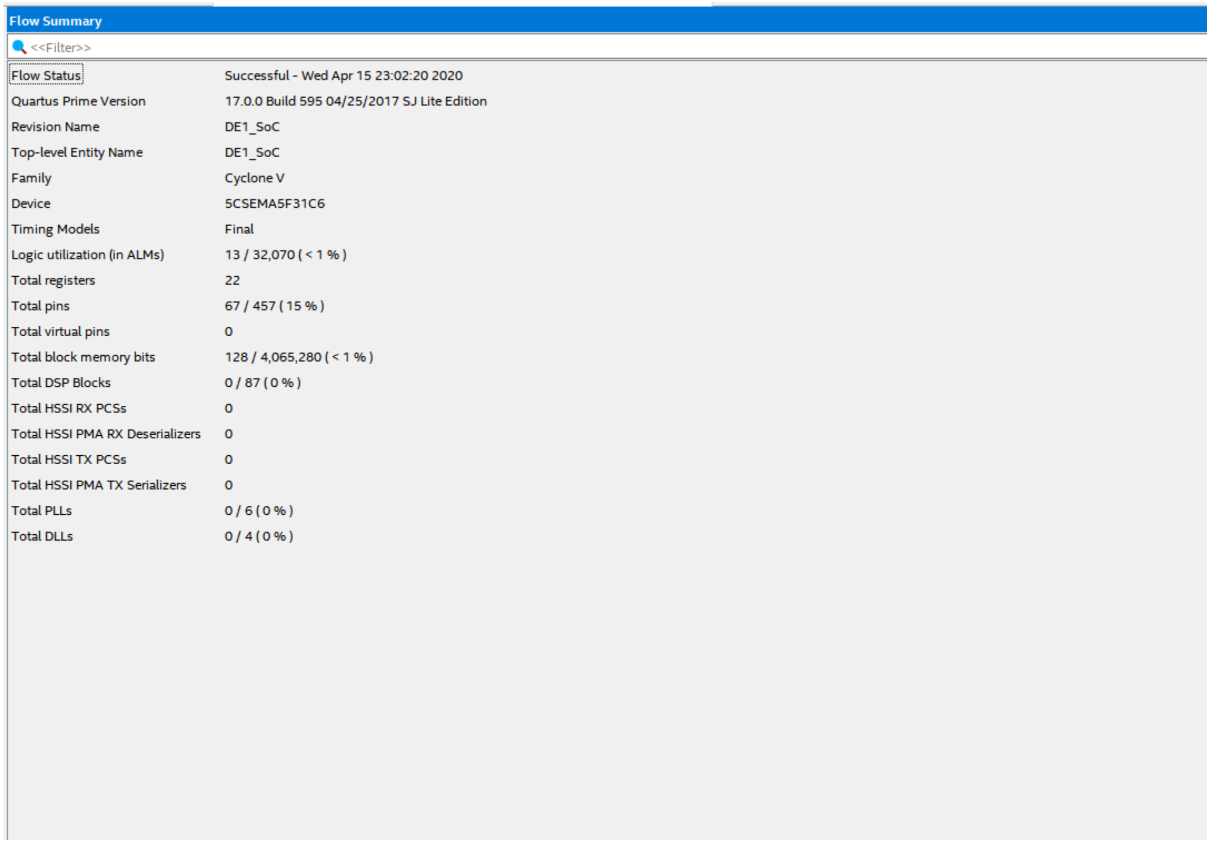
Over-writes the old 4’b1111 in 5’b00010 by 4’b0000 (0x0 in 0x02).



Next is the testbench for the *DE1_SoC* module (*DE1_SoC_task3)*, which shows again the same behavior as the *DE1_SoC* module for task 2 but using the memory from *ram32D_4W*. The initial red signal from 0ns to 0.15 ns occurs because ModelSim does not know the initial values of the RAM (which is fine since it is volatile memory, but it observed that MultiSim can predict the value when it uses the memory from the IP catalog since it seems to get initialized to zero). Similarly, uncommenting `define TESTING disables the synchronizers which are unnecessary for simulation to make the simulation easier to read.



The Flow Summary also shows the same resource usage: 13 ALMs, 22 Registers, and 128 Total block memory bits (32 words at 4 bits per word is 32x4=128bits). This also confirms the statement in the laboratory document that predicts the use of Block Memory Bits.


Flow Summary

<<Filter>>

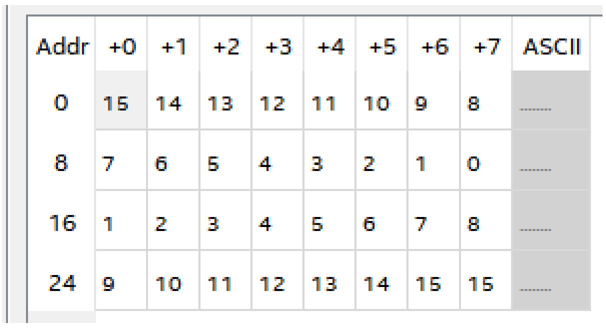
Flow Status	Successful - Wed Apr 15 23:02:20 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	13 / 32,070 (< 1 %)
Total registers	22
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	128 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Results for Task 4

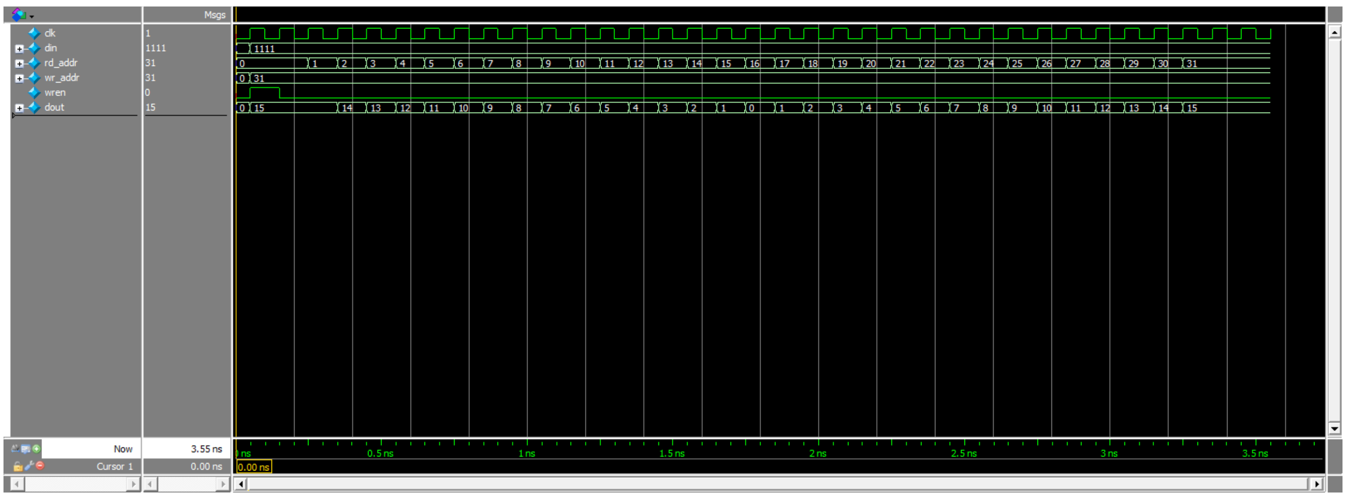
First, the testbench for the 32x4 2-port RAM performs the following:

1. Writes 4’b1111 in 5’b11111 to check that writing works.

2. The testbench goes through every location in memory reading words which can be verified in the screenshot below. The memory initialization file was shown earlier in the report and is repeated here for convenience:

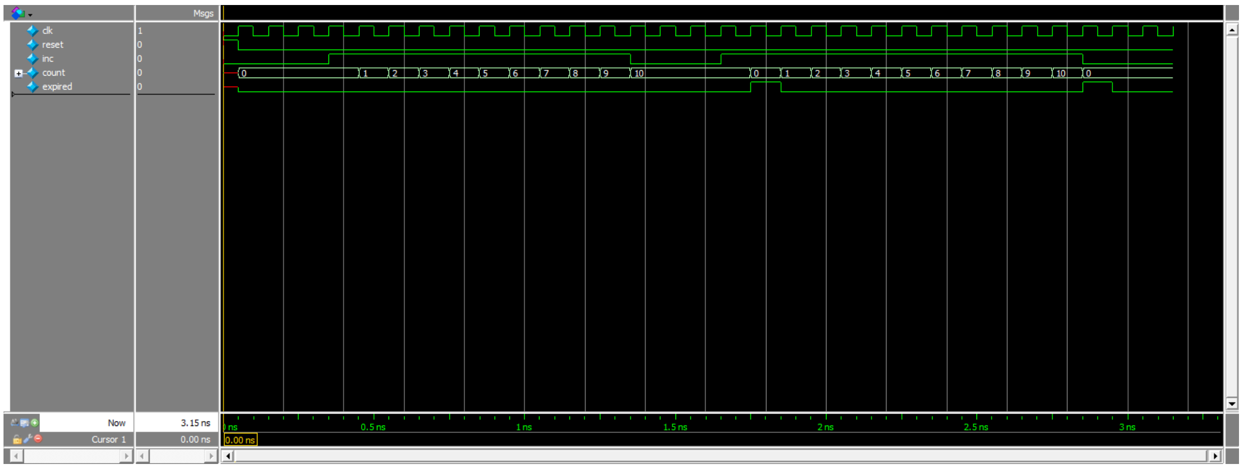


Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	15	14	13	12	11	10	9	8	....
8	7	6	5	4	3	2	1	0	....
16	1	2	3	4	5	6	7	8	....
24	9	10	11	12	13	14	15	15	....



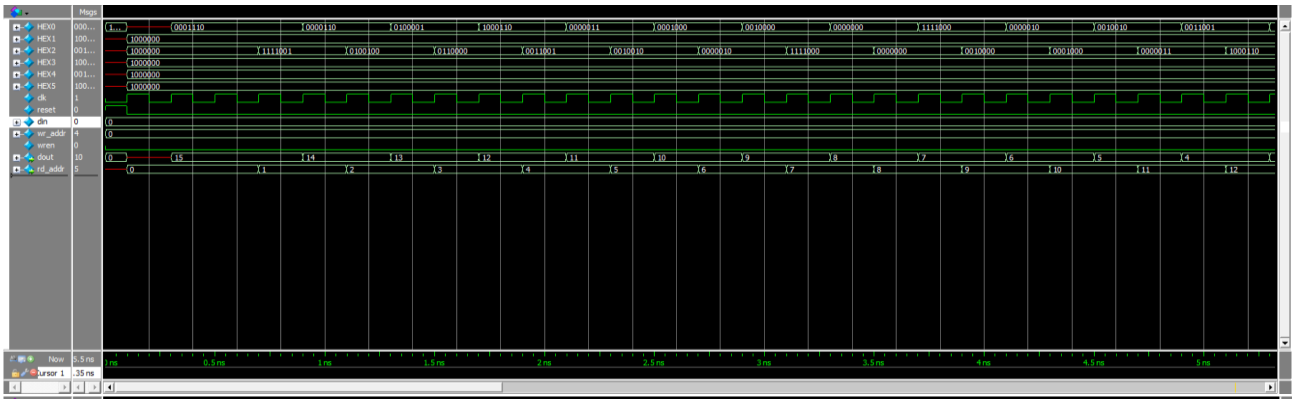
Next is the controlled counter (*controlled_cntr.sv)*:

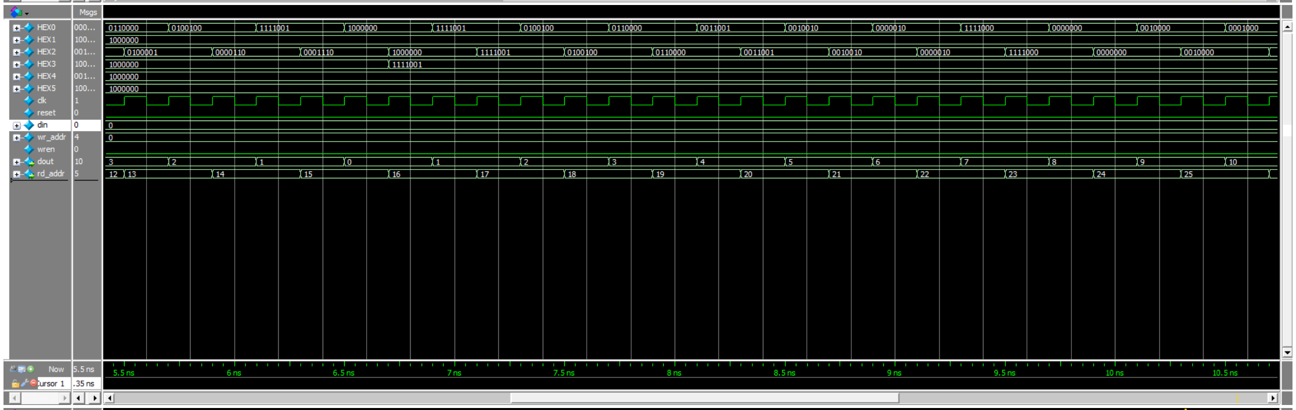
1. Tests that the counter doesn't increase when *inc* is FALSE.
2. *Inc* goes TRUE and the counter starts increasing.
3. *Inc* goes FALSE and the counter stops at 10 (maximum count for the parameters of this testbench).
4. The value is held.
5. Then *inc* goes TRUE and the counter expires (*expired* held TRUE for one clock cycle) and the counter wraps around to 0.
6. Then it counts again until it wraps again to 0.

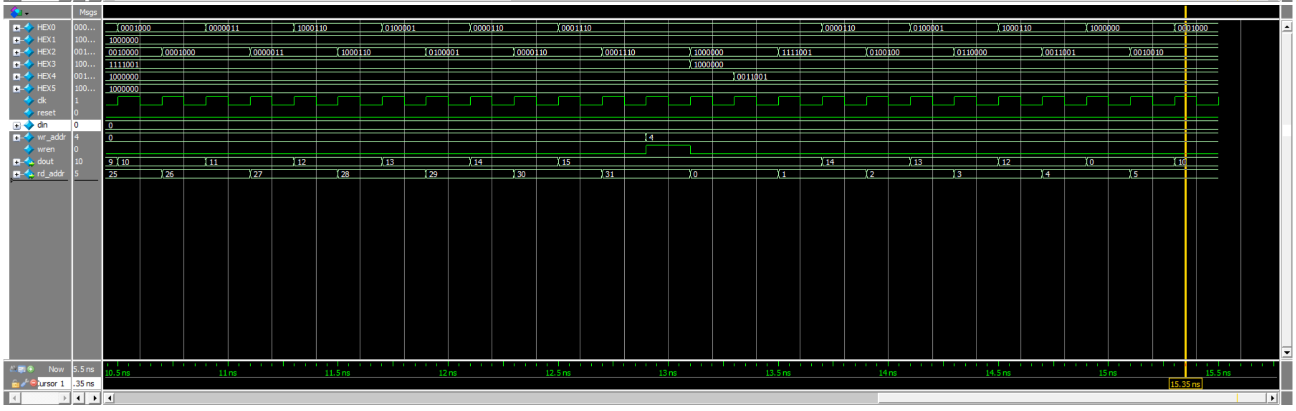


Finally, the *DE1_SoC* file for this module (*DE1_SoC_task4.sv)* tests the following (Note that uncommenting `define TESTING makes the count value very small since we don't want to wait 50 million cycles during simulation):

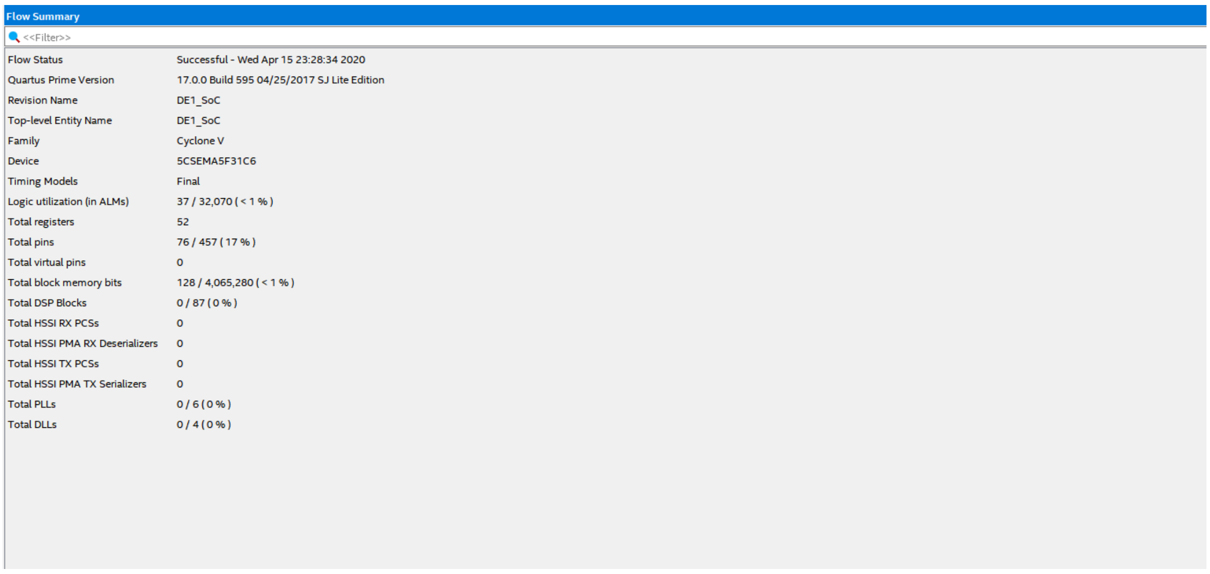
1. Each position in the memory initialization file is read (0 to 12.75ns)
2. Writes 4'b0000 in 5'b0100 (writes 0 in 0x04) and checks the change took effect.







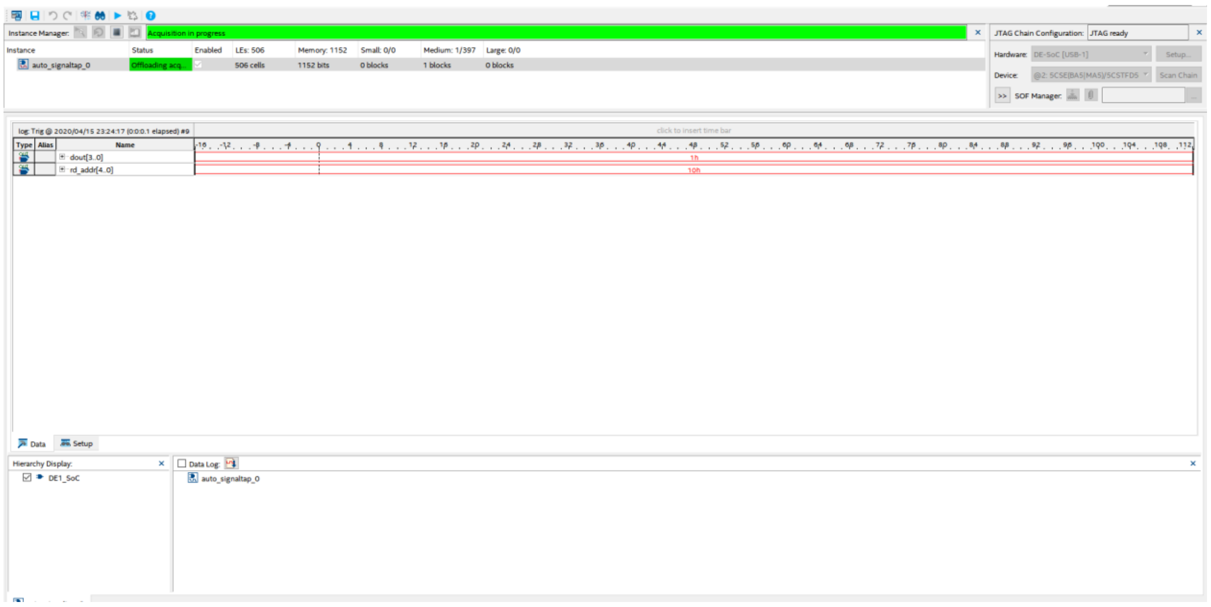
This module indeed goes through every address about once every second and shows the read address and the contents at that address and allows the user to over-write the memory (the write address and write data are shown too correctly). The Flow Summary shows 37 ALMs, 52 Registers, and again 128 Total Block Memory Bits:



Flow Summary

Flow Status	Successful - Wed Apr 15 23:28:34 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	37 / 32,070 (< 1 %)
Total registers	52
Total pins	76 / 457 (17 %)
Total virtual pins	0
Total block memory bits	128 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

SignalTap can also be used to monitor the signals of data output and the reading address:



Instance Manager: Acquisition in progress

JTAG Chain Configuration: JTAG ready

Instance	Status	Enabled	LEs: 506	Memory: 1152	Small: 0/0	Medium: 1/397	Large: 0/0
auto_signaltap_0	Offloading acq...		506 cells	1152 bits	0 blocks	1 blocks	0 blocks

Hardware: DE-SoC [USB-1]

Device: @2: 5CSE(BA5|MA5)/5CSTFD5

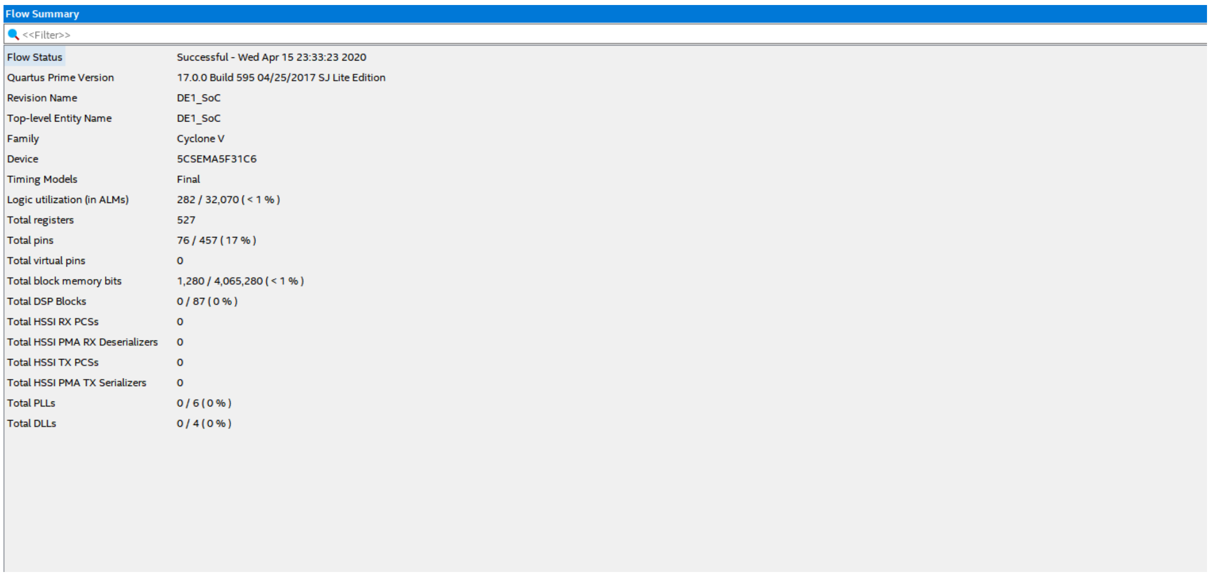
log: Trig @ 2020/04/15 23:24:17 (0:0:0.1 elapsed) #9

Type	Alias	Name	
		dout[3..0]	1h
		rd_addr[4..0]	10h

Experience Report

One challenge was to use the 50MHz module while producing a new reading address (task 4) only once every second. This was satisfactorily solved by letting the clock be at 50MHz as required but to provide a slower signal (the counter with the count output) that only increases based on another counter that expires every 50 million cycles (the counter with the *inc* output).

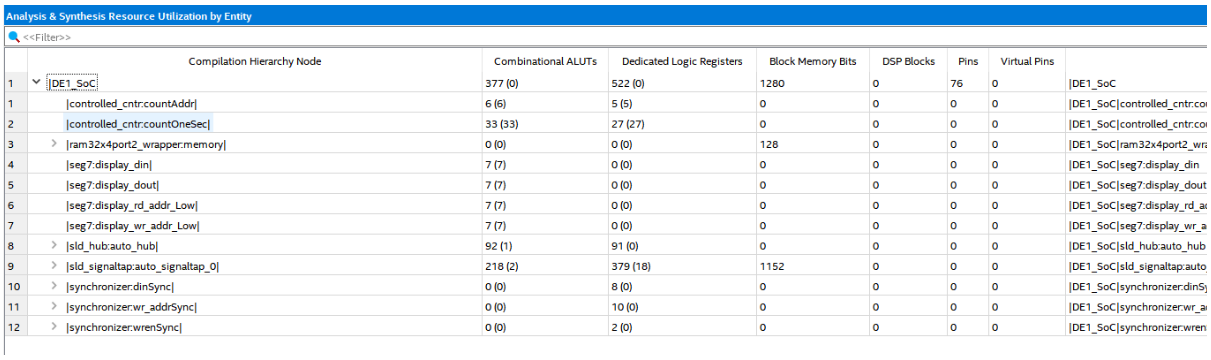
I also was doubtful when I took a screenshot of the Flow Summary for task 4. At first I had this:


Flow Summary

<<Filter>>

Flow Status	Successful - Wed Apr 15 23:33:23 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	282 / 32,070 (< 1 %)
Total registers	527
Total pins	76 / 457 (17 %)
Total virtual pins	0
Total block memory bits	1,280 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Upon further inspection, I figured that SignalTap takes a good amount of resources (92+218=310 ALUTs). Thus it is important to remember to look at the usage by entity to calculate how much is actually being used by your design:


Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	
1	|DE1_SoC	377 (0)	522 (0)	1280	0	76	0	|DE1_SoC
1	|controlled_cntr:countAddr|	6 (6)	5 (5)	0	0	0	0	|DE1_SoC|controlled_cntr:co
2	|controlled_cntr:countOneSec|	33 (33)	27 (27)	0	0	0	0	|DE1_SoC|controlled_cntr:co
3	|ram32x4port2_wrapper:memory|	0 (0)	0 (0)	128	0	0	0	|DE1_SoC|ram32x4port2_wr
4	|seg7:display_din|	7 (7)	0 (0)	0	0	0	0	|DE1_SoC|seg7:display_din
5	|seg7:display_dout|	7 (7)	0 (0)	0	0	0	0	|DE1_SoC|seg7:display_dout
6	|seg7:display_rd_addr_Low|	7 (7)	0 (0)	0	0	0	0	|DE1_SoC|seg7:display_rd_a
7	|seg7:display_wr_addr_Low|	7 (7)	0 (0)	0	0	0	0	|DE1_SoC|seg7:display_wr_a
8	|sld_hub:auto_hub|	92 (1)	91 (0)	0	0	0	0	|DE1_SoC|sld_hub:auto_hub
9	|sld_signaltap:auto_signaltap_0|	218 (2)	379 (18)	1152	0	0	0	|DE1_SoC|sld_signaltap:auto
10	|synchronizer:dinSync|	0 (0)	8 (0)	0	0	0	0	|DE1_SoC|synchronizer:dinS
11	|synchronizer:wr_addrSync|	0 (0)	10 (0)	0	0	0	0	|DE1_SoC|synchronizer:wr_a
12	|synchronizer:wrenSync|	0 (0)	2 (0)	0	0	0	0	|DE1_SoC|synchronizer:wren

A few tips for future laboratories:

- Many times, in the past I've re-developed counters and synchronizers since I make them too specific. This time I made a counter that I am sure I can use later and the same goes for the synchronizer.
- SignalTap Logic Analyzer inserts logic so that it can probe your signals. Thus, it is necessary to remember to subtract the usage of SignalTap to measure the logic used by your design.

Briefly explain why SignalTap was unnecessary for Task #4: The signals in task #4 are already displayed in the hexadecimal display, so there was no need to further display the output data and the reading address since they don't contain any information that we don't have. The idea of the logic analyzer is to look at the signals inside the FPGA which we would not normally be able to measure since the hardware is enclosed and it is very small. Also, the output signals are changing slowly (about once per second), so we don't get the benefit of recording signals for later analysis. For example, If you were using a logic analyzer to debug your UART protocol then it would make sense because the signals change fast and you need to record a failed message and then analyze it to discover possible areas of error. But in task #4 we can clearly see the address changing slowly.

The feedback for the laboratory was positive since I feel that I learned a lot. One good thing I learned is to use the IP catalog so that I can use modules that have been very well tested by a group of experts. Nevertheless, I also gained the knowledge to create my own modules that fit my application. This way, if I don't find the module that I want in the library, I can always make my own module that does exactly what I want it to do. Another valuable skill is the ability to use the SignalTap logic analyzer to debug designs, which I expect to be very useful during next laboratory (VGA).

Estimated total time to complete the laboratory: Around 11 to 14 hours.