Vincent Kwok & Hunter Hatzenbeler
EE/CSE 371
April 12, 2023
Lab 2 Report

Design Procedure:

Lab 2 centers around using System Verilog to program memory functionality on a remote FPGA. The several tasks in this lab show how the memory that can be programmed is highly customizable, with already created structures and templates for the programmer to utilize. A conceptual diagram of Random Access Memory (Figure 1) is provided below. For all tasks, the code will model the structure and operation of 32x3 RAM.

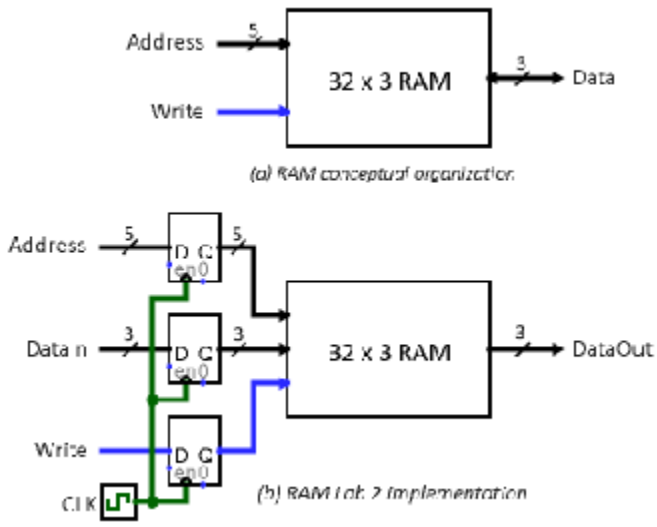


Figure 1: Conceptual diagram of Random Access Memory (32x3 RAM)

Task 1:

The procedure of task 1 creates RAM using library modules. By using the IP catalog in Quartus, a 1 port RAM with 3 bit width and 32 bit words of memory can be easily created in a ram32x3.v file to be implemented in a module to complete the task through instantiation. To verify the functionality of this file and its implementation, a testbench was created.

Task 2:

Task 2 requires the same outcome as task 1, utilizing the same testbench. While task 1 created RAM with a library module already created by Quartus, task 2 required us to program the RAM ourselves. In figure 2, line 12 represents the memory storage structure, with the word size of the RAM being the exponent of 2. A single address is representative of both the read and write address. As can be seen in line 16 and 20. Even when the write enable is on or off, the address bits will direct either an input or output. The module below is then initialized in a separate module (not shown in figure) to enable

interaction with FPGA hardware such as the switches for address and data selection, as well as displays for data and address information.

```
 3   module task2RAM ( address, CLOCK_50, data, wren, q);
 4      input logic [4:0]  address;
 5      input logic        CLOCK_50;
 6      input logic [2:0]  data;
 7      input logic        wren;
 8      output logic [2:0]  q;
 9
10      assign clock = CLOCK_50;
11
12      logic [2:0] memory_array [31:0];
13
14      always_ff @(posedge clock) begin
15         if(wren) begin
16            memory_array[address] <= data;
17            q <= data;
18
19         end else begin
20            q <= memory_array[address];
21         end
22
23      end
24
25   endmodule
```

Figure 2: 32x3 Ram module from task 2

Task 3:

Task 3 is a repetition of task 1 and 2, but the RAM module and specifications are different. Again the Quartus IP catalog is used to create a file representing a 2-port 32x3 RAM. With 1 port representing the write and another port representing read. The file is saved as a .mif (Memory Initialization File) file. Using the code from task 2, the testbench and module are modified to accommodate for the newly added port and separate functionality of both ports. Read and write data in this new module are represented on a HEX display while the switches on the FPGA specify the write data and address. Finally, a switch is used to swap between the task 3 and task 2 RAM. In figure 3, line 48 and line 51 initialize the modules required. Since both modules output data of 3 bits, 3 MUXs are created to select which memory register is being used. The FPGA hardware is also referred to in this module, with select being switch 9 and seven segment displays for addresses taking an input from the switches.

```
42      // Muxes to select between the output of task 2 and task 3 memory registers
43      mux2_1 Doutmux (.out(DataOut[0]), .i0(DoutT2[0]), .i1(DoutT3[0]), .sel(SW[9]));
44      mux2_1 Doutmux2 (.out(DataOut[1]), .i0(DoutT2[1]), .i1(DoutT3[1]), .sel(SW[9]));
45      mux2_1 Doutmux3 (.out(DataOut[2]), .i0(DoutT2[2]), .i1(DoutT3[2]), .sel(SW[9]));
46
47      // instantiates task 2
48      task2RAM t2 (.address(count), .CLOCK_50(CLOCK_50), .data(Din), .wren(~SW[9] & SW
49
50      // instantiates task 3
51      ram32x3port2 t3 (.clock(CLOCK_50), .data(Din), .rdaddress(count), .wraddress(SW[
52
53      seg7 h5 (.hex({3'b000, SW[8]}), .leds(HEX5)); // Hex 5, displays wraddress
54      seg7 h4 (.hex(SW[7:4]), .leds(HEX4));         // Hex 4, displays wraddress
55      seg7 h3 (.hex({3'b000, count[4]}), .leds(HEX3)); // Hex3, displays rdaddress
56      seg7 h2 (.hex(count[3:0]), .leds(HEX2)); // Hex 2, displays rdaddress
57      seg7 h1 (.hex(SW[3:1]), .leds(HEX1));  // Hex 1, displays Data in
58      seg7 h0 (.hex(DataOut), .leds(HEX0)); // Hex 0, displays data out
59
```

Figure 3: Task 3 module

Overall System:

The block diagram for task 2 (Figure 3) is essentially an expanded version of the Conceptual diagram from Figure 1. Each address and data switch is specified as well as the write and clock inputs. HEX outputs for the 7 segment displays on the FPGA are shown as well. Both block diagrams are color coded to represent structure, output, input, and use of the bits. There is a flip flop for each unique type of data to synchronize all inputs that go into the RAM, making the memory synchronous.

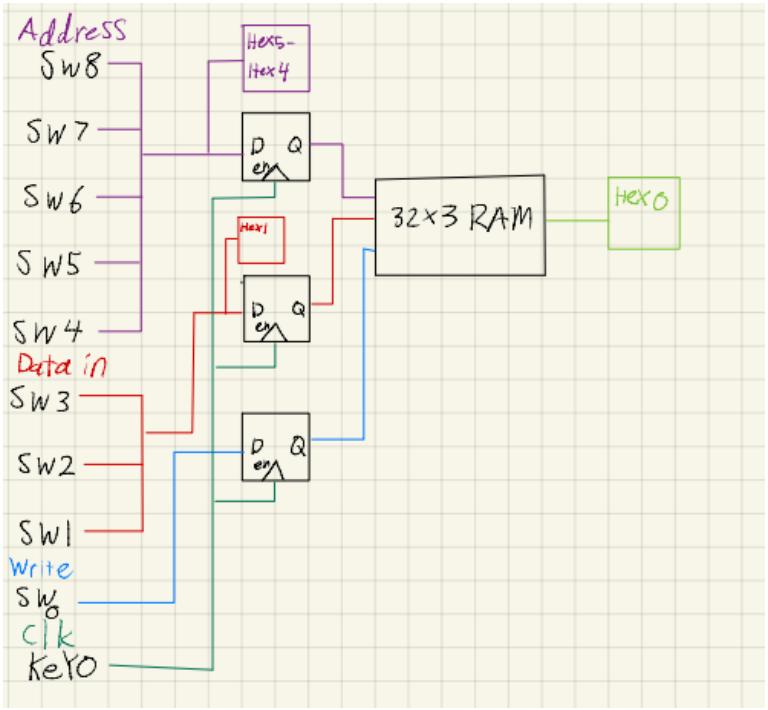


Figure 3: Task 2 block diagram

The task 3 block diagram (Figure 4) is quite similar to the task 2 block diagram. Instead, there is a switch that enables the read/write functionality between 2 32x3 RAM units. Due to this selection, a MUX must be connected to the switch as the selection to ensure the correct data output is chosen between the 2 RAM units. A counter is used and connected to clock to periodically change the data that is shown by the hex outputs. All 7 segment displays are used to show the that is being read, written, and address values. KEY 3 is used as a reset to clear all the data from both RAMs.

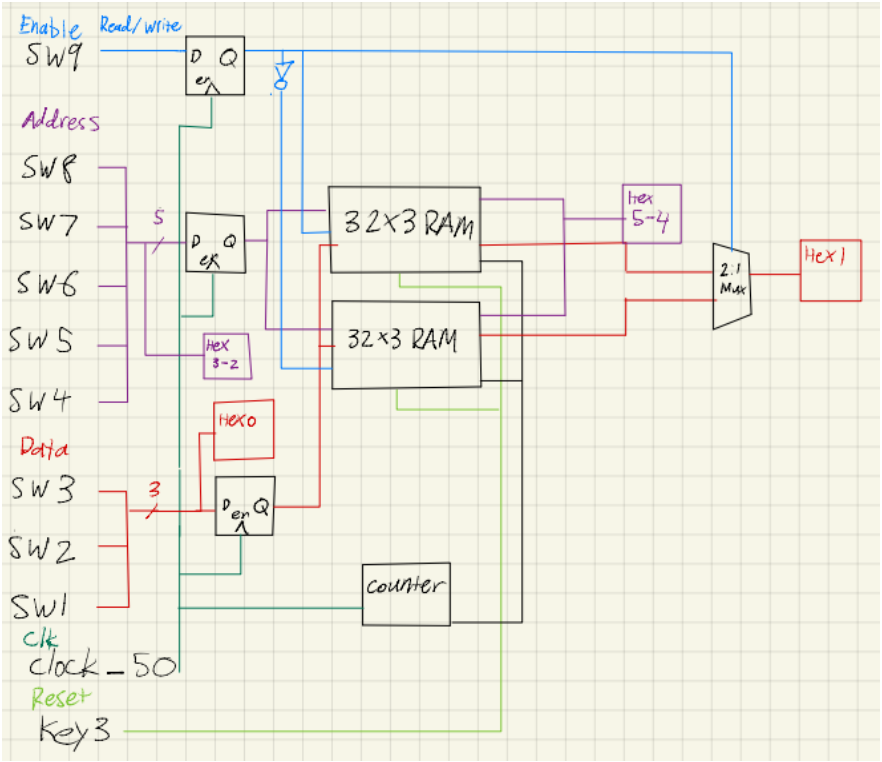


Figure 4: Task 3 block diagram

Results:

Task 1:

Figure 5: Task 1 testbench

The task 1 testbench was written very thoroughly, going through the 2 settings of enabling write, the 32 variations of 5 bits of address, the 8 variations of 3 bits of data, there are a total of 512 unique bit combinations that are all tested. Due to RAM having a flip flop type of functionality, any data that is written in is read out 1 clock cycle later to account for the time delay of the flip flop. The write is disabled for the first half of the testbench waveform (Figure). This causes no information to be read in the whole duration. After that, the output waveform always models the input waveform delayed by a single clock cycle.

Pictured in the screenshot is the expected behavior for data being written into the ram module. Initially, the output q for the module at address 8 is 0, even though the data in is 3'b111. However, when write is enabled, q outputs 3'b111 as expected. When the address is changed, q reads 0 again, and when address goes back to 8, the value 3'b111 is remembered.

Task2:

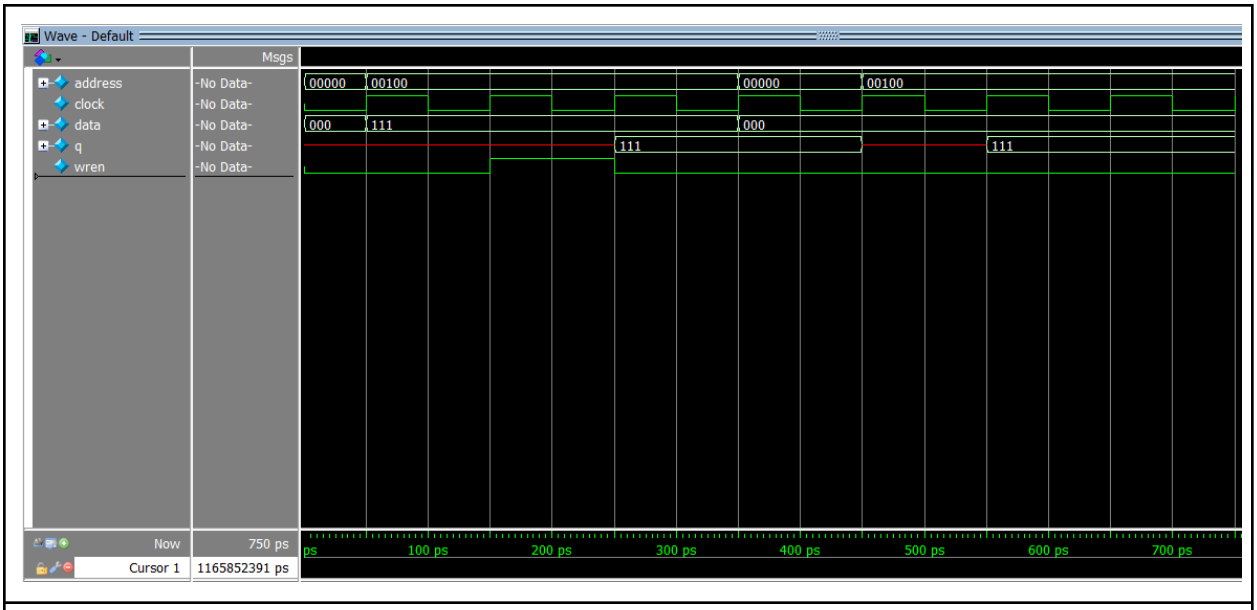
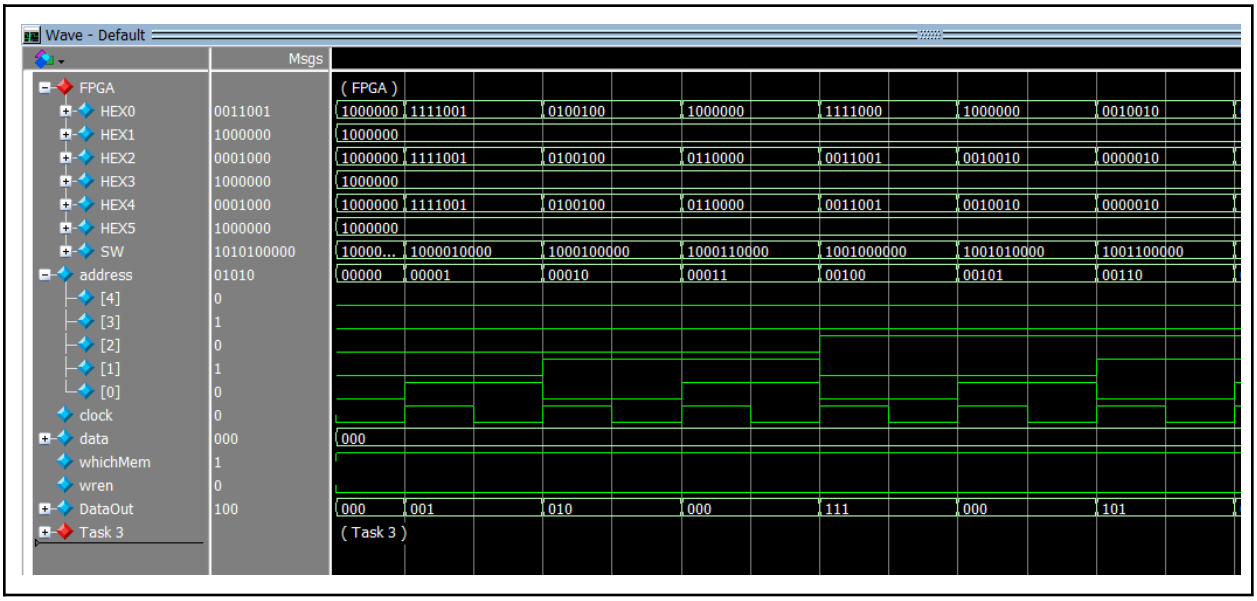


Figure 5: Task 2 testbench

Using the same testbench as task 1, the output waveform of task 2 also matched the output waveform of task 1. Because of this outcome, we concluded that the functionality of the module created by task 1 is the same as the functionality as the module created for task 2. The only difference between the two is that before an address has been written to, the output goes z, which is undefined, instead of 0.

Task 3:



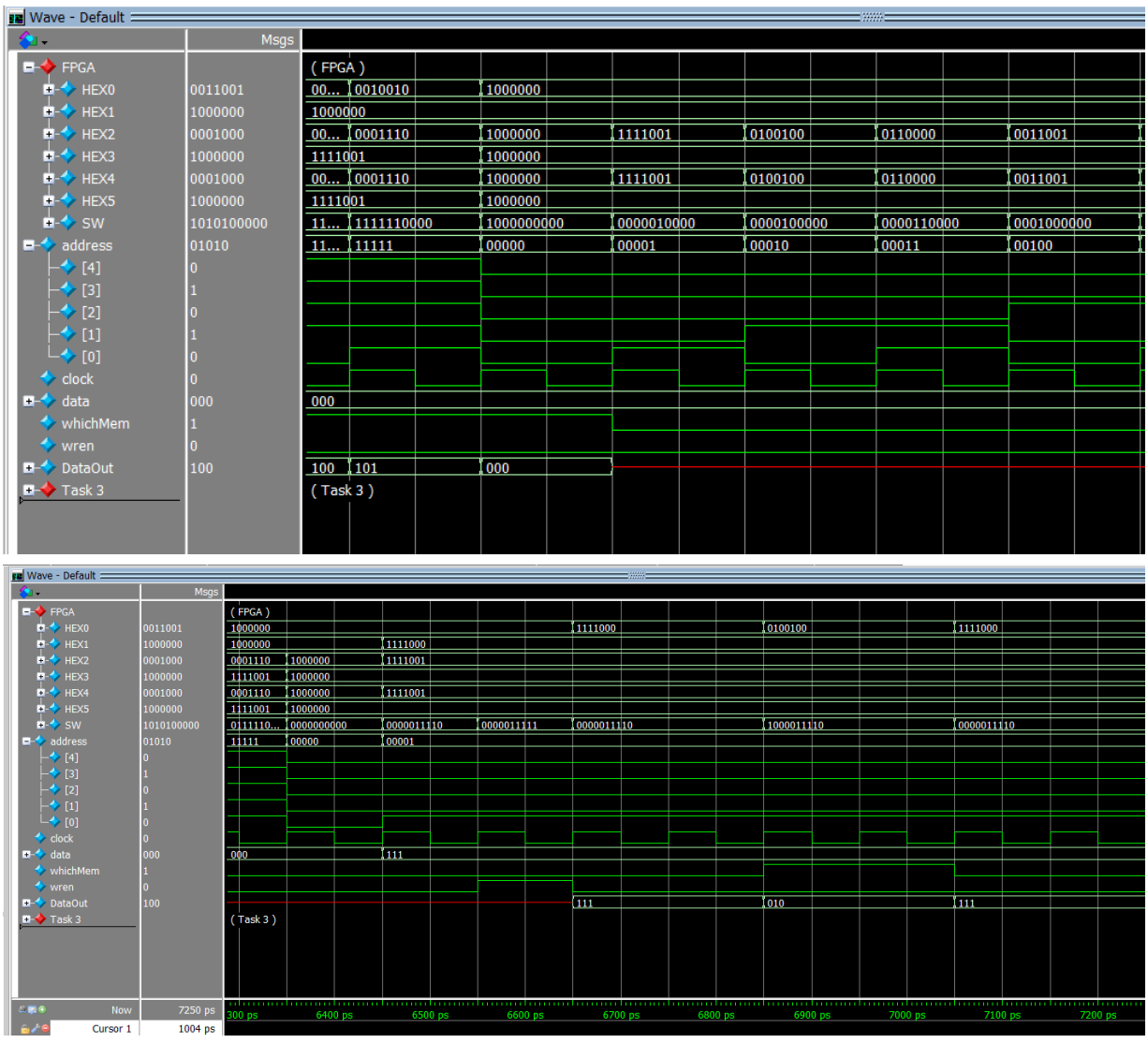


Figure 5: Task 3 testbench

Since task 1 and task 2 are the same in functionality but different in the method of creation, the outcome of task 3 is dependent on these tasks, as task 3 involves switching between a newly created RAM and another from task 2. The testbench of task 3 involves the implementation of the switches, 7 segment (HEX) displays, the switch between the task 3 and task 2 memory modules.

The first screenshot shows the first case for task 3, where the rdaddress is incremented without having written to the file. Since SW[9] is high, it is reading from the .mif file ram, and so as you can see in the data out line there are already values written, which is the expected behavior.

The second screenshot shows the second case for task 3. Rdaddress is again incremented, but this time since SW[9] is low, it is reading from the task2 ram file. This time, the dataout is empty, which is the expected behavior since nothing has been written to this ram.

The third screenshot shows the third case for task 3. First, it writes 7 into register 10 of task 2 ram , which then shows up in the output. Then, SW9 switches on, and dataout changes as it reads from the task3 ram. When SW9 switches back off, dataout returns to the value 7 that was written there previously.

Flow Summary:

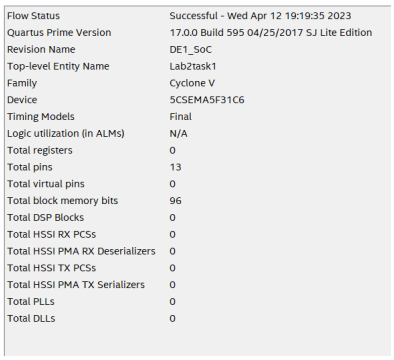
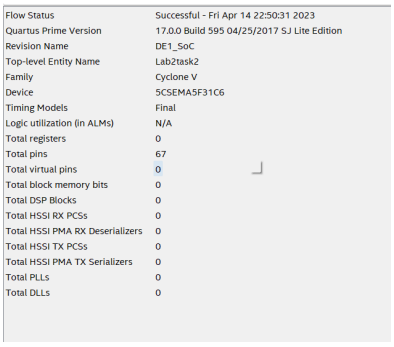
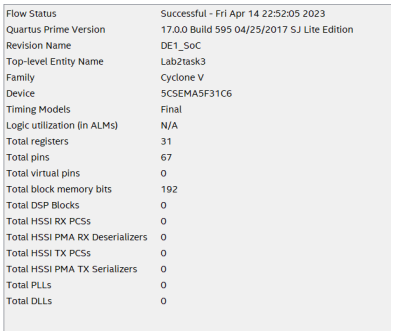
		

Figure 6:The ModelSim Flow Summary of the task 1 module compilation	Figure 7:The ModelSim Flow Summary of the task 2 module compilation	Figure 8:The ModelSim Flow Summary of the task 3 module compilation

Experience Report:

We found this lab to be straightforward due to a large portion of the lab being often focused on using library modules. In lab 2, we ran into a strange problem that made it impossible for the runlab.do file to detect new task files and had to re-create the whole project, causing our debugging time to greatly increase. The slides of code in lecture were very helpful in giving us a hint as to how to program the lab.

Overall the design/ programming process of this lab went smoothly other than that error, the most time was spent on the coding and debugging of the code. Design of the block diagrams took some tweaking and the reading, though dense, was also often a simple list of instructions that did not have to be read over several times.

This lab took approximately 8 hours, broken down as follows:

- Reading - 30 minutes
- Planning - 40 minutes
- Design- 80 minutes
- Coding- 150 minutes
- Testing- 60 minutes
- Debugging- 120 minutes