# CSE 371 Lab 2 Report

Elijah Melton, 2164822
Ankith Tunuguntla, 2234509

## Design Procedure

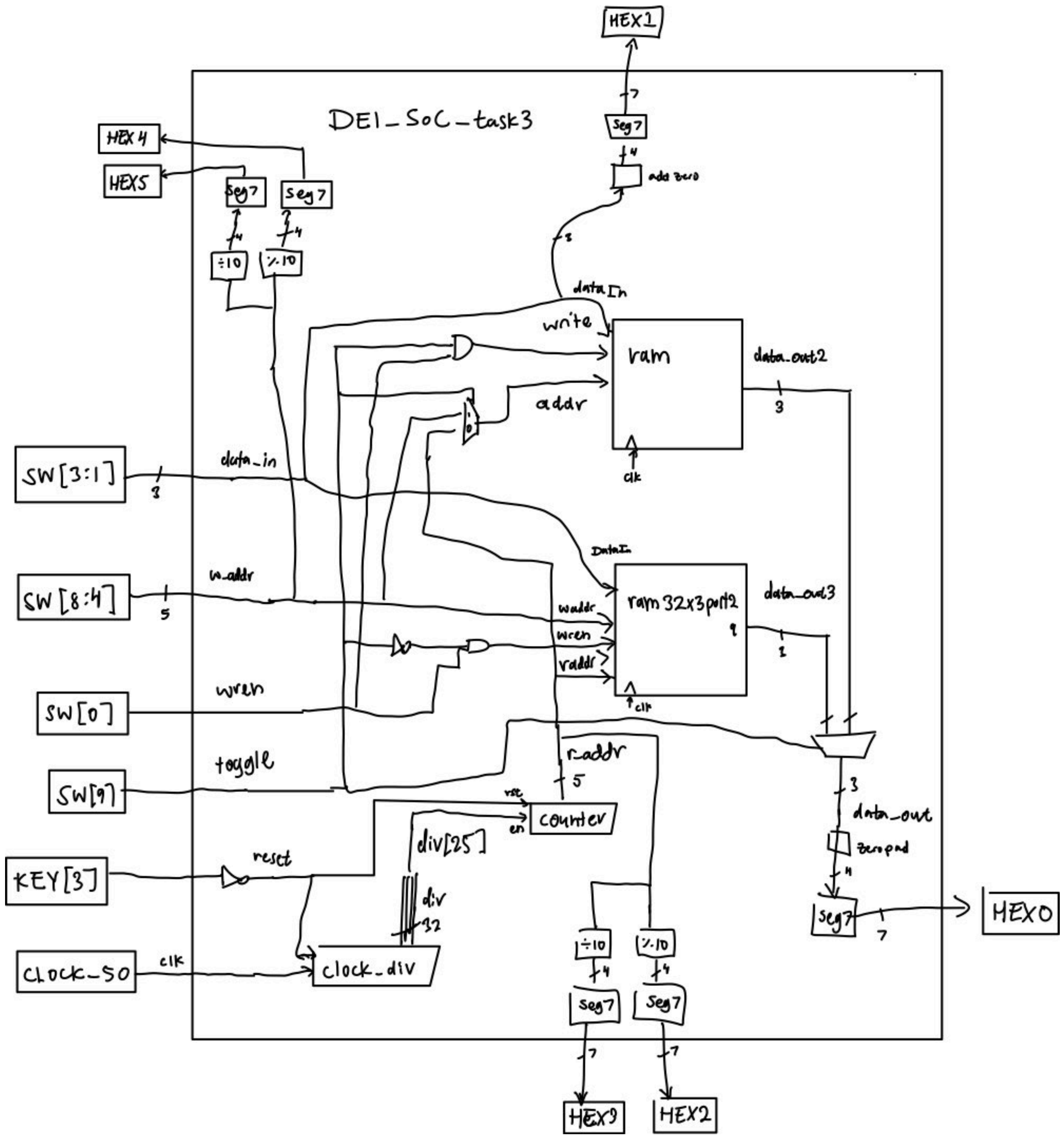
### Block diagram for Task 3 (including task2)

Below is our top level diagram for task 3, which also contains our task 2 module. This module takes in inputs from hardware including the write address, reset signal, write enable, data in, , toggle, and clock. Each of these are controlled by peripherals (e.g. `SW`, `KEY`). Furthermore, it outputs the read address, write address, data out, and data in to the `HEX` displays.

It uses a clock divider to control the read address, counting up one by one from 0-32 on each clock div edge.

The toggle is used to control both the write enable signals to our two ram modules, and to demultiplex the data output from our two ram modules. When toggle is high, `ram` is active, and conversely when `toggle` is low, `ram32x3port2` is active. The data output is fed into `seg7` modules, and then that output is hooked up to `HEX0`. Similarly, data in is put through a `seg7` module and fed into `HEX1`, and the same for the read address and write address to HEX 2, 3 and 4, 5 respectively. Notably, we need to split the read and write addresses into 10s place and ones places using /10 and %10 so the overall address is displayed correctly on the HEX displays.

Lastly, we need to make sure we can only write to one ram module at a time. To do this, the write enable signal is anded with `toggle` and `~toggle` for `ram` and `ram32x3port2` resp so they are only write enabled when they are active.


DE1_SoC_task3
HEX1
7
Seg7
4
add zero
3
HEX4
HEX5
Seg7
Seg7
4
4
÷10
%10
dataIn
write
ram
data_out2
3
addr
clk
SW[3:1]
data_in
3
DataIn
SW[8:4]
w_addr
5
waddr
wren
raddr
ram32x3port2
q
data_out3
3
clk
SW[0]
wren
SW[9]
toggle
r_addr
5
3
data_out
zeropad
4
seg7
7
HEX0
rst
en
counter
div[25]
KEY[3]
reset
div
32
clock_div
CLOCK_50
clk
÷10
%10
4
4
Seg7
Seg7
7
7
HEX3
HEX2

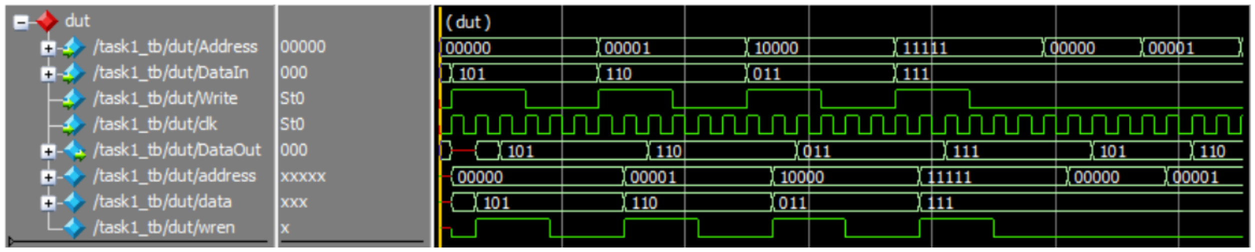
## Results

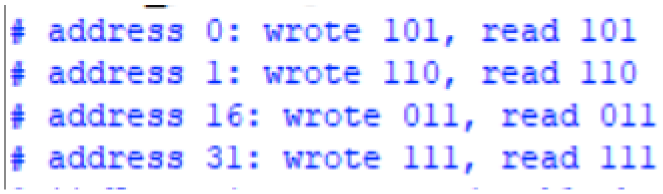
### Task 1 (ram32x3)

#### Implementation Details

In task 1 we implemented a wrapper for the ip ram module generated by Quartus. This was a fairly simple module to implement, since it basically only needed registers in front of the inputs, and everything else was a direct wrapper for the ip ram. These registers can introduce an extra clock cycle of delay before reads/writes, but otherwise have no effect. This is nice for metastability, so that inputs are synchronized with the clock cycle.

#### Simulation





```
# address 0: wrote 101, read 101
# address 1: wrote 110, read 110
# address 16: wrote 011, read 011
# address 31: wrote 111, read 111
```

This test bench also contains assertions to make sure the values can be read again at the end of the test. All assertions pass.

### Task 2 (ram)

#### Implementation Details

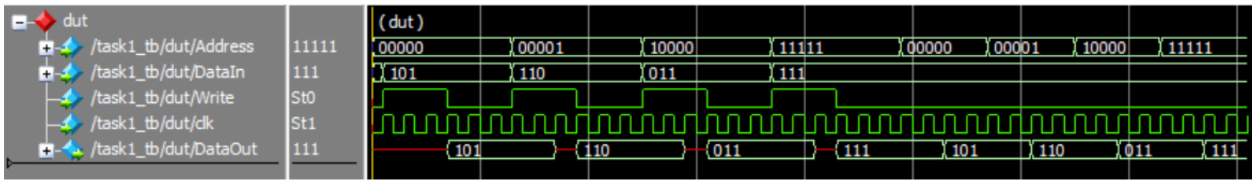
`ram` was very similar to `task1`, with the only difference being we needed to manage the memory ourselves. We used the same register scheme, but instead of passing inputs/outputs directly into another module, we instead used the following:

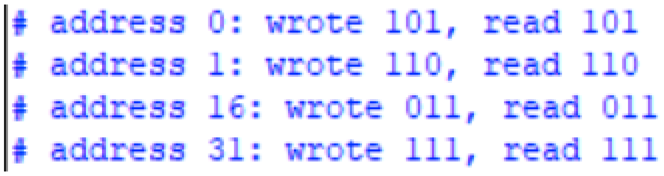
```
logic [2:0] memory_array [0:31];
// ...
always_ff @(posedge clk) begin
    if (write_reg) begin
```

```
            memory_array[address_reg] <= data_reg;
        end
        DataOut <= memory_array[address_reg];
    end
```

## Simulation

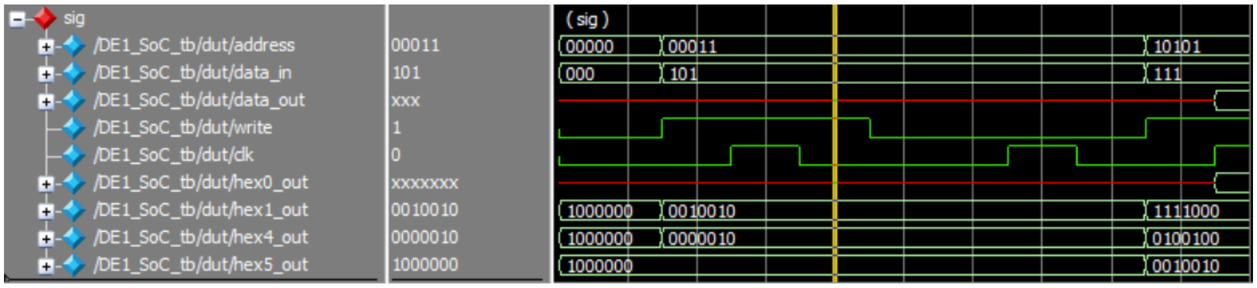
I tested `ram` with the same test bench. The simulation and output appeared identical, with the one exception being there is no default initialized value, and DataIn doesn't overwrite DataOut, so DataOut is undefined before it is written. All assertions pass as well.

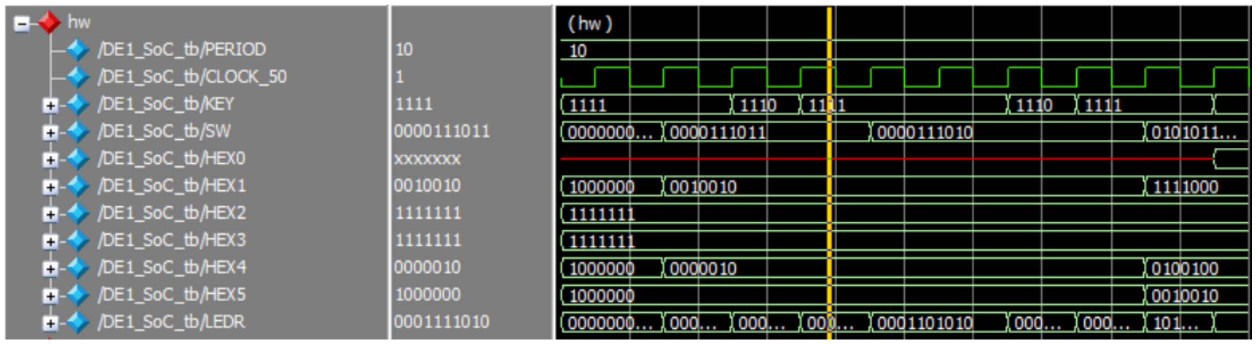


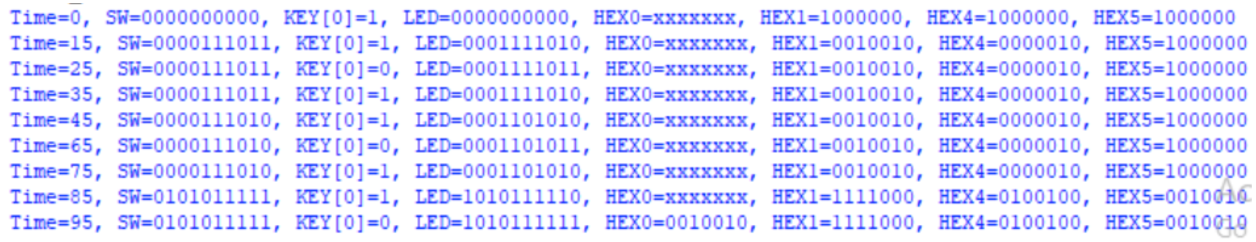


```
# address 0: wrote 101, read 101
# address 1: wrote 110, read 110
# address 16: wrote 011, read 011
# address 31: wrote 111, read 111
```

## Task 2 (DE1_SoC)


sig
/DE1_SoC_tb/dut/address 00011
/DE1_SoC_tb/dut/data_in 101
/DE1_SoC_tb/dut/data_out xxx
/DE1_SoC_tb/dut/write 1
/DE1_SoC_tb/dut/clk 0
/DE1_SoC_tb/dut/hex0_out xxxxxxx
/DE1_SoC_tb/dut/hex1_out 0010010
/DE1_SoC_tb/dut/hex4_out 0000010
/DE1_SoC_tb/dut/hex5_out 1000000
(sig)
00000 00011 10101
000 101 111
1000000 0010010 1111000
1000000 0000010 0100100
1000000 0010010





```
Time=0, SW=0000000000, KEY[0]=1, LED=0000000000, HEX0=xxxxxxx, HEX1=1000000, HEX4=1000000, HEX5=1000000
Time=15, SW=0000111011, KEY[0]=1, LED=0001111010, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=25, SW=0000111011, KEY[0]=0, LED=0001111011, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=35, SW=0000111011, KEY[0]=1, LED=0001111010, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=45, SW=0000111010, KEY[0]=1, LED=0001101010, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=65, SW=0000111010, KEY[0]=0, LED=0001101011, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=75, SW=0000111010, KEY[0]=1, LED=0001101010, HEX0=xxxxxxx, HEX1=0010010, HEX4=0000010, HEX5=1000000
Time=85, SW=0101011111, KEY[0]=1, LED=1010111110, HEX0=xxxxxxx, HEX1=1111000, HEX4=0100100, HEX5=0010010
Time=95, SW=0101011111, KEY[0]=0, LED=1010111111, HEX0=0010010, HEX1=1111000, HEX4=0100100, HEX5=0010010
```

## Task 3 (DE1_SoC_task3)

### Implementation Details

For the combined task 2 and 3, we used a clock divider for the read address counter

To prevent writes to both rams, we conditioned write enable on the toggle signal

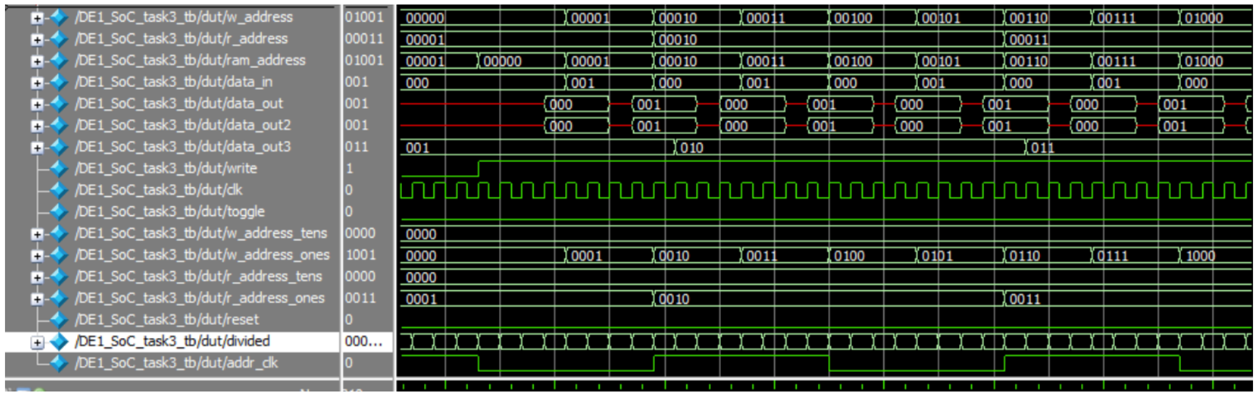
```
assign toggle    = SW[9];    // SW9 to toggle between dual and single port RAM

// ...

// Single-port RAM (task2)
 ram ram_task2_inst (
     .Address(ram_address),  // Use the selected address
     .DataIn(data_in),
     .Write(write & ~toggle),
     .clk(clk),
     .DataOut(data_out2)
 );

 // Dual-port RAM (task3)
 ram32x3port2 ram_task3_inst (
     .clock(clk),
     .data(data_in),
     .rdaddress(r_address),  // Read port always uses counter address
     .wraddress(w_address),  // Write port always uses switch address
     .wren(write & toggle),
     .q(data_out3)
 );
```

Otherwise, both rams are constantly reading values, but the actual data_out value is muxed by our `toggle` variable. To handle both reads and writes for our single port ram, we muxed the `w_address` and `r_address` depending on whether `write` was enabled.

```
    // select address for single-port RAM
    assign ram_address = write ? w_address : r_address;
    // Select which RAM's output to display based on toggle switch
    assign data_out = toggle ? data_out3 : data_out2;
```

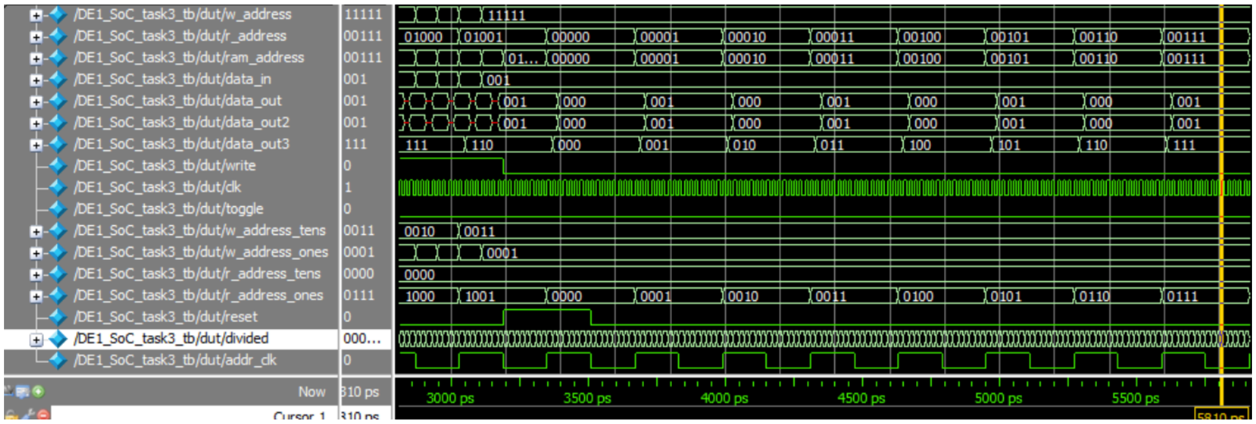
## Simulation

First I start by writing all of the task2 memory to have the value addr % 2:



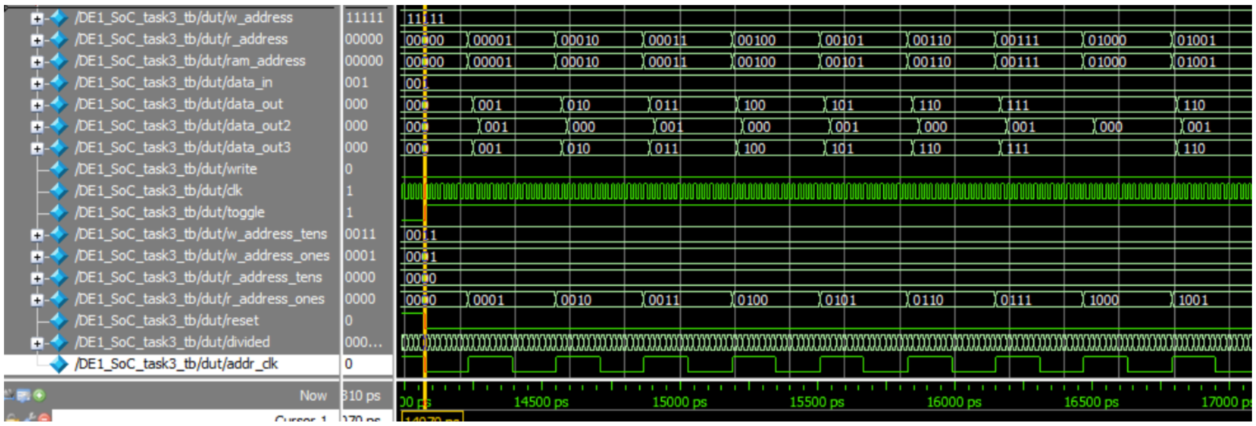
```
/DE1_SoC_task3_tb/dut/w_address         01001
/DE1_SoC_task3_tb/dut/r_address         00011
/DE1_SoC_task3_tb/dut/ram_address       01001
/DE1_SoC_task3_tb/dut/data_in           001
/DE1_SoC_task3_tb/dut/data_out          001
/DE1_SoC_task3_tb/dut/data_out2         001
/DE1_SoC_task3_tb/dut/data_out3         011
/DE1_SoC_task3_tb/dut/write             1
/DE1_SoC_task3_tb/dut/clk               0
/DE1_SoC_task3_tb/dut/toggle            0
/DE1_SoC_task3_tb/dut/w_address_tens    0000
/DE1_SoC_task3_tb/dut/w_address_ones    1001
/DE1_SoC_task3_tb/dut/r_address_tens    0000
/DE1_SoC_task3_tb/dut/r_address_ones    0011
/DE1_SoC_task3_tb/dut/reset             0
/DE1_SoC_task3_tb/dut/divided           000...
/DE1_SoC_task3_tb/dut/addr_clk          0
```

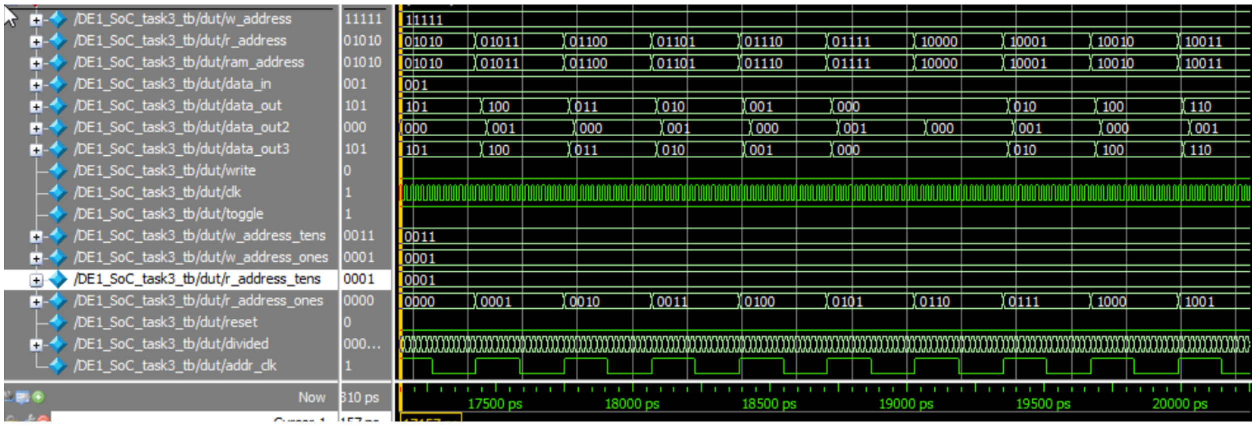
This continues on for a while, until I write all addresses. Then, I read back all the addresses from said ram to make sure it was saved:


/DE1_SoC_task3_tb/dut/w_address 11111
/DE1_SoC_task3_tb/dut/r_address 00111
/DE1_SoC_task3_tb/dut/ram_address 00111
/DE1_SoC_task3_tb/dut/data_in 001
/DE1_SoC_task3_tb/dut/data_out 001
/DE1_SoC_task3_tb/dut/data_out2 001
/DE1_SoC_task3_tb/dut/data_out3 111
/DE1_SoC_task3_tb/dut/write 0
/DE1_SoC_task3_tb/dut/clk 1
/DE1_SoC_task3_tb/dut/toggle 0
/DE1_SoC_task3_tb/dut/w_address_tens 0011
/DE1_SoC_task3_tb/dut/w_address_ones 0001
/DE1_SoC_task3_tb/dut/r_address_tens 0000
/DE1_SoC_task3_tb/dut/r_address_ones 0111
/DE1_SoC_task3_tb/dut/reset 0
/DE1_SoC_task3_tb/dut/divided 000...
/DE1_SoC_task3_tb/dut/addr_clk 0
3000 ps
3500 ps
4000 ps
4500 ps
5000 ps
5500 ps

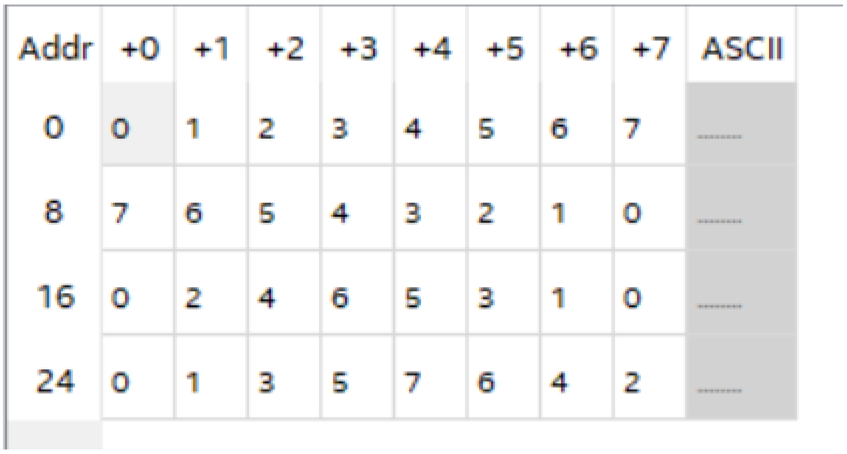
As expected, all values match addr % 2

Lastly, I reset, toggle to switch to the other ram, and read all the cells in the two-port ram:


/DE1_SoC_task3_tb/dut/w_address 11111
/DE1_SoC_task3_tb/dut/r_address 00000
/DE1_SoC_task3_tb/dut/ram_address 00000
/DE1_SoC_task3_tb/dut/data_in 001
/DE1_SoC_task3_tb/dut/data_out 000
/DE1_SoC_task3_tb/dut/data_out2 000
/DE1_SoC_task3_tb/dut/data_out3 000
/DE1_SoC_task3_tb/dut/write 0
/DE1_SoC_task3_tb/dut/clk 1
/DE1_SoC_task3_tb/dut/toggle 1
/DE1_SoC_task3_tb/dut/w_address_tens 0011
/DE1_SoC_task3_tb/dut/w_address_ones 0001
/DE1_SoC_task3_tb/dut/r_address_tens 0000
/DE1_SoC_task3_tb/dut/r_address_ones 0000
/DE1_SoC_task3_tb/dut/reset 0
/DE1_SoC_task3_tb/dut/divided 000...
/DE1_SoC_task3_tb/dut/addr_clk 0
14500 ps
15000 ps
15500 ps
16000 ps
16500 ps
17000 ps

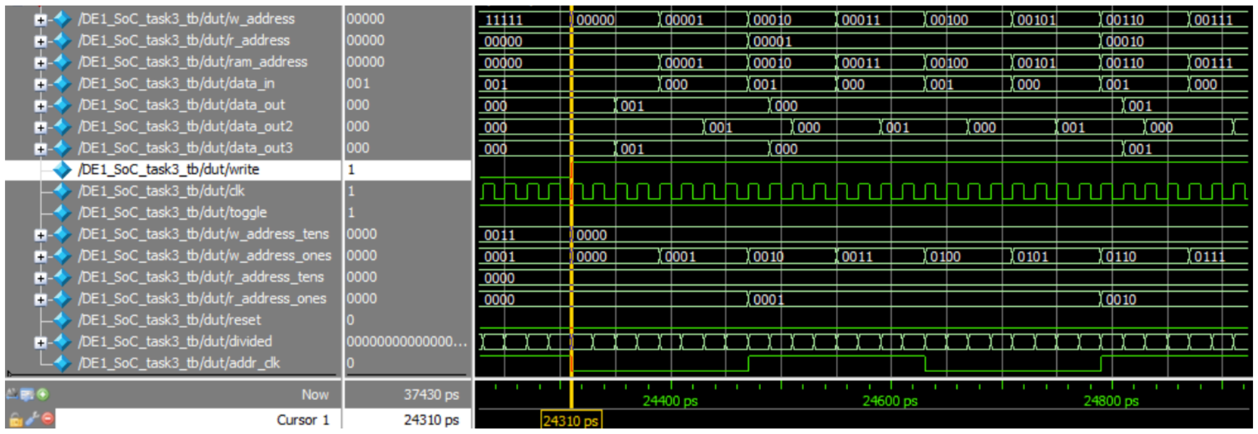

/DE1_SoC_task3_tb/dut/w_address 11111
/DE1_SoC_task3_tb/dut/r_address 01010
/DE1_SoC_task3_tb/dut/ram_address 01010
/DE1_SoC_task3_tb/dut/data_in 001
/DE1_SoC_task3_tb/dut/data_out 101
/DE1_SoC_task3_tb/dut/data_out2 000
/DE1_SoC_task3_tb/dut/data_out3 101
/DE1_SoC_task3_tb/dut/write 0
/DE1_SoC_task3_tb/dut/clk 1
/DE1_SoC_task3_tb/dut/toggle 1
/DE1_SoC_task3_tb/dut/w_address_tens 0011
/DE1_SoC_task3_tb/dut/w_address_ones 0001
/DE1_SoC_task3_tb/dut/r_address_tens 0001
/DE1_SoC_task3_tb/dut/r_address_ones 0000
/DE1_SoC_task3_tb/dut/reset 0
/DE1_SoC_task3_tb/dut/divided 000...
/DE1_SoC_task3_tb/dut/addr_clk 1
17500 ps
18000 ps
18500 ps
19000 ps
19500 ps
20000 ps

This also continues on for a while. Comparing the values read from the dual port ram, we can see we read the expected values (counting up from 0-7, then down from 7-0).



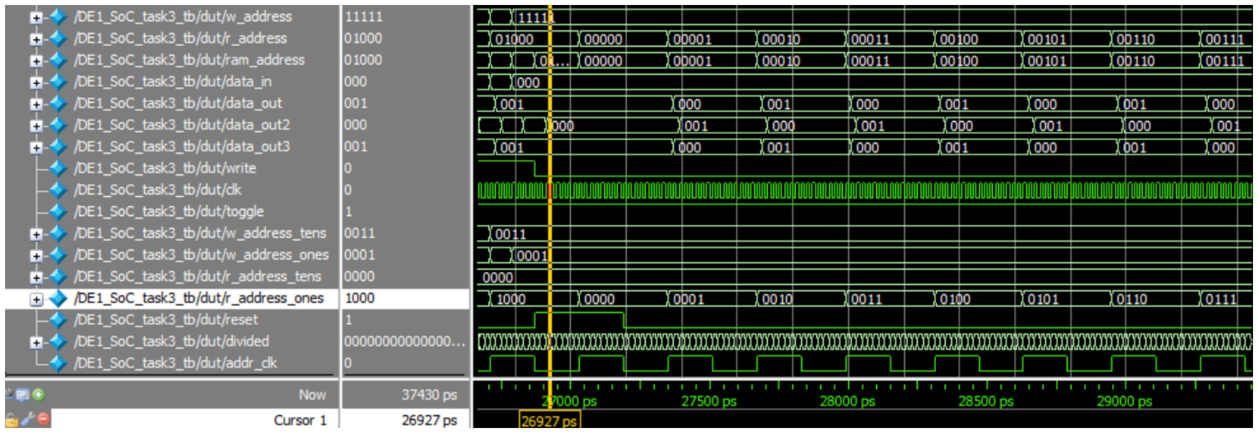
| Addr | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | ASCII |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ........ |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ........ |
| 16 | 0 | 2 | 4 | 6 | 5 | 3 | 1 | 0 | ........ |
| 24 | 0 | 1 | 3 | 5 | 7 | 6 | 4 | 2 | ........ |

Lastly, I test writing to the dual port ram by writing every address with `~(addr % 2)`

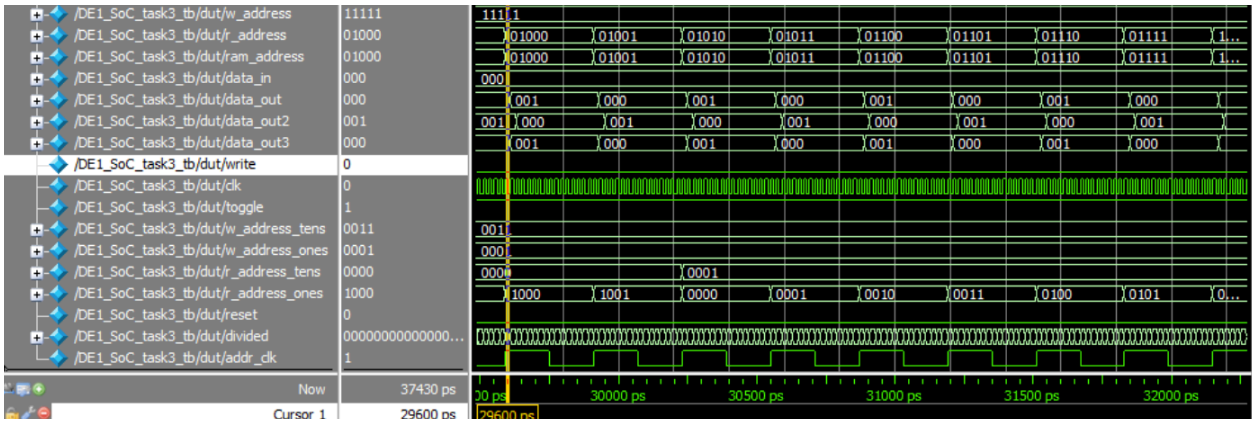


| | |
|---|---|
| /DE1_SoC_task3_tb/dut/w_address | 00000 |
| /DE1_SoC_task3_tb/dut/r_address | 00000 |
| /DE1_SoC_task3_tb/dut/ram_address | 00000 |
| /DE1_SoC_task3_tb/dut/data_in | 001 |
| /DE1_SoC_task3_tb/dut/data_out | 000 |
| /DE1_SoC_task3_tb/dut/data_out2 | 000 |
| /DE1_SoC_task3_tb/dut/data_out3 | 000 |
| /DE1_SoC_task3_tb/dut/write | 1 |
| /DE1_SoC_task3_tb/dut/clk | 1 |
| /DE1_SoC_task3_tb/dut/toggle | 1 |
| /DE1_SoC_task3_tb/dut/w_address_tens | 0000 |
| /DE1_SoC_task3_tb/dut/w_address_ones | 0000 |
| /DE1_SoC_task3_tb/dut/r_address_tens | 0000 |
| /DE1_SoC_task3_tb/dut/r_address_ones | 0000 |
| /DE1_SoC_task3_tb/dut/reset | 0 |
| /DE1_SoC_task3_tb/dut/divided | 00000000000000... |
| /DE1_SoC_task3_tb/dut/addr_clk | 0 |
| Now | 37430 ps |
| Cursor 1 | 24310 ps |

And then read every address back. Note that this is the opposite scheme of the one used for the ram from task2, so we'd expect them to always be eachother's complement. As you can see below, this holds true, meaning writing to both ram modules works, and writes don't overwrite eachother.



| | |
|---|---|
| /DE1_SoC_task3_tb/dut/w_address | 11111 |
| /DE1_SoC_task3_tb/dut/r_address | 01000 |
| /DE1_SoC_task3_tb/dut/ram_address | 01000 |
| /DE1_SoC_task3_tb/dut/data_in | 000 |
| /DE1_SoC_task3_tb/dut/data_out | 001 |
| /DE1_SoC_task3_tb/dut/data_out2 | 000 |
| /DE1_SoC_task3_tb/dut/data_out3 | 001 |
| /DE1_SoC_task3_tb/dut/write | 0 |
| /DE1_SoC_task3_tb/dut/clk | 0 |
| /DE1_SoC_task3_tb/dut/toggle | 1 |
| /DE1_SoC_task3_tb/dut/w_address_tens | 0011 |
| /DE1_SoC_task3_tb/dut/w_address_ones | 0001 |
| /DE1_SoC_task3_tb/dut/r_address_tens | 0000 |
| /DE1_SoC_task3_tb/dut/r_address_ones | 1000 |
| /DE1_SoC_task3_tb/dut/reset | 1 |
| /DE1_SoC_task3_tb/dut/divided | 00000000000000... |
| /DE1_SoC_task3_tb/dut/addr_clk | 0 |
| Now | 37430 ps |
| Cursor 1 | 26927 ps |



```
/DE1_SoC_task3_tb/dut/w_address        11111
/DE1_SoC_task3_tb/dut/r_address        01000
/DE1_SoC_task3_tb/dut/ram_address      01000
/DE1_SoC_task3_tb/dut/data_in          000
/DE1_SoC_task3_tb/dut/data_out         000
/DE1_SoC_task3_tb/dut/data_out2        001
/DE1_SoC_task3_tb/dut/data_out3        000
/DE1_SoC_task3_tb/dut/write            0
/DE1_SoC_task3_tb/dut/clk              0
/DE1_SoC_task3_tb/dut/toggle           1
/DE1_SoC_task3_tb/dut/w_address_tens   0011
/DE1_SoC_task3_tb/dut/w_address_ones   0001
/DE1_SoC_task3_tb/dut/r_address_tens   0000
/DE1_SoC_task3_tb/dut/r_address_ones   1000
/DE1_SoC_task3_tb/dut/reset            0
/DE1_SoC_task3_tb/dut/divided          00000000000000...
/DE1_SoC_task3_tb/dut/addr_clk         1
                                 Now   37430 ps
                            Cursor 1   29600 ps
```

# Experience Report

## Feedback

This lab was more challenging than the last, but still manageable. One piece of feedback is that I was a little unclear initially that we were supposed to use the counter for reading from both the dual port and single port ram in task3. At first I thought we were also supposed to switch all the hardware mappings to match task2 exactly for the single port ram.

## Significant Issues

None to speak of

## Tips/Tricks

- Design before you implement!
- Always reset at the start of your test benches so sequential logic has a chance to initialize

## Time Spent

~13 hours