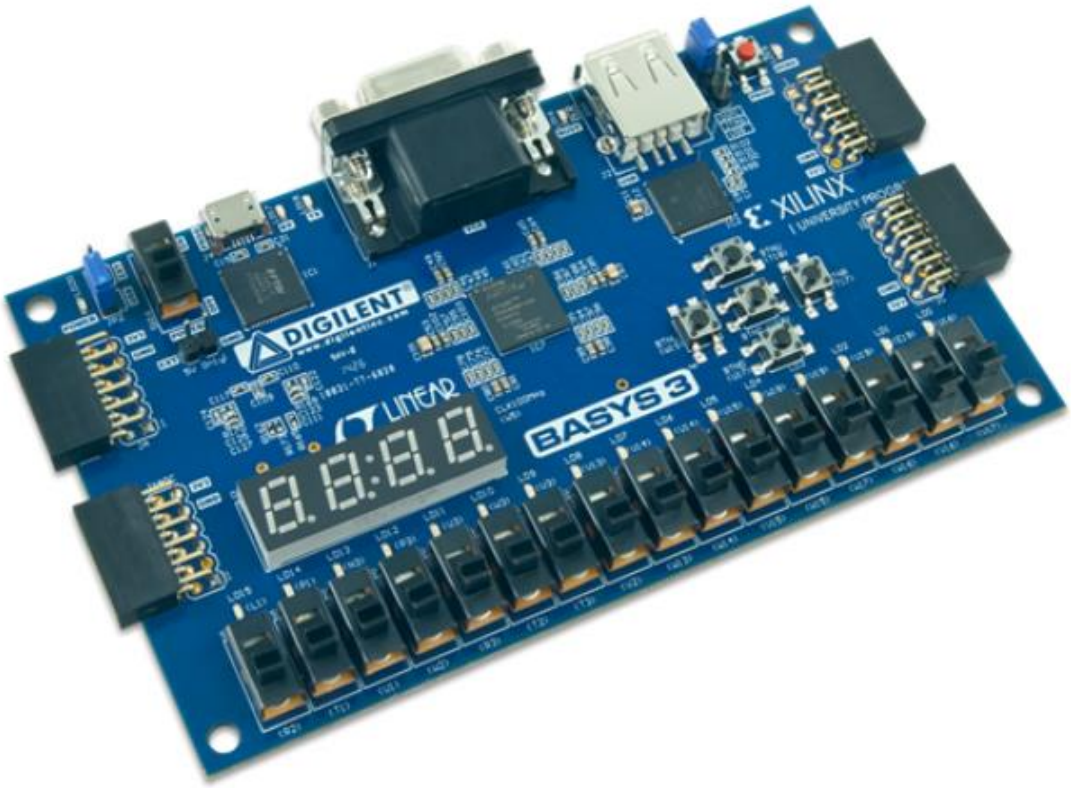
FIFO LAB - ECE351 Digital Logic Design

Ryan Pauly
The University of Tennessee Knoxville



Date Completed: 3/11/2019

The main goal of lab 2 is to create a first-in first-out (FIFO) controller. Task1 requires that we utilize the IP catalog's Block RAM to create this FIFO controller. Task2 requires that we use the IP catalog's built in FIFO generator to again create a FIFO controller. Specifications for the lab requires that we use read/write width of 8 and a read/write depth of 16 for both task1 and task2. Read and write width essentially is the data that is being input or output. The width is how many bits we want to store with input and read with output. Depth is how many locations for the input to be stored and read which is 16. This means there are 16 total locations where data can be written and consequently read after being written. We are using a single-clock (synchronous) to write and read with both task1 and task2, and the clock will be based off a button on the Basys3 board, specifically the center button (BTNC) or pin U18. This allows the input and output to be done at a pace that is visible to us. If it were simply tied to the board's 100Mhz clock, we would have a very difficult time understanding what is occurring even with the 7segment implemented. Also, we need to be able to determine when we wish to read or write, so we have a read and write switch that we can use to toggle between reading or writing, this is SW15 or pin R2. In my implementation, when the SW15 is pushed upward (1) then we are in write mode, and when it is in the lower position (0) then we are in read mode. Next, we need to be able to determine what we would like to write, so switches 7 down to 0 are our data input switches. As mentioned previously we also will need to use the seven-segment display to show the current address and what is being written or read to that address. For my implementation my left-most digit (the third digit) outputs in HEX the address value. The two right-most digits show the data, both in HEX. Lastly, we use several LEDs, specifically LEDS 15-10, to show several statuses. The LEDs will show if we are in read or write mode, and the LEDs will show if we are full (have written to our depth), if we are empty (have read to our depth), and almost-empty and almost-full which will just light up if we are about to hit full or empty.

This lab proved to be quite difficult as there seemed to be a huge delay on the addresses between reading and writing that proved to be quite a serious issue in task1. The problem that task1 posed is that we are using a synchronous block RAM IP to create this FIFO controller, and fortunately I discovered that there was a checkbox that was adding a whole extra clock delay.

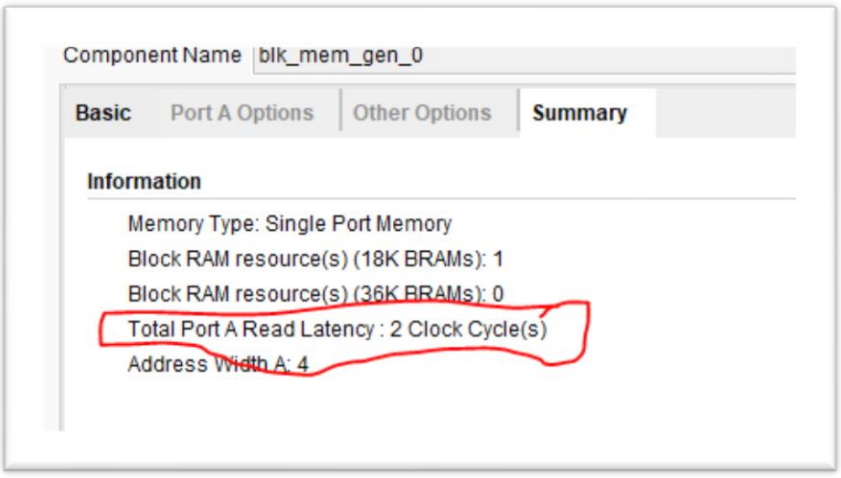


Figure 1. Image showing the clock delay within the Block RAM's settings and summary.

Figure 1 shows the clock cycle latency which displays 2 Clock Cycles of latency, this is not good for the implementation that we want. The way to fix this was to uncheck a check-box that gave us a primitive reading mode.

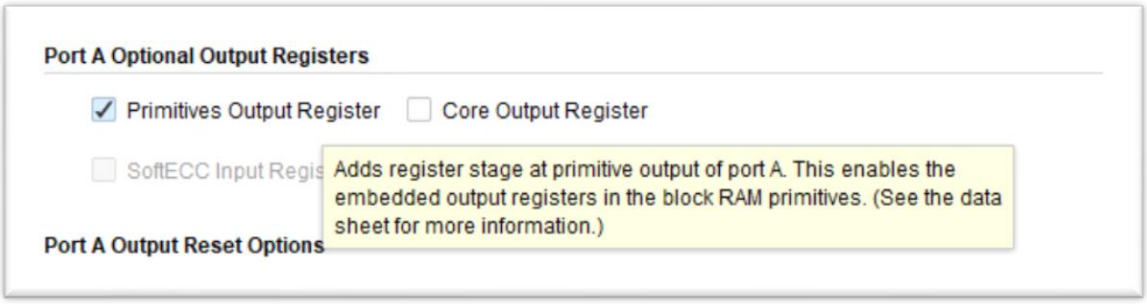


Figure 2. The 'Primitives Output Register' seemed to be the source of the additional Clock Cycle latency.

With this checkbox unchecked from Figure 2, we will then only have 1 clock cycle of latency. This dramatically improved the situation and made finishing this lab much simpler.

The logic to complete this lab centered heavily on the addresses of both reading and writing. The way to understand this lab is to think of a wheel with 16 different locations to read and write (0 to 15). To implement the lab, I created two "pointers" which count from 0000 to 1111 in binary. My pointer for writing increments whenever we write (if we are not full), and my read point increments whenever we read (if we are not empty). Additionally, we want to keep track of how full or empty we might be, so we need another signal which I called "counter." Counter will increment when we write and decrement when we read. Strangely enough we need a range of 0 to 16 for our counter. The counter can go up to 16, but when it does it knows we are full, because each time we write, the counter is incrementing, so truthfully if we only had 0 to 15 for the range of counter, to fill up the entire FIFO we would end up with counter incrementing one more past 15. I made this counter signal an integer with a fixed range of 0 to 16. A portion of this logic can be seen in Figure 3 below. In retrospect this could have been more simplified if perhaps I tried using "if-statements" instead of case statements.

Overall the lab was interesting but also challenging. There was a lot of 'nitty-gritty' logic that had to be taken care of throughout implementing task1 and task2, specifically with clock cycle latency. The most difficult part for me was keeping track of addresses when writing and reading and ensuring that when you change the mode you are in (reading/writing) that the pointers are looking at the correct location. Thankfully we did have an extension on this lab that allowed for more time to be spent on this assignment. While I was working on the lab I would save the code I had into a Notepad++ VHDL file and then rewrite or change code and noticed after I had finally received my signature after a fully working implementation, I discovered some of my code was rather redundant and could have been simplified. If I had more time to work on the assignment I would spend it revising my code to be more efficient and shorter.

APPENDIX

```
    action_process  :   process(new_btn)
    begin
        if falling_edge(new_btn) then

            --dataSig <= Data;
--          address <= ptr_write when SW15 = "1" else ptr_read;
--          adrSig  <= address;
            if counter < 15 then
                latch <= '0';
            else
                latch <= '1';
            end if;

            if SW15 = "1" then -- WRITE
                --ptr_read <= ptr_read;
                case counter is
                    when 0 to 14    =>
                        counter     <=  counter + 1;
                        --ptr_write   <=  ptr_write + "0001";
                        adrSig      <= ptr_write;
                        ptr_write <= ptr_write + "0001";
                    when 15         =>
                        counter <= counter + 1;
                        adrSig  <= ptr_write;
                        latch   <= '1';
                    when 16         =>  -- FULL
                        counter <= 16;
                        latch   <= '1';
                    when others     =>
                        counter     <=  0;
                        latch       <= '0';
                        ptr_write   <= "0000";
                end case;

            else
                --ptr_write <= ptr_write;
                case counter is
                    when 0      =>
                        counter     <= 0;
                        --ptr_read    <=  ptr_read;
                    when 1 to 15    =>
                        counter     <=  counter - 1;
                        ptr_read    <=  ptr_read + "0001";
                        adrSig      <=  ptr_read;
                    when 16         =>
                        latch       <= '0';
                        ptr_write   <= ptr_write + "0001";
                        ptr_read    <= ptr_read + "0001";
                        counter     <= counter - 1;
                        adrSig      <=  ptr_read;
                    when others     =>
                        counter     <= 0;
                        ptr_read    <= "0000";
                end case;
                --adrSig  <= ptr_read;
            end if;
        end if;
    end process;
```

Figure 3. The main chunk of logic used for implementing task1 of lab2.

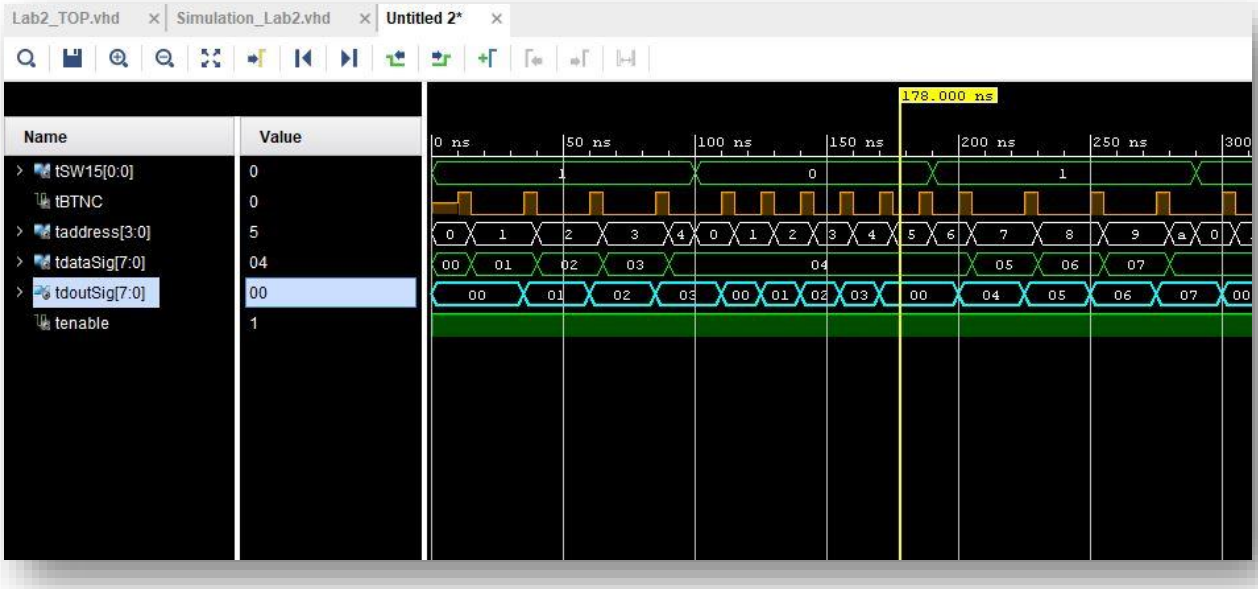


Figure 4. Simulation of the Block RAM FIFO-Controller from task1.

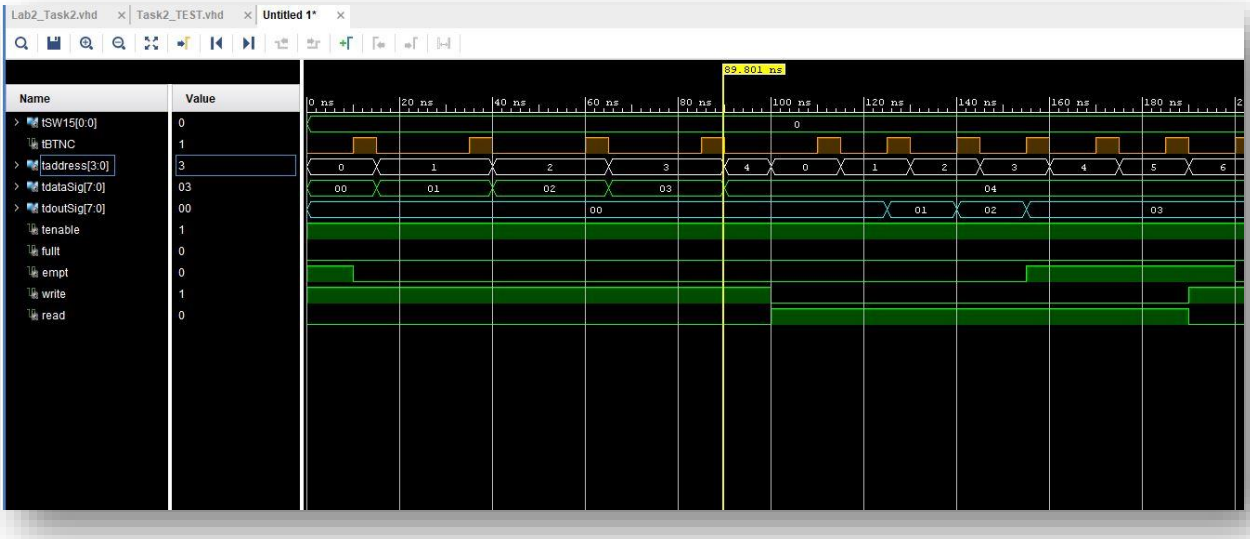


Figure 5. Simulation of the FIFO-controller from task2.