Lab 2: Block Ram and FIFOs in Memory

This lab implements a FIFO using Block RAM (BRAM) on the BASYS 3 Artix-7 Board through VHDL code written in Vivado. The lab procedure is split into two tasks. In Task 1, I created a custom-designed controller for an 8-bit wide, 16-word long FIFO memory design using a single-port BRAM-based IP in Vivado. In Task 2 I utilized a FIFO included on the Artix-7 FPGA to accomplish the same objective as Task 1. For both tasks my code displays the read/write mode and storage status on LEDs, as well as the address and corresponding data on the 7-segment displays.

Mark Lee

ECE 357

October 10, 2019

Part 1 works

off by 1

TA Date: _____

TA Signature: _____ ✓

Introduction

Computer and electrical engineers are expected to be adept at hardware design, especially when it comes to memory and data storage, one of the most central functions of a computer. Therefore, as a part of their education, electrical and computer engineering students work quite a bit with various memory hardware and all of its associated characteristics. This lab explores the memory resources available on the Artix-7 FPGA. Distributed memory is implemented using hardware traditionally used to execute logic (e.g., flip-flops) and is not specifically designed to function as memory. In contrast, Block RAM (BRAM) is a memory array inherent to the FPGA board that is specifically meant to store data. BRAM on 7 series FPGAs includes a capacity of up to 36 Kbits of data, programmable FIFO logic, optional error-correction capabilities, and overall a high level of customization available. One feature of this customization is programmable data width; for this lab we use a BRAM with read/write width 8 and depth 16. We also utilize/implement a synchronous (single-clock) FIFO that provides the storage flags full, empty, almost full, and almost empty.

This lab exercise focuses on BRAM usage in VHDL and on the BASYS3 board. The goal of this lab exercise is twofold. Task 1 involves implementing a FIFO memory module 8 bits wide and 16 words long using a single-port synchronous BRAM module located in the Vivado IP catalog. This customized BRAM is to then be instantiated in VHDL code that tracks the read/write addresses and displays them (along with the data at those addresses) on the 7-segment display on the BASYS3 board. 6 LEDs show read/write mode and the 4 storage flags, while the inputs consist of 8 data switches and 1 read/write mode switch. The Block RAM module is synchronized with a pushbutton on the FPGA.

Task 2 is almost identical to Task 1, but instead of implementing a custom FIFO memory controller, the objective is to use a pre-existing FIFO module on the FPGA. The result of this task should mimic the functionality of Task 1, except the address is not displayed on the 7-segment display.

By accomplishing these tasks, electrical and computer engineering students will obtain a higher degree of competency and versatility in their manipulation of memory resources on FPGAs and similar devices.

Procedure

The Lab 2 writeup seemed both fascinating and challenging from the start. I have a limited amount of experience employing memory hardware, so before beginning to write any code I researched the memory resources available on the FPGA via Youtube and the Xilinx memory reference guide. Once I had a good idea of how BRAM and FIFOs operated, I instantiated a single-port Block RAM into my project from the IP catalog. Then I dredged up my debounced pulse code from ECE 255, because the design specifications required that the BRAM be clocked by the center button on the BASYS3 board. Once I included both these things as components into my main VHDL code, I began formulating the logic to complete Task 1. I first focused on the 7-segment displays. Lines 59-110 of the Task 1 code (located at the end of this report) represent the first portion of code following the instantiation of the components and entity/architecture declarations. I copied this code from Lab 1, as it efficiently alternates the four 7-segment displays with the proper data. It does this by dividing the clock rate (100 MHz) to 10ms per anode, causing the displays to alternate lighting up faster than the human eye can notice. The depicted data on the right two displays is the 8-bit read/write data -- shown in binary coded decimal (BCD) -- and the leftmost display shows the address at which the data is located (in hex). The next process (lines 114-123) updates the current memory address displayed on the 7-segment display, depending on whether the system is in read mode or write mode. The next process (lines 125-147) is the "main" process of the code, incrementing the read/write memory "pointers" whenever the button is pressed. The final notable portion of code is the process from lines 149-194, which features several comparisons to determine which flags to display on the LEDs (read, write, full, empty, almost full, almost empty). After several hours of debugging, I finally secured a functioning result on my board.

Task 2 was thankfully a bit easier than Task 1, or at least less time consuming. Most of my code (which is located below the Task 1 code at the end of the report) is the same or similar to the VHDL code of Task 1. The major difference is that instead of implementing the FIFO logic myself (keeping track of the read/write addresses, updating the full/empty flags, etc.), I utilized the built-in FIFO generator within the Vivado IP library. This nifty module takes care of the status flags for me, and so I don't have to worry about writing as much logic. The first 60

lines of code are relatively the same, except that I instantiate a FIFO generator as a component instead of Block RAM. The 7-segment logic is mostly the same, so the next real difference in code arrives at lines 118-142, where I tweak the LED flags generated by the FIFO module, as they weren't quite right originally. Often, the LEDs would display the empty/almost empty and full/almost full LEDs at the same time, which is of course impossible and incoherent, so I wrote this process to generate the right output. Lines 144-152 update the displayed data on the 7-segment displays, and the final process (lines 154-168) toggles the read/write mode output and the read/write enable for the FIFO whenever the button is pressed.

That's essentially all there is to my VHDL code. Much of the code between the two programs is the same as Task 1 and Task 2 accomplish nearly the same operation on the board. The method of implementation is all that is different, and using these different approaches has increased my proficiency in using the memory resources available on the FPGA.

Results

The simulation for Tasks 1 and 2 is shown below in Figure 1. Since Task 1 and Task 2 are so similar, this simulation -- along with the successful performance of my implementation during lab hours -- should suffice to prove the correctness of my design. The main features of this simulation are the 8 input bits (the 8-bit vector "input_sim") and the full/empty flags (the 6-bit vector "status_sim"), which correspond to the vectors "input" and "status_flags" in my main code. For clarity, the bits in "status_sim" from 5 to 0 are read, write, full, empty, almost full, and almost empty, like in the lab instructions. It can be observed that there are 20 cycles of the pushbutton BTNCC_sim. I chose to do 16 consecutive writes to show the accuracy of the full/empty/almost full/almost empty flags, then do 4 reads to show that it works in the opposite direction. Before the first write, status_sim(2) (empty) is set to '1', then once the second write occurs then status_sim(0) (almost empty) is set to '1'. Once the 16-word memory module is full, it can be seen that the full/almost full flags work in a similarly correct manner. Also note that the enable bit is write ('1') for 16 cycles, and that everything works as it's supposed to.

The simulation code for this lab can be found at the very end of the lab report, following the Task 1 Code and Task 2 Code sections. As previously stated, I fed 16 consecutive writes

followed by 4 consecutive reads into the Task 1 code. For simplicity, I fed the input data over the 16 writes from values 0-15, and over the 4 reads from values 0-3. The simulation data applies to pretty much both tasks, especially after I fixed that problem in Task 2 with multiple LEDs shining at the same time. The cathode values from bits 6 to 0 can be seen at the very bottom of Figure 1, kind of squished in there. I didn't think they were that important to show, but they are also correct for the given input data.

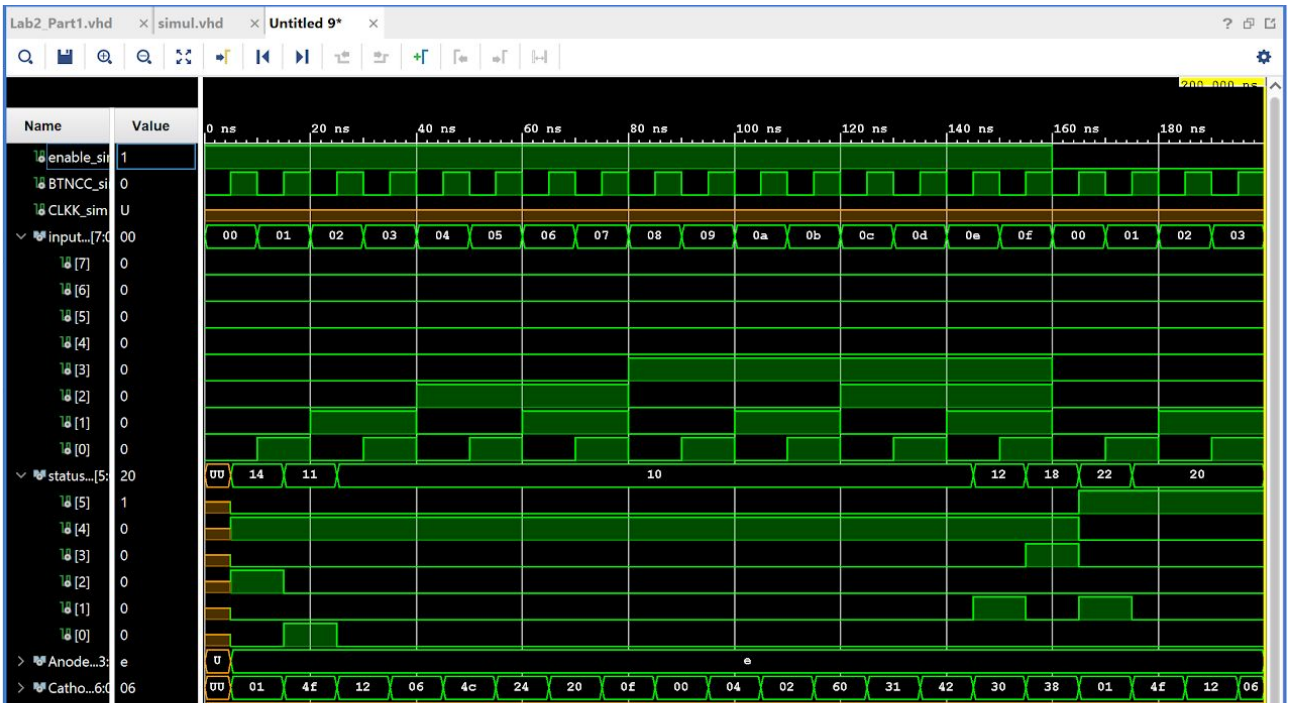

Figure 1. Simulation data for Task 1 and Task 2.

Conclusions

My implementation of Task 1 was perfect, and my implementation of Task 2 was almost perfect. The only hiccup in my design and code for Task 2 is that when switching from a write to a read, the output data for the first read lags behind by one cycle -- however, the address is correct (weird error, right?). I spent a fair bit of time trying to fix it, but I couldn't quite conquer the problem, and ultimately decided to cut my losses and move on from the quandary. This error was small, though, and for the most part my implementation was quite good. My code is clean, fairly efficient, concise, well-formatted, and thoroughly commented. However, I recognize that I

definitely have room for improvement. I could have made my simulation more rigorous by alternating more between reads and writes. Additionally, in my Task 1 code I suspect that the last process statement has a few redundant comparisons when checking the read and write pointers for equality (i.e, I think there are a couple cases where I evaluate the same condition twice, such as when read_sig equals write_sig, read_sig equals write_sig + 1, etc.). More generally, another aspect of VHDL and Vivado that I need to improve upon is writing simulations before uploading my code to the BASYS3 board. I often fail to resist the temptation to code a whole program and debug it via observing what's going wrong on the FPGA. If I break this bad habit, I can save myself a great deal of time and headache, and also take advantage of what simulations provide, which is comprehensive insight into how the code will work. Most significantly, I need to start on these labs earlier, as I have a tendency to procrastinate very badly, and this often results in a painful late-night scramble to finish the lab report.

Potential improvements aside, I also ran into a few issues while completing this lab. First was the conceptual hurdle of understanding what Block RAM and FIFOs were on the FPGA and how they operated. Then, as always when working with VHDL, I encountered a bunch of weird errors during elaboration/synthesis/implementation that I either fixed by overriding it in the .xdc file or by altering my coding approach entirely. I already mentioned my issue with the LEDs in Task 2, so I won't divulge much more on that, any more than the fact that it was a soul-draining experience. Lastly, as I already alluded, I am not great at writing simulations, so I had to spend a long time just figuring out how to include the signals in the simulation file. Writing testbenches is a skill that I have to practice at to progress.

Overall, I'm really pleased with the outcome of this lab. I put a lot of time and effort into getting my code to work, during which I learned much about VHDL and the various mechanisms of Vivado and the BASYS3 board. I feel much more comfortable with VHDL components, writing simulations (although I still need more practice), and using the Vivado IP catalog. Most of all, I have a much more profound understanding of the memory resources available on the FPGA, and how memory operates on the most fundamental level.

Task 1 Code

Project Summary | Schematic | Lab2_Part1.vhd * | CLK_Component.vhd

```
1  -- Mark Lee     ECE 351/357 Lab 2       11 October 2019
2  --
3  -- This VHDL program utilizes a Block RAM memory module from the
4  -- Vivado IP catalog to implement a FIFO structure. I use a debounced
5  -- pulse from a button on the BASYS3 board to clock the reading and
6  -- writing of data.
7
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use ieee.numeric_std.ALL;
11 use ieee.std_logic_unsigned.all;
12
13 entity Lab2_Part1 is
14     Port ( enable, BTNCC, CLKK: in std_logic;
15            input: in std_logic_vector (7 downto 0);
16            status_flags: out std_logic_vector(5 downto 0);
17            Anodes: out std_logic_vector (3 downto 0);
18            Cathodes: out std_logic_vector (6 downto 0)
19     );
20 end Lab2_Part1;
21
22 architecture Behavioral of Lab2_Part1 is
23
24 signal DB_pulse: std_logic;
25 signal enable_sig: std_logic := '1';
26 signal sig: std_logic := '0';
27 signal read_sig: std_logic_vector(3 downto 0) := "0000";
28 signal write_sig: std_logic_vector(3 downto 0) := "0000";
29 signal full_status: std_logic_vector(5 downto 0) := "100100";
30 signal wea_sig: std_logic_vector(0 downto 0);
31 signal address: std_logic_vector(3 downto 0) := "0000";
32 signal output: std_logic_vector(7 downto 0);
33 signal LED_BCD: std_logic_vector(3 downto 0);
34 signal counter: std_logic_vector(19 downto 0);
35
```

```
--Instantiate the debounced pulse functionality and Block RAM component
component DB_CLK
    port(CLK, BTNC: in std_logic;
         DB: out std_logic);
end component;

COMPONENT blk_mem_gen_0
  PORT (
    clka : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;

begin
    --Instantiate the debounced pulse functionality and Block RAM component
    CLK_Component : DB_CLK port map(CLK => CLKK, BTNC => BTNCC, DB => DB_pulse);
    BRAM : blk_mem_gen_0 PORT MAP (clka => DB_pulse, ena => enable_sig, wea => wea_sig,
           addra => address, dina => input, douta => output);

    --Increment counter until counter(18) changes value --> approximately 10ms has passed
    process(CLKK)
    begin
        if(rising_edge(CLKK)) then
            counter <= counter + 1;
        end if;
    end process;

    --When the value for LED_BCD changes, change the output on the display
    process(LED_BCD)
    begin
        case LED_BCD is
            when "0000" => Cathodes <= "0000001"; -- "0"
            when "0001" => Cathodes <= "1001111"; -- "1"
            when "0010" => Cathodes <= "0010010"; -- "2"
            when "0011" => Cathodes <= "0000110"; -- "3"
            when "0100" => Cathodes <= "1001100"; -- "4"
            when "0101" => Cathodes <= "0100100"; -- "5"
            when "0110" => Cathodes <= "0100000"; -- "6"
            when "0111" => Cathodes <= "0001111"; -- "7"
            when "1000" => Cathodes <= "0000000"; -- "8"
            when "1001" => Cathodes <= "0000100"; -- "9"
            when "1010" => Cathodes <= "0000010"; -- "A"
            when "1011" => Cathodes <= "1100000"; -- "b"
            when "1100" => Cathodes <= "0110001"; -- "C"
            when "1101" => Cathodes <= "1000010"; -- "d"
            when "1110" => Cathodes <= "0110000"; -- "E"
            when "1111" => Cathodes <= "0111000"; -- "F"
            when others =>
        end case;
    end process;
```

```
90
91      --When 10ms has passed, alternate displays. The leftmost display
92      --shows the address, and the right two displays show the read/written
93      --data in binary coded decimal (BCD)
94      process(counter(18))
95      begin
96          case counter(19 downto 18) is
97              when "00" =>
98                  Anodes <= "1110";
99                  LED_BCD <= output(3 downto 0);
100             when "01" =>
101                 Anodes <= "1101";
102                 LED_BCD <= output(7 downto 4);
103             when "10" =>
104                 Anodes <= "0111";
105                 LED_BCD <= address;
106             when others =>
107                 Anodes <= "0111";
108                 LED_BCD <= address;
109         end case;
110     end process;
111
112     --Update the address and read/write signals when enable
113     --changes and the read/write signals change
114     process(enable, read_sig, write_sig)
115     begin
116         if(enable = '0') then
117             address <= read_sig;
118             wea_sig <= "0";
119         else
120             address <= write_sig;
121             wea_sig <= "1";
122         end if;
123     end process;
124
```

```
124
125     process is
126     begin
127         wait until falling_edge(DB_pulse);
128
129         if(enable = '0') then
130             sig <= '0';
131
132             --If the memory is not empty on read, read the value
133             --from memory and increment the read pointer
134             if(full_status(3 downto 0) /= "0100") then
135                 read_sig <= read_sig + 1;
136             end if;
137
138         else
139             sig <= '1';
140
141             --If the memory is not full on write, write the value
142             --to memory and increment the write pointer
143             if(full_status(3 downto 0) /= "1000") then
144                 write_sig <= write_sig + 1;
145             end if;
146         end if;
147     end process;
148
```

```
148
149     process(read_sig, write_sig)
150     begin
151
152         --If the read pointer was incremented and is only one behind
153         --the write pointer, then there is only one value in memory
154         --(almost empty)
155         if(read_sig + 1 = write_sig and sig = '0') then
156             full_status <= "100001";
157         --If the read pointer was incremented and is only one ahead
158         --of the write pointer, then memory is almost full
159         elsif(read_sig = write_sig + 1 and sig = '0') then
160             full_status <= "100010";
161         --If the read pointer was incremented and now equals the write
162         --pointer, then the memory is now empty
163         elsif(read_sig = write_sig and sig = '0') then
164             full_status <= "100100";
165         --There is no scenario where a value is read and memory is full.
166         --If none of the other conditions are met, then turn
167         --status_flags(3 downto 0) off
168         elsif(sig = '0') then
169             full_status <= "100000";
170         end if;
171
```

```
171
172         --If the write pointer was incremented and is only one ahead
173         --of the read pointer, then there is only one value in memory
174         --(almost empty)
175         if(write_sig = read_sig + 1 and sig = '1') then
176            full_status <= "010001";
177
178         ---If the write pointer was incremented and is only one behind
179         --the read pointer, then memory is almost full
180         elsif(write_sig + 1 = read_sig and sig = '1') then
181             full_status <= "010010";
182
183         --If the write pointer was incremented and now equals the read
184         --pointer, then the memory is now full
185         elsif(write_sig = read_sig and sig = '1') then
186             full_status <= "011000";
187
188         --There is no scenario where a value is written and memory is empty.
189         --If none of the other conditions are met, then turn
190         --status_flags(3 downto 0) off
191         elsif(sig = '1') then
192             full_status <= "010000";
193         end if;
194     end process;
195
196     --Set LED flags
197     process(full_status)
198     begin
199         status_flags <= full_status;
200     end process;
201
202 end Behavioral;
203
```

Task 2 Code

Lab2_Part2.vhd *

C:/Users/Alexander/Desktop/School_Assignments/ECE_357_Labs/Lab2_Part2/Lab2_Part2.srcs/sources_1/new/Lab2_Part2.vhd

```
1  -- Mark Lee     ECE 351/357 Lab 2       11 October 2019
2  --
3  -- This VHDL program is similar to Task 1, but now uses a built-in FIFO
4  -- generator customized from the Vivado IP catalog instead of my own
5  -- implementation. I again use a debounced pulse from a button on the
6  -- BASYS3 board to clock the reading and writing of data.
7
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use ieee.numeric_std.ALL;
11 use ieee.std_logic_unsigned.all;
12
13 entity Lab2_Part2 is
14     Port( enable, BTNCC, CLKK: in std_logic;
15           input: in std_logic_vector (7 downto 0);
16           status_flags: out std_logic_vector(5 downto 0);
17           Anodes: out std_logic_vector (3 downto 0);
18           Cathodes: out std_logic_vector (6 downto 0)
19     );
20 end Lab2_Part2;
21
```

```
21
22 architecture Behavioral of Lab2_Part2 is
23
24 signal DB_pulse: std_logic;
25 signal renable_sig: std_logic := '1';
26 signal sig: std_logic := '0';
27 signal read_sig: std_logic_vector(3 downto 0) := "0000";
28 signal write_sig: std_logic_vector(3 downto 0) := "0000";
29 signal full_status: std_logic_vector(5 downto 0);
30 signal wea_sig: std_logic;
31 signal address: std_logic_vector(3 downto 0) := "0000";
32 signal output: std_logic_vector(7 downto 0);
33 signal LED_BCD: std_logic_vector(3 downto 0);
34 signal counter: std_logic_vector(19 downto 0);
35 signal data: std_logic_vector(7 downto 0);
36
37 --Again, I use a debounced pulse from ECE 255 and the FIFO component
38 component DB_CLK
39     port(CLK, BTNC: in std_logic;
40          DB: out std_logic);
41 end component;
42
```

```
43 COMPONENT fifo_generator_0
44   PORT (
45     clk : IN STD_LOGIC;
46     din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
47     wr_en : IN STD_LOGIC;
48     rd_en : IN STD_LOGIC;
49     dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
50     full : OUT STD_LOGIC;
51     almost_full : OUT STD_LOGIC;
52     empty : OUT STD_LOGIC;
53     almost_empty : OUT STD_LOGIC
54   );
55 END COMPONENT;
56
57 begin
58     CLK_Component : DB_CLK port map(CLK => CLKK, BTNC => BTNCC, DB => DB_pulse);
59     FIFO : fifo_generator_0 PORT MAP (clk => DB_pulse, din => input, wr_en => wea_sig,
60           rd_en => renable_sig, dout => output, full => full_status(3),
61           almost_full => full_status(1), empty => full_status(2), almost_empty => full_status(0));
62
```

```
62
63      --Increment counter until counter(18) changes value --> approximately 10ms has passed
64      process(CLKK)
65      begin
66          if(rising_edge(CLKK)) then
67              counter <= counter + 1;
68          end if;
69      end process;
70
71      --When the value for LED_BCD changes, change the output on the display
72      process(LED_BCD)
73      begin
74          case LED_BCD is
75              when "0000" => Cathodes <= "0000001"; -- "0"
76              when "0001" => Cathodes <= "1001111"; -- "1"
77              when "0010" => Cathodes <= "0010010"; -- "2"
78              when "0011" => Cathodes <= "0000110"; -- "3"
79              when "0100" => Cathodes <= "1001100"; -- "4"
80              when "0101" => Cathodes <= "0100100"; -- "5"
81              when "0110" => Cathodes <= "0100000"; -- "6"
82              when "0111" => Cathodes <= "0001111"; -- "7"
83              when "1000" => Cathodes <= "0000000"; -- "8"
84              when "1001" => Cathodes <= "0000100"; -- "9"
85              when "1010" => Cathodes <= "0000010"; -- "A"
86              when "1011" => Cathodes <= "1100000"; -- "b"
87              when "1100" => Cathodes <= "0110001"; -- "C"
88              when "1101" => Cathodes <= "1000010"; -- "d"
89              when "1110" => Cathodes <= "0110000"; -- "E"
90              when "1111" => Cathodes <= "0111000"; -- "F"
91              when others =>
92          end case;
93      end process;
94
```

```
95      --When 10ms has passed, alternate displays. The right
96      --two displays show the read/written data in binary
97      --coded decimal (BCD)
98      process(counter(18))
99      begin
100       --  if(enable = '1') then
101           case counter(19 downto 18) is
102               when "00" =>
103                   Anodes <= "1110";
104                   LED_BCD <= data(3 downto 0);
105               when "01" =>
106                   Anodes <= "1101";
107                   LED_BCD <= data(7 downto 4);
108               when "10" =>
109                   Anodes <= "1110";
110                   LED_BCD <= data(3 downto 0);
111               when "11" =>
112                   Anodes <= "1101";
113                   LED_BCD <= data(7 downto 4);
114               when others=>
115           end case;
116     end process;
117
```

```
    --The status flags (full, empty, almost full, almost empty) from the
    --built-in FIFO were weird --> they often displayed multiple flags simultaneously.
    --This process fixes that error.
    process(full_status)
    begin
        if(enable = '0') then
            if(full_status(3 downto 0) = "0101") then
                status_flags <= "100100";
            elsif(full_status(3 downto 0) = "1010") then
                status_flags <= "101000";
            else
                status_flags(5 downto 4) <= "10";
                status_flags(3 downto 0) <= full_status(3 downto 0);
            end if;
        else
            if(full_status(3 downto 0) = "0101") then
                status_flags <= "010100";
            elsif(full_status(3 downto 0) = "1010") then
                status_flags <= "011000";
            else
                status_flags(5 downto 4) <= "01";
                status_flags(3 downto 0) <= full_status(3 downto 0);
            end if;
        end if;
    end process;
```

```

    --Update the displaed data on the 7-segment displays
    process(enable, renable_sig, wea_sig)
    begin
        if(enable = '0') then
            data <= output;
        else
            data <= input;
        end if;
    end process;

    --When the button is pressed, update the read/write mode and
    --the read/write enable signals
    process is
    begin
        wait until falling_edge(DB_pulse);
        if(enable = '0') then
            full_status(5 downto 4) <= "10";
            renable_sig <= '1';
            wea_sig <= '0';
        else
            full_status(5 downto 4) <= "01";
            renable_sig <= '0';
            wea_sig <= '1';
        end if;
    end process;
end Behavioral;
```

Simulation Code for Tasks 1 and 2

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity simul is
5  --  Port ( );
6  end simul;
7
8  architecture Behavioral of simul is
9      signal enable_sim, BTNCC_sim, CLKK_sim: std_logic;
10     signal input_sim, output_sim: std_logic_vector (7 downto 0);
11     signal status_sim: std_logic_vector(5 downto 0) := "100100";
12     signal Anodes_sim: std_logic_vector (3 downto 0);
13     signal Cathodes_sim: std_logic_vector (6 downto 0);
14     signal read_sim, write_sim, addr_sim: std_logic_vector(3 downto 0);
15     signal pulse_sim: std_logic;
16 begin
17     Part1: entity work.Lab2_Part1(Behavioral) port map(enable => enable_sim,
18     BTNCC => BTNCC_sim, CLKK => CLKK_sim, input => input_sim,
19     status_flags => status_sim, Anodes => Anodes_sim, Cathodes => Cathodes_sim);
20
21     output_sim <= << signal Part1.output : std_logic_vector(7 downto 0) >> ;
22     read_sim <= << signal Part1.read_sig:std_logic_vector(3 downto 0) >> ;
23     write_sim <= << signal Part1.write_sig:std_logic_vector(3 downto 0) >> ;
24     addr_sim <= << signal Part1.address:std_logic_vector(3 downto 0) >> ;
25     pulse_sim <= << signal Part1.DB_pulse:std_logic >> ;
26
27
```

```
26
27
28      process
29      begin
30          BTNCC_sim <= '0';
31          enable_sim <= '1';
32
33          input_sim <= "00000000";
34          wait for 5ns;
35          BTNCC_sim <= '1';
36          wait for 5ns;
37          BTNCC_sim <= '0';
38
39          input_sim <= "00000001";
40          wait for 5ns;
41          BTNCC_sim <= '1';
42          wait for 5ns;
43          BTNCC_sim <= '0';
44
45          input_sim <= "00000010";
46          wait for 5ns;
47          BTNCC_sim <= '1';
48          wait for 5ns;
49          BTNCC_sim <= '0';
50
51          input_sim <= "00000011";
52          wait for 5ns;
53          BTNCC_sim <= '1';
54          wait for 5ns;
55          BTNCC_sim <= '0';
56
```

```
56
57            input_sim <= "00000100";
58            wait for 5ns;
59            BTNCC_sim <= '1';
60            wait for 5ns;
61            BTNCC_sim <= '0';
62
63            input_sim <= "00000101";
64            wait for 5ns;
65            BTNCC_sim <= '1';
66            wait for 5ns;
67            BTNCC_sim <= '0';
68
69            input_sim <= "00000110";
70            wait for 5ns;
71            BTNCC_sim <= '1';
72            wait for 5ns;
73            BTNCC_sim <= '0';
74
75            input_sim <= "00000111";
76            wait for 5ns;
77            BTNCC_sim <= '1';
78            wait for 5ns;
79            BTNCC_sim <= '0';
80
```

```
80
81          input_sim <= "00001000";
82          wait for 5ns;
83          BTNCC_sim <= '1';
84          wait for 5ns;
85          BTNCC_sim <= '0';
86
87          input_sim <= "00001001";
88          wait for 5ns;
89          BTNCC_sim <= '1';
90          wait for 5ns;
91          BTNCC_sim <= '0';
92
93          input_sim <= "00001010";
94          wait for 5ns;
95          BTNCC_sim <= '1';
96          wait for 5ns;
97          BTNCC_sim <= '0';
98
99          input_sim <= "00001011";
100         wait for 5ns;
101         BTNCC_sim <= '1';
102         wait for 5ns;
103         BTNCC_sim <= '0';
104
```

```
104
105         input_sim <= "00001100";
106         wait for 5ns;
107         BTNCC_sim <= '1';
108         wait for 5ns;
109         BTNCC_sim <= '0';
110
111         input_sim <= "00001101";
112         wait for 5ns;
113         BTNCC_sim <= '1';
114         wait for 5ns;
115         BTNCC_sim <= '0';
116
117         input_sim <= "00001110";
118         wait for 5ns;
119         BTNCC_sim <= '1';
120         wait for 5ns;
121         BTNCC_sim <= '0';
122
123         input_sim <= "00001111";
124         wait for 5ns;
125         BTNCC_sim <= '1';
126         wait for 5ns;
127         BTNCC_sim <= '0';
128
129
130         enable_sim <= '0';
131
```

```
132
133         input_sim <= "00000000";
134         wait for 5ns;
135         BTNCC_sim <= '1';
136         wait for 5ns;
137         BTNCC_sim <= '0';
138
139         input_sim <= "00000001";
140         wait for 5ns;
141         BTNCC_sim <= '1';
142         wait for 5ns;
143         BTNCC_sim <= '0';
144
145         input_sim <= "00000010";
146         wait for 5ns;
147         BTNCC_sim <= '1';
148         wait for 5ns;
149         BTNCC_sim <= '0';
150
151         input_sim <= "00000011";
152         wait for 5ns;
153         BTNCC_sim <= '1';
154         wait for 5ns;
155         BTNCC_sim <= '0';
156
157     end process;
158 end Behavioral;
159
```