

**Introduction**

In this lab, we are assigned to build the different storage building blocks we learned in class. The ones we needed to build were the 32x32 register file, and first-in-first-out queue, abbreviated as FIFO. Our group’s task was to design the individual components, run test bench simulations for it, and implement the FIFO onto our FPGA board.

**Design Methodology**

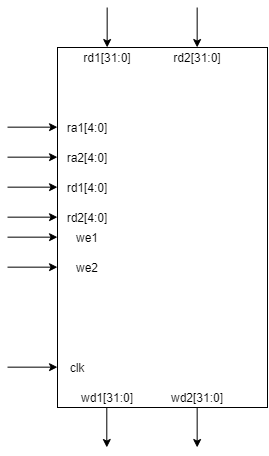
To make the 32x32 register file, first we needed to declare a 2D-array. One dimension for the width of the data and one dimension for the amount of data to be stored. Then, we needed to create reg for reading the address location, and the data value to be capable of reading and writing. Since our register file required us to have 2 inputs and 2 outputs, we created 2 of each reg. The register is able to read if given a valid address and have the write enable signal low, giving us an output at rd[31:0]. The register is able to write if given a valid address, valid data, and have the write enable signal high, which will then write to the address specified. Since this device has two inputs and outputs, there can sometimes be conflict. For example, if both we1 and we2 are high (meaning write is enabled), then our program will only write on one of them since we can only write one at a time. In addition, if both ra1 and ra2 are the same with different data, we can not write at it both at the same time because we can only perform one action at a time. Below are the list of signals and their function, module names and definitions, as well as the block diagram for the 32x32 register file.

|  |  |
| --- | --- |
| **Signal Names** | **Definitions** |
| rd1[31:0] | Read data port 1 |
| ra1[4:0] | Read access for read data port 1 |
| rd2[31:0] | Read data port 2 |
| ra2[4:0] | Read access for read data port 2 |
| wd1[31:0] | Write data port 1 |
| wa1[4:0] | Write address for write data port 1 |
| we1 | Write enable for write data port 1 |
| wd2[:31:0] | Write data port 2 |
| wa2[4:0] | Write address for write data port 2 |
| we2 | Write enable for write data port 2 |
| clk | Clock signal |

*Table 1: Signal names used for 32x32 register file*

|  |  |
| --- | --- |
| **Module name** | **Definition** |
| regfile2.v | Implements 32x32 register file |
| refile2TB.v | Simulates test bench for regfile2 |

*Table 2: Modules used for 32x32 register file*



*Figure 1:Block diagram for 32x32 register file*

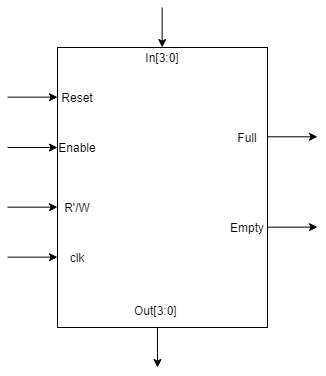
For the first in first out queue (FFIO), we were assigned to make the FPGA board store up to 8, 4 bit data. This time, to store data we did not need any addressing because we placed it in a queue, where it organizes where the data is stored and retrieved. Instead, we had an enable pin that ensures that the FIFO is on, and a R/W pin to choose whether to read or write, with 1 being write and 0 being read. The reset pins would remove all the values in the FFIO, and the clk signal was generated from a button using the debouncer.v file we made in the previous lab since using a normal 5/10GHZ clock would be too fast for us to read or write any values. Our FIFO will have there outputs. When the FFIO is full, we will have the ‘full’ flag on, and when the FIFO is empty, we will have the ‘empty’ flag on which will be shown via LED on the FPGA board. And lastly, we will have out[3:0] read from the bottom of our queue via 7-segment LED. The truth table for the FIFO, modules used, and the block diagram will be shown below.

|  |  |  |
| --- | --- | --- |
| **FIFO Control**  **R’/W EN** | | **Operation** |
| x | 0 | nop |
| 0 | 1 | read |
| 1 | 1 | write |

*Table 3: Truth table for FIFO control*

|  |  |
| --- | --- |
| **Modules** | **Definition** |
| FIFO.v | Implements first in first out queue |
| clk\_gen | Generates clock signal |
| button\_debouncer.v | Generates clock signal from button press on FPGA board |
| led\_display.v | Generates decimal number on 7-segment LED on FPGA board |
| fifo\_tb.v | Simulates test bench for FIFO.v |
| FIFO\_fpga.v | Implements FIFO.v on FPGA board |

*Table 4: Modules used for FIFO control*

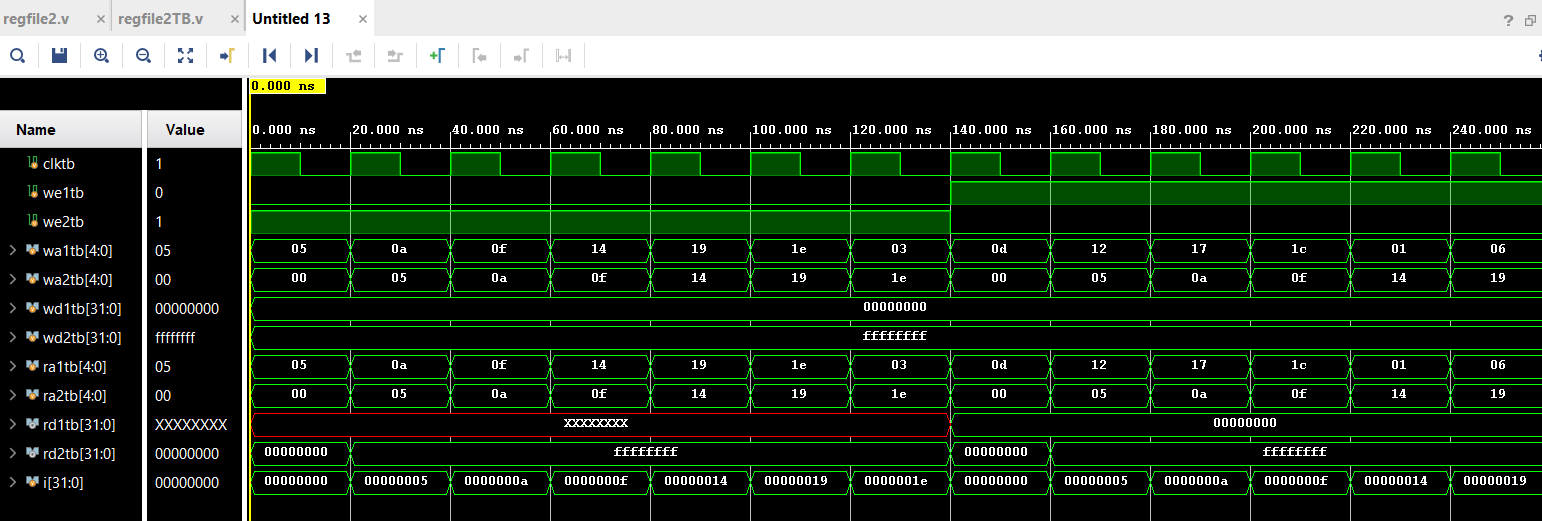


*Figure 2: Block diagram for FIFO*

**Simulation Results**

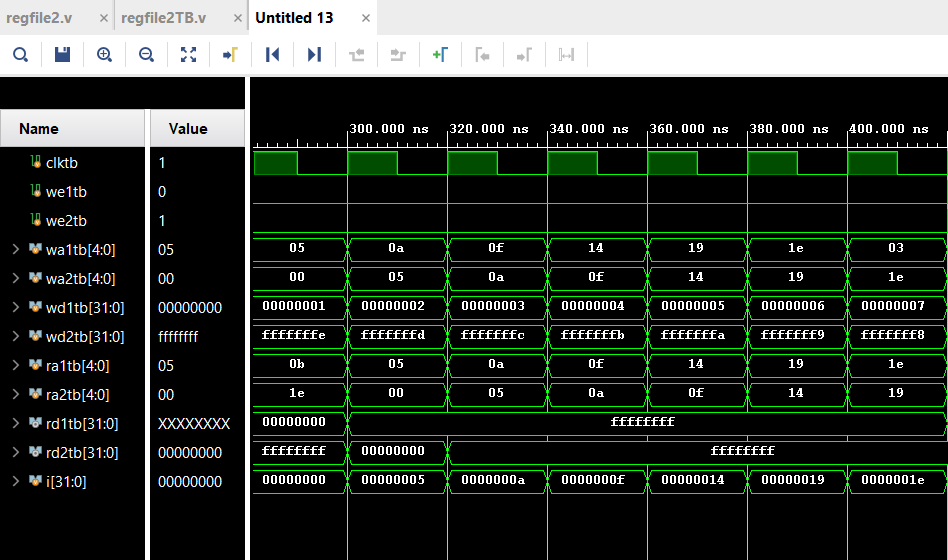
Part 1: Register file

This screenshot shows a simple case where we1 is on for the first half and we2 is on for the last half. The data is written when clk is high, and when the appropriate write enable signal is high. Then, after one clock cycle, we can read what the value was saved in that location.



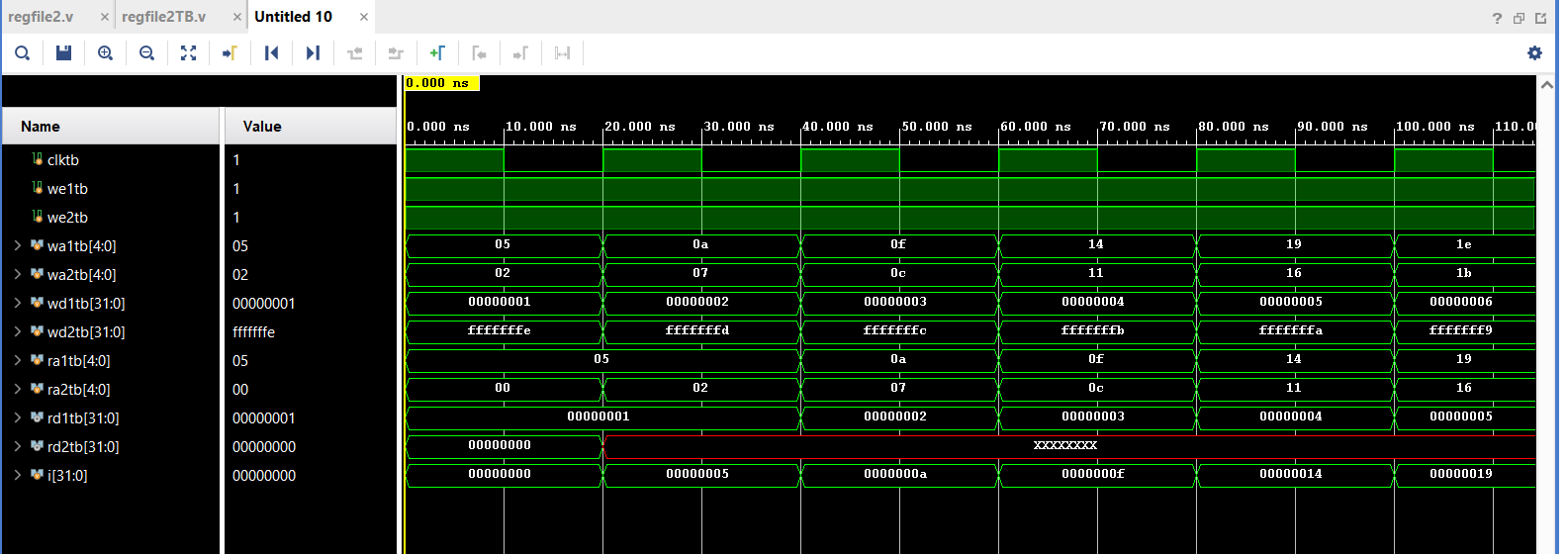
*Figure 3: Simple one enable on, different address*

In this screenshot, we made it so both write enable is low and is a continuation of figure 3. When both are low, we see that the data is not changing looking at the rd1tb and rd2tb. This is because we are not selecting either ports to write, therefore not updating the data inside.



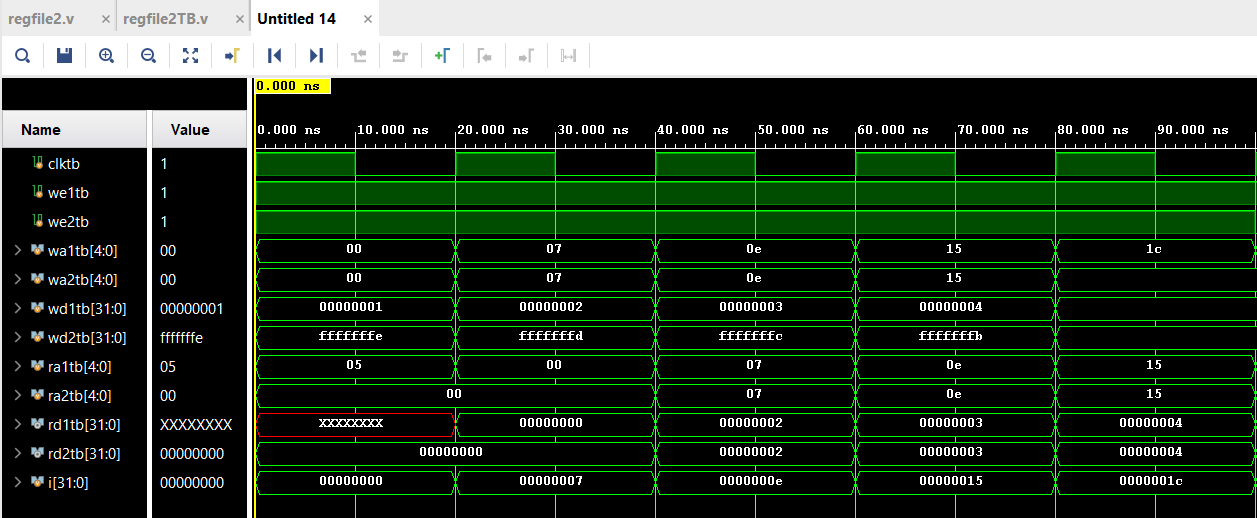
*Figure 4: Both write enable low*

This screenshot shows when both write enable signals are high. When both are high it is supposed to write to both addresses specified by wa1tb and wa2tb. However, since we can only write on one address at a time, we can only choose one of them. In this simulation the first port was chosen, therefore was kept updating and port 2 was not chosen, therefore nothing was written on it.



*Figure 5: Both write enable high*

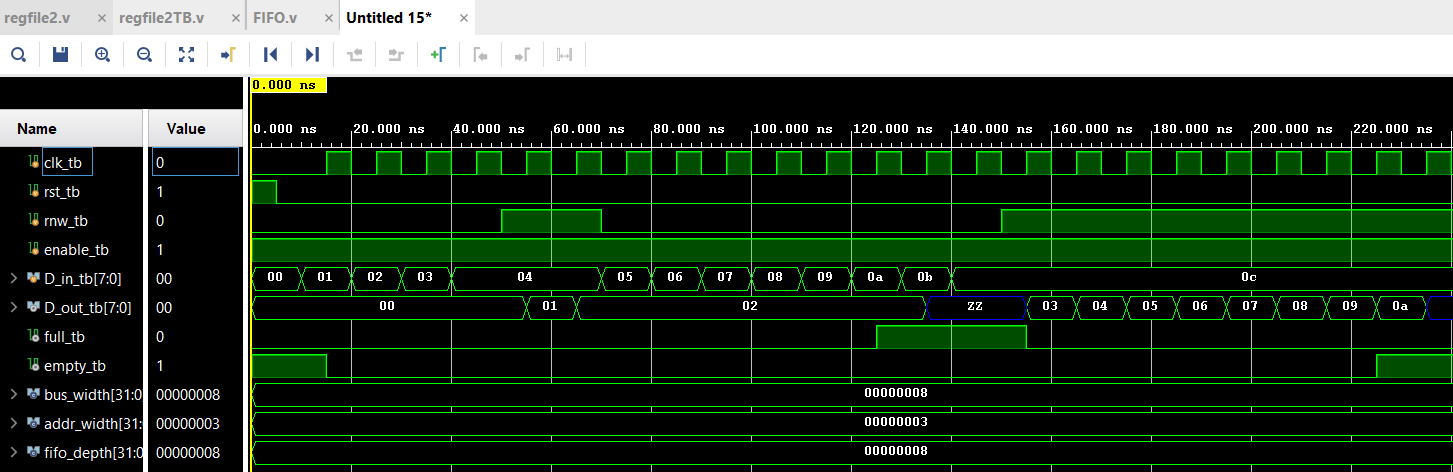
Similar to the last screenshot, this one also has both write enable signals on high. However, in this case, we have both addresses pointed to the same place with different data values. Like previously said, since the register file can only write one data at a time, it can not write both data from port 1 and port 2. Therefore, only one data is chosen, and the screenshot shows data 1 being saved into the register.



*Figure 6: Both write enable high with same address*

Part 2

The screenshot below is from the test bench simulation from the FIFO portion of this lab. At first, we see the empty flag set to 0 since we do not have any value saved in the queue. However, after the first cycle with the write signal on, the empty flag is reset back to 0. We keep saving some values until rnw\_tb is set to high, which then we read the values out of the queue. First we stored the value 1,2,3,4 but after the two read cycles, we read the value 1, and we are reading the value 2 when the write cycle starts again. Our queue current looks like this: 3,4. Then we keep writing some values and adding it into our queue. After storing the value a, our queue is full since it has 8 stored values, therefore our full flag is set to 1 and we are unable to write some values until we read them out. Then, we read until we emptied our queue, which at last showed us the empty flag being set to 1 and d\_out also being set to ‘zz’.



*Figure 7:FIFO testing simulation*

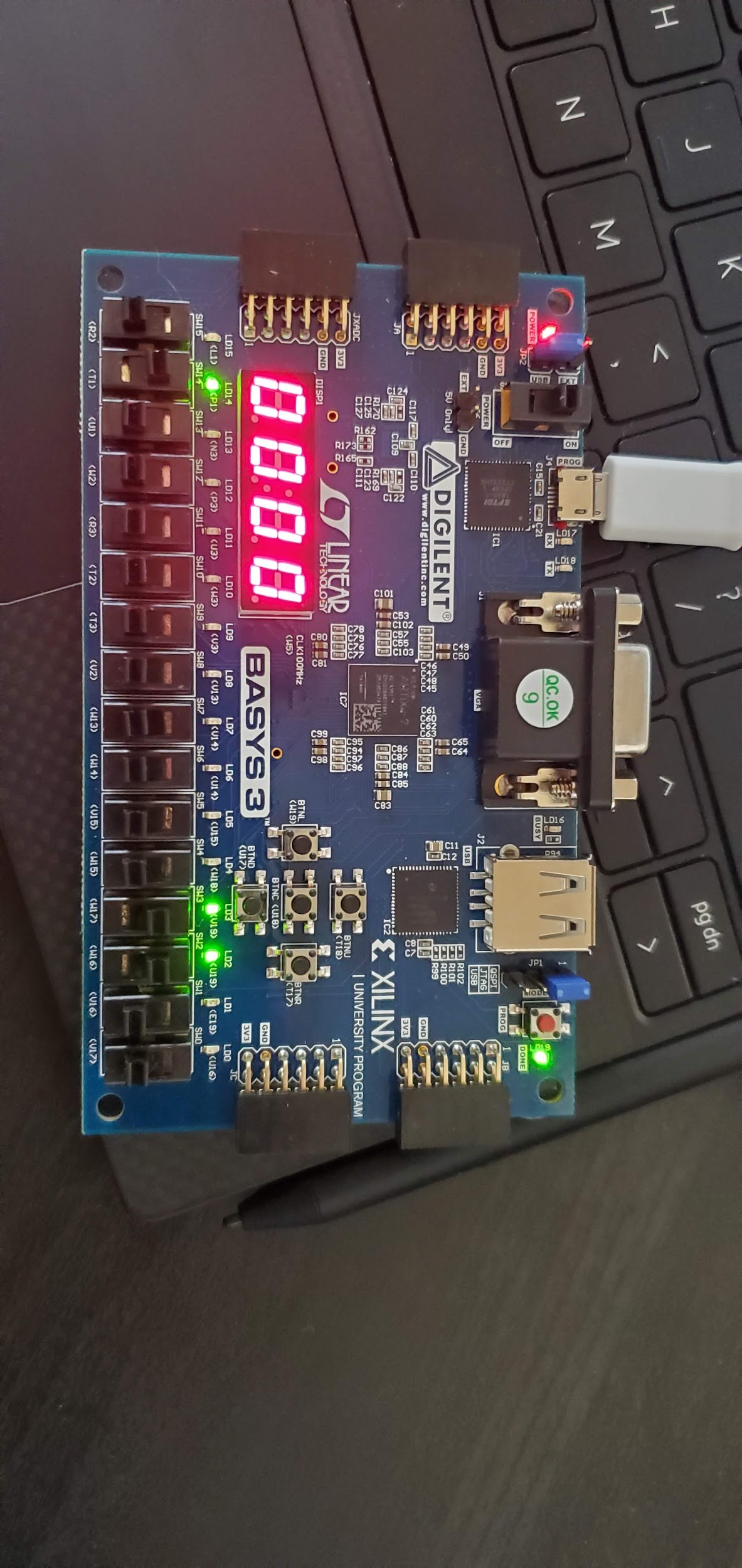
**FPGA Validation**

Our board has 4 input switches on the right for the value to be stored in the FIFO, and read/write signal on the far left, and enable pin on the 2nd to left switch. Then we have the empty pin on the 5th LED from the left and the full pin on the 4th LED from the left.

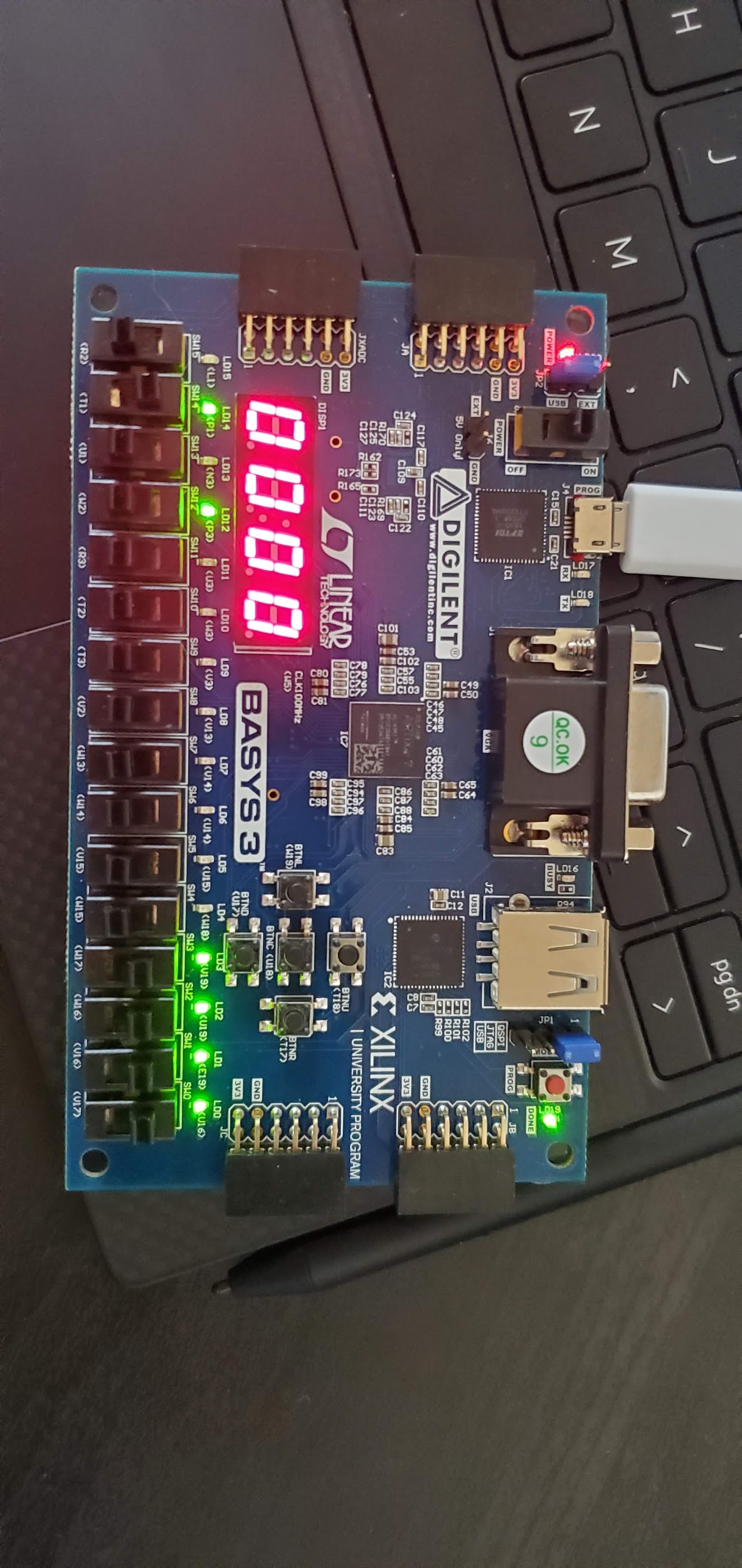
First, we have enable on and the empty set to high. Then, we started writing some numbers into the FIFO, so the empty flag was reset back to 0. After entering 8 numbers, the full flag is set. Then we start reading these values and after reading 8 values, we have the empty flag set to 1 again. This sequence is shown in the pictures below.



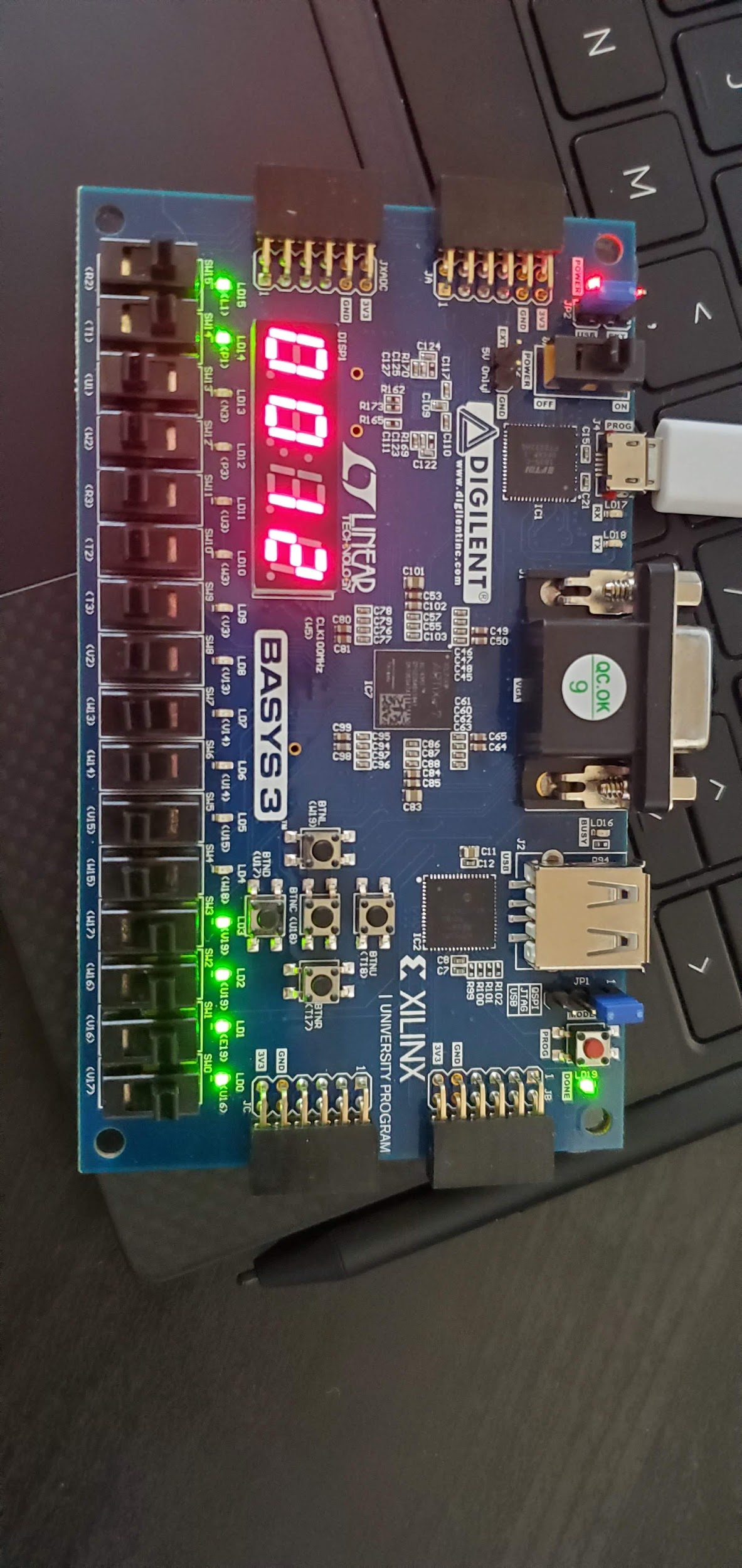
*Figure 8: Empty flag on*



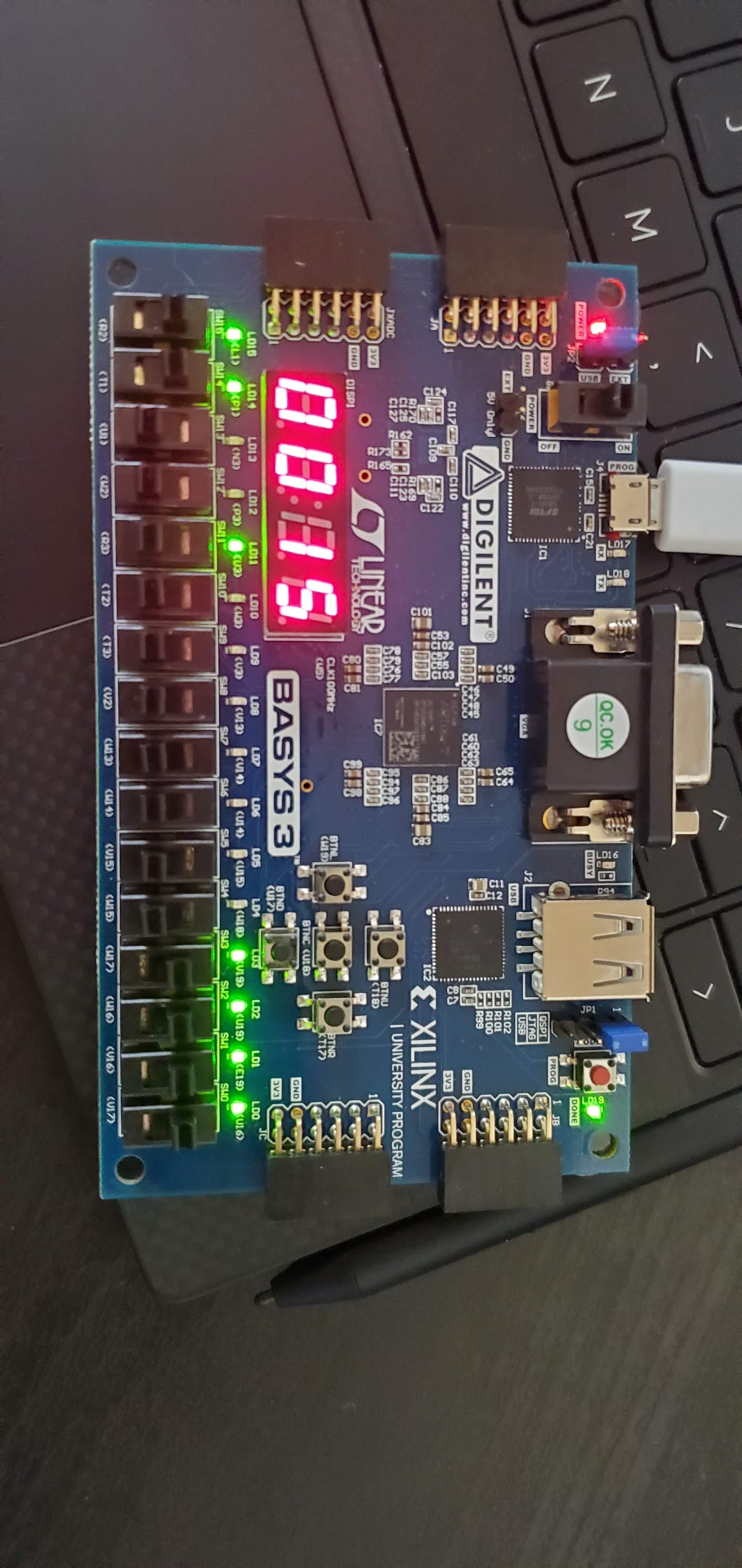
*Figure 9: Empty flag set to 0 after writing values*



*Figure 10:Full flag set after storing max values*



*Figure 11: Full flag set to 0 after reading*



*Figure 11:Empty flag set after reading all values*

**Conclusion**

Overall, our team was able to successfully complete both portions of this lab. The goal of this lab was to learn about storage building blocks, and we learned about these items with the two main tasks of creating a 32x32 register file, as well as a first in first out queue. Both of these tasks required us to learn about assigning data to a certain address within the files.

For the first task, we created a file called regfile2.v which can be seen in the appendix which uses a 2D array to be able to store data. Within this reg, there are actually 2 separate locations where data can be stored, and there are 2 inputs and 2 outputs. The file has multiple enable pins, such as pins to which control whether the write is enabled for each of the register inputs, address write inputs, address write data, read data output, and read address. Because of the way we designed this file, the data will only be input to the correct location on each clock rising edge. To make sure that we do not have any conflict, we designed our file so that only a single address can be written to at the same time. We did this by using an if statement, so that we1 will have a higher priority than we2. We were successful in completing this task.

The next task was to create a FIFO queue and implement this onto the FPGA board, which we were also able to successfully complete. The logic of this is simple. The data that is input into the queue first, will be the first data to come out. After properly setting up switches on the FPGA board to control whether we are reading or writing, and after setting up a button to be our debounced clock, we were able to successfully input data one clock cycle at a time, and then read the data one clock cycle at a time while ensuring that our flag LEDs for the empty and full flag were still working.

**Appendix**

|  |
| --- |
| regfile2.v |
| `timescale 1ns / 1ps  module regfile2(  input clk,  input we1, we2,  input [4:0] wa1, wa2,  input [31:0] wd1, wd2,  input [4:0] ra1, ra2,  output [31:0] rd1, rd2  );    reg[31:0] rf[31:0];    always @(posedge clk) begin    if(we1) rf[wa1] = wd1;  else if(we2) rf[wa2] = wd2;  end    assign rd1 = (ra1 != 0) ? rf[ra1] : 0; //read1  assign rd2 = (ra2 != 0) ? rf[ra2] : 0; //read2  //assign rd1 = rf[ra1];  //assign rd2 = rf[ra2];      endmodule |

|  |
| --- |
| regfile2TB.v |
| `timescale 1ns / 1ps  module regfile2TB;  reg clktb;  reg we1tb, we2tb;  reg [4:0] wa1tb, wa2tb;  reg [31:0] wd1tb, wd2tb;  reg [4:0] ra1tb, ra2tb;  wire [31:0] rd1tb, rd2tb;  regfile2 DUT(  clktb,  we1tb, we2tb, wa1tb, wa2tb,  wd1tb, wd2tb, ra1tb, ra2tb,  rd1tb, rd2tb  );    integer i = 0;  // integer exp\_rd1tb;  // integer exp\_rd2tb;  // integer errorCount = 0;    task tick;  begin  clktb = 1;  #10;  clktb = 0;  #10;  end  endtask      initial begin  we1tb = 0; //First, testing for when write enables are opposites  we2tb = 1;  wd1tb = 32'h00000000; //initial input data  wd2tb = 32'hffffffff;  ra1tb = 5'b00101;  ra2tb = 0;    for(i = 0; i< 5'b11111; i = i + 5'b00101) begin  wa1tb = i + 5'b00101;  wa2tb = i;    ra1tb = i + 5'b00101;  ra2tb = i;  tick;  end;    we1tb = 1; //Second, flipping the write enables  we2tb = 0;  for(i = 0; i< 5'b11111; i = i + 5'b00101) begin  wa1tb = i + 5'b01101;  wa2tb = i;    ra1tb = i + 5'b01101;  ra2tb = i;  tick;  end;  we1tb = 1; //Third, both write enables are high  we2tb = 1;  wd1tb = 32'h00000000; //initial input data  wd2tb = 32'hffffffff;  for(i = 0; i< 5'b11111; i = i + 5'b00101) begin  wd1tb=wd1tb+1;  wd2tb=wd2tb-1;  wa1tb = i + 5'b00101;  wa2tb = i+2;  tick;  ra1tb = i + 5'b00101;  ra2tb = i+2;  end;    we1tb = 0; //Fourth, both write enables are low  we2tb = 0;  wd1tb = 32'h00000000; //initial input data  wd2tb = 32'hffffffff;        for(i = 0; i< 5'b11111; i = i + 5'b00101) begin  wd1tb=wd1tb+1;  wd2tb=wd2tb-1;  wa1tb = i + 5'b00101;  wa2tb = i;  tick;  ra1tb = i + 5'b00101;  ra2tb = i;    end;    we1tb = 1; //Fifth, both read addresses and a write address are the same  we2tb = 1;    for(i = 0; i < 5'b11111; i = i + 5'b00111) begin  wd1tb=wd1tb+1;  wd2tb=wd2tb-1;  wa1tb = i;  wa2tb = i;  tick;  ra1tb = i;  ra2tb = i;  end  we2tb = 0;  we1tb = 1;  for(i = 5'b00000; i<11111; i = i+ 5'b00001)begin  wa1tb = i;  ra1tb = i;    tick;  end  end  endmodule |

|  |
| --- |
| FIFO.v |
| `timescale 1ns / 1ps  module FIFO (D\_in, D\_out, empty, full, clk, rst, rnw, enable);  parameter bus\_width = 8;  parameter addr\_width = 3;  parameter fifo\_depth = 8; // 2^addr\_width = fifo\_depth  input clk; // clock  input rst; // Asynchronous reset  input rnw; // read (1) or write (0) control  input enable; // enables the FIFO  input [bus\_width-1:0] D\_in; // Data input to the FIFO  output reg [bus\_width-1:0] D\_out; // Data output from the FIFO  output reg full; // Asserted when the FIFO is full  output reg empty; // Asserted when the FIFO is empty  //reg [bus\_width-1:0] D\_out;  //reg full, empty;  reg [addr\_width:0] r\_ptr, w\_ptr; // read and write pointers  // Pay attention on their size!!!  reg [bus\_width-1:0] mem [fifo\_depth -1:0]; // memory used by the FIFO  always @ (posedge clk, posedge rst)  begin  if (rst)  begin r\_ptr = 0; w\_ptr = 0; D\_out = 0; end  else if (!enable)  begin D\_out ='bz; end  else if (rnw && !empty) // start reading  begin D\_out = mem[r\_ptr[addr\_width-1:0]]; r\_ptr = r\_ptr + 1; end  else if (!rnw && !full) // start writing  begin mem[w\_ptr[addr\_width-1:0]] = D\_in; w\_ptr = w\_ptr + 1; end  else  begin D\_out ='bz; end  end  always @ (r\_ptr, w\_ptr) // update the flags based on the read/write pointers  begin  if (r\_ptr == w\_ptr)  begin empty = 1; full = 0; end  else if (r\_ptr[addr\_width -1:0] == w\_ptr[addr\_width-1:0])  begin empty = 0; full = 1; end  else  begin empty = 0; full = 0; end  end  endmodule |

|  |
| --- |
| fifo\_tb.v |
| `timescale 1ns / 1ps  module fifo\_tb;  parameter bus\_width = 8;  parameter addr\_width = 3;  parameter fifo\_depth = 8;  reg clk\_tb;  reg rst\_tb;  reg rnw\_tb;  reg enable\_tb;  reg [bus\_width-1:0] D\_in\_tb;  wire [bus\_width-1:0] D\_out\_tb;  wire full\_tb, empty\_tb;  FIFO DUT(  .D\_in(D\_in\_tb), // Data input to the FIFO  .D\_out(D\_out\_tb), // Data output from the FIFO  .empty(empty\_tb), // Asserted when the FIFO is empty  .full(full\_tb), // Asserted when the FIFO is full  .clk(clk\_tb), // clock  .rst(rst\_tb), // Asynchronous reset  .rnw(rnw\_tb), // read (1) or write (0) control  .enable(enable\_tb) // enables the FIFO  );  integer i;  integer errorCounter;  task tick;  begin  clk\_tb = 0;  #5;  clk\_tb = 1;  #5;  end  endtask  task reset;  begin  rst\_tb = 1;  #5;  rst\_tb = 0;  #5;  end  endtask  task checkFlags; //task to check flags and output to console  begin  if(full\_tb) begin  $display("FIFO is full");  end  else if(empty\_tb) begin  $display("FIFO is empty");  end  end  endtask  task checkErrors;  begin    end  endtask    initial begin  i = 0;  enable\_tb = 1;  D\_in\_tb = 0;  clk\_tb = 0;  rnw\_tb = 0;  reset; //reset the fifo to initialize the pointers    //6 tests  //1. Empty  //2. Partially Empty  //3. Partially Full  //4. Full  //5. Adding to a full queue  //6. Deleting all items from queue    /\*1.\*/ checkFlags; // FIFO is EMPTY  $display("1. Empty Queue");  //2. FIFO is partially empty  $display("2. Partially empty Queue");  rnw\_tb = 0; //write three times  for(i = 0; i<3'b100; i = i + 1'b1) begin  D\_in\_tb = D\_in\_tb + 1'b1;  tick;  end;  rnw\_tb = 1; //read twice  tick;  tick;  checkFlags; //FIFO is partially empty  //It has 1 element inside at the moment    $display("3. Partially Full Queue");  //3. FIFO is partially full  rnw\_tb = 0; //change to write - write 4 times  for(i = 0; i<3'b100; i = i + 1'b1) begin  D\_in\_tb = D\_in\_tb + 1'b1;  tick;  end;  checkFlags;  //write 4 times  //QUEUE now has 5 elements    $display("4. Full Queue");  //4. FIFO is full  for(i = 0; i<3'b011; i = i + 1'b1) begin  D\_in\_tb = D\_in\_tb + 1'b1;  tick;  end; //write 3 times  checkFlags;    $display("5. Adding to a full Queue");  //5. Attempting to add another element  D\_in\_tb = D\_in\_tb + 1'b1;  tick;    $display("Deleting untill empty");  //6. Deleting all items from queue  //Empty flag should go back to 1  rnw\_tb = 1; //read elements  for(i = 0; i < 4'b1010; i = i + 1'b1) begin  tick;  end  checkFlags;  #5;  end  endmodule |

|  |
| --- |
| FIFO\_fpga.v |
| `timescale 1ns / 1ps  module FIFO\_fpga(  input [7:0]Din,    input clk\_FPGA,  input button\_clk,  input button\_reset,  input rnw\_fpga,  input enable\_fpga,    output [7:0]ledDisplay,  output [3:0]ledPins\_fpga,  output reg rnw\_LED,  output reg enable\_LED,  output wire full\_LED,  output wire empty\_LED,    output reg [7:0]Dout  );    always @ (\*) begin  Dout = Din;  rnw\_LED = rnw\_fpga;  enable\_LED = enable\_fpga;  end  // wire full\_fpga;  // wire empty\_fpga;      wire[7:0] FIFO\_out;  wire fiveKhz\_clk;  wire button\_w;  wire button\_FPGA;  wire DONT\_USE;    FIFO f(  .D\_in(Din), // Data input to the FIFO  .D\_out(FIFO\_out), // Data output from the FIFO  .empty(empty\_LED), // Asserted when the FIFO is empty  .full(full\_LED), // Asserted when the FIFO is full  .clk(button\_FPGA), // clock  .rst(button\_reset), // Asynchronous reset  .rnw(rnw\_fpga), // read (1) or write (0) control  .enable(enable\_fpga)  );    clk\_gen clk (  .clk\_input(clk\_FPGA),  .rst(1'b0),  .clk\_4sec(DONT\_USE),  .clk\_5KHz(fiveKhz\_clk)  );  button\_debouncer db(  .clk(fiveKhz\_clk),  .button(button\_clk),  .debounced\_button(button\_FPGA)  );  wire [3:0] led3\_w = FIFO\_out % 10;  wire [3:0] led2\_w = ((FIFO\_out % 100) - led3\_w)/10;  wire [3:0] led1\_w = ((FIFO\_out % 1000) - led2\_w)/100;  wire [3:0] led0\_w = ((FIFO\_out % 10000) - led1\_w)/1000;  led\_display segs(  .clk\_input(clk\_FPGA),  .led0(led0\_w),  .led1(led1\_w),  .led2(led2\_w),  .led3(led3\_w),  .led\_out(ledDisplay),  .ledPins(ledPins\_fpga)  );  endmodule |

|  |
| --- |
| clk\_gen.v |
| `timescale 1ns / 1ps  module clk\_gen (  input wire clk\_input,  input wire rst,  output reg clk\_4sec,  output reg clk\_5KHz  );  integer count1, count2;  always @ (posedge clk\_input) begin  if (rst) begin  count1 = 0; clk\_4sec = 0;  count2 = 0; clk\_5KHz = 0;  end else begin  if (count1 == 200000000) begin  clk\_4sec = ~clk\_4sec;  count1 = 0;  end  if (count2 == 10000) begin  clk\_5KHz = ~clk\_5KHz;  count2 = 0; end  count1 = count1 + 1;  count2 = count2 + 1;  end  end  endmodule |

|  |
| --- |
| button\_debouncer.v |
| module button\_debouncer #(parameter depth = 16) (  input wire clk, /\* 5 KHz clock \*/  input wire button, /\* Input button from constraints \*/  output reg debounced\_button  );    localparam history\_max = (2\*\*depth)-1;  /\* History of sampled input button \*/  reg [depth-1:0] history;  always @ (posedge clk) begin  /\* Move history back one sample and insert new sample \*/  history <= { button, history[depth-1:1] };    /\* Assert debounced button if it has been in a consistent state throughout history \*/  debounced\_button <= (history == history\_max) ? 1'b1 : 1'b0;  end    endmodule |

|  |
| --- |
| seven\_seg\_display.v |
| module led\_display(  input clk\_input,  input [3:0]led0,  input [3:0]led1,  input [3:0]led2,  input [3:0]led3,  output reg [7:0]led\_out,  output reg [3:0]ledPins  );  reg clk\_5KHz;  reg [1:0]led\_sel;  reg [3:0]led\_out\_bcd;  integer count1, count2;  always @ (posedge clk\_input) begin  if(count1 == 200000000) begin  count1 = 0;  end  if (count2 == 10000) begin  clk\_5KHz = ~clk\_5KHz;  count2 = 0;  end  count1 = count1 + 1;  count2 = count2 + 1;  end  always @ (posedge clk\_5KHz) begin  led\_sel = led\_sel + 1;  case(led\_sel)  2'b00: begin  ledPins = 4'b0111;  led\_out\_bcd = led0;  end  2'b01: begin  ledPins = 4'b1011;  led\_out\_bcd = led1;  end  2'b10: begin  ledPins = 4'b1101;  led\_out\_bcd = led2;  end  2'b11: begin  ledPins = 4'b1110;  led\_out\_bcd = led3;  end  endcase  case(led\_out\_bcd)  4'd0: led\_out = 8'b11000000;  4'd1: led\_out = 8'b11111001;  4'd2: led\_out = 8'b10100100;  4'd3: led\_out = 8'b10110000;  4'd4: led\_out = 8'b10011001;  4'd5: led\_out = 8'b10010010;  4'd6: led\_out = 8'b10000010;  4'd7: led\_out = 8'b11111000;  4'd8: led\_out = 8'b10000000;  4'd9: led\_out = 8'b10010000;  4'b10: led\_out = 8'b01111111;  default: led\_out = 8'b11111111;  endcase  end  endmodule |

|  |
| --- |
| FIFO\_constraint.xdc |
| # Clock input  set\_property -dict {PACKAGE\_PIN W5 IOSTANDARD LVCMOS33} [get\_ports {clk\_FPGA}];  create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clk\_FPGA}];  # Button (debouncer and reset)  set\_property -dict {PACKAGE\_PIN U18 IOSTANDARD LVCMOS33} [get\_ports {button\_clk}]; # Center Button  set\_property -dict {PACKAGE\_PIN T17 IOSTANDARD LVCMOS33} [get\_ports {button\_reset}]; #Right button  # Input Switches  set\_property -dict {PACKAGE\_PIN V17 IOSTANDARD LVCMOS33} [get\_ports {Din[0]}]; # Din0  set\_property -dict {PACKAGE\_PIN V16 IOSTANDARD LVCMOS33} [get\_ports {Din[1]}]; # Din1  set\_property -dict {PACKAGE\_PIN W16 IOSTANDARD LVCMOS33} [get\_ports {Din[2]}]; # Din2  set\_property -dict {PACKAGE\_PIN W17 IOSTANDARD LVCMOS33} [get\_ports {Din[3]}]; # Din3  set\_property -dict {PACKAGE\_PIN W15 IOSTANDARD LVCMOS33} [get\_ports {Din[4]}]; # Din4  set\_property -dict {PACKAGE\_PIN V15 IOSTANDARD LVCMOS33} [get\_ports {Din[5]}]; # Din5  set\_property -dict {PACKAGE\_PIN W14 IOSTANDARD LVCMOS33} [get\_ports {Din[6]}]; # Din6  set\_property -dict {PACKAGE\_PIN W13 IOSTANDARD LVCMOS33} [get\_ports {Din[7]}]; # Din7  set\_property -dict {PACKAGE\_PIN R2 IOSTANDARD LVCMOS33} [get\_ports {rnw\_fpga}]; # Read (1)/Write(0)  set\_property -dict {PACKAGE\_PIN T1 IOSTANDARD LVCMOS33} [get\_ports {enable\_fpga}]; # enable  # LED Output  set\_property -dict {PACKAGE\_PIN U16 IOSTANDARD LVCMOS33} [get\_ports {Dout[0]}]; # Dout0  set\_property -dict {PACKAGE\_PIN E19 IOSTANDARD LVCMOS33} [get\_ports {Dout[1]}]; # Dout1  set\_property -dict {PACKAGE\_PIN U19 IOSTANDARD LVCMOS33} [get\_ports {Dout[2]}]; # Dout2  set\_property -dict {PACKAGE\_PIN V19 IOSTANDARD LVCMOS33} [get\_ports {Dout[3]}]; # Dout3  set\_property -dict {PACKAGE\_PIN W18 IOSTANDARD LVCMOS33} [get\_ports {Dout[4]}]; # Dout4  set\_property -dict {PACKAGE\_PIN U15 IOSTANDARD LVCMOS33} [get\_ports {Dout[5]}]; # Dout5  set\_property -dict {PACKAGE\_PIN U14 IOSTANDARD LVCMOS33} [get\_ports {Dout[6]}]; # Dout6  set\_property -dict {PACKAGE\_PIN V14 IOSTANDARD LVCMOS33} [get\_ports {Dout[7]}]; # Dout7  set\_property -dict {PACKAGE\_PIN L1 IOSTANDARD LVCMOS33} [get\_ports {rnw\_LED}]; # rnw LED  set\_property -dict {PACKAGE\_PIN P1 IOSTANDARD LVCMOS33} [get\_ports {enable\_LED}]; # enable LED  set\_property -dict {PACKAGE\_PIN P3 IOSTANDARD LVCMOS33} [get\_ports {full\_LED}]; # full LED  set\_property -dict {PACKAGE\_PIN U3 IOSTANDARD LVCMOS33} [get\_ports {empty\_LED}]; # empty LED  # 7 Segment Display  set\_property -dict {PACKAGE\_PIN W7 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[0]}]; # CA  set\_property -dict {PACKAGE\_PIN W6 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[1]}]; # CB  set\_property -dict {PACKAGE\_PIN U8 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[2]}]; # CC  set\_property -dict {PACKAGE\_PIN V8 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[3]}]; # CD  set\_property -dict {PACKAGE\_PIN U5 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[4]}]; # CE  set\_property -dict {PACKAGE\_PIN V5 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[5]}]; # CF  set\_property -dict {PACKAGE\_PIN U7 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[6]}]; # CG  set\_property -dict {PACKAGE\_PIN V7 IOSTANDARD LVCMOS33} [get\_ports {ledDisplay[7]}]; # DP  set\_property -dict {PACKAGE\_PIN U2 IOSTANDARD LVCMOS33} [get\_ports {ledPins\_fpga[0]}]; # AN0  set\_property -dict {PACKAGE\_PIN U4 IOSTANDARD LVCMOS33} [get\_ports {ledPins\_fpga[1]}]; # AN1  set\_property -dict {PACKAGE\_PIN V4 IOSTANDARD LVCMOS33} [get\_ports {ledPins\_fpga[2]}]; # AN2  set\_property -dict {PACKAGE\_PIN W4 IOSTANDARD LVCMOS33} [get\_ports {ledPins\_fpga[3]}]; # AN3 |