

San Jose State University
Department of Computer Engineering

CMPE 125 Lab Report

Lab 6 Report

Title Storage Building Blocks

Semester FALL 2019

Date 10-21-19

by

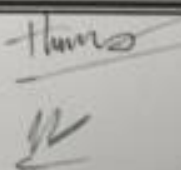

Name HUNG NGUYEN
(typed)

SID 012392081
(typed)

Name PHAT LE
(typed)

SID 012067666
(typed)

Lab Checkup Record

Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
		100%*		

* Detailed descriptions must be given in the report.

* Diagram needed.

Introduction

The purpose of this lab is to become familiar with storage building block, with the design, verification and validation flow and EDA tools. Specifically, the main task is to create a register file that can read and write data from the given addresses, data, and write enable signals. Another task is to create a FIFO (First in First out) with an input signal to select write or read function and two outputs Full and Empty to show if the file is full or empty. The lab also requires functional verification and FPGA validation.

Design Methodology

The first task is to design and functionally verify a register file that have 2 read ports and 2 write ports. The read function is combinational and the write function is sequential. Unlike the register file in the lecture slide which only have one write port, the register file for this lab need to handle special cases since it can write two data at the same clock cycle. The problem occurs when both write signals want to write to the same address of the file. Our solution is to write neither data from two write ports when the address is the same. Furthermore, both read ports can read from the same address at any time.

Table 1 : List of modules for Task 1

Module	Function
filereg2	This module is the design for the register file that contains both sequential and combinational section that read and write data based on the given address signal (ra1, ra2, wa1, wa2). The combination section includes 2 read output ports(rd1 & rd2) that will read out data from file whenever the enable signal of desired port(re1 & re2) is ON(1). The sequential section includes 2 write input ports(wd1 & wd2) that would write to the file only when they are enabled (we1 & we2) and having difference write address (wa1 & wa2).
filereg2_tb	This module test the functionality of the filereg2.v design file only for special corner cases. The first case would check the file when one of the write enable inputs(we) is ON(1) and when the read and write addresses are the same (ra1 = ra2 = wa1). The second case will check the file when both write addresses are the same.

The second task of the lab is to design, functionally verify, and FPGA validate a FIFO that have 8 bit deep and 4 bit wide. The primary function is the first data input is also the first data that output. Since it has 8 bit deep, FIFO can store 4-bits inputs at 8 different address locations. To

control the function, FIFO use WNR to select write(1) and read(0) instructions. It also use enable, clock, and reset to make it sequential. Moreover, the FIFO also includes Full output signal when it already fill up data in all eight addresses or empty when there is no data in all eight addresses.

Table 2 : List of module for task 2

Module	Function
FIFO_FPGA	This is the top level module for hardware validation. It connects FIFO module with a button_debouncer module which connect to clk_gen module as a clock input. It also control the connection of input switches and output LED.
FIFO	This module is the main design module of task 2. It uses two pointers for pointing to read and write address. Every time new data is read or write, the read and write pointers are increments by 1. When both pointers are at the same location, it outputs empty (1) and full(0). When the write pointer incrementing until it meets the read pointer gain, FIFO would output full(1) and empty (0)
button_debouncer	This module acts as the clock control input
clk_gen	This module produce 5kHz clock signal for the button_debouncer module.
FIFO_tb	This is a testbench module that push in data(write) from empty to full and pop data from full to empty (read).

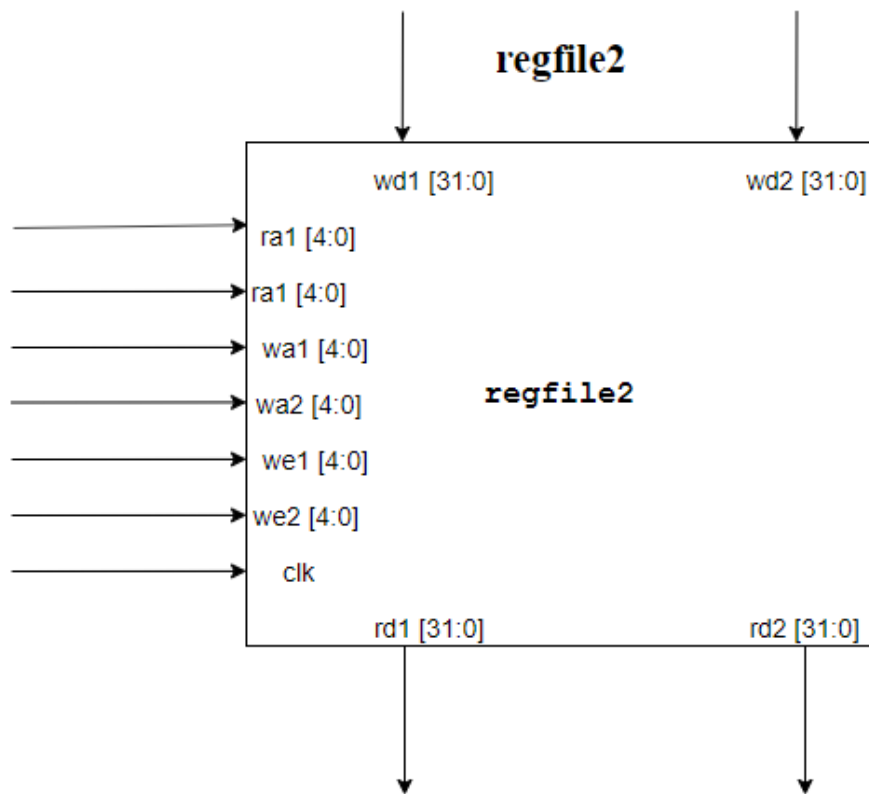


Figure 1: Block diagram for Task 1

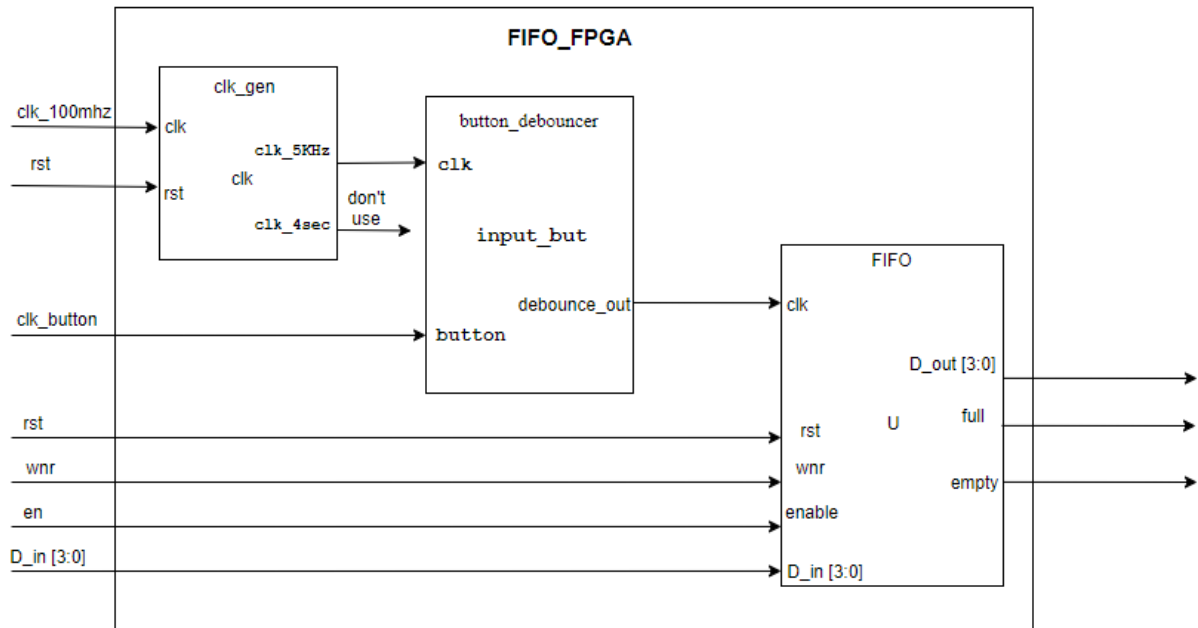


Figure 2: Block diagram for Task 2

Simulation Result

Task 1

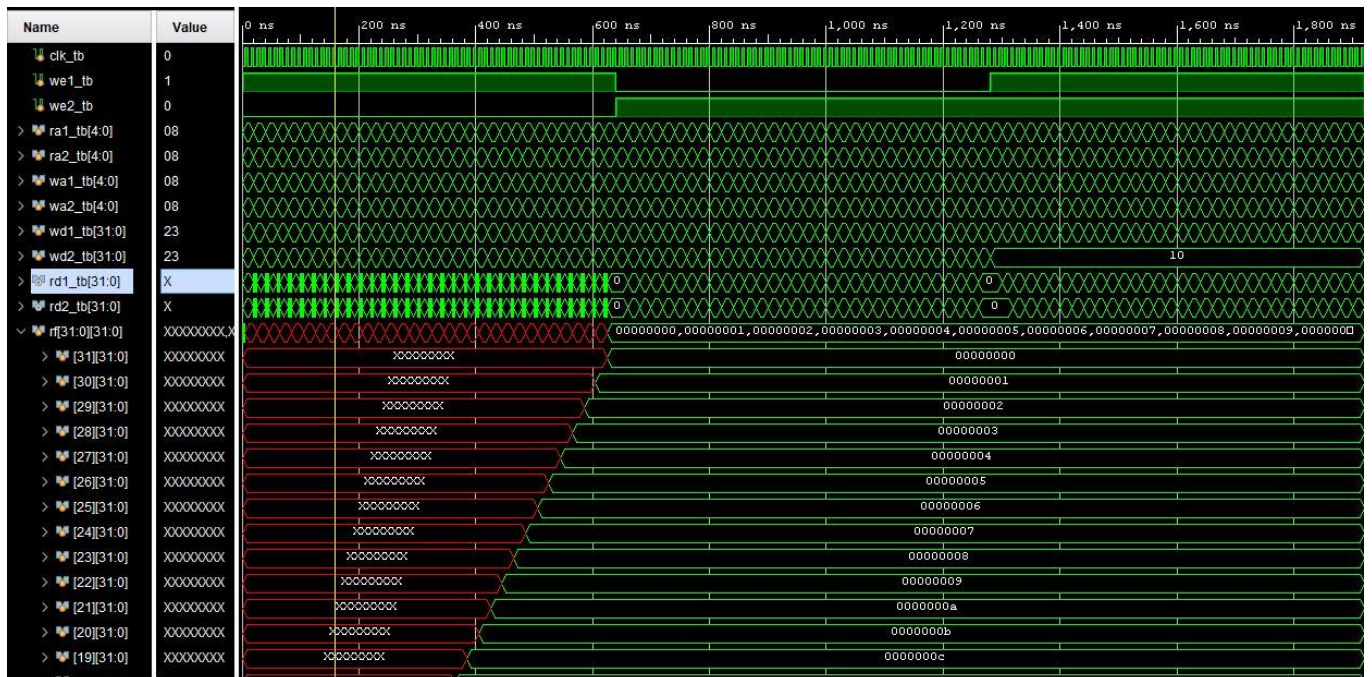


Figure 3: Simulation waveforms produced from Task 1 for all corner cases

The testbench for this task would check three cases (when we1 is enabled, when we2 is enabled, when both we1 and we2 are enabled). Since there are so many possible input cases, the testbench would only cover the case when all read and write addresses are the same in one of the write enable is ON. We also handle the case when the wire addresses are the same.

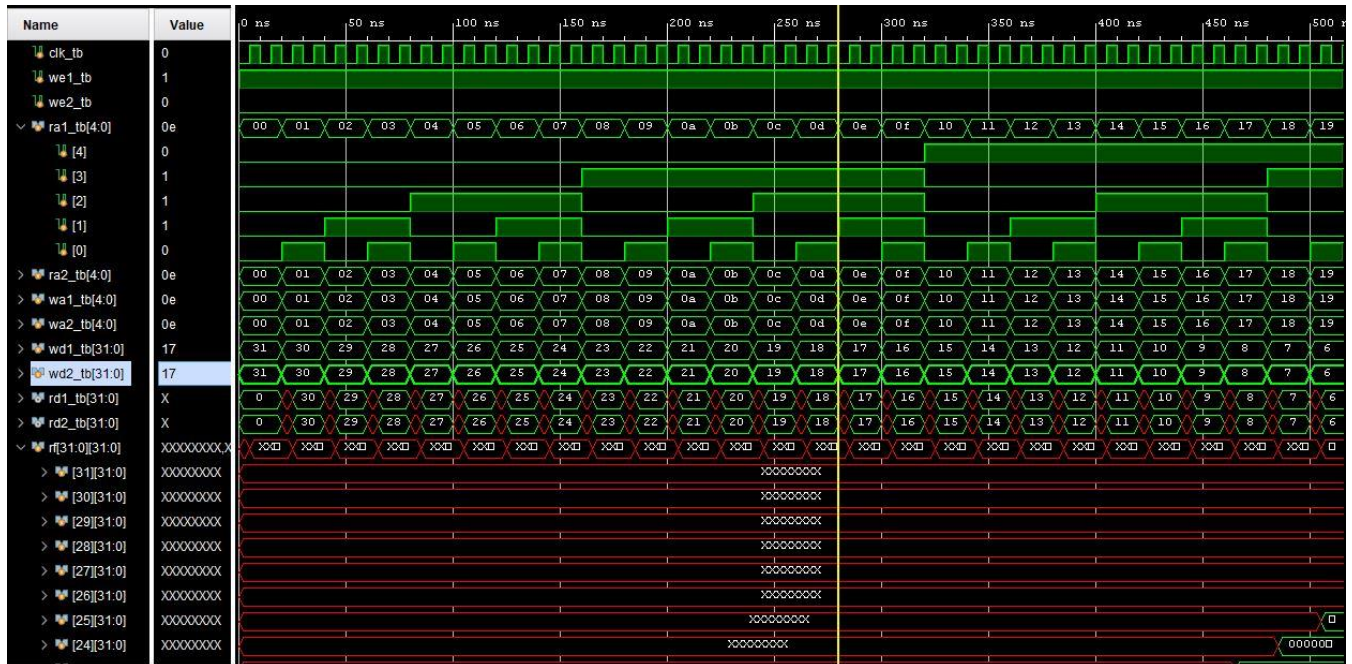


Figure 4: Simulation waveforms produced from Task 1 when only write port 1 is enable (we1_tb = 1, we2_tb = 0).

According to Figure 4, only we1 is enabled. The testbench only handle the case when ra1= ra2 = wa1. We can observe that the write input data (wd1_tb) is the same as the read outputs (rd1_tb and rd2_tb) as expected. Since the read function is combinational, it reads the data instantly once it received a new address location. In this case, the read data from a new address location only appears at the first rising edge because the location does not have any content yet (displayed as red waveform for half cycle). As soon as the new data is written on that address (at the rising edge of the next clock cycle), the rd begin to display proper data waveform (green waveform).

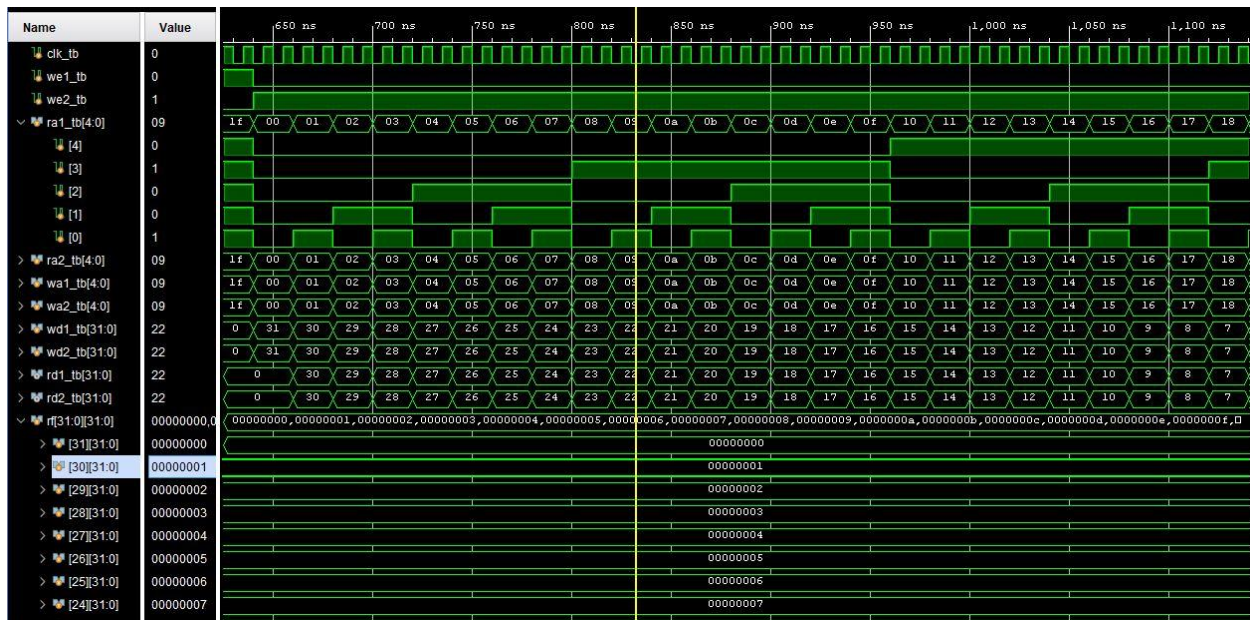


Figure 5:Simulation waveforms produced from Task 1 when only write port 2 is enabled($we2_tb = 1$, $we1_tb = 0$).

Similar to the above case, this testbench also handle the case when write port 2 were enabled and $rd1=rd2=wd2$. The read data for this case did not appear read waveforms because the same data were already written to the same addresses in the case above. Furthermore, we could observe that the $wd2$ waveform was the same as $rd1$ and $rd2$ as expected.

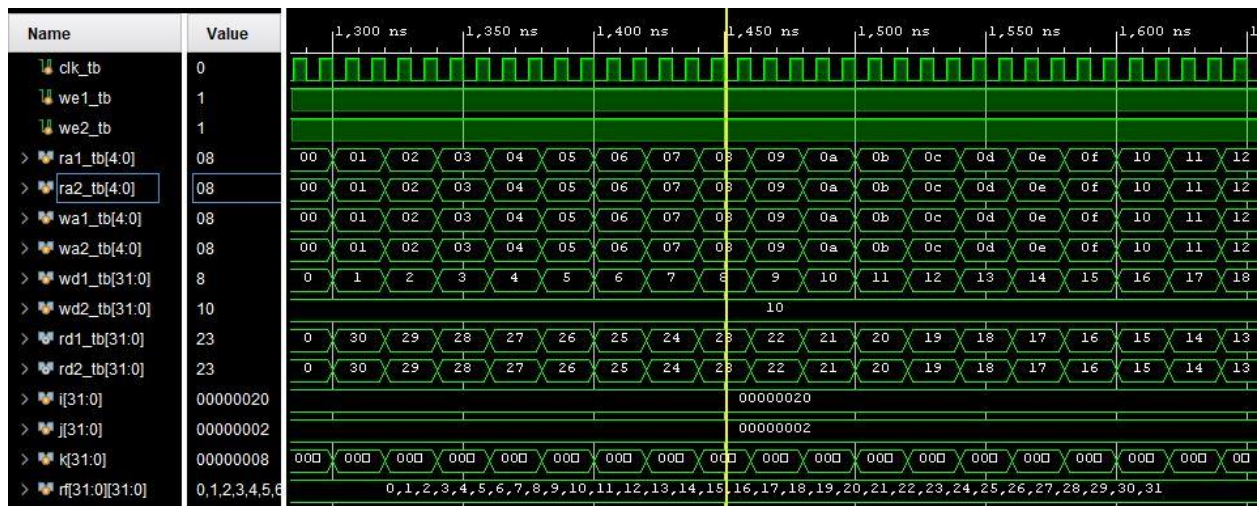


Figure 6:Simulation waveforms produced from Task 1 only when both write port 1 and 2 are enabled($we2_tb = 1$, $we1_tb = 1$).

When both write ports were enabled ($we1 = we2 = 1$), the new data would not write to the assigned address location. According to the figure above, write data of port 2 was given a fixed value of 10 and the write data of 1 was a chain of number from 0-31. In this case, although all the read and write ports were given the same address, the read data did not match with the write

data since nothing new data had written to the file. For this reason, the simulation verification for task 1 was successfully built because the waveforms is as expected.

Task 2

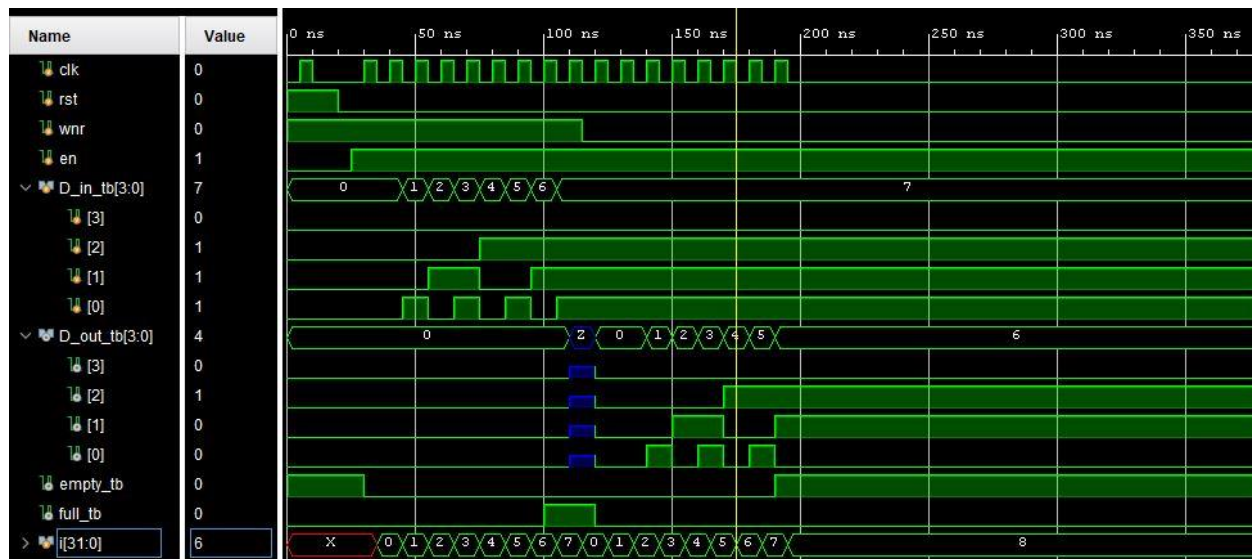


Figure 4: Simulation waveforms produced from Task 2

For task 2, the goal for the testbench were filling up the empty file until it is full, then pop the data out until the file is empty. In this case, the process of pushing(write) the input data(D_in_tb) was a for loop of 8 number from 0 to 7 so that the file is full. The pop process (read) was also a for loop that pop out the output data(D_out_tb) one by one until it empty. We could observe from the figure above that the file is empty at the beginning (empty_tb = 1, full_tb = 0). During the write process, the file was partially empty and partially full(empty_tb = 0, full_tb = 0). When all eight values had written in the file, it became full (empty_tb = 0, full_tb = 1). Thus, the functional verification for task 2 was successfully made.

FPGA Validation

For the FPGA validation process, the rightmost four switches represents 4-bit input D_in and the four LEDs above them are output D_out. The leftmost three switches represents reset(rst), enable(en), and W/R(wnr) respectively. Furthermore, the whole process is controlled by a clock button. When the reset switch was ON(1), it cleared all the data in the file which turned on the empty LED. The board could read or write only when the reset switch was OFF(0) and enable switch was ON(1). For the wnr switch, when it was ON(1), the board can write the data from the D_in input to the file and the empty LED would turn OFF(0). As eight inputs were written in the file (for 8 clock cycles press of the button), the full LED would turn ON(1). Similar to the write instruction, the read instruction begin once wnr switch was OFF(0). As eight data from file were read out (4-bit D_out output LEDs) for 8 clock cycles press of the button, the full LED would turn OFF(0), and the empty LED is ON again(1). Since the FPGA board behaviour matches the expected functionality of FIFO design, the FPGA validation process was successful..

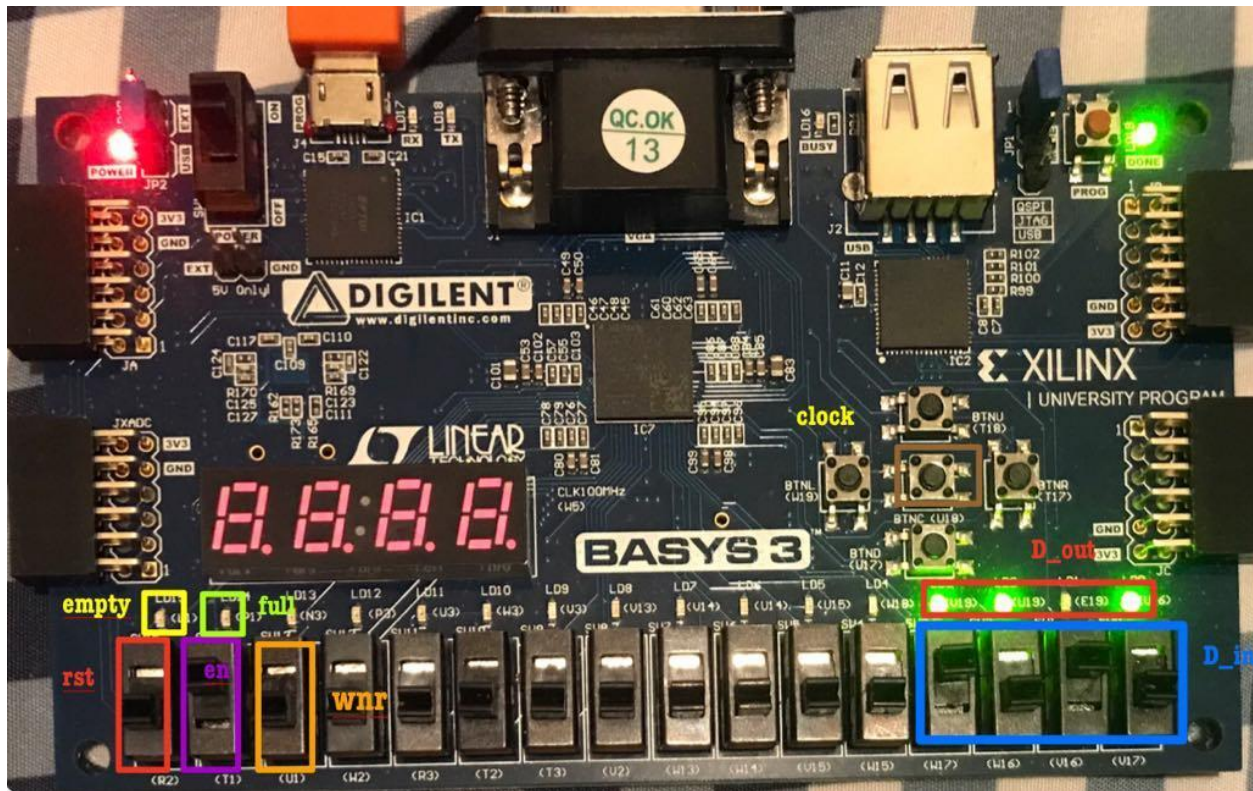


Figure 10: inputs and outputs layout for the FPGA validation process of task 2.

Conclusion

Overall, both the simulation and FPGA validation process for Task 1 and 2 was successfully performed since the result is as expected. This lab experiment helps strengthen our knowledge on functionality of reading and writing to file by closely observe the changes between reset, enable, and address signals. We also learned that although the read instruction is combinational(instantly read data from file), it still requires a delay time of next rising edge of the clock to write and read new data. One problem that we faced during the testbench was that we could not read any data during the simulation process. Luckily, the TA helped us debug the problem by tracing back the unknown signal from file. He added the rf file to the simulated waveform to observe if the data was written to the file.

Appendix

a. Source Code

Task 1

regfile2.v

```

module regfile2(input clk,
                input we1, we2,    ///write enable
                input [4:0] ra1, ra2, wa1, wa2,    // read address
                input[31:0] wd1, wd2,    //write data
                output [31:0] rd1, rd2);
    reg [31:0] rf [31:0]; //read file
    always @(posedge clk)
    begin
        if(we1 && we2) // if both are on
        begin
            if(wa1!=wa2)
            begin
                if(we1) rf[wa1] <= wd1;//write to one of enable wrie
                if(we2) rf[wa2] <= wd2;
            end
        end
        //when only one of the we is enable
        else
        begin
            if(we1) rf[wa1] <= wd1;//write to one of enable wrie
            if(we2) rf[wa2] <= wd2;
        end
    end

    assign rd1 = (ra1 != 0) ? rf[ra1]:0;
    assign rd2 = (ra2 != 0) ? rf[ra2]:0;
endmodule

```

regfile2_tb.v

```

module regfile2_tb;
    reg clk_tb;
    reg we1_tb;
    reg we2_tb;
    reg [4:0] ra1_tb;
    reg [4:0] ra2_tb;
    reg [4:0] wa1_tb;
    reg [4:0] wa2_tb;
    reg [31:0] wd1_tb;
    reg [31:0] wd2_tb;

    wire[31:0] rd1_tb;
    wire[31:0] rd2_tb;

    integer i, j,k;

    regfile2 DUT(.clk(clk_tb),
                .we1(we1_tb),
                .we2(we2_tb),
                .ra1(ra1_tb),
                .ra2(ra2_tb),
                .wa1(wa1_tb),
                .wa2(wa2_tb),
                .rd1(rd1_tb),
                .wd1(wd1_tb),
                .wd2(wd2_tb),

```

```

        .rd2(rd2_tb));

task ticktock;
begin
    #5 clk_tb = ~clk_tb;
    #5 clk_tb = ~clk_tb;
end
endtask

initial begin
    clk_tb = 0;
    we1_tb = 0; //      disable both write
    we2_tb = 0;

    //case2
    for (j=0; j<2; j= j+1)
    begin

        we1_tb = 1-j; // enable write 1 when j =0
        we2_tb = j; //disable write 2 when j =0
        for (i=0; i<32; i=i+1)
        begin
            wa1_tb = i;
            wa2_tb = i;
            ra1_tb = wa1_tb;
            ra2_tb = wa2_tb;
            //initialize when we1 is enable
            wd1_tb = 31-i; //let the data run from 31-0
            wd2_tb = 31-i; //let the data run from 31-0
            ticktock();
            ticktock();
        end
    end

    //case1
    wd2_tb = 10;
    //case1
    for(k =0; k<32; k=k+1)
    begin
        we1_tb = 1;
        we2_tb = 1; //they both enable
        wa1_tb = k;
        wa2_tb = k; //wire address is the same
        ra1_tb = wa1_tb;
        ra2_tb = wa2_tb;
        //initialize when we1 is enable
        wd1_tb = k; //let the data run from 31-0
        //wd2_tb = 10; //let the data run from 31-0
        ticktock();
        ticktock();

    End
    $display("test successful");
    $finish;

end

```

```
endmodule
```

Task 2

FIFO.v

```
module FIFO#(parameter bus_width = 4,
              parameter addr_width = 3,
              parameter fifo_depth= 8)// 2^addr_width = fifo_depthinputclk;// clock
(input clk,
 input rst, // Asynchronous reset
 input wnr, // read (0) or write (1) 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 [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(!wnr && !empty) // start reading
        begin
            D_out = mem[r_ptr[addr_width-1:0]];
            r_ptr = r_ptr + 1;
        end
    else if(wnr && !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
end
endmodule

```

clk_gen.v

```

module clk_gen (
    input wire clk100MHz,
    input wire rst,
    output reg clk_4sec,
    output reg clk_5KHz
);

integer count1, count2;

always @ (posedge clk100MHz) begin
    if (rst) begin
        count1 = 0;
        count2 = 0;
        clk_5KHz = 0;
        clk_4sec = 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

```



```

        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

```

FIFO_FPGA.v

```

module FIFO_FPGA(input wire [3:0] D_in,
    output wire [3:0] D_out,
    input wire clk100MHz,
    input wire rst,en, wnr,
    output wire empty,full,
    input wire clk_button
);
    wire DONT_USE;
    wire clk_5KHz;
    wire debounced_clk;
    //wire [3:0] D_out;

```

```

clk_gen clk(.clk100MHz(clk100MHz),
            .rst(rst),
            .clk_4sec(DONT_USE),
            .clk_5KHz(clk_5KHz));

button_debouncer input_but(.clk(clk_5KHz),
                           .button(clk_button),
                           .debounced_button(debounced_clk));

FIFO U(.clk(debounced_clk),
       .rst(rst),
       .wnr(wnr),
       .enable(en),
       .D_in(D_in),
       .D_out(D_out),
       .full(full),
       .empty(empty));

endmodule

```

FIFO_tb.v

```

module FIFO_tb;
    reg clk,rst,wnr,en;
    reg [3:0] D_in_tb;
    wire [3:0] D_out_tb;
    wire empty_tb, full_tb;

    integer i;
    FIFO DUT(.clk(clk),
            .rst(rst),
            .wnr(wnr),
            .enable(en),
            .D_in(D_in_tb),
            .D_out(D_out_tb),
            .empty(empty_tb),
            .full(full_tb));

    task ticktock;
    begin
        #5 clk = ~clk;
        #5 clk = ~clk;
    end
    endtask

    initial begin
        D_in_tb = 0;
        clk = 0;
        rst = 1;
        wnr = 1;//write
        en=0;
        ticktock();

        #5 rst = 1; // clear all unknowns
        #5 rst = 0; // clear all unknowns
        #5 en = 1; //enable to begin process
        ticktock();
    end
endmodule

```

```

        for (i = 0; i < 8; i=i+1)
        begin
            D_in_tb = i;
            wnr = 1;
            ticktock();
        end

        for (i = 0; i < 8; i=i+1)
        begin
            wnr = 0;
            ticktock();
        end
    end
end
endmodule

```

FIFO_constraints.xdc

```

# Clock signal
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33} [get_ports
{clk100MHz}];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
{clk100MHz}];

# input switches
#D_in
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports
{D_in[0]}];
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports
{D_in[1]}];
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports
{D_in[2]}];
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33} [get_ports
{D_in[3]}];

#rst
set_property -dict {PACKAGE_PIN R2 IOSTANDARD LVCMOS33} [get_ports {rst}];
#en
set_property -dict {PACKAGE_PIN T1 IOSTANDARD LVCMOS33} [get_ports {en}];
#rnw
set_property -dict {PACKAGE_PIN U1 IOSTANDARD LVCMOS33} [get_ports {wnr}];

#clock button
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports
{clk_button}];

#D_out
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports
{D_out[0]}];
set_property -dict {PACKAGE_PIN E19 IOSTANDARD LVCMOS33} [get_ports
{D_out[1]}];
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports
{D_out[2]}];
set_property -dict {PACKAGE_PIN V19 IOSTANDARD LVCMOS33} [get_ports
{D_out[3]}];

```

```
#empty
set_property -dict {PACKAGE_PIN L1 IOSTANDARD LVCMOS33} [get_ports {empty}];

#full
set_property -dict {PACKAGE_PIN P1 IOSTANDARD LVCMOS33} [get_ports {full}];
```