Hardware Design Lab_4 Report

Name:

林奕為

Department:

CS 25

Student’s ID:

110062271

Contents

What I have learned from the Lab

In this lab, I spend lots of time especially on the Advanced_Question_4 and FPGA_implementation for hours to debug,

In question 4 I examined the module
carefully, and followed the wave form provided in the spec pdf to eventually modified the FIFO_8 module and make the result output the same as the example.

In FPGA_implementation I also debug it for hours, in the process I continue correcting the wrong code and eventually finding that I set the wrong top module in the Vivado.

Since I have no one to count on but myself, I significantly improve my debugging skill and coding skill with respect to the Verilog.

There are many possibilities when your Verilog or FPGA module goes wrong, you must check the wave form, design the testbench, and test your assumptions one by one, this is a arduous process, but after many days stayed up late being at Education building, next time the period between picking out the wrong possibilities should be greatly shorter.

Also, I didn’t score well in the previous lab, probably due to the incompleteness of the report, so in this lab, I improve my work on the report and organized my report in a more professional and detailed way.

This is another thing I learned from this lab, the skill about report writing can transfer to other courses as well, and it should benefit me about professional skill especially with respect to writing soon.

Here’s the Diagram link in google drive:
https://drive.google.com/drive/folders/1udhDYFbK_WbuNwmLSdtH1L5CW1JL9XcH?usp=sharing

Advanced Question 1: 4-bit Ping-Pong Counter

How Do I Design Source Code:

I followed the instructions, when rst_n == 1’b0, the counter resets its value to 4’b0000, and the direction to 1’b1, and if enable == 1’b1, the counter begins its operation.

I update out <= next_out and direction <= next_direction each posedge clk is triggered.

I use always (*) to update next_out and next_direction.
When out reaches its upper bound or (lower bound AND direction == 1’b0),
I flip the direction.

If direction == 1’b1, next_out equals out + 1’b1 otherwise out – 1’b1;

Here’s the source code:

```
`timescale 1ns/1ps

module Ping_Pong_Counter (clk, rst_n, enable, direction, out);
input clk, rst_n;
input enable;
output direction;
output [4-1:0] out;

reg direction;
reg next_direction;
reg [3:0] out;
reg [3:0] next_out;

always @(posedge clk) begin
    if(!rst_n) begin
        out <= 4'd0;
        direction <= 1'b1;
    end
    else begin
```

```
        out <= next_out;
        direction <= next_direction;
    end
end

always @(*) begin
    if(enable) begin
        if(out == 4'd15 || (out == 4'd0 && direction == 1'b0))
next_direction = !direction;
        if(next_direction == 1'b0) next_out = out - 1;
        else if(next_direction == 1'd1) next_out = out + 1;
    end
    else begin
        next_out = out;
        next_direction = direction;
    end
end

endmodule
```

How Do I Design Testbench:

Each 5ns the clk signal will be flipped.

At First cycle, I set enabled = 1’b0
2nd cycle, I set rst_n = 1’b0
3rd cycle, I set rst_n = 1’b1
Next 3 cycles, I set rst_n = 1’b0 to examine the output remains the same

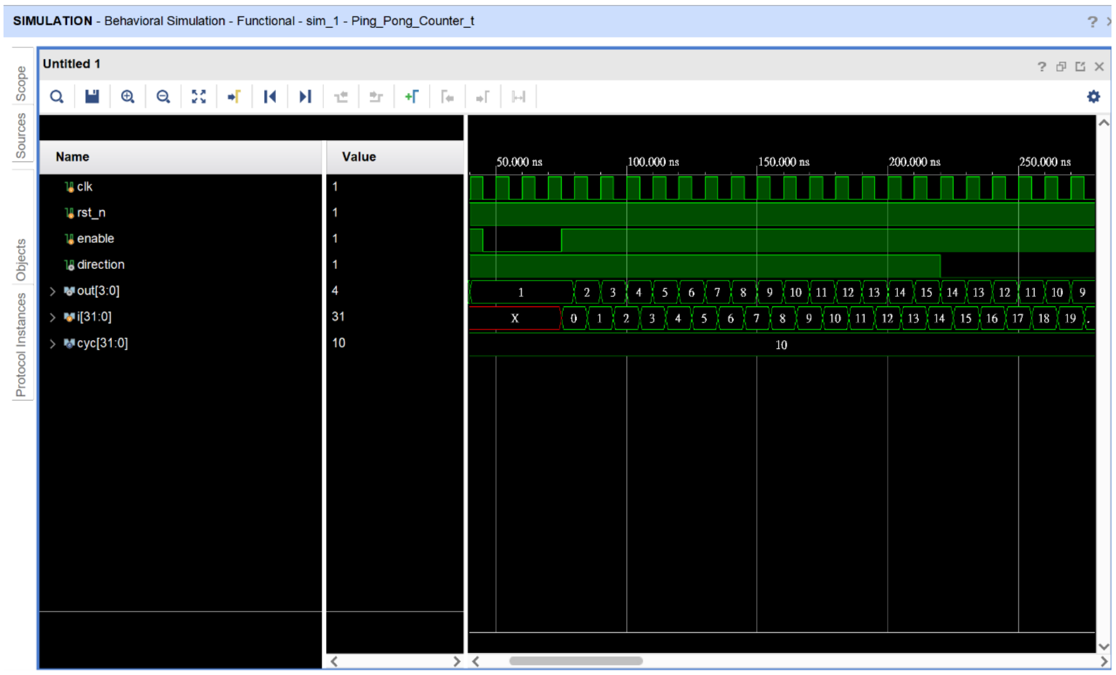
Last, I use a for loop to run 30 cycles to test if the counter counts correctly when direction == 1’b1 and 1’b0;

Here’s the testbench code:

```
`timescale 1ns/1ps

module Ping_Pong_Counter_t;
reg clk = 1'b1;
reg rst_n, enable;
wire direction;
wire [3:0] out;


// specify duration of a clock cycle.
parameter cyc = 10;
integer  i;

// generate clock.
always#(cyc/2)clk = !clk;

// Ping_Pong_Counter (clk, rst_n, enable, direction, out);
Ping_Pong_Counter P1(clk, rst_n, enable, direction, out);

initial begin
    @(negedge clk)
        enable = 1'b0;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
        rst_n = 1'b0;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
        rst_n = 1'b1;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
        $display("out : %d, direction: %b", out, direction);
        enable = 1'b1;
    @(negedge clk)
        enable = 1'b0;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
```

```
        enable = 1'b0;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
        enable = 1'b0;
        $display("out : %d, direction: %b", out, direction);
    @(negedge clk)
        enable = 1'b1;
        $display("out : %d, direction: %b", out, direction);

    for (i = 0; i <= 30; i = i+1) begin
        @(negedge clk)
            $display("out : %d, direction: %b", out, direction);
    end
end

endmodule
```

Here's the partial waveform result:

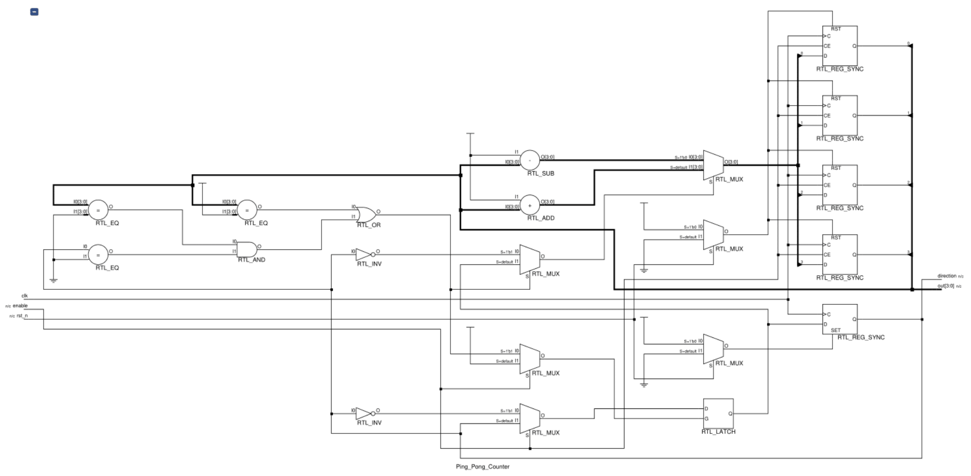


Name	Value
clk	1
rst_n	1
enable	1
direction	1
out[3:0]	4
i[31:0]	31
cyc[31:0]	10

```
out[3:0]  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 14 13 12 11 10 9
i[31:0]   X 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
cyc[31:0] 10
```

Diagram:

Ping Pong Counter :



Advanced Question 2: First-In First Out (FIFO) Queue

How Do I Design Source Code:

I use a reg [7:0] mem [7:0] to record data, and reg [2:0] read_ptr, write_ptr to represent index.

If ((ren && !wen) || (ren && wen)) read operation is enabled.
When empty it output error otherwise it reads out data and immediately set the empty signal to 1'b0. Afterwards, the read_ptr index is incremented in a circular fashion. If the operation results the queue empty, set empty to 1'b1.

Similarly, if (wen && !ren) write operation is enabled.
When full it outputs error otherwise it writes in data and immediately set the full signal to 1'b0. Afterwards, the write_ptr index is incremented in a circular fashion. If the operation results the queue full, set full to 1'b1.

Here's the source code:

```
module FIFO_8(clk, rst_n, wen, ren, din, dout, error);
input clk;
input rst_n;
input wen, ren;
input [8-1:0] din;
output [8-1:0] dout;
output error;

reg [7:0] dout;
reg error;

reg [7:0] mem [7:0];
reg full, empty;
reg [2:0] read_ptr, write_ptr;

always @(posedge clk) begin
    if(~rst_n) begin
        read_ptr <= 3'd0;
        write_ptr <= 3'd0;
        dout <= 8'd0;
        error <= 1'b0;
        empty <= 1'b1;
        full <= 1'b0;
    end
    else begin
        if((ren && !wen) || (ren && wen)) begin
            if(empty) error <= 1'b1;
            else begin
                error <= 1'b0;
                dout <= mem[read_ptr];
                full <= 1'b0;
                if(read_ptr == 3'd7) begin
                    if(write_ptr == 3'd0) empty <= 1'b1;
                    else empty <= 1'b0;
                end
                else begin
                    if(read_ptr + 3'd1 == write_ptr) empty <= 1'b1;
                    else empty <= 1'b0;
                end
                read_ptr <= (read_ptr == 3'd7) ? 3'd0 : read_ptr + 3'd1;
            end
        end
        else if(wen && !ren) begin
            if(full) error <= 1'b1;
            else begin
                error <= 1'b0;
                mem[write_ptr] <= din;
                empty <= 1'd0;
                if(write_ptr == 3'd7) begin
                    if(read_ptr == 3'd0) full <= 1'b1;
                    else full <= 1'b0;
                end
                else begin
                    if(write_ptr + 3'd1 == read_ptr) full <= 1'b1;
                    else full <= 1'b0;
                end
                write_ptr <= (write_ptr == 3'd7) ? 3'd0 : write_ptr +
3'd1;
            end
        end
        else dout <= dout; // ren == 1'b0 && wen == 1'b0
    end
end

endmodule
```

How Do I Design Testbench:

I first write in 8 input data to the queue and read them out.

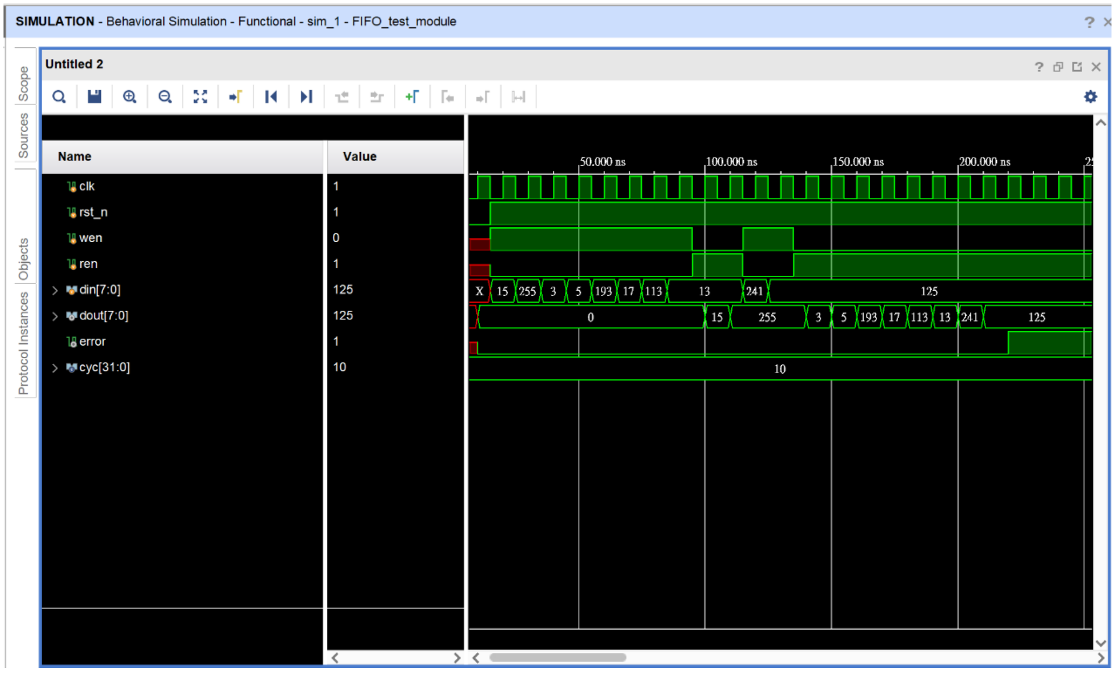
Then, I write in 2 additional data to test if the circular function works and read them out respectively.

Here’s the testbench =code:

```
module FIFO_test_module;
reg clk = 1'b1;
reg rst_n, wen, ren;
reg [7:0] din;
wire [7:0] dout;
wire error;

// specify duration of a clock cycle.
parameter cyc = 10;

// generate clock.
always#(cyc/2)clk = !clk;

// FIFO_8(clk, rst_n, wen, ren, din, dout, error);
FIFO_8 F1(clk, rst_n, wen, ren, din, dout, error);

initial begin
    @(negedge clk)
        rst_n = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
        wen = 1'b1;
        ren = 1'b0;
        din = 8'b00001111;
    @(negedge clk)
        din = 8'b11111111;
    @(negedge clk)
        din = 8'b00000011;
    @(negedge clk)
        din = 8'b00000101;
    @(negedge clk)
        din = 8'b11000001;
    @(negedge clk)
        din = 8'b00010001;
    @(negedge clk)
        din = 8'b01110001;
    @(negedge clk)
        din = 8'b00001101;
    @(negedge clk)
        wen = 1'b0;
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b0;
        wen = 1'b1;
        din = 8'b11110001;
    @(negedge clk)
        din = 8'b01111101;
    @(negedge clk)
        wen = 1'b0;
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b1;
    @(negedge clk)
```

```
        ren = 1'b1;
    @(negedge clk)
        ren = 1'b1;

end

endmodule
```

Here's the wave form diagram:

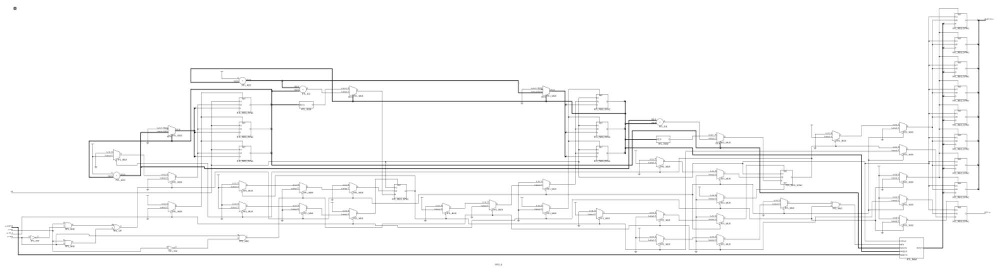

SIMULATION - Behavioral Simulation - Functional - sim_1 - FIFO_test_module

Untitled 2

Name	Value
clk	1
rst_n	1
wen	0
ren	1
din[7:0]	125
dout[7:0]	125
error	1
cyc[31:0]	10

Diagram:

FIFO_8 module:



Advanced Question 3: Multi-Bank Memory

How Do I Design Source Code:

I reuse the Memory module from Basic question, and design two additional module, Bank_Memory which is made up of 4 Memory modules and Multi_Bank_Memory which is made up of 4 Bank_Memory modules.

Inside the Bank_Memory, it determines which Memory module to operate on and decides to what value of ren and wen to pass into each Memory based on raddr[8:7] and waddr[8:7].

Similarly in Multi_Bank_Memory, it operates in a similar way to Bank_Memory, besides t it pass value into 4 Bank_Memory modules respectively.

Additionally, if(ren == 1'b1 && (raddr[8:7] == waddr[8:7])) and wen equals 1’b1, only read operation is performed.

Other details listed in the spec are also implemented.

Here’s the source code:

```
`timescale 1ns/1ps

module Memory (clk, ren, wen, addr, din, dout);
input clk;
input ren, wen;
input [7-1:0] addr;
input [8-1:0] din;
output [8-1:0] dout;

reg [8-1:0] dout;
reg [8-1:0] memory [127:0];

always @(posedge clk) begin
    if(ren && wen) begin
```

```
        dout [8-1:0] <= memory[addr];
    end
    else if(ren && !wen) begin
        dout [8-1:0] <= memory[addr];
    end
    else if(wen && !ren) begin
        memory[addr] <= din[8-1:0];
        dout <= 0;
    end
    else begin
        dout <= 0;
    end
end

endmodule

module Bank_Memory(clk, ren, wen, waddr, raddr, din, dout);
input clk;
input ren, wen;
input [11-1:0] waddr;
input [11-1:0] raddr;
input [8-1:0] din;
output [8-1:0] dout;

reg ren1, ren2, ren3, ren4, wen1, wen2, wen3, wen4;
wire [7:0] out1, out2, out3, out4;
reg [7:0] dout;

always @(*) begin
    if(ren == 1'b1) begin
        ren1 = (raddr[8:7] == 2'b00) ? 1'b1 : 1'b0;
        ren2 = (raddr[8:7] == 2'b01) ? 1'b1 : 1'b0;
        ren3 = (raddr[8:7] == 2'b10) ? 1'b1 : 1'b0;
        ren4 = (raddr[8:7] == 2'b11) ? 1'b1 : 1'b0;
    end
    else begin
        ren1 = 1'b0;
        ren2 = 1'b0;
        ren3 = 1'b0;
        ren4 = 1'b0;
    end

    if(wen == 1'b1) begin
        if(ren == 1'b1 && (raddr[8:7] == waddr[8:7])) begin
            wen1 = 1'b0;
            wen2 = 1'b0;
            wen3 = 1'b0;
            wen4 = 1'b0;
        end
        else begin
            wen1 = (waddr[8:7] == 2'b00) ? 1'b1 : 1'b0;
            wen2 = (waddr[8:7] == 2'b01) ? 1'b1 : 1'b0;
            wen3 = (waddr[8:7] == 2'b10) ? 1'b1 : 1'b0;
            wen4 = (waddr[8:7] == 2'b11) ? 1'b1 : 1'b0;
        end

    end

end

wire [6:0] addr1, addr2, addr3, addr4;
assign addr1 = (ren1 == 1'b1) ? raddr[6:0] : (wen1 == 1'b1) ?
waddr[6:0] : 4'b0;
assign addr2 = (ren2 == 1'b1) ? raddr[6:0] : (wen2 == 1'b1) ?
waddr[6:0] : 4'b0;
assign addr3 = (ren3 == 1'b1) ? raddr[6:0] : (wen3 == 1'b1) ?
waddr[6:0] : 4'b0;
assign addr4 = (ren4 == 1'b1) ? raddr[6:0] : (wen4 == 1'b1) ?
waddr[6:0] : 4'b0;


Memory m1(clk, ren1, wen1, addr1, din, out1);
Memory m2(clk, ren2, wen2, addr2, din, out2);
Memory m3(clk, ren3, wen3, addr3, din, out3);
Memory m4(clk, ren4, wen4, addr4, din, out4);

always @(*) begin
```

```
    if(ren == 1'b1) begin
        if(ren1 == 1'b1) begin
        dout = out1;
        end
        else if(ren2 == 1'b1) begin
            dout = out2;
        end
        else if(ren3 == 1'b1) begin
            dout = out3;
        end
        else if(ren4 == 1'b1)begin
            dout = out4;
        end
    end
end

endmodule


module Multi_Bank_Memory (clk, ren, wen, waddr, raddr, din, dout);
input clk;
input ren, wen;
input [11-1:0] waddr;
input [11-1:0] raddr;
input [8-1:0] din;
output [8-1:0] dout;


reg ren1, ren2, ren3, ren4, wen1, wen2, wen3, wen4;
wire [7:0] out1, out2, out3, out4;
reg [7:0] dout;

always @(*) begin
    if(ren == 1'b1) begin
        ren1 = (raddr[10:9] == 2'b00) ? 1'b1 : 1'b0;
        ren2 = (raddr[10:9] == 2'b01) ? 1'b1 : 1'b0;
        ren3 = (raddr[10:9] == 2'b10) ? 1'b1 : 1'b0;
        ren4 = (raddr[10:9] == 2'b11) ? 1'b1 : 1'b0;
    end
    else begin
        ren1 = 1'b0;
        ren2 = 1'b0;
        ren3 = 1'b0;
        ren4 = 1'b0;
    end

    if(wen == 1'b1) begin
        if(ren == 1'b1 && (raddr[10:9] == waddr[10:9])) begin
            wen1 = 1'b0;
            wen2 = 1'b0;
            wen3 = 1'b0;
            wen4 = 1'b0;
        end
        else begin
            wen1 = (waddr[10:9] == 2'b00) ? 1'b1 : 1'b0;
            wen2 = (waddr[10:9] == 2'b01) ? 1'b1 : 1'b0;
            wen3 = (waddr[10:9] == 2'b10) ? 1'b1 : 1'b0;
            wen4 = (waddr[10:9] == 2'b11) ? 1'b1 : 1'b0;
        end
    end
end



// Bank_Memory(clk, ren, wen, waddr, raddr, din, dout);
Bank_Memory m1(clk, ren1, wen1, waddr, raddr, din, out1);
Bank_Memory m2(clk, ren2, wen2, waddr, raddr, din, out2);
Bank_Memory m3(clk, ren3, wen3, waddr, raddr, din, out3);
Bank_Memory m4(clk, ren4, wen4, waddr, raddr, din, out4);

always @(*) begin
    if(ren == 1'b1) begin
        if(ren1 == 1'b1) begin
        dout = out1;
        end
        else if(ren2 == 1'b1) begin
            dout = out2;
        end
        else if(ren3 == 1'b1) begin
            dout = out3;
        end
        else if(ren4 == 1'b1)begin
            dout = out4;
        end
    end

end

endmodule
```

How Do I Design Testbench:

Nothing fancy, I pass in two input data to two different address, doing write and read operation on them respectively.

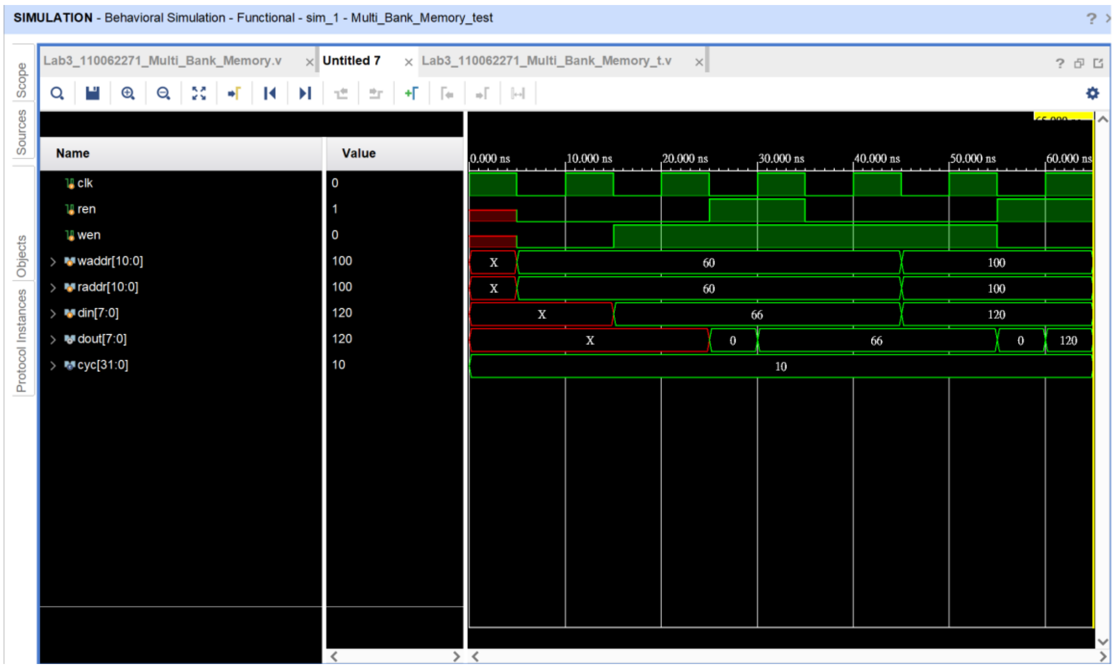
Also, I add additional test to try to read a value from an empty memory.

Here's the testbench code:

```
`timescale 1ns/1ps

module Multi_Bank_Memory_test;
reg clk = 1'b1;
reg ren, wen;
reg [11-1:0] waddr;
reg [11-1:0] raddr;
reg [8-1:0] din;
wire [8-1:0] dout;

Multi_Bank_Memory M1(clk, ren, wen, waddr, raddr, din, dout);

parameter cyc = 10;

always#(cyc/2)clk = !clk;
```

```verilog
initial begin
        @(negedge clk)
        ren = 0;
        wen = 0;
        waddr = 11'd60;
        raddr = 11'd60;

        @(negedge clk)
        wen = 1'b1;
        din = 8'd66;

        @(negedge clk)
        // Test read operation
        ren = 1'b1;

        @(negedge clk)
        ren = 1'b0;

        @(negedge clk)
        waddr = 11'd100;
        raddr = 11'd100;
        din = 8'd120;
        wen = 1'b1;

        @(negedge clk)
        wen = 1'b0;
        ren = 1'b1;

        @(negedge clk)


        $finish;
    end



endmodule
```

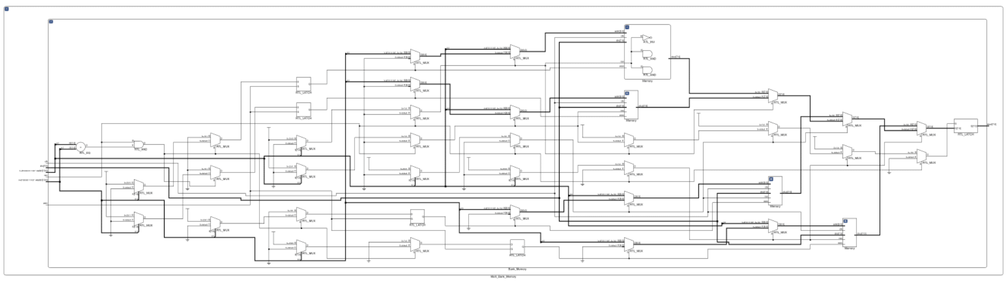
Here’s the waveform result:


SIMULATION - Behavioral Simulation - Functional - sim_1 - Multi_Bank_Memory_test
Lab3_110062271_Multi_Bank_Memory.v
Untitled 7
Lab3_110062271_Multi_Bank_Memory_t.v
Scope
Sources
Objects
Protocol Instances
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns

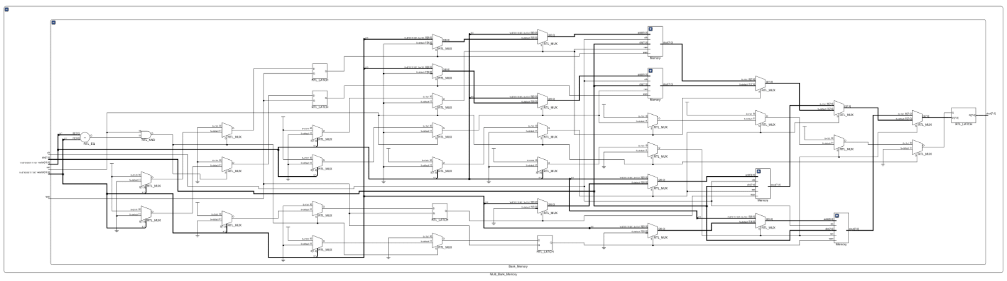
Name	Value
clk	0
ren	1
wen	0
waddr[10:0]	100
raddr[10:0]	100
din[7:0]	120
dout[7:0]	120
cyc[31:0]	10

Diagram:

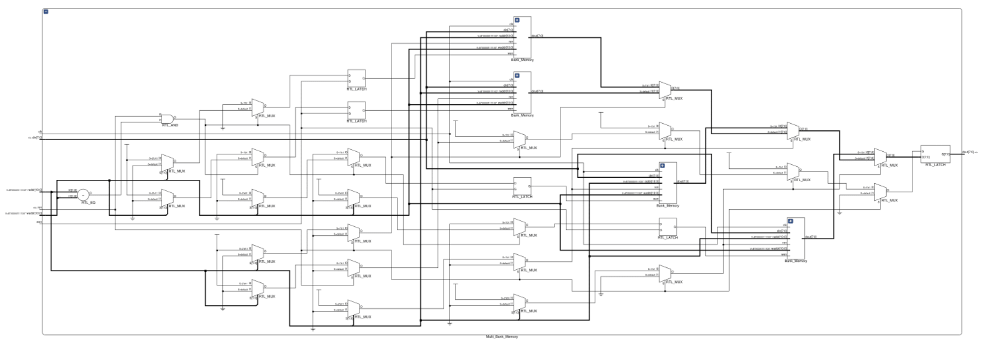
Memory Module:



Bank_Memory Module:



Multi_Bank:



Advanced_Question_4: Round-Robin FIFO Arbiter

How Do I Design Source Code:

I reuse the FIFO_8 module from the previous question and followed the spec restrictions and pass ren and wen accordingly to each FIFO_8 and using a 2-bit counter to determine which FIFO_8 module to read.

Additionally, I revised the FIFO_8 code to make it do the write operation when ren == 1’b1 && wen == 1’b1 to make the example wave form result showed on the spec pdf possible.

Here’s the source code:

```verilog
`timescale 1ns/1ps

module FIFO_8(clk, rst_n, wen, ren, din, dout, error);
input clk;
input rst_n;
input wen, ren;
input [8-1:0] din;
output [8-1:0] dout;
output error;

reg [7:0] dout;
reg error;

reg [7:0] mem [7:0];
reg full, empty;
reg [2:0] read_ptr, write_ptr;

always @(posedge clk) begin
    if(~rst_n) begin
        read_ptr <= 3'd0;
        write_ptr <= 3'd0;
        dout <= 8'd0;
        error <= 1'b0;
        empty <= 1'b1;
        full <= 1'b0;
    end
    else begin
        if(ren && !wen) begin
            if(empty) error <= 1'b1;
            else begin
                error <= 1'b0;
                dout <= mem[read_ptr];
                full <= 1'b0;
                if(read_ptr == 3'd7) begin
                    if(write_ptr == 3'd0) empty <= 1'b1;
                    else empty <= 1'b0;
                end
                else begin
                    if(read_ptr + 3'd1 == write_ptr) empty <= 1'b1;
                    else empty <= 1'b0;
                end
                read_ptr <= (read_ptr == 3'd7) ? 3'd0 : read_ptr + 3'd1;
            end
        end
        else if((wen && !ren) || (ren && wen)) begin
            if(full) error <= 1'b1;
            else begin
                error <= 1'b0;
                mem[write_ptr] <= din;
                empty <= 1'd0;
                if(write_ptr == 3'd7) begin
                    if(read_ptr == 3'd0) full <= 1'b1;
                    else full <= 1'b0;
                end
                else begin
                    if(write_ptr + 3'd1 == read_ptr) full <= 1'b1;
                    else full <= 1'b0;
                end
                write_ptr <= (write_ptr == 3'd7) ? 3'd0 : write_ptr +
3'd1;
```

```
            end
        end
        else begin
            if(empty) begin
                dout <= 8'b0;
                error <= 1'b1;
            end
            else begin
                dout <= dout;
                error <= 1'b0;
            end
        end
    end
end

endmodule

module Round_Robin_FIFO_Arbiter(clk, rst_n, wen, a, b, c, d, dout,
valid);
input clk;
input rst_n;
input [4-1:0] wen;
input [8-1:0] a, b, c, d;
output [8-1:0] dout;
output valid;

reg[7:0] dout;
reg valid;

reg ren1, ren2, ren3, ren4;
wire err1, err2, err3, err4;
reg [1:0] counter;
reg [7:0] result;
reg [1:0] sel;
wire [7:0] out1, out2, out3, out4;


always @(posedge clk) begin
    if(~rst_n) begin
```

```
        dout <= 8'b0;
        valid <= 1'b0;
        counter <= 2'b00;
    end
    else begin
        case(counter)
        2'b00:
            if(wen[0] || err1) begin
                valid <= 1'b0;
                dout <= 8'b0;
            end
            else begin
                valid <= 1'b1;
                sel <= 2'b00;
            end
        2'b01:
            if(wen[1] || err2) begin
                valid <= 1'b0;
                dout <= 8'b0;
            end
            else begin
                valid <= 1'b1;
                sel <= 2'b01;
            end
        2'b10:
            if(wen[2] || err3) begin
                valid <= 1'b0;
                dout <= 8'b0;
            end
            else begin
                valid <= 1'b1;
                sel <= 2'b10;
            end
        default:
            if(wen[3] || err4) begin
                valid <= 1'b0;
                dout <= 8'b0;
            end
            else begin
                valid <= 1'b1;
                sel <= 2'b11;
            end
        endcase
        counter <= (counter == 2'b11) ? 2'b00 : (counter + 2'b01);
    end
end

always @(posedge clk) begin
    if(~rst_n) begin
        ren1 <= 1'b1;
        ren2 <= 1'b0;
        ren3 <= 1'b0;
        ren4 <= 1'b0;
        result <= out1;
    end
    else begin
        ren1 <= (counter == 2'b11) ? 1'b1 : 1'b0;
        ren2 <= (counter == 2'b00) ? 1'b1 : 1'b0;
        ren3 <= (counter == 2'b01) ? 1'b1 : 1'b0;
        ren4 <= (counter == 2'b10) ? 1'b1 : 1'b0;
    end
end

always @(*) begin
    case(sel)
        2'b00: dout = out1;
        2'b01: dout = out2;
        2'b10: dout = out3;
        2'b11: dout = out4;
    endcase
end
```

How Do I Design Testbench:

I implemented the example wave form on the spec pdf to my testbench, and the testing result is the same as the example.

Here’s the testbench:

```
module Round_Robin_FIFO_Arbiter_test;
  reg clk;
  reg rst_n;
  reg [4-1:0] wen;
  reg [8-1:0] a, b, c, d;
  wire [8-1:0] dout;
  wire valid;

  // Instantiate the module
  Round_Robin_FIFO_Arbiter r1(clk, rst_n, wen, a, b, c, d, dout, valid);

  // Clock generation
  always begin
    #5 clk = ~clk;
  end

  // Stimulus
  initial begin
    clk = 1'b0;
    rst_n = 1'b0;

    @(negedge clk)
        rst_n = 1'b1;
        wen = 4'b1111;
        a = 8'd87;
        b = 8'd56;
        c = 8'd9;
        d = 8'd13;

    @(negedge clk)
        wen = 4'b1000;
        a = 8'd0;
        b = 8'd0;
        c = 8'd0;
        d = 8'd85;
    @(negedge clk)
        wen = 4'b0100;
        c = 8'd139;
        d = 8'd0;
    @(negedge clk)
        wen = 4'b0000;
        c = 8'd0;
    @(negedge clk)
        wen = 4'b0000;
    @(negedge clk)
        wen = 4'b0000;
    @(negedge clk)
        wen = 4'b0001;
        a = 8'd51;
    @(negedge clk)
        wen =4'b0000;
        a = 8'd0;
    @(negedge clk)
        wen =4'b0000;
    @(negedge clk)
        wen =4'b0000;
    @(negedge clk)
        wen =4'b0000;
  end
endmodule
```

Here’s the wave form graph: (numbers in decimal)

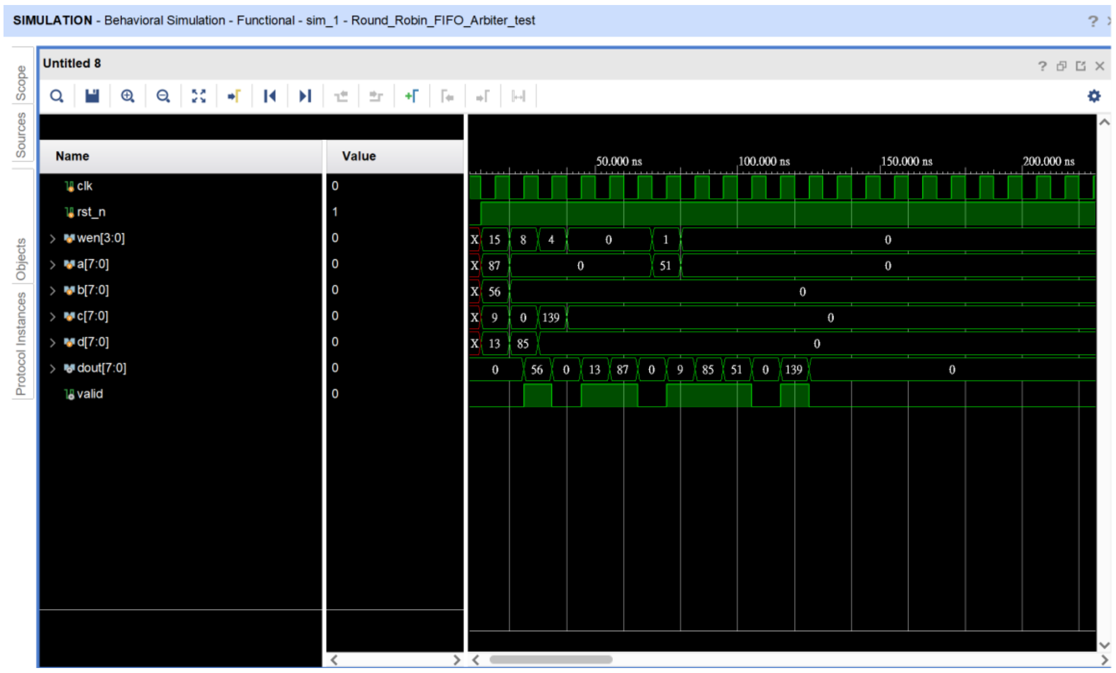
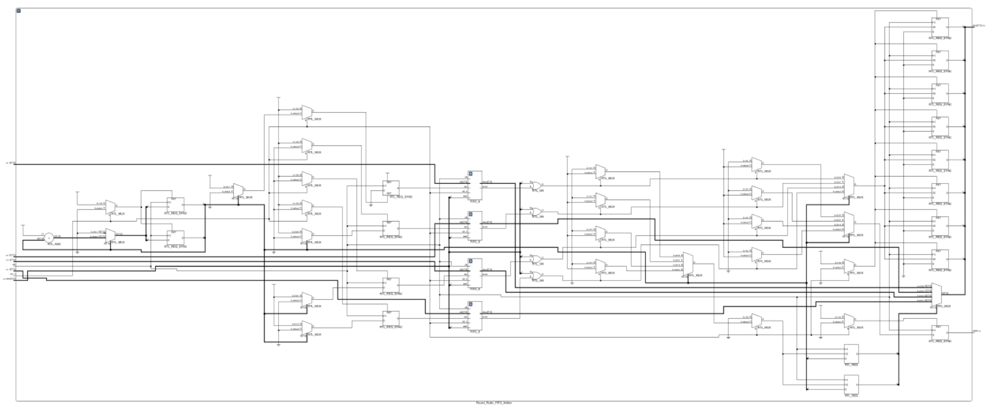
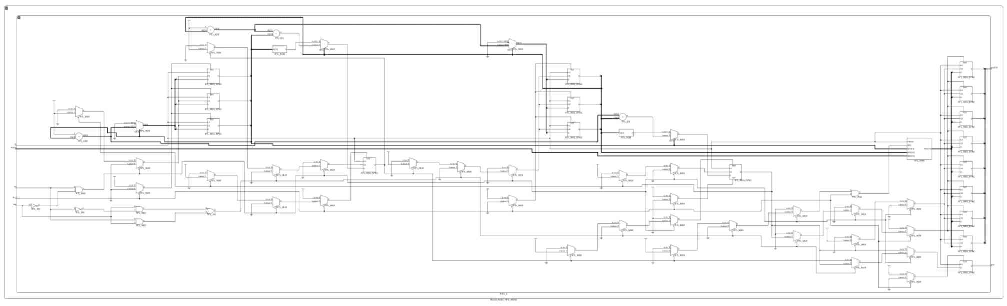


Diagram:

Round-Robin FIFO Arbiter:



FIFO_8: (modified version)



Advanced Question 5: 4-bit Parameterized Ping-Pong Counter

How Do I Design Source Code:

I reuse the Ping-Pong Counter module I previously designed, but I modified the upper bound and the lower bound, making them input [4-1:0] max and input [4-1:0] min respectively.

Also, I check whether ((flip == 1'b1) && (out < max) && (out > min)), if so I flip the direction bit.

Here’s the source code:

```verilog
`timescale 1ns/1ps

module Parameterized_Ping_Pong_Counter (clk, rst_n, enable, flip, max,
min, direction, out);
input clk, rst_n;
input enable;
input flip;
input [4-1:0] max;
input [4-1:0] min;
output direction;
output [4-1:0] out;

reg direction;
reg next_direction;
reg [3:0] out;
reg [3:0] next_out;

always @(posedge clk) begin
    if(!rst_n) begin
        out <= min;
        direction <= 1'b1;
    end
    else begin
        out <= next_out;
        direction <= next_direction;
    end
```

```
end

always @(*) begin
    if(enable) begin
        if(((max > min) && (out == max || (out == min && direction ==
1'b0))) || ((flip == 1'b1) && (out < max) && (out > min))) begin
            next_direction = !direction;
            if(next_direction == 1'b0) next_out = out - 1'b1;
            else next_out = out + 1'b1;
        end
        else if((max == min) || (max < min) || (out > max) || (out <
min)) begin
            next_out = out;
            next_direction = direction;
        end
        else begin
            next_direction = direction;
            if(next_direction == 1'b0) next_out = out - 1'b1;
            else next_out = out + 1'b1;
        end


    end
    else begin
        next_out = out;
        next_direction = direction;
    end
end


endmodule
```

How Do I Design Testbench:

I make 3 testbench cases, each of them corresponding to the example waveform on the spec pdf.

Only one case is enabled at a time, other cases are commented when not being tested.

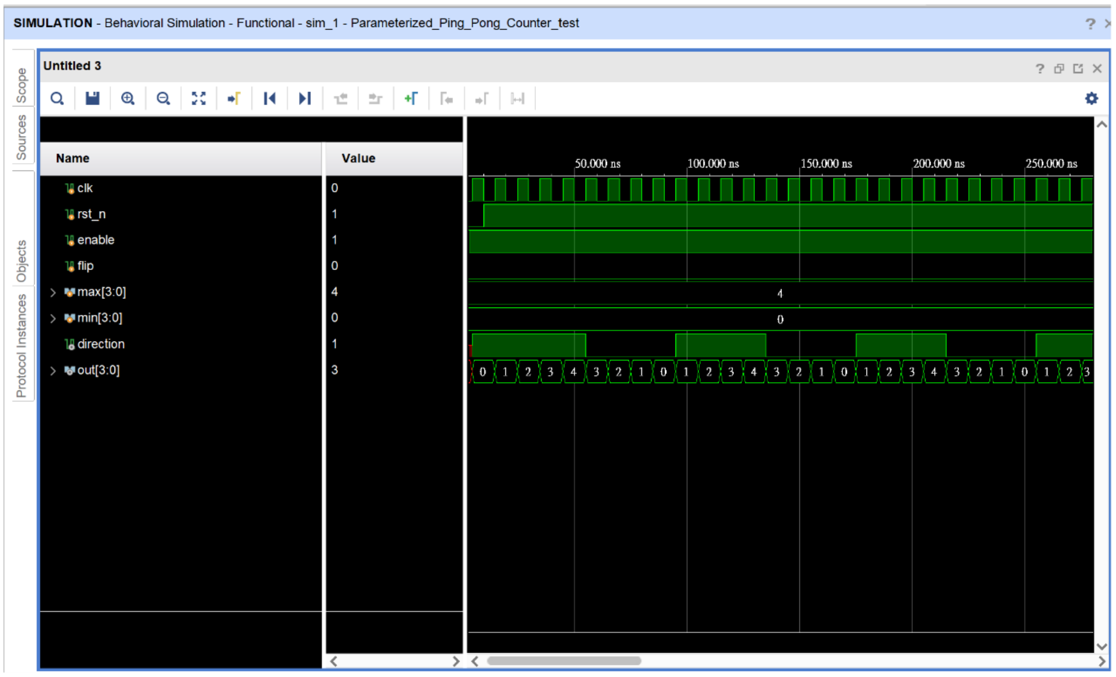
Here's the testbench code:

```
`timescale 1ns/1ps

module Parameterized_Ping_Pong_Counter_test;
reg clk, rst_n, enable, flip;
reg [3:0] max, min;
wire direction;
wire [3:0] out;

always #5 clk = !clk;

// Parameterized_Ping_Pong_Counter (clk, rst_n, enable, flip, max, min,
direction, out);
Parameterized_Ping_Pong_Counter P1(clk, rst_n, enable, flip, max, min,
direction, out);

/* testcase 1 An example waveform where flip is set to 1'b0
    and enable is set to 1'b1
    In this example min = 4'd0 and max = 4'd4 */
/*
initial begin
    clk = 1'b0;
    rst_n = 1'b0;
    enable = 1'b1;
    flip = 1'b0;
    min = 4'd0;
    max = 4'd4;

    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
end  */


/* testcase 2 An example waveform where there is one flip and enable is
set  to 1'b1
In this example min = 4’d0 and max = 4'd4 */
/*
initial begin
    clk = 1'b0;
    rst_n = 1'b0;
    enable = 1'b1;
    flip = 1'b0;
    min = 4'd0;
    max = 4'd4;

    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        flip = 1'b1;
    @(negedge clk)
        flip = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
end */

/* testcase 3 An example waveform where there are two flips and enable
is
set to 1'b1
In this example min = 4'd0 and max = 4'd4 */
initial begin
    clk = 1'b0;
    rst_n = 1'b0;
    enable = 1'b1;
    flip = 1'b0;
    min = 4'd0;
    max = 4'd4;

    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        flip = 1'b1;
    @(negedge clk)
        flip = 1'b0;
    @(negedge clk)
        flip = 1'b1;
    @(negedge clk)
        flip = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
```

```
end
endmodule
```

Here's the wave form diagram:

Case 1:

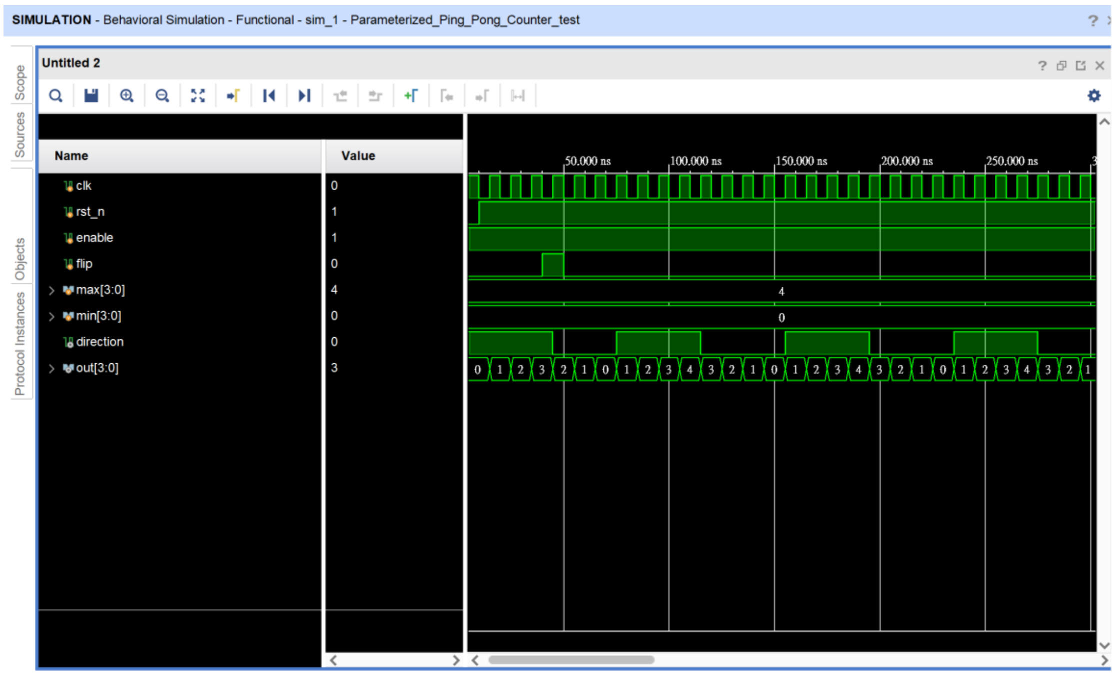


SIMULATION - Behavioral Simulation - Functional - sim_1 - Parameterized_Ping_Pong_Counter_test

Untitled 3

Name	Value
clk	0
rst_n	1
enable	1
flip	0
max[3:0]	4
min[3:0]	0
direction	1
out[3:0]	3

Case 2:

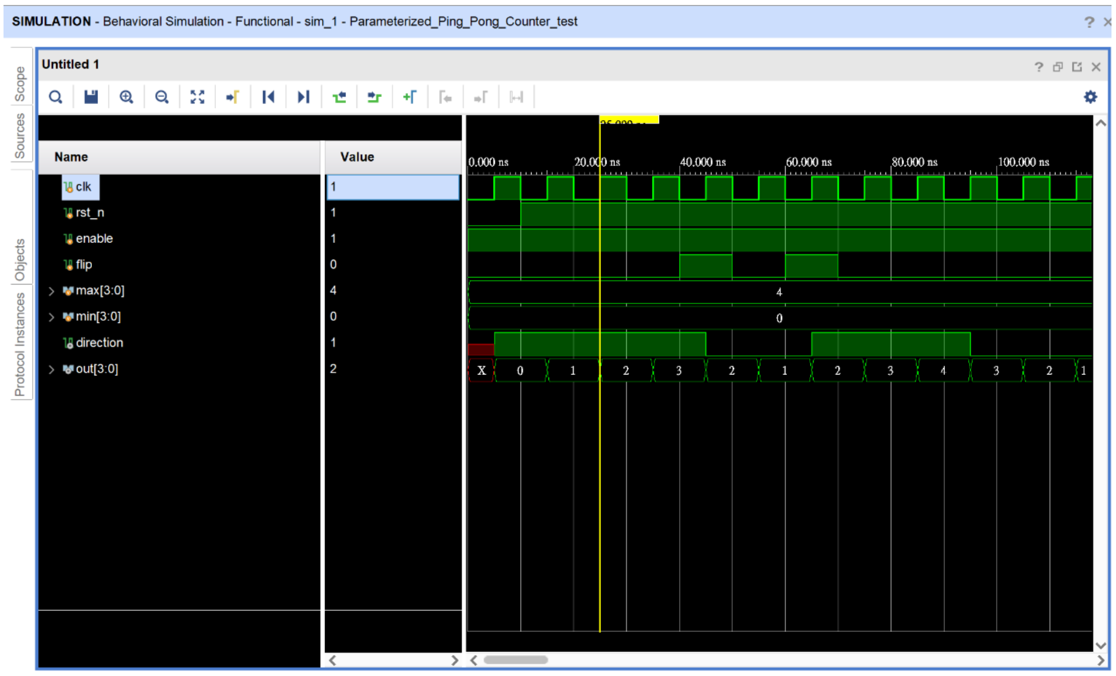


SIMULATION - Behavioral Simulation - Functional - sim_1 - Parameterized_Ping_Pong_Counter_test

Untitled 2

Name	Value
clk	0
rst_n	1
enable	1
flip	0
max[3:0]	4
min[3:0]	0
direction	0
out[3:0]	3

Case 3:

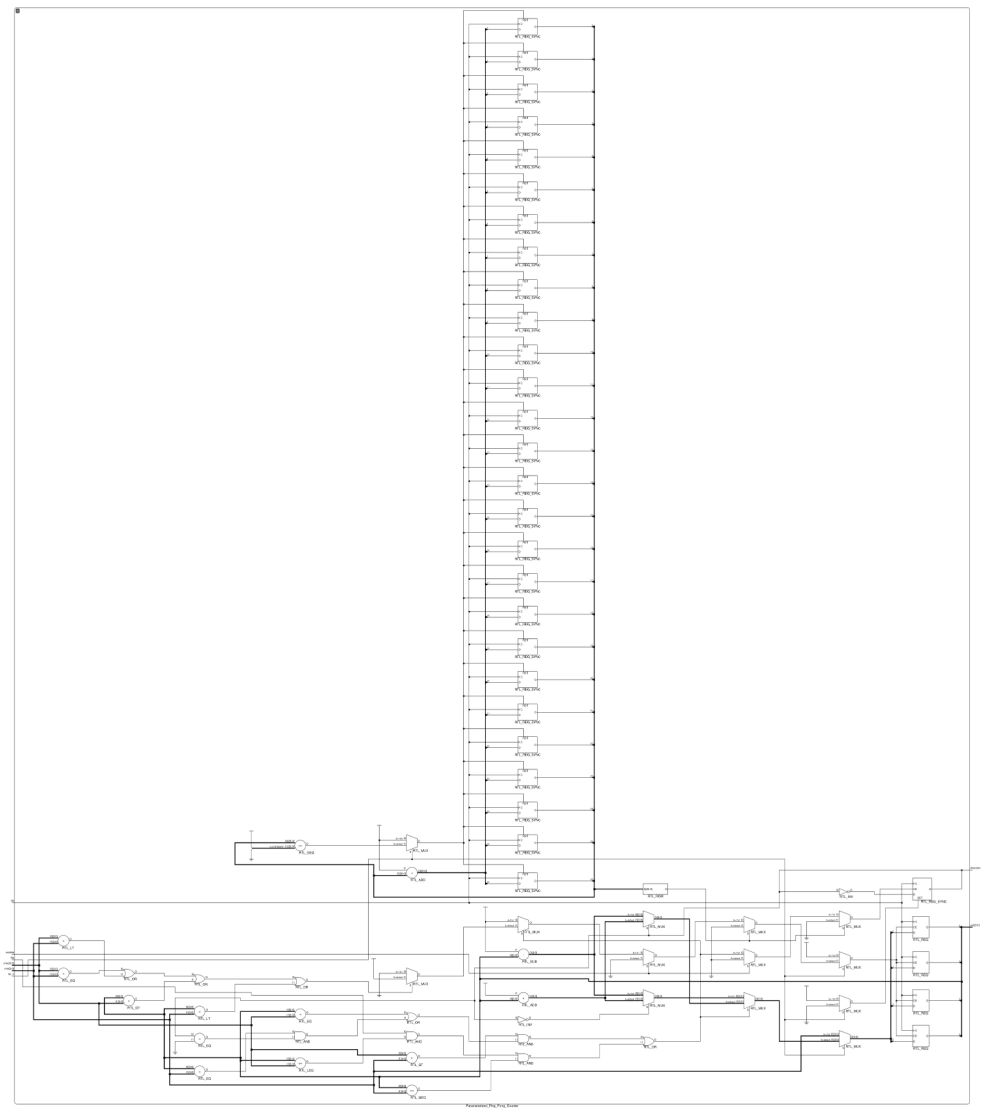

SIMULATION - Behavioral Simulation - Functional - sim_1 - Parameterized_Ping_Pong_Counter_test

Untitled 1

Name	Value
clk	1
rst_n	1
enable	1
flip	0
max[3:0]	4
min[3:0]	0
direction	1
out[3:0]	2

Diagram:

Parameterized_Ping_Pong_Counter:



FPGA Implementation

How I Design Source Code:

I design the one_pulse and debounced and incorporate in a module called one_signal(clk, pb, pb_one_pulse).

I also referenced this website www.fpga4student.com about how to blink the LED and update the result to the LED roughly every one second.

I also modified the Parameterized Ping Pong cCounter module used to let it flip the direction but when ((flip == 1'b1) && (out <= max) && (out >= min)) is true.

Here's the source code:

```
module Parameterized_Ping_Pong_Counter (clk, rst_n, enable, flip, max,
min, direction, out);
input clk, rst_n;
input enable;
input flip;
input [4-1:0] max;
input [4-1:0] min;
output direction;
output [4-1:0] out;

reg direction;
reg next_direction;
reg [3:0] out;
reg [3:0] next_out;

reg[26:0] one_second_counter;
wire one_second_enable;

always @(posedge clk) begin
    if(~rst_n) one_second_counter <= 0;
    else begin
        if(one_second_counter >= 99999999) one_second_counter <= 0;
        else one_second_counter <= one_second_counter + 1;
    end
```

```
end


assign one_second_enable = (one_second_counter == 99999999) ? 1'b1 :
1'b0;


always @(posedge clk) begin
    if(!rst_n) begin
        out <= min;
        direction <= 1'b1;
    end
    else if(one_second_enable)begin
        out <= next_out;
        direction <= next_direction;
    end
end

always @(*) begin
    if(enable) begin // fpga modified version
        if(((max > min) && (out == max || (out == min && direction ==
1'b0))) || ((flip == 1'b1) && (out <= max) && (out >= min))) begin
            next_direction = !direction;
            if(next_direction == 1'b0) next_out = out - 1'b1;
            else next_out = out + 1'b1;
        end
        else if((max == min) || (max < min) || (out > max) || (out <
min)) begin
            next_out = out;
            next_direction = direction;
        end
        else begin
            next_direction = direction;
            if(next_direction == 1'b0) next_out = out - 1'b1;
            else next_out = out + 1'b1;
        end
    end
    else begin
        next_out = out;
```

```verilog
        next_direction = direction;
    end
end


endmodule

module debounce(clk, pb, pb_debounced);
input clk, pb;
output pb_debounced;
reg [3:0] temp;

always @(posedge clk) begin
    temp[3:1] <= temp[2:0];
    temp[0] <= pb;
end

assign pb_debounced = (temp == 4'b1111) ? 1'b1 : 1'b0;

endmodule

module onepulse(clk, pb_debounced, pb_one_pulse);
input clk, pb_debounced;
output reg pb_one_pulse;

reg pb_debounced_delay;

always @(posedge clk) begin
    pb_one_pulse <= pb_debounced & (!pb_debounced_delay);
    pb_debounced_delay <= pb_debounced;
end

endmodule

module one_signal(clk, pb, pb_one_pulse);
input clk, pb;
output pb_one_pulse;

wire pb_debounced;
```

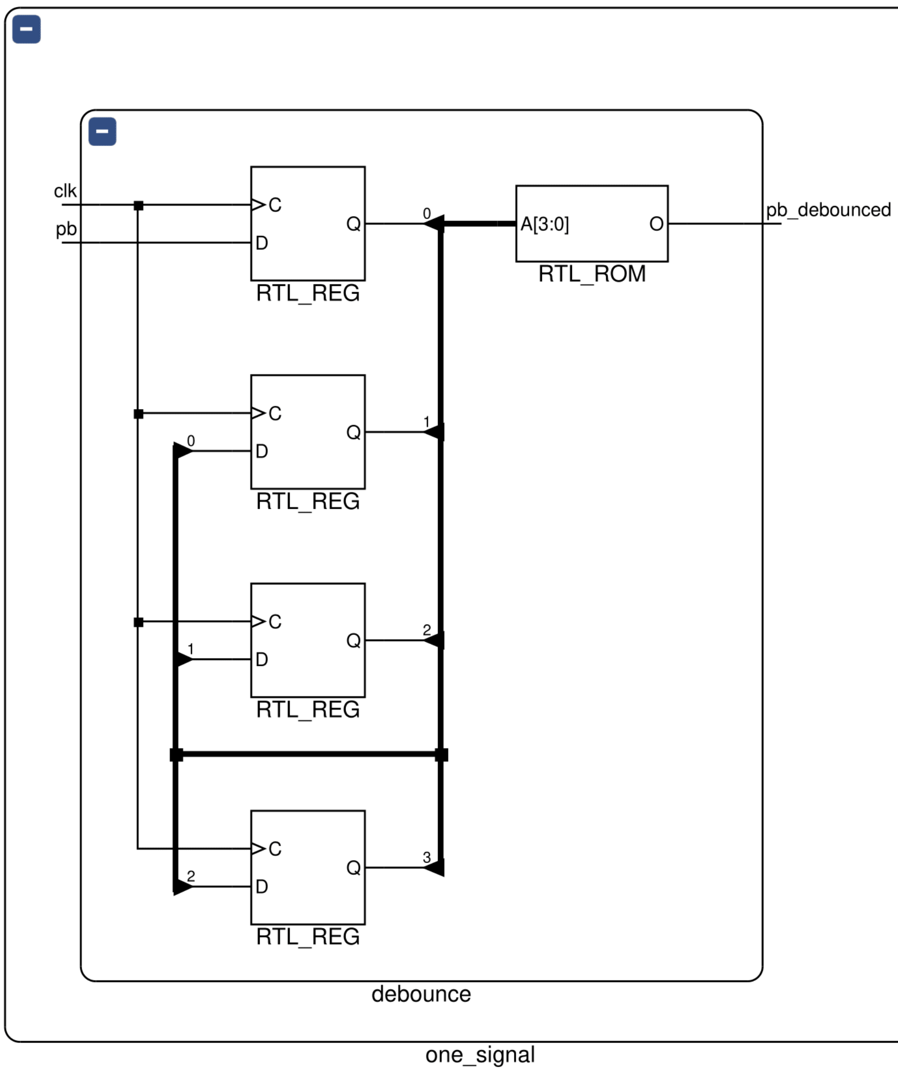
```
debounce d1(clk, pb, pb_debounced);
onepulse d2(clk, pb_debounced, pb_one_pulse);

endmodule


module  FPGA_7segemnt(SW, clk, btn_up, btn_down, LED_out, anode_digit);
input [8:0] SW;
input clk, btn_up, btn_down;
wire rst_n, flip, direction;

reg [19:0] refresh_counter;
wire [1:0] sel;

reg [26:0] one_second_counter;
wire one_second_enable;
wire [3:0] dout;
reg reset_signal;
output reg [6:0] LED_out;
output reg [3:0] anode_digit;

one_signal d1(clk, btn_up, rst_n);
one_signal d2(clk, btn_down, flip);

// Parameterized_Ping_Pong_Counter (clk, rst_n, enable, flip, max, min,
direction, out);
Parameterized_Ping_Pong_Counter C1(clk, !rst_n, SW[8], flip, SW[7:4],
SW[3:0], direction, dout);


always @(posedge clk or posedge rst_n) begin
    if(rst_n == 1) refresh_counter <= 0;
    else refresh_counter <= refresh_counter + 1;
end

assign sel = refresh_counter[19:18];

always @(sel) begin
```

```
        if(sel == 2'b00) begin
            anode_digit = 4'b1110;
            if(direction == 1'b0) LED_out = 7'b1100011;
            else LED_out = 7'b0011101;
        end
        else if(sel == 2'b01) begin
            anode_digit = 4'b1101;
            if(direction == 1'b0) LED_out = 7'b1100011;
            else LED_out = 7'b0011101;
        end
        else if(sel == 2'b10) begin
            anode_digit = 4'b1011;
            case(dout)
                4'd0: LED_out = 7'b0000001;
                4'd1: LED_out = 7'b1001111;
                4'd2: LED_out = 7'b0010010;
                4'd3: LED_out = 7'b0000110;
                4'd4: LED_out = 7'b1001100;
                4'd5: LED_out = 7'b0100100;
                4'd6: LED_out = 7'b0100000;
                4'd7: LED_out = 7'b0001111;
                4'd8: LED_out = 7'b0000000;
                4'd9: LED_out = 7'b0000100;
                4'd10: LED_out = 7'b0000001;
                4'd11: LED_out = 7'b1001111;
                4'd12: LED_out = 7'b0010010;
                4'd13: LED_out = 7'b0000110;
                4'd14: LED_out = 7'b1001100;
                4'd15: LED_out = 7'b0100100;
                default: LED_out = 7'b0000000;
            endcase
        end
        else begin
            anode_digit = 4'b0111;
            if(dout >= 4'b1010) LED_out = 7'b1001111; // 1
            else LED_out = 7'b0000001; // 0
        end
end
```

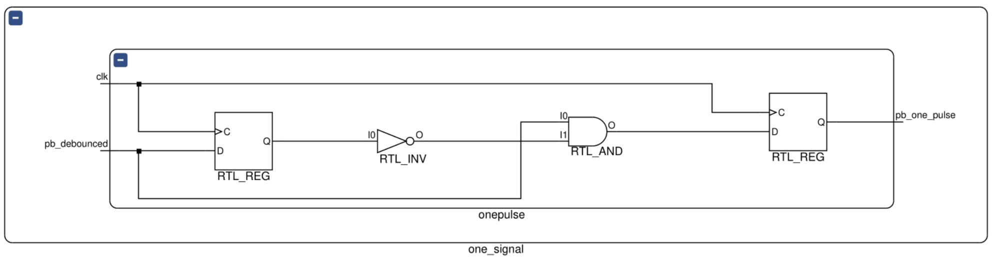
```



endmodule
```

Diagram:

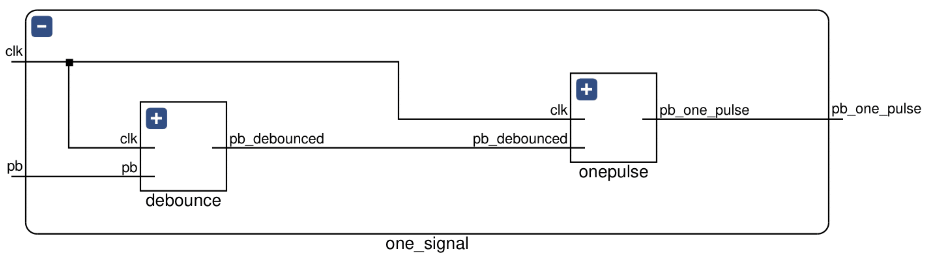
debounce:



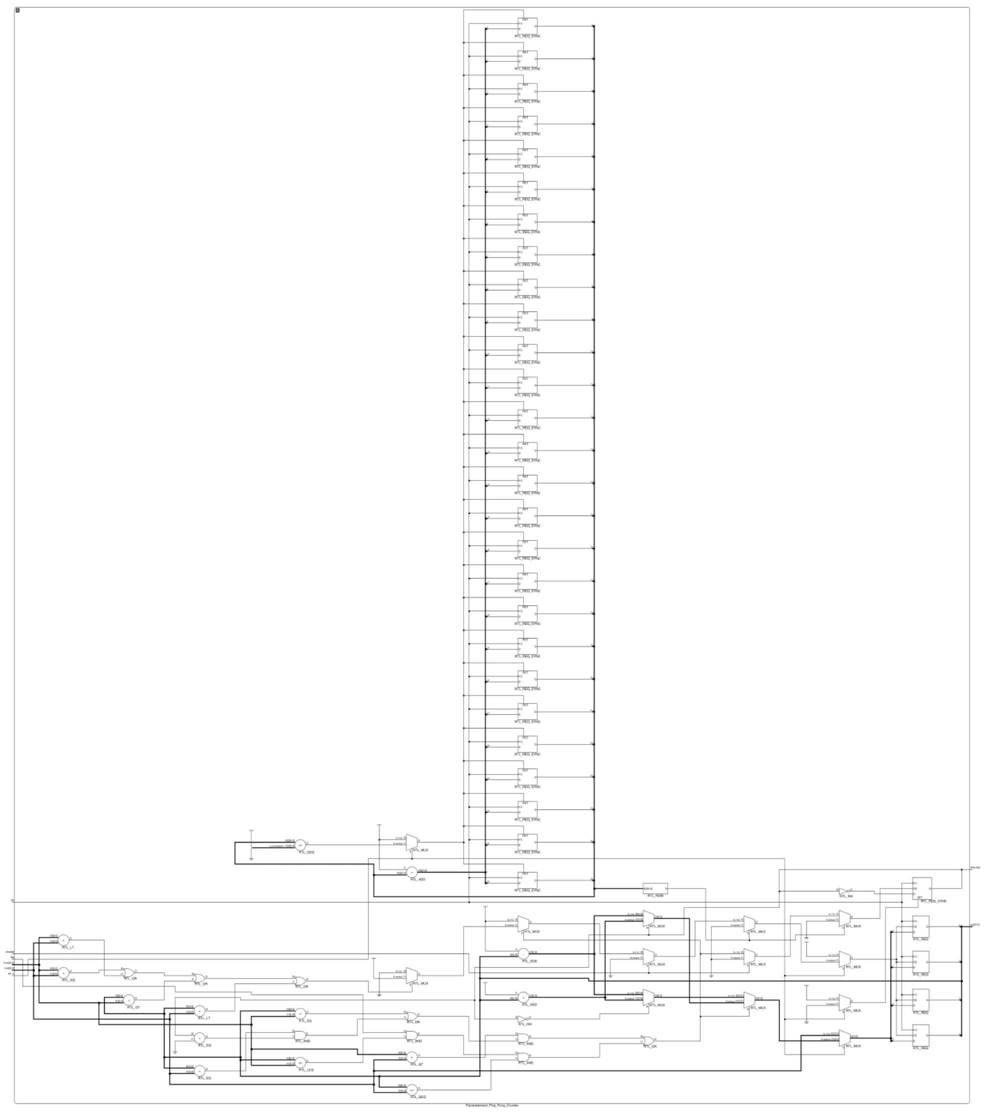
onepulse:


clk
pb_debounced
C
D
Q
RTL_REG
I0
O
RTL_INV
I0
I1
O
RTL_AND
C
D
Q
RTL_REG
pb_one_pulse
onepulse
one_signal

one_signal:


clk
pb
clk
pb
debounce
pb_debounced
pb_debounced
clk
onepulse
pb_one_pulse
pb_one_pulse
one_signal

Parameterized Ping Pong Counter:



FPGA_7segemnt: