Hardware Design Lab\_4 Report

Name:

**林奕為**

Department:

**CS 25**

Student’s ID:

**110062271**

Contents

[What I have learned from the Lab 3](#_Toc149296643)

[Advanced Question 1: 4-bit Ping-Pong Counter 4](#_Toc149296644)

[How Do I Design Source Code: 4](#_Toc149296645)

[How Do I Design Testbench: 5](#_Toc149296646)

[Diagram: 8](#_Toc149296647)

[Advanced Question 2: First-In First Out (FIFO) Queue 9](#_Toc149296648)

[How Do I Design Source Code: 9](#_Toc149296649)

[How Do I Design Testbench: 11](#_Toc149296650)

[Diagram: 14](#_Toc149296651)

[Advanced Question 3: Multi-Bank Memory 15](#_Toc149296652)

[How Do I Design Source Code: 15](#_Toc149296653)

[How Do I Design Testbench: 20](#_Toc149296654)

[Diagram: 23](#_Toc149296655)

[Advanced\_Question\_4: Round-Robin FIFO Arbiter 26](#_Toc149296656)

[How Do I Design Source Code: 26](#_Toc149296657)

[How Do I Design Testbench: 31](#_Toc149296658)

[Diagram: 34](#_Toc149296659)

[Advanced Question 5: 4-bit Parameterized Ping-Pong Counter 36](#_Toc149296660)

[How Do I Design Source Code: 36](#_Toc149296661)

[How Do I Design Testbench: 37](#_Toc149296662)

[Diagram: 43](#_Toc149296663)

[FPGA Implementation 44](#_Toc149296664)

[How I Design Source Code: 44](#_Toc149296665)

[Diagram: 50](#_Toc149296666)

# 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

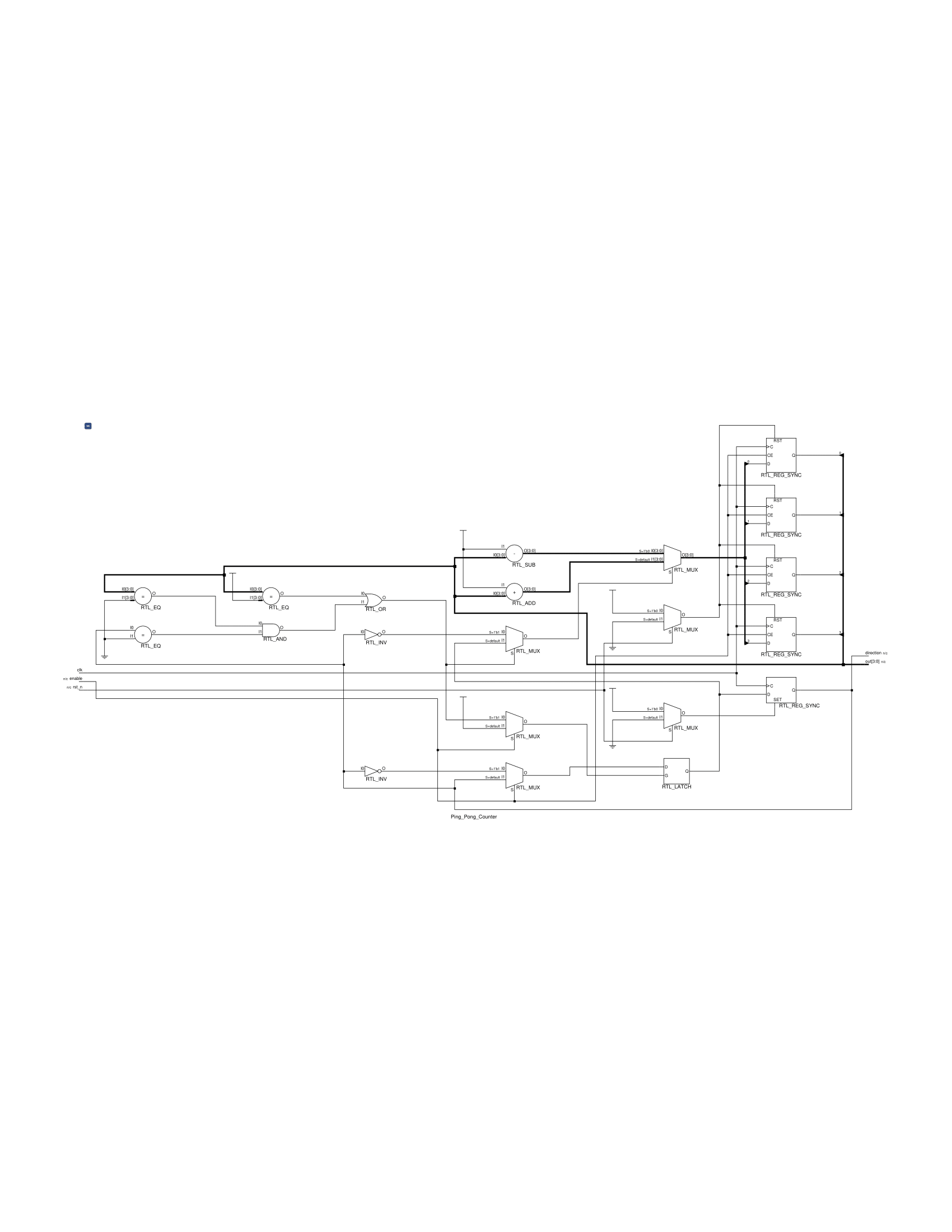
Here’s the partial waveform result:

一張含有 螢幕擷取畫面, 文字, 軟體, 多媒體軟體 的圖片

自動產生的描述

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:

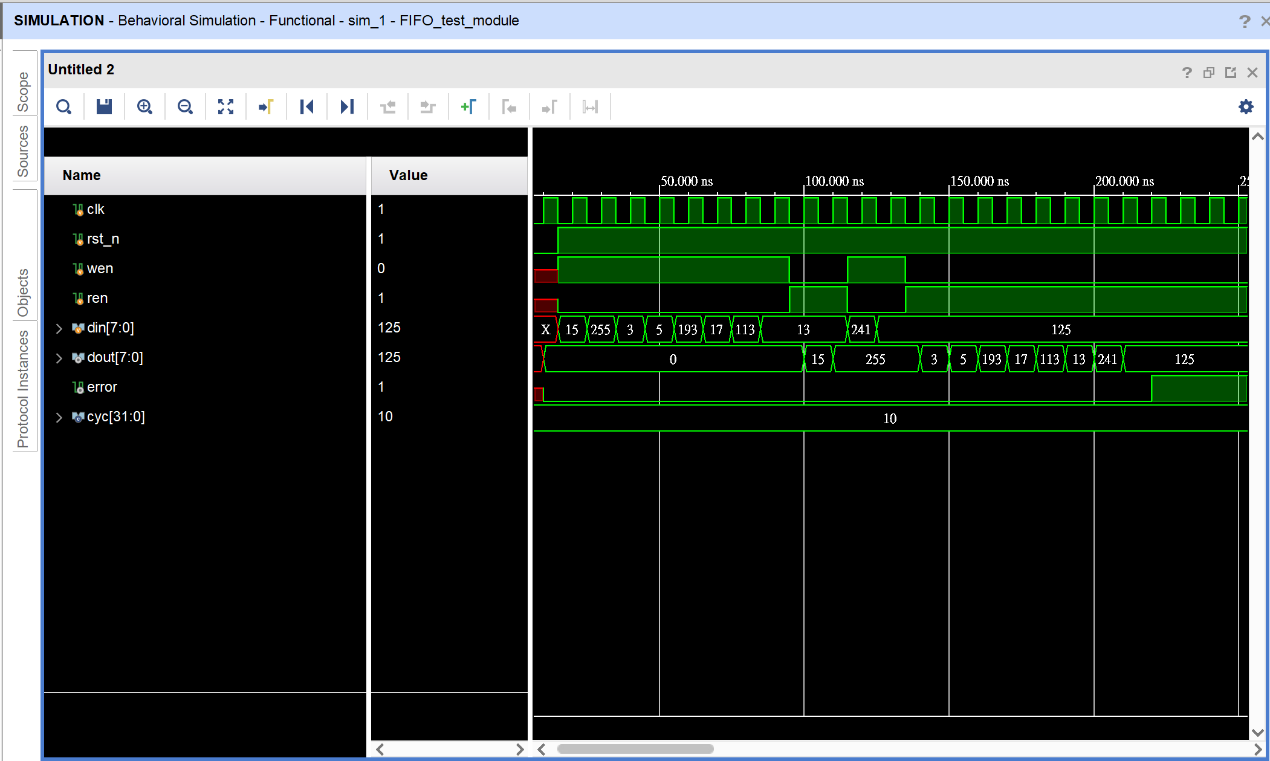


Diagram:

FIFO\_8 module:

一張含有 寫生, 圖畫, 圖表, 工程製圖 的圖片

自動產生的描述

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;

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:

一張含有 螢幕擷取畫面, 文字, 軟體, 陳列 的圖片

自動產生的描述

Diagram:

Memory Module:  
一張含有 寫生, 圖畫, 圖表, 工程製圖 的圖片

自動產生的描述

Bank\_Memory Module:

一張含有 寫生, 圖畫, 圖表, 行 的圖片

自動產生的描述

Multi\_Bank:

一張含有 寫生, 圖畫, 圖表, 工程製圖 的圖片

自動產生的描述

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:

`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

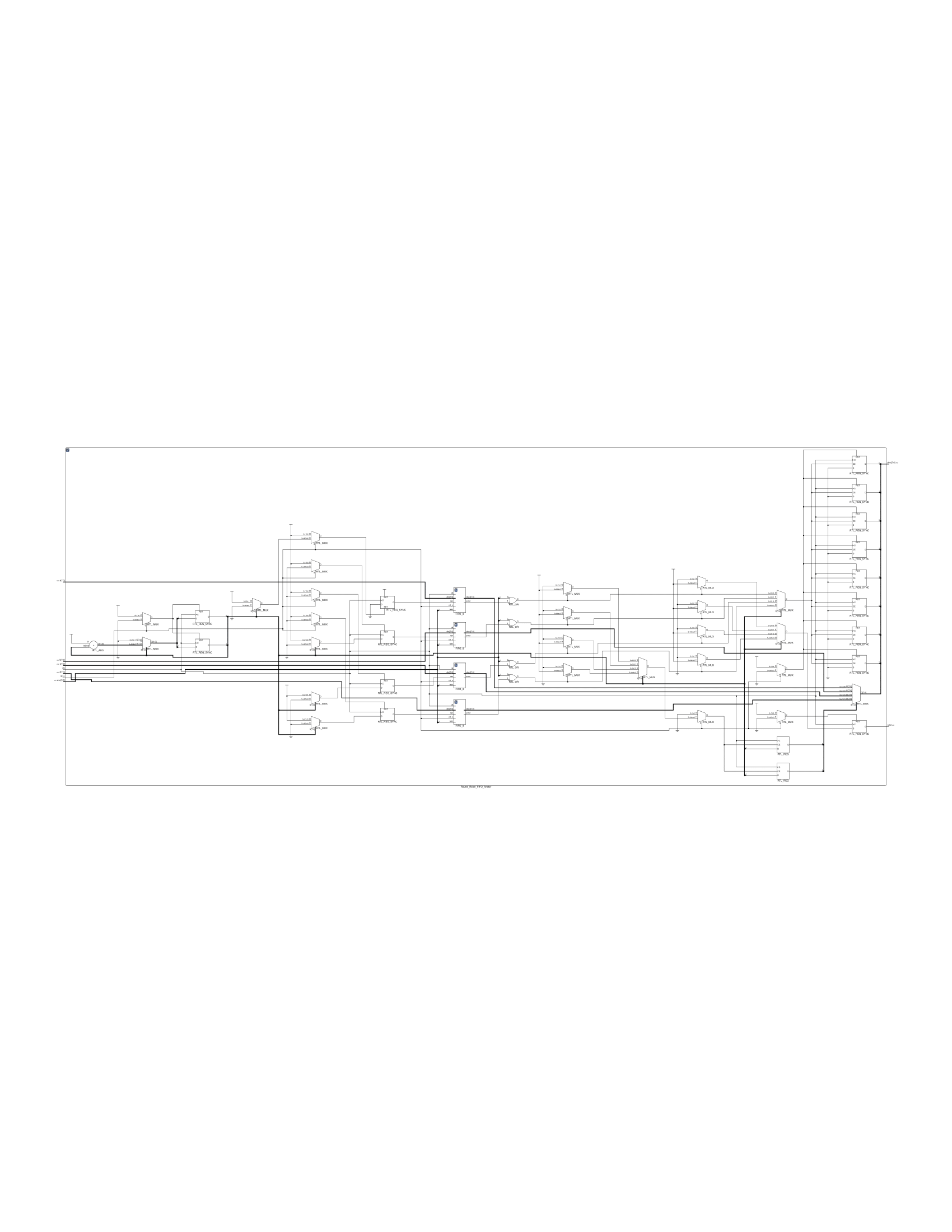
Here’s the wave form graph: (numbers in decimal)

一張含有 文字, 螢幕擷取畫面, 陳列, 軟體 的圖片

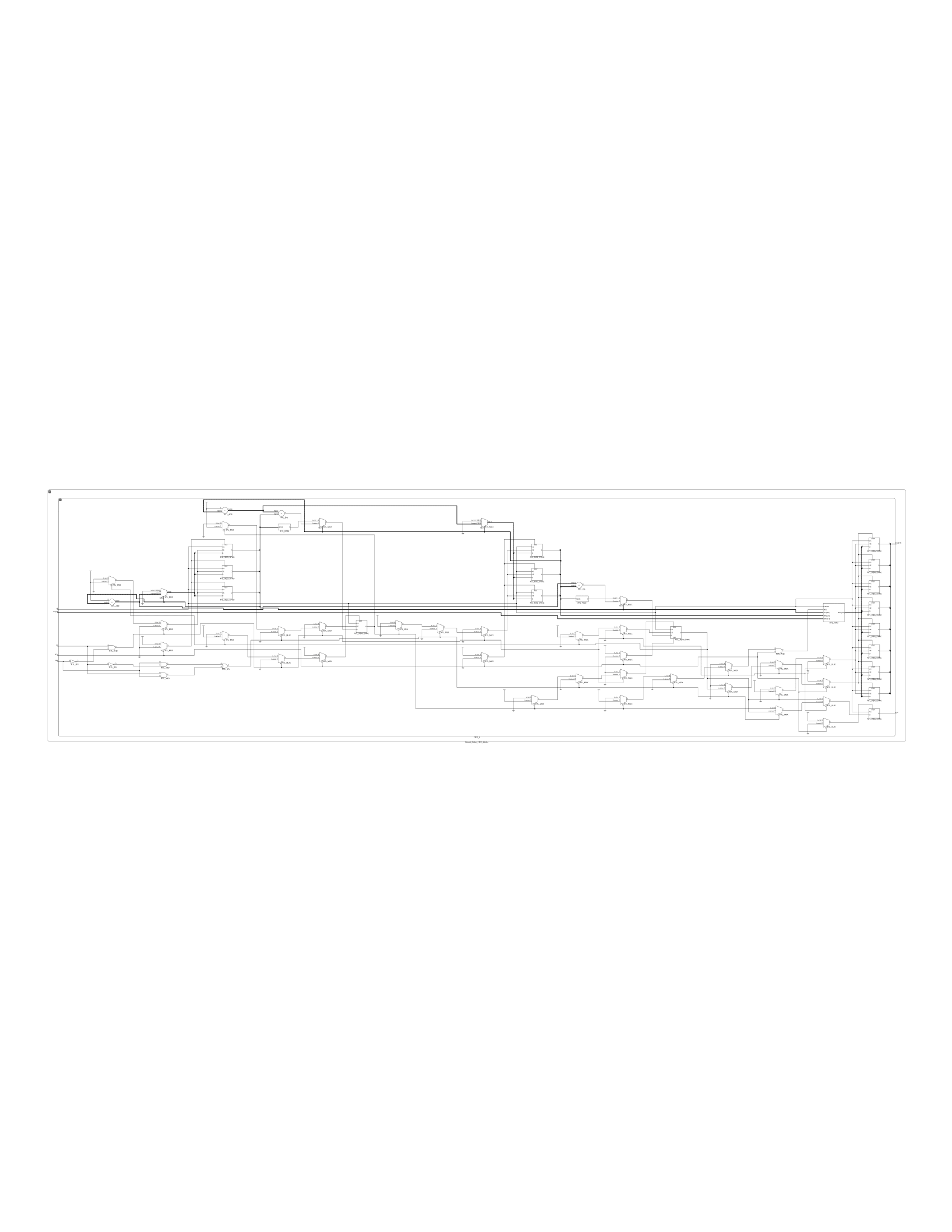
自動產生的描述

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:

`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)

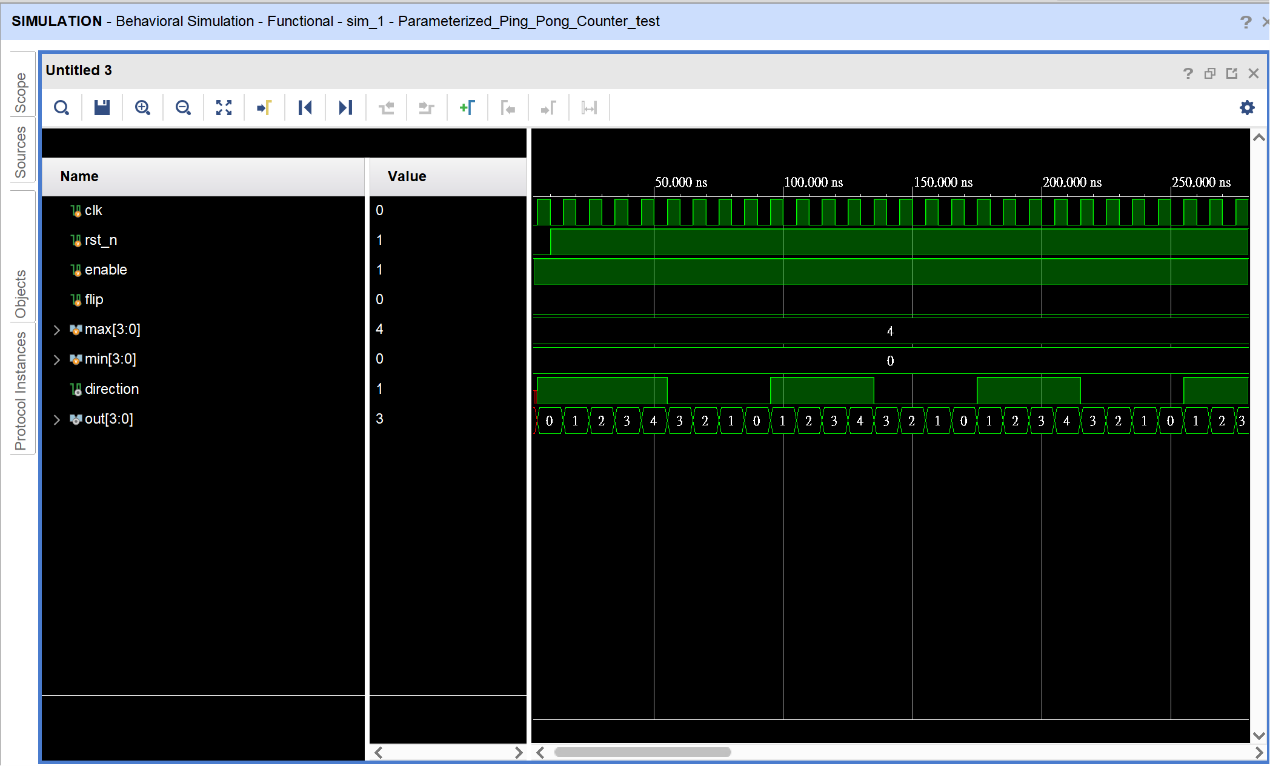
        rst\_n = 1'b1;

end

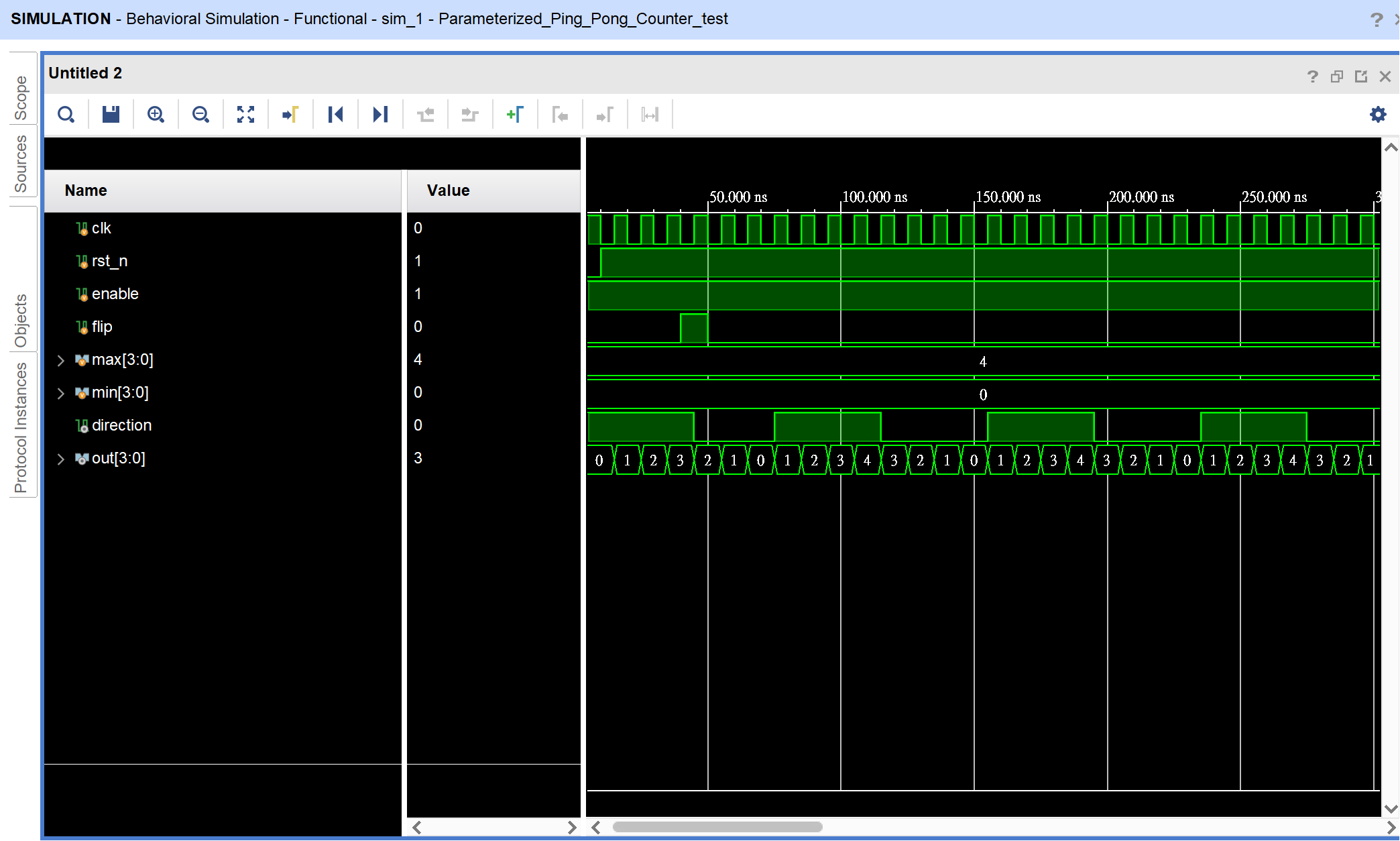
endmodule

Here’s the wave form diagram:

Case 1:



Case 2:



Case 3:

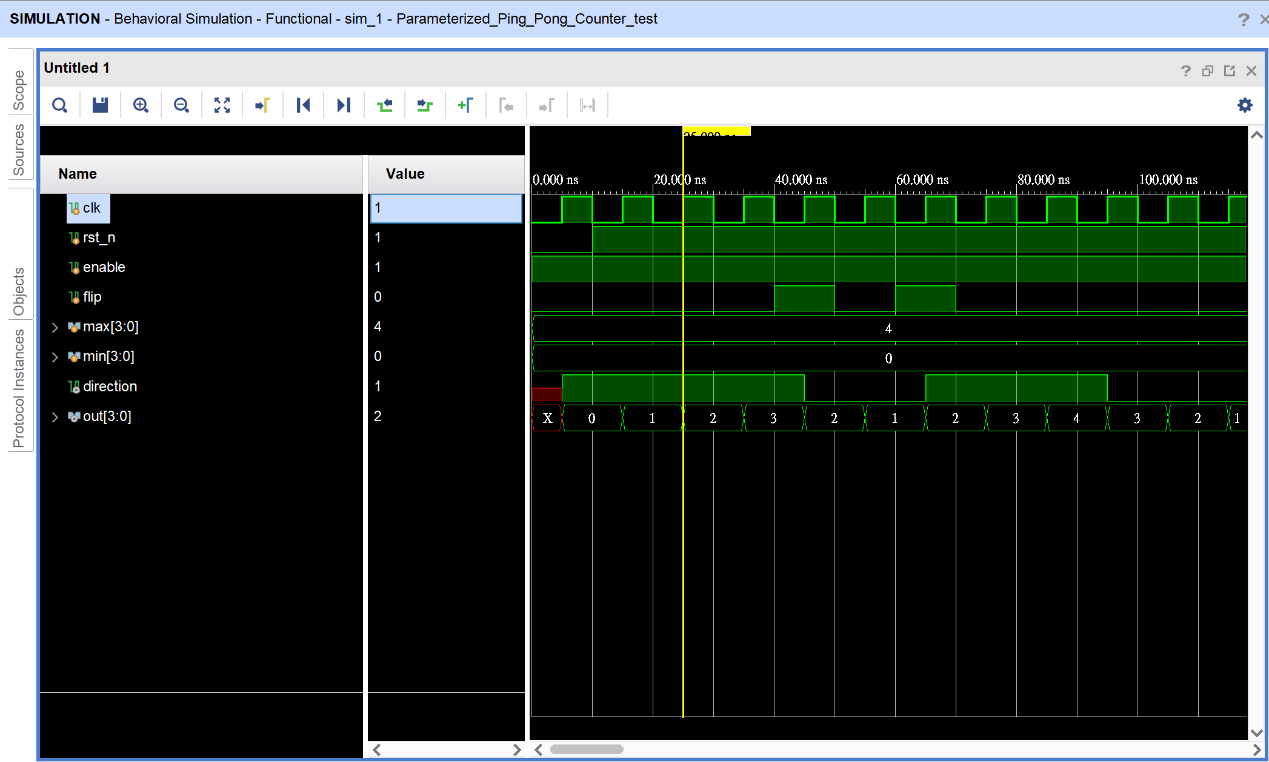
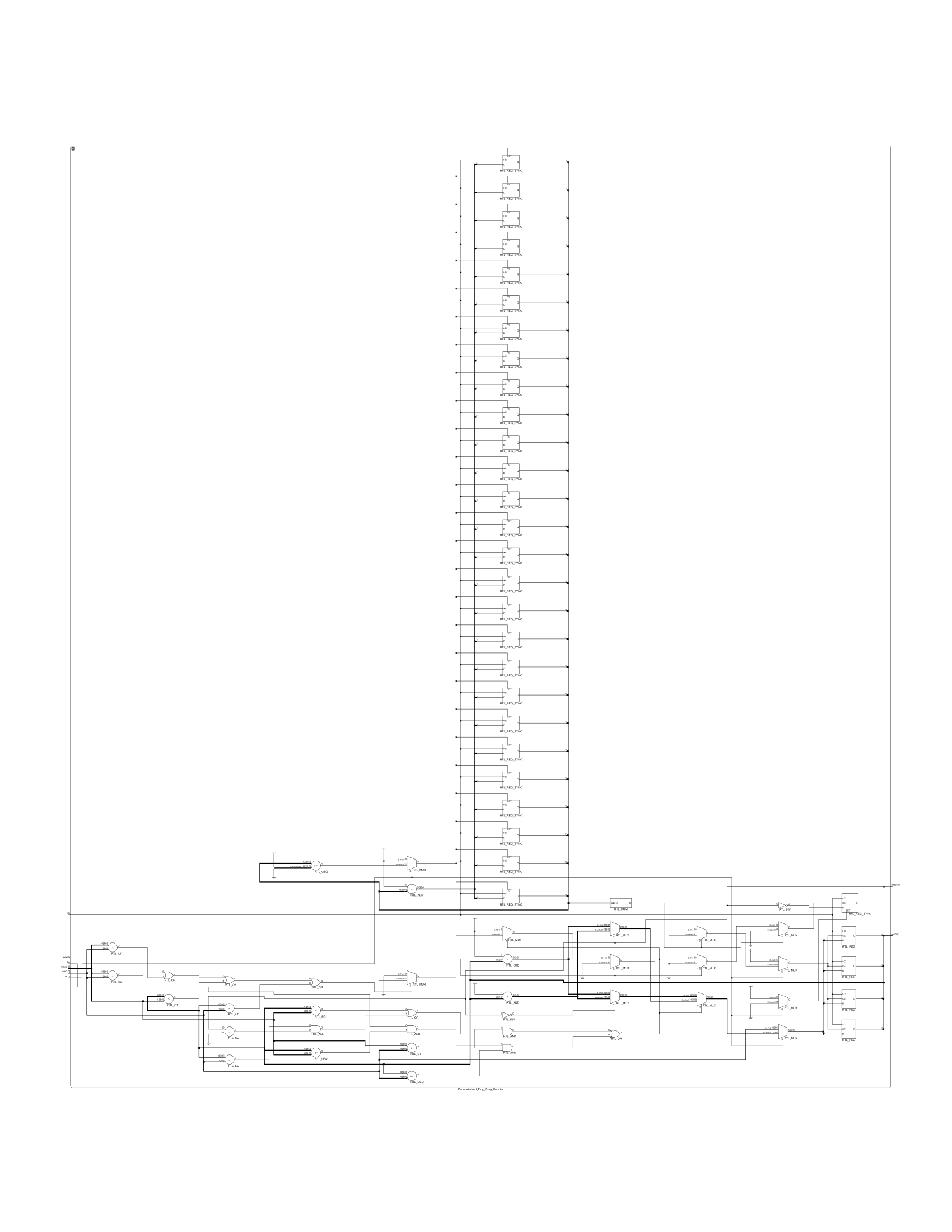


Diagram:

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](https://www.fpga4student.com/2017/09/seven-segment-led-display-controller-basys3-fpga.html) 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;

        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

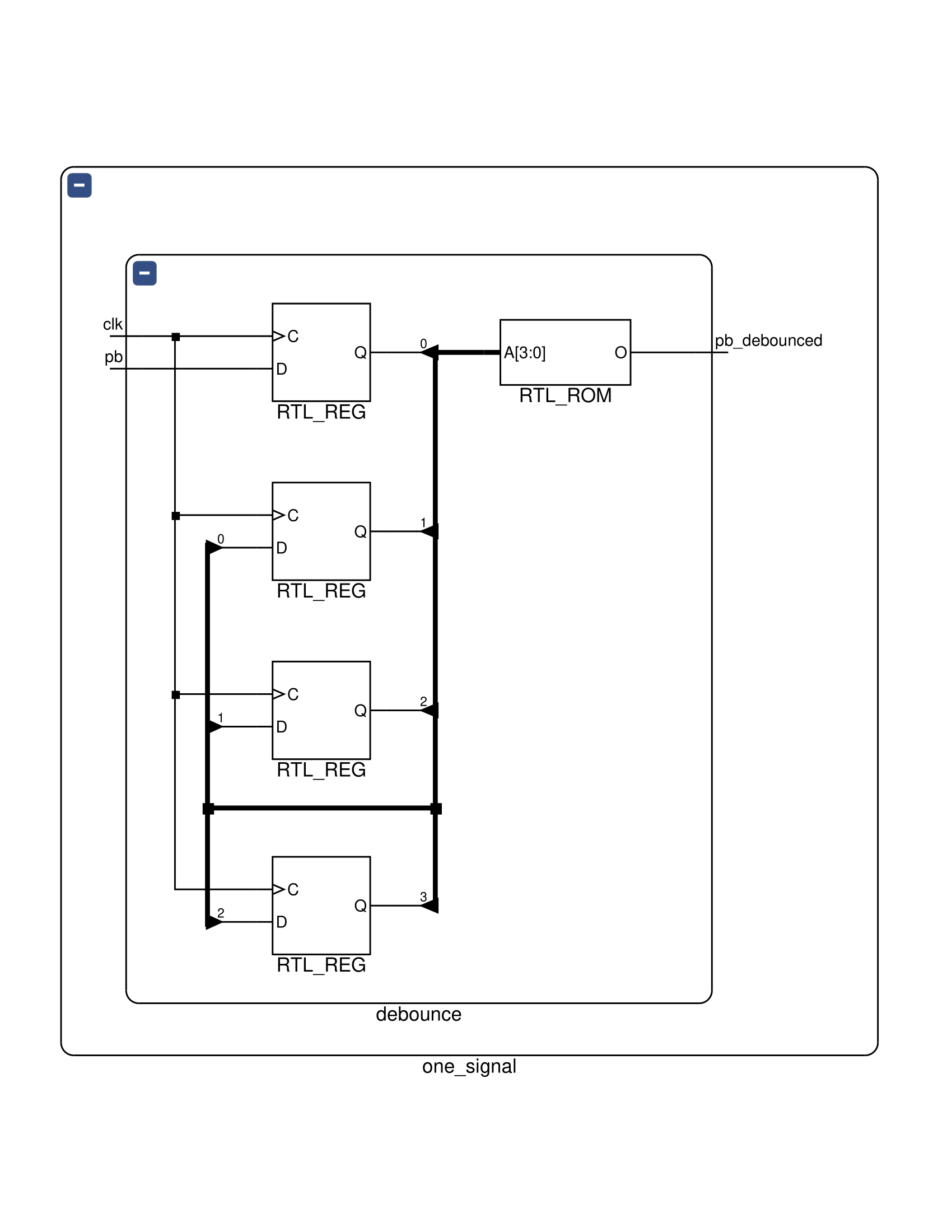
        end

end

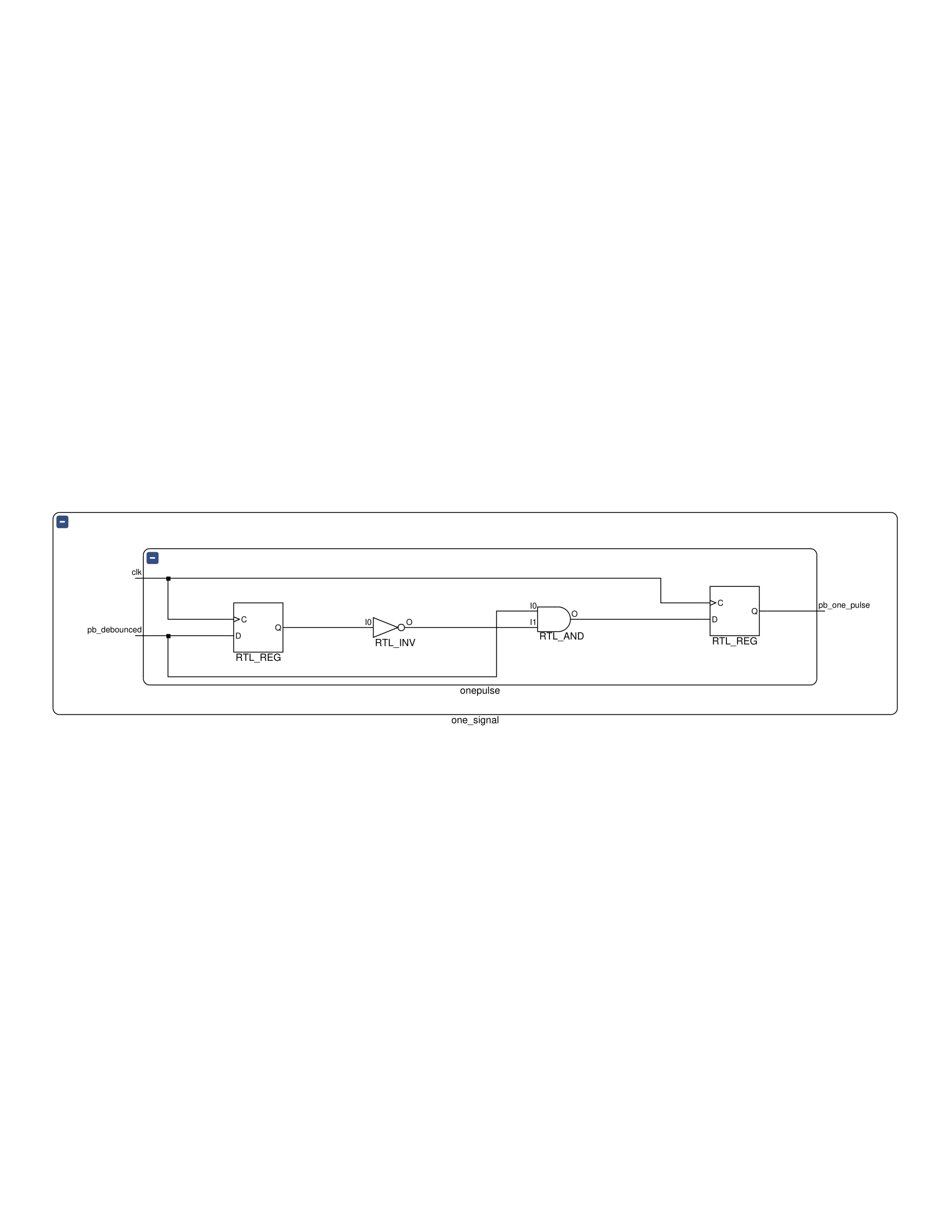
endmodule

Diagram:

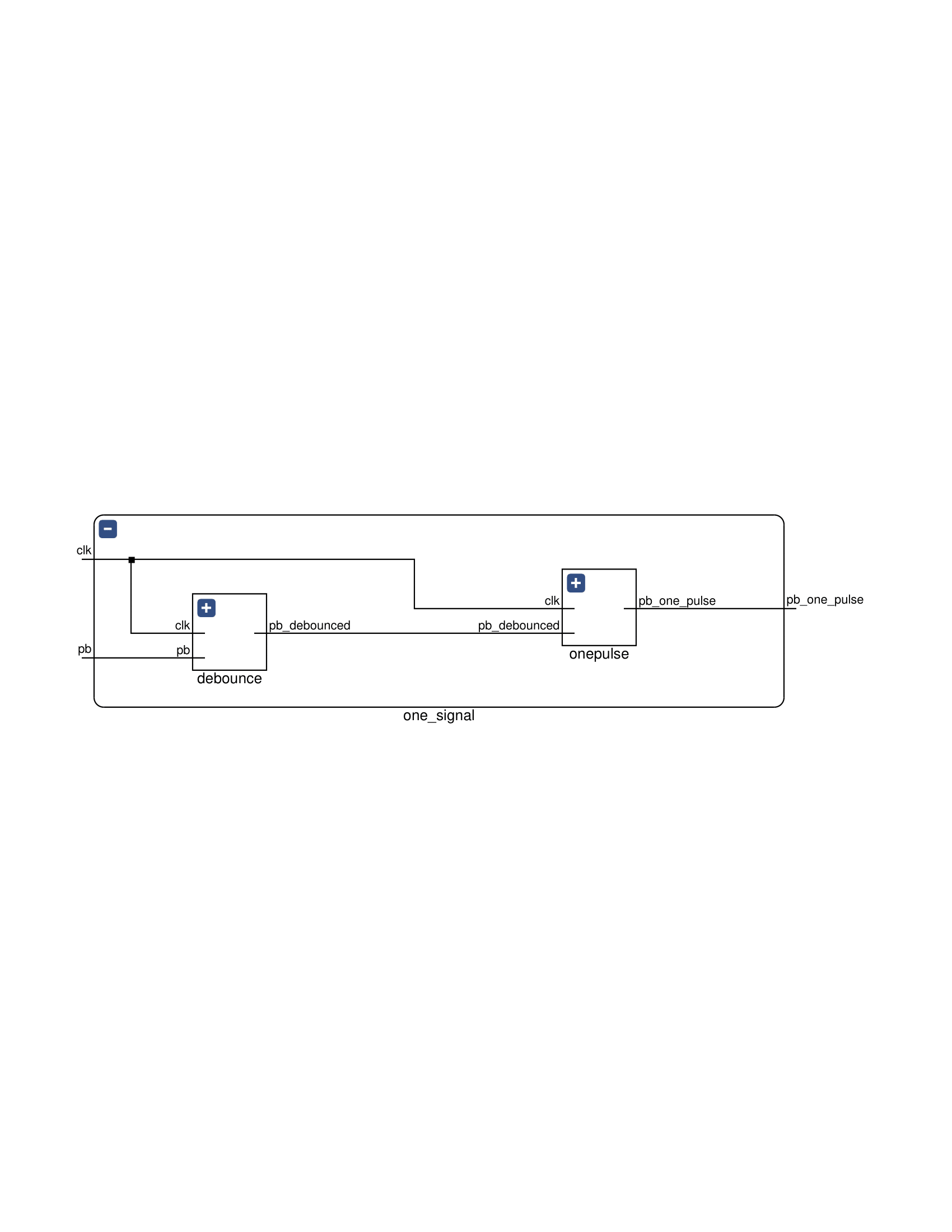
debounce:



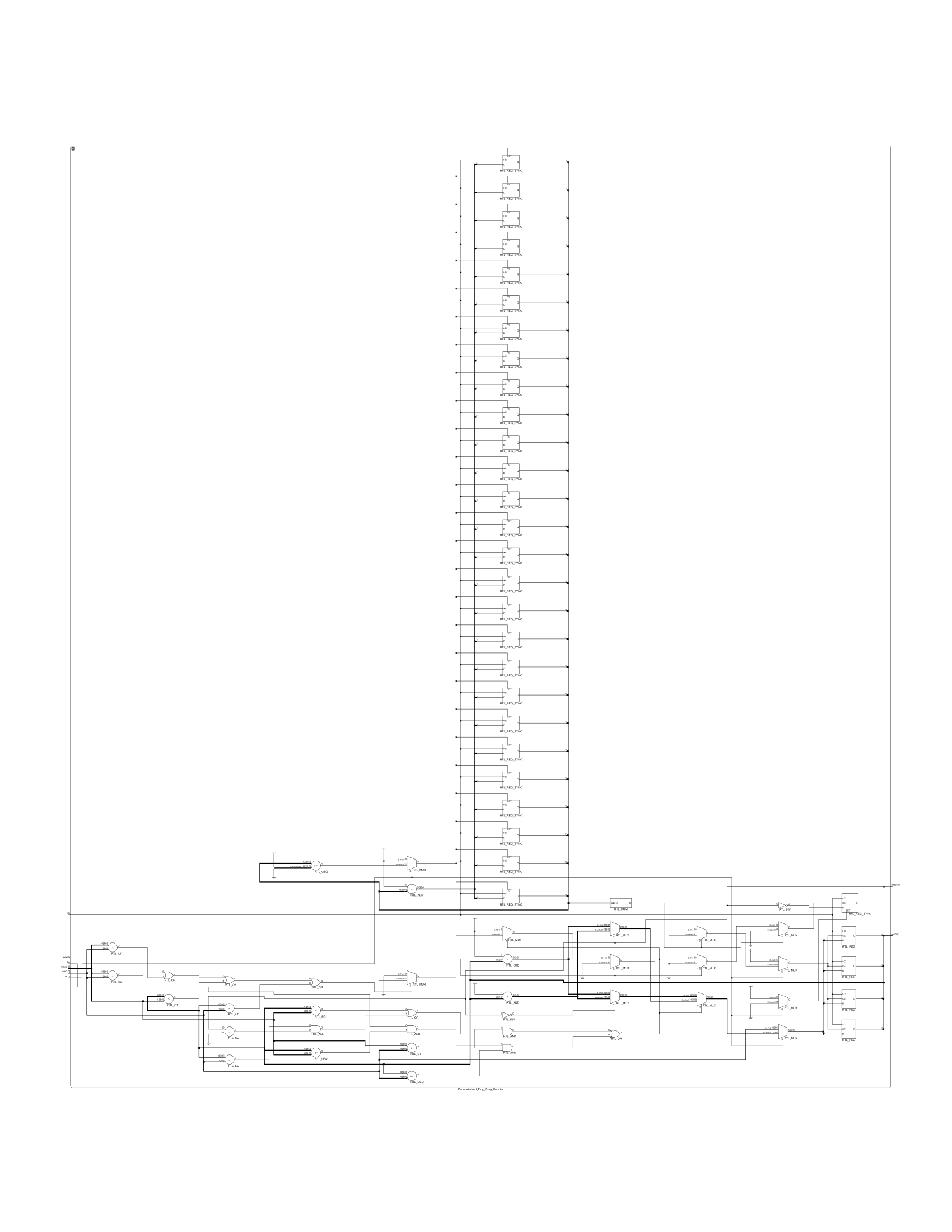
onepulse:



one\_signal:



Parameterized Ping Pong Counter:



FPGA\_7segemnt:

