University of Colorado Boulder
ECEE Department

ECEN 2350 - Digital Logic - Fall 2023
Location: Engineering Center, ECCR 1B40, MWF 1:25PM - 2:15PM
Instructor: Dr. Mona ElHelbawy
Final Project
Lab Title: Digital Stopwatch
Date of Experiment: December 19th, 2023
Names: Connor Sorrell

Introduction

This project report outlines the process of our experimentation building and designing a digital stopwatch, which will be implemented onto our Basys 3 board using System Verilog in Vivado.
Some properties of the stopwatch are as follows:

- The stopwatch begins when the pin V17 is switched into the "1" position
- The stopwatch will stop at the current value when the switch is turned back to "0"
- The stopwatch will count until the timer reaches 999, then reset to 0
- The button U18 on the basys3 board will also reset the stopwatch to 0
- How fast the stopwatch moves can be easily adjusted in the top level module
- Circuit takes in 8-bit binary numbers and converts to BCD, where each decimal number then becomes represented by four bits.

Despite recognizing the opportunities for improvement, it was a good learning experience having to come up with code purely by ourselves, and we take pride in completing the task at hand with the functional stopwatch. We have all agreed that this project forced us to think outside the box, honed our problem-solving skills, and bettered us at working innovatively and collaboratively.

Top Level Module:

Block Diagram

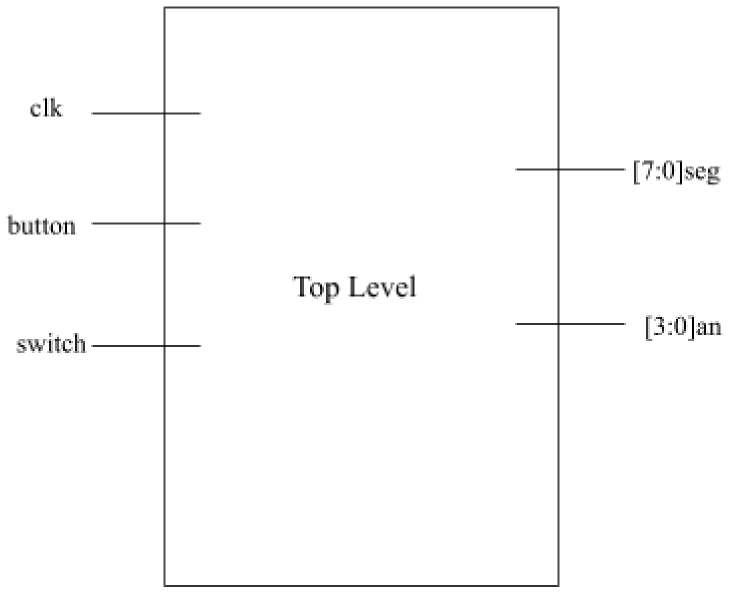

Figure 1.0.1: Black box diagram for top level module

Structural Model

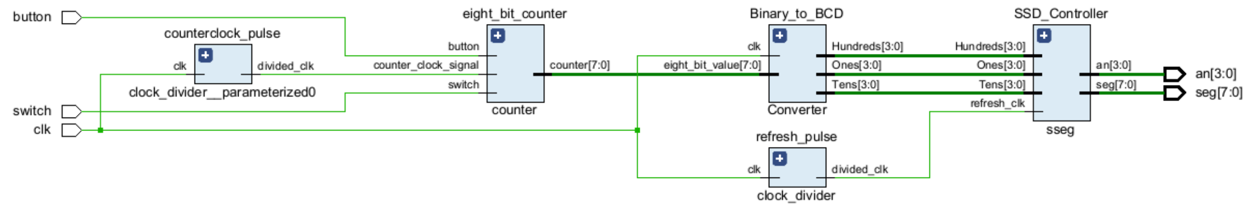

Figure 1.0.2: Structural Model of top level module

Simulation

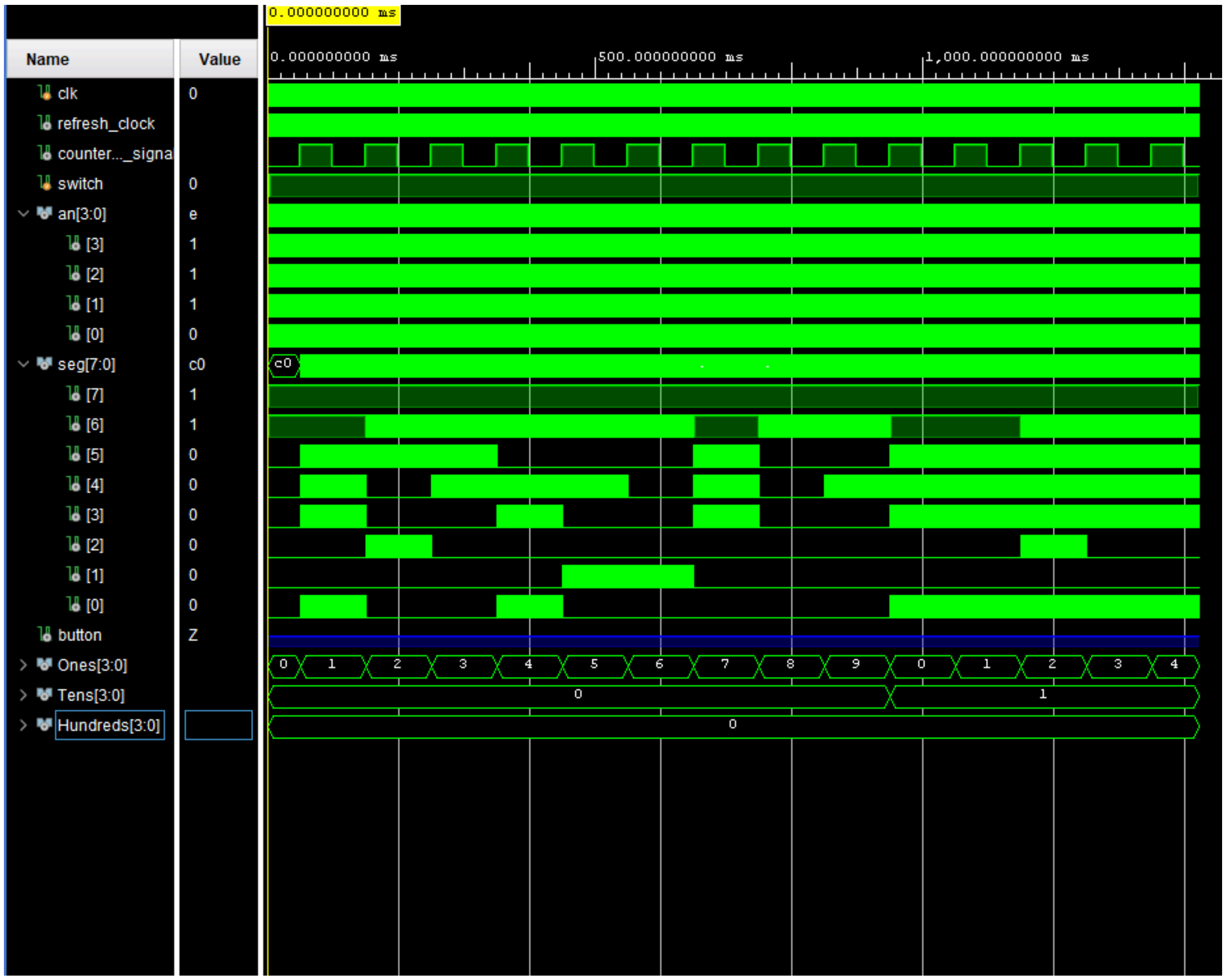

Figure 1.0.3: Very zoomed out screenshot of the top level simulation model. Shows the ones, tens, hundreds adding, the segs are turning on accordingly, and the clocks are working as expected.

Source Code

```
module toplevel(

   input wire clk,
   input wire switch,
   input wire button,

   output wire [3:0] an,
   output wire [7:0] seg

   );
   wire refresh_clock;
   wire counter_clock_signal;
```

```
    wire [7:0] eight_bit_counter_value;
    wire [3:0] Ones;
    wire [3:0] Tens;
    wire [3:0] Hundreds;

    clock_divider #(4999) refresh_pulse (clk, refresh_clock); //10 kHz
    clock_divider #(4999999) counterclock_pulse (clk, counter_clock_signal ); //10 Hz
    counter eight_bit_counter (counter_clock_signal, switch, button, eight_bit_counter_value);
    Converter Binary_to_BCD(clk, eight_bit_counter_value, Ones, Tens, Hundreds);
    sseg SSD_Controller (refresh_clock, Ones, Tens, Hundreds, an, seg);
endmodule
```

Simulation Code

```
module toplevelsim();

reg clk = 0;
reg switch = 0;
wire [3:0] an;
wire [7:0] seg;

toplevel testbench(clk, switch, button, an, seg);

always #5 clk = ~clk;

initial begin
    #500 switch = 1;
end

endmodule
```

Clock Divider Module

Block Diagram

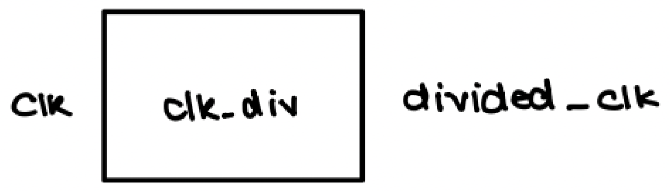

Figure 1.0.4: Black Box Diagram of clock divider module

Structural Model

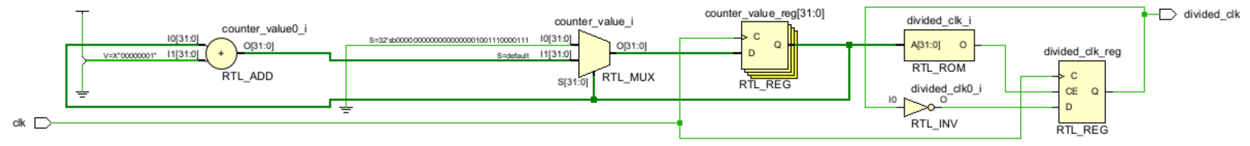

Figure 1.0.5: Structural Model from the synthesis of the clock divider module

Simulation

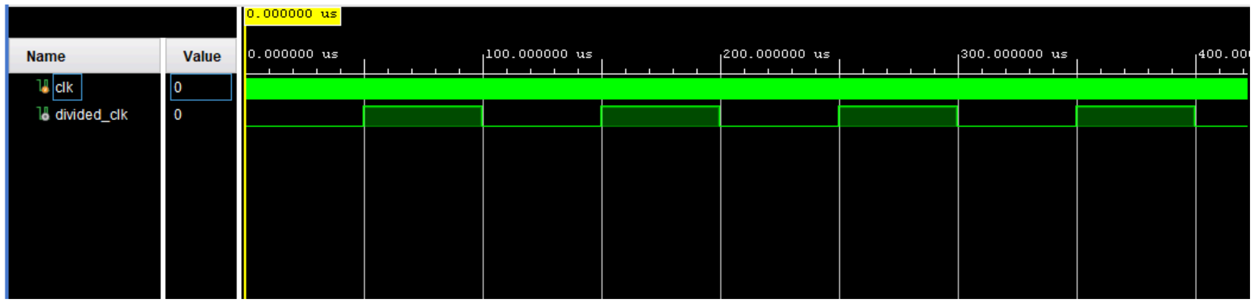

Figure 1.0.6: Very zoomed out screenshot of the clock_divider simulation model, so that the divided_clk can be seen. clk itself has too high of a frequency to be seen, but both frequencies are what we expected.

Source Code

```
module clock_divider #(parameter div_value = 4999)(

    output reg divided_clk = 0
    );



    integer counter_value = 0;

    always@ (posedge clk)
    begin

        if (counter_value == div_value)
            counter_value <= 0;
        else
            counter_value <= counter_value+1;
    end

    always @(posedge clk)
    begin
        if(counter_value == div_value)
            divided_clk <= ~divided_clk;
        else
        divided_clk <= divided_clk ;
    end

endmodule
```

Simulation Code

```
module clockdivsim();
    reg clk = 0;
    wire divided_clk;

    clock_divider uut (.clk(clk), .divided_clk(divided_clk));

    always #5 clk = ~clk;

endmodule
```

Binary to BCD Converter Module

Block Diagram

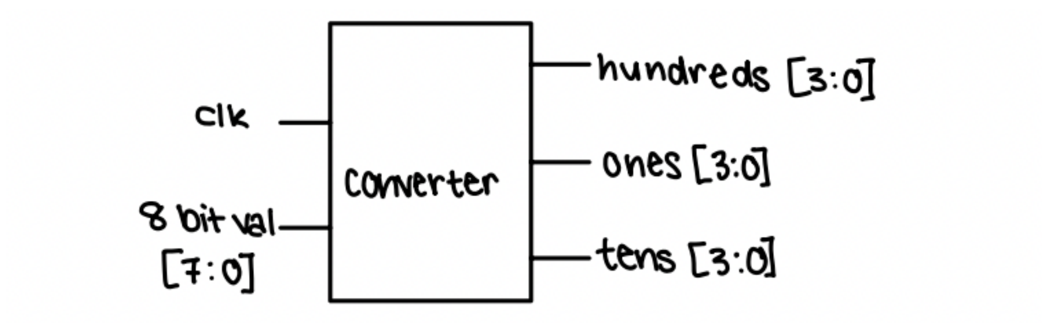

Figure 1.0.7: Black Box Diagram of the Binary to BCD Converter Module

Structural Model

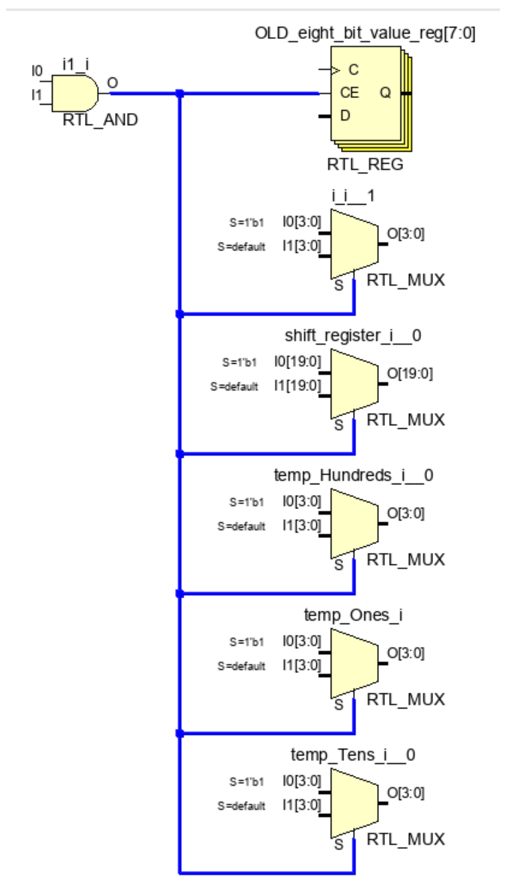

Figure 1.0.8:Structural Model obtained from synthesis of the Binary to BCD Converter Module

Simulation

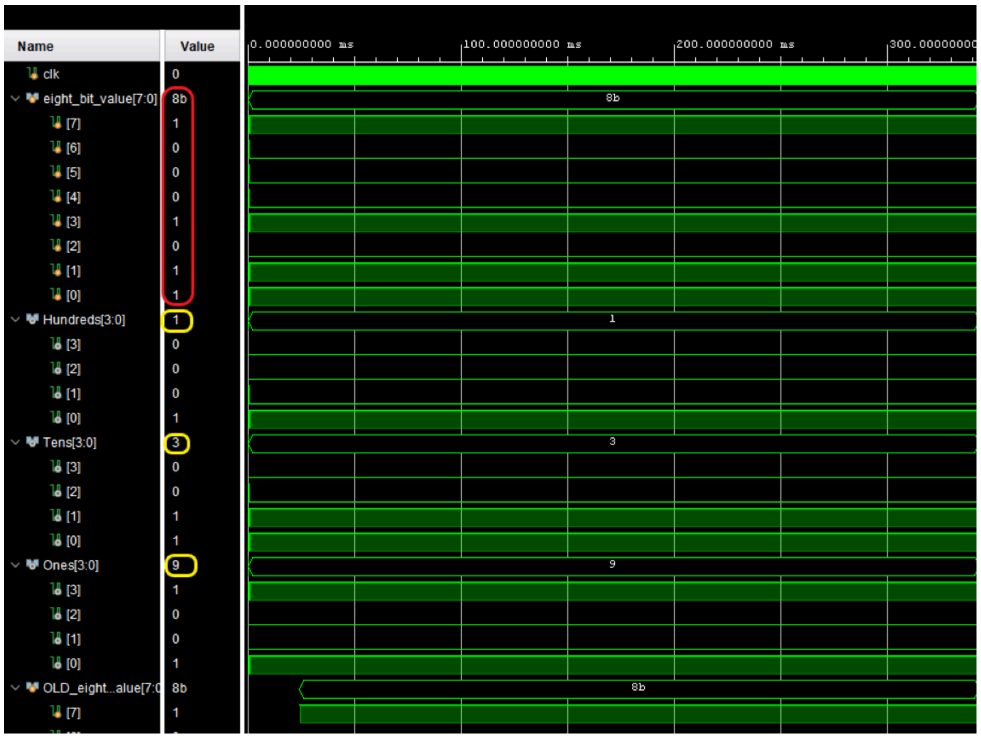

Figure 1.0.9: screenshot of the BCD to binary converter simulation model. Shows that the 8 bit binary value accurately becomes represented by the 3 4-bit values, ones, tens, hundreds.

Source Code

```
module Converter(
   input clk,
   input [7:0] eight_bit_value,
   output reg [3:0] Ones = 0,
   output reg [3:0] Tens = 0,
   output reg [3:0] Hundreds = 0

   );

   reg[3:0] i = 0;
   reg[19:0] shift_register = 0;

   reg[3:0] temp_Hundreds = 0;
   reg[3:0] temp_Tens = 0;
   reg[3:0] temp_Ones = 0;



   reg[7:0] OLD_eight_bit_value = 0;
```

```
    always @ (posedge clk)
    begin

    if (i==0 & (OLD_eight_bit_value != eight_bit_value)) begin
    shift_register = 20'b0;

    OLD_eight_bit_value = eight_bit_value;

    shift_register[7:0] = eight_bit_value;
    temp_Hundreds = shift_register[19:16];
    temp_Tens = shift_register[15:12];
    temp_Ones = shift_register[11:8];
    i = i+1;

    end

    if(i < 9 & i > 0) begin

    if(temp_Hundreds >=5) temp_Hundreds = temp_Hundreds +3;
    if(temp_Tens >=5) temp_Tens = temp_Tens +3;
    if(temp_Ones >=5) temp_Ones = temp_Ones +3;

    shift_register[19:8] = {temp_Hundreds,temp_Tens,temp_Ones};

    shift_register = shift_register <<1;

    temp_Hundreds = shift_register[19:16];
    temp_Tens = shift_register[15:12];
    temp_Ones = shift_register[11:8];
    i = i+1; // repeats until i = 9

    end

    if(i == 9) begin
    i = 0;
    Hundreds = temp_Hundreds;
    Tens = temp_Tens;
    Ones = temp_Ones;
    end
end

endmodule
```

Simulation Code

```
module BCDsim();
reg clk = 0;
reg [7:0] eight_bit_value=0;
wire [3:0] Ones;
wire [3:0] Tens;
wire [3:0] Hundreds;

Converter testbench (clk, eight_bit_value, Ones, Tens, Hundreds);

always #5 clk = ~clk;

initial begin

    eight_bit_value = 0;
    # 500 eight_bit_value = 10;
    # 500 eight_bit_value = 248;
    # 500 eight_bit_value = 139;
end

endmodule
```

Seven Segment Top Module

Block Diagram

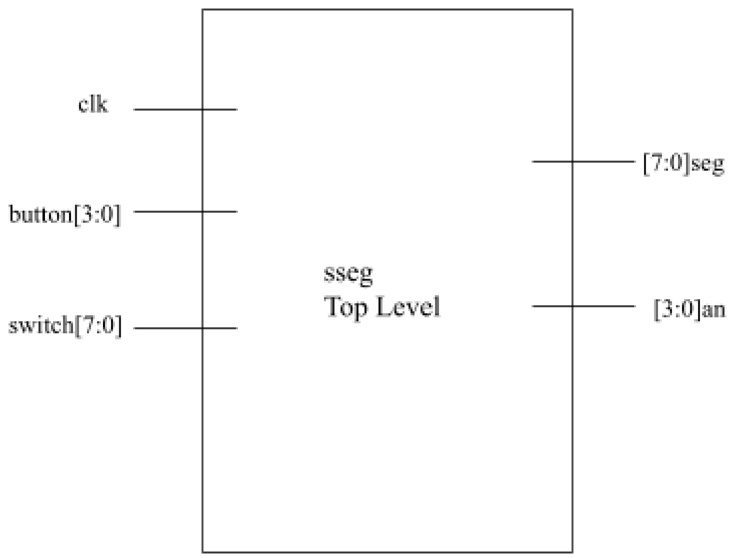

Figure 1.1.0: Black Box Diagram of the Seven Segment Top Module

Structural Model

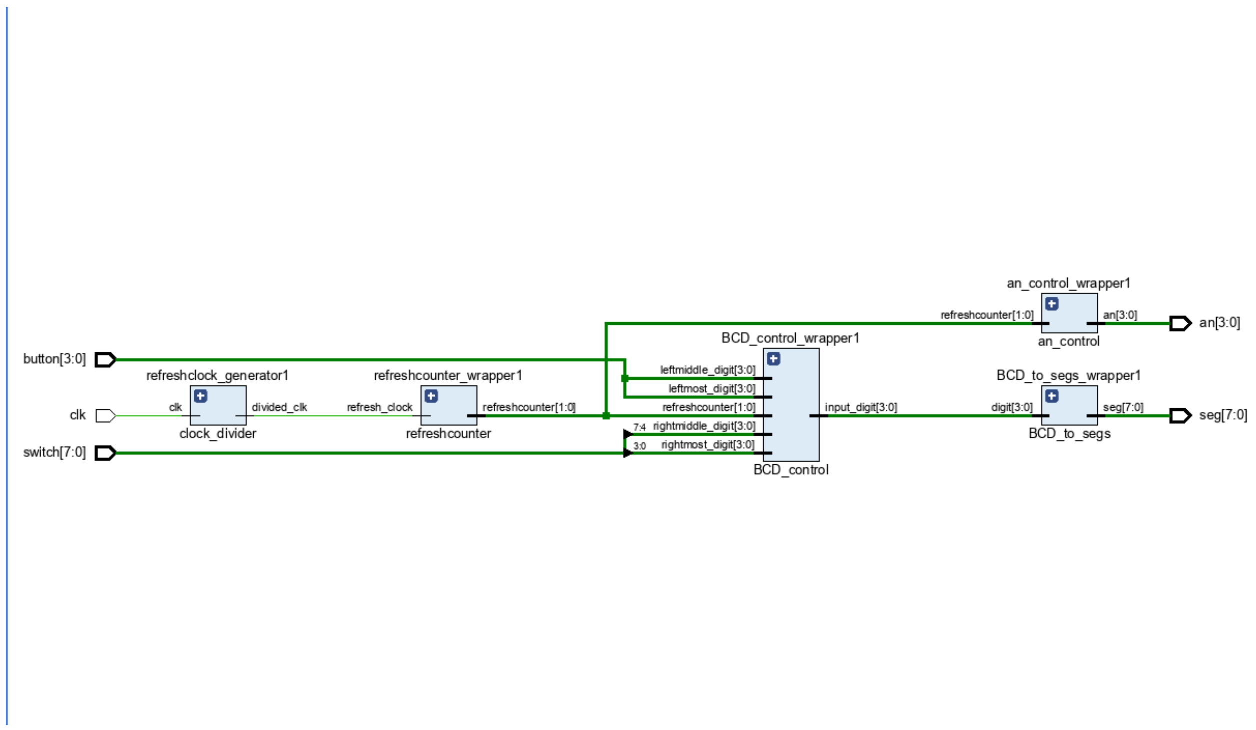

Figure 1.1.1: Structural Model obtained from synthesis of the Seven Segment Top Module

Simulation

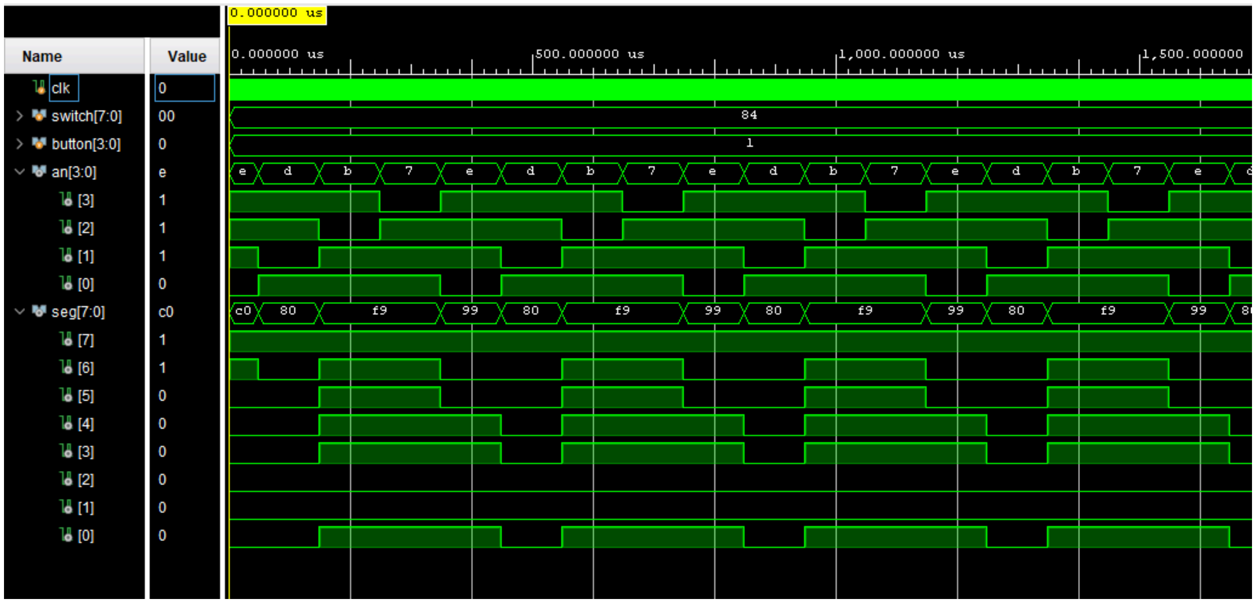

Figure 1.1.2: screenshot of the top level seven segment simulation model. Shows the segs resulting at the right times as well as the anodes.

Source Code

```
module ssegtop(
    input wire clk, // from the clock divider, this is 10kHz (is based on the local param div value that we set )
    input wire [7:0] switch,
    input wire [3:0] button,
    output wire [3:0] an,
    output wire [7:0] seg
    );

    wire refresh_clock;
    wire [1:0] refreshcounter;
    wire [3:0] input_digit;

    clock_divider refreshclock_generator1 (.clk(clk), .divided_clk(refresh_clock));
    refreshcounter refreshcounter_wrapper1 (.refresh_clock(refresh_clock), .refreshcounter(refreshcounter ));
    an_control an_control_wrapper1 (.refreshcounter(refreshcounter), .an(an));
    BCD_control BCD_control_wrapper1 (.rightmost_digit(switch[3:0]), .rightmiddle_digit(switch[7:4]),
.leftmiddle_digit(button[3:0]),
    .leftmost_digit(button[3:0]), .refreshcounter(refreshcounter), .input_digit(input_digit));
    BCD_to_segs BCD_to_segs_wrapper1 (.digit(input_digit), .seg(seg));


endmodule
```

Simulation Code

```
module ssegsim();
    reg clk = 0;
    reg [7:0] switch = 0;
    reg [3:0] button = 0;
    wire [3:0] an;
    wire [7:0] seg;

    ssegtop UUT (clk, switch, button, an, seg);
    always #5 clk = ~clk;

    initial
    begin
    #100 switch[3:0] = 4;
    #1000 switch [7:4] = 8;
    #100 button = 1;

    end



endmodule
```

8-Bit Counter Module

Block Diagram

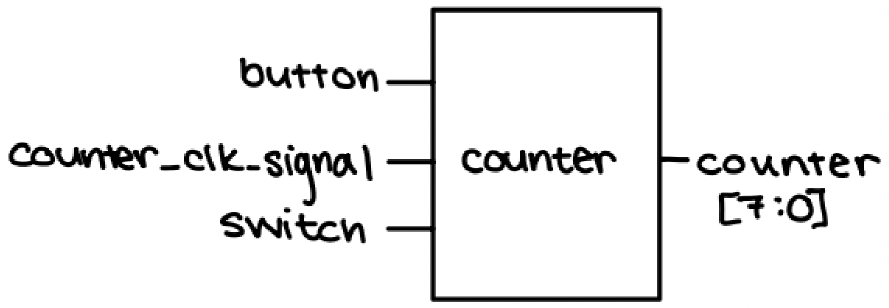

Figure 1.1.3: Black Box Diagram of the 8-Bit Counter Module

Structural Model

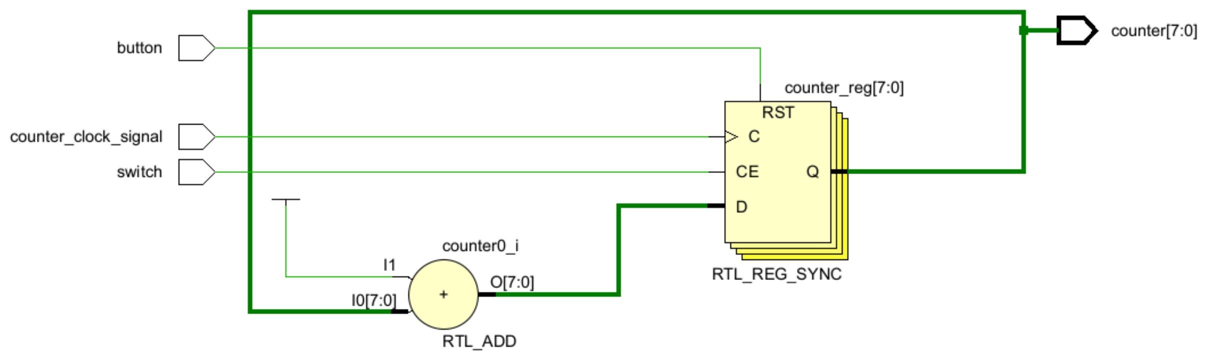

Figure 1.1.4: Structural Model obtained from synthesis of the 8-Bit counter module

Simulation

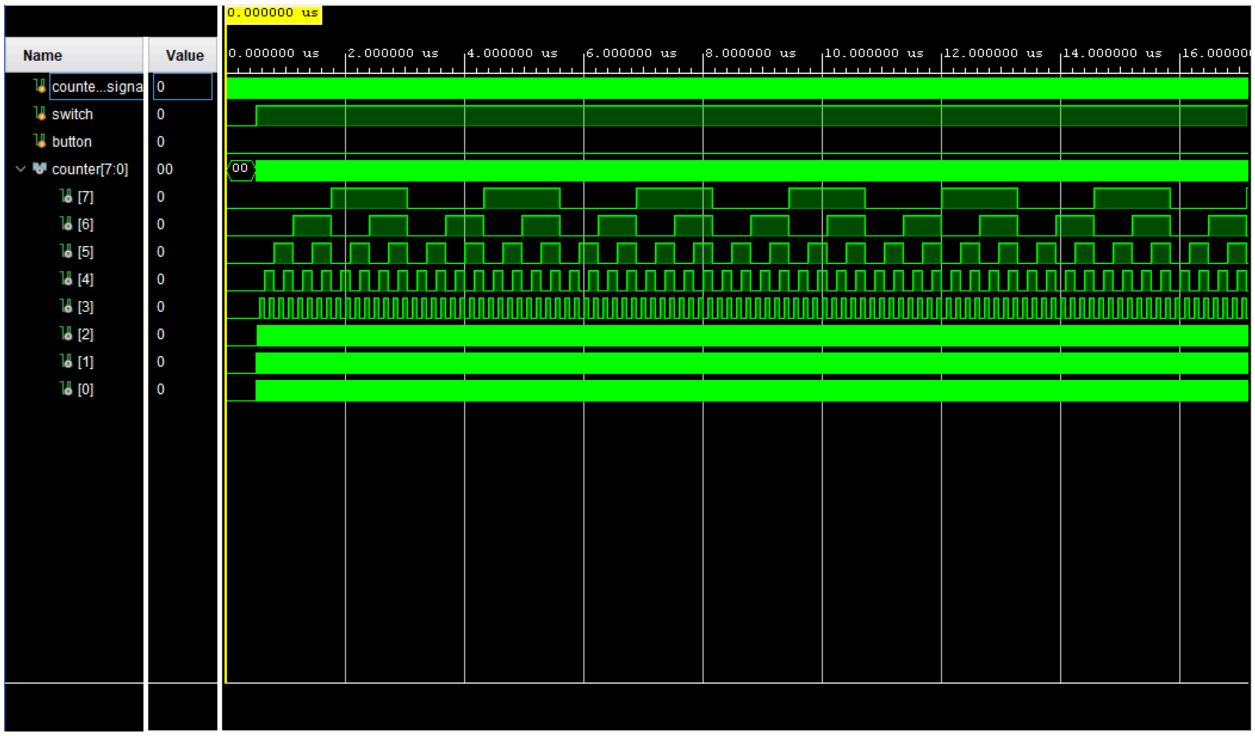

Figure 1.1.5: Screenshot of the counter simulation model. Shows how the 8-bit is added and everything is as expected.

Source Code

```
module counter(
    input wire counter_clock_signal,
    input wire switch,
    input wire button,
    output reg [7:0] counter = 0

    );


    always@(posedge counter_clock_signal)
    begin

        if (switch == 0 || button == 0)

            begin
                counter <= counter;
            end
        if (switch == 1)

            begin

            if (counter == 9999)

                begin
                    counter <= 0;

                    end else

                        begin

                            counter <=counter +1;
                        end
                 end


        if (button == 1)
        begin
            counter <= 0;
        end
    end
endmodule
```

Simulation Code

```
module countersim();

    reg counter_clock_signal = 0;
    reg switch = 0;
    reg button = 0;
    wire [7:0] counter;


    counter testbench (counter_clock_signal, switch, button, counter);

    always #5 counter_clock_signal = ~counter_clock_signal;
    initial begin

        #500 switch = 1;
    end


endmodule
```

Seven Seg Controller

Block Diagram

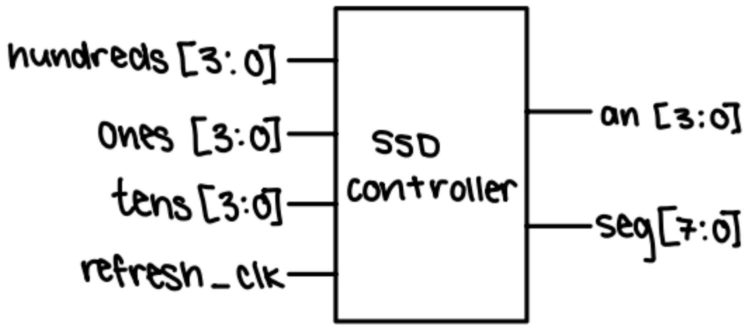

Figure 1.1.6: Black Box Diagram of the SSD Controller

Structural Model

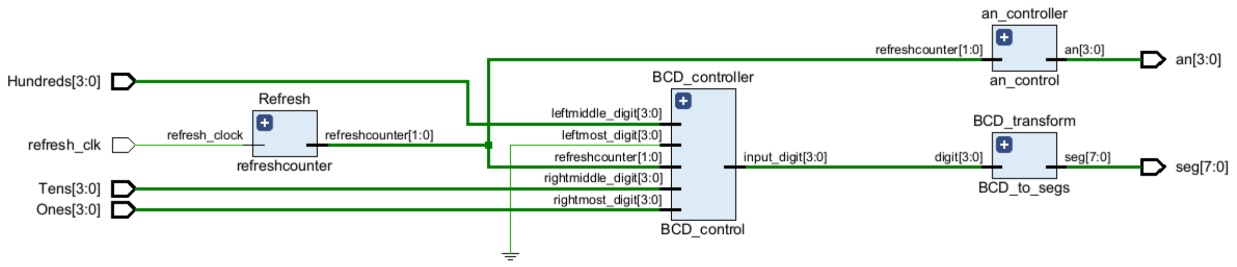

Figure 1.1.7: Structural Model obtained from synthesis of the Seven Seg Controller

Simulation

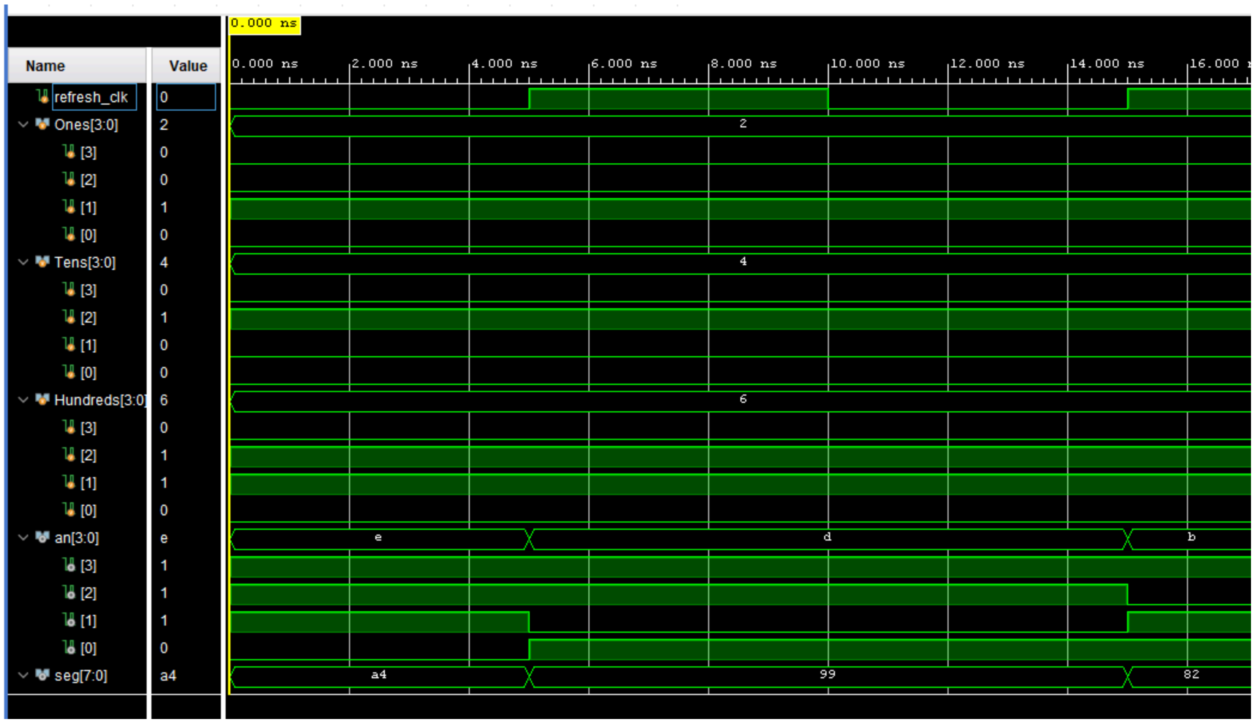

Figure 1.1.8: Screenshot of the SSD controller simulation model. Shows that ones, tens and hundreds are working correctly, as they should be outputting 2,4,6 respectively. Also shows refresh_clk is working as intended and the anodes are too.

Source Code

```
module sseg(

   input wire refresh_clk,
   input wire [3:0] Ones,
   input wire [3:0] Tens,
   input wire [3:0] Hundreds,
   output wire [3:0] an,
   output wire [7:0] seg

   );
   wire [1:0] refreshcounter;
   wire [3:0] input_digit;

   refreshcounter Refresh(.refresh_clock(refresh_clk), .refreshcounter(refreshcounter));
   an_control an_controller(.refreshcounter(refreshcounter), .an(an));
   BCD_control BCD_controller(.rightmost_digit(Ones), .rightmiddle_digit(Tens), .leftmiddle_digit(Hundreds),
.leftmost_digit(4'd0), .refreshcounter(refreshcounter),
   .input_digit(input_digit));
```

```
    BCD_to_segs BCD_transform(.digit(input_digit), .seg(seg));

endmodule
```

Simulation Code

```
module segcontrollersim();
    reg refresh_clk = 0;
    reg [3:0] Ones = 0;
    reg [3:0] Tens = 0;
    reg [3:0] Hundreds = 0;

    wire [3:0] an;
    wire [7:0] seg;

    sseg testbench (refresh_clk, Ones, Tens, Hundreds, an, seg);
    always #5 refresh_clk = ~refresh_clk;

    initial
    begin
    Ones = 4'b0010;
    Tens = 4'b0100;
    Hundreds = 4'b0110;

    end



endmodule
```

Small Helper Modules

- These modules are "controllers" for the rest of the circuit. They do not need to be simulated because any issues they have show up immediately when running any of the connectors or top modules. They consist of case statements, assigning power, digits, etc.

An control circuit

Black Box

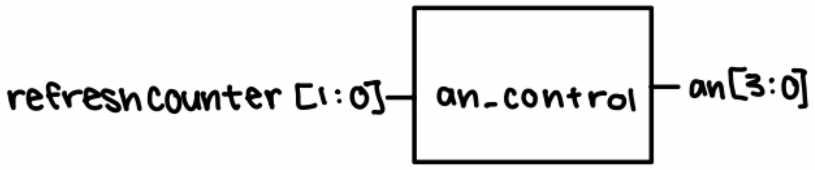

Figure 1.1.9: Black Box Diagram modeling the an control circuit

Schematic

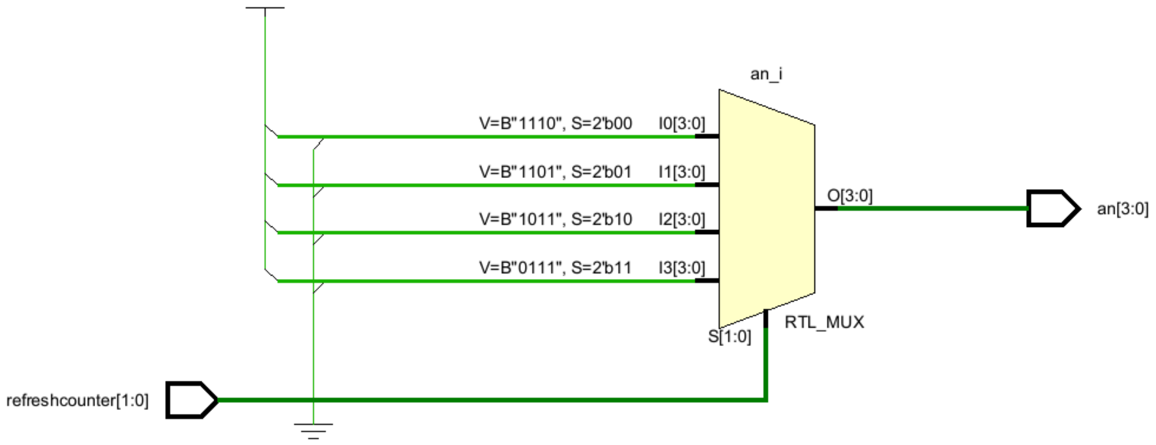

Figure 1.2.0: Structural Model obtained from synthesis of the An control circuit

Source Code

```
module an_control(

    input [1:0] refreshcounter,
    output reg[3:0] an = 0
    );

    always @(refreshcounter)
    begin
        case(refreshcounter) //turns these digits on
        2'b00:
            an = 4'b1110; //  (rightmost)
        2'b01:
            an = 4'b1101;
        2'b10:
            an = 4'b1011;
        2'b11:
            an = 4'b0111; // (leftmost digit)
        endcase
 end
endmodule
```

BCD control circuit

Black Box

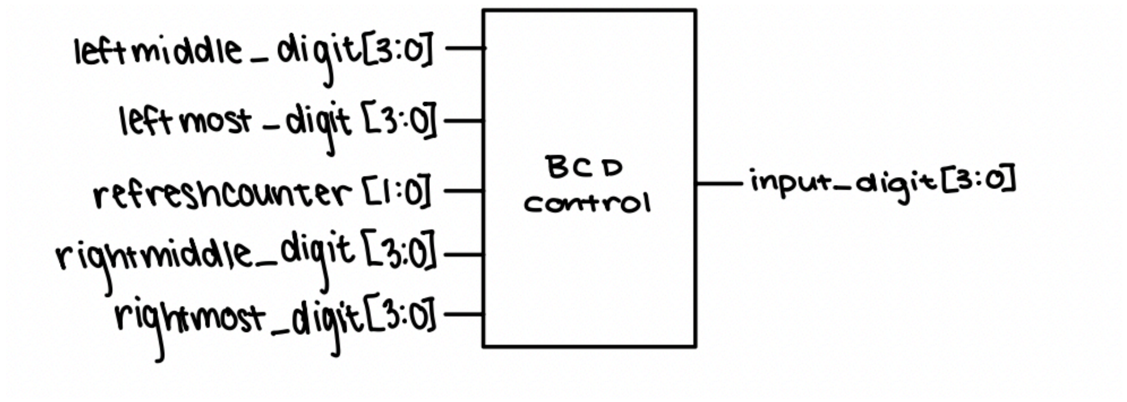

Figure 1.2.1: Black Box Diagram modeling the BCD control circuit

Schematic

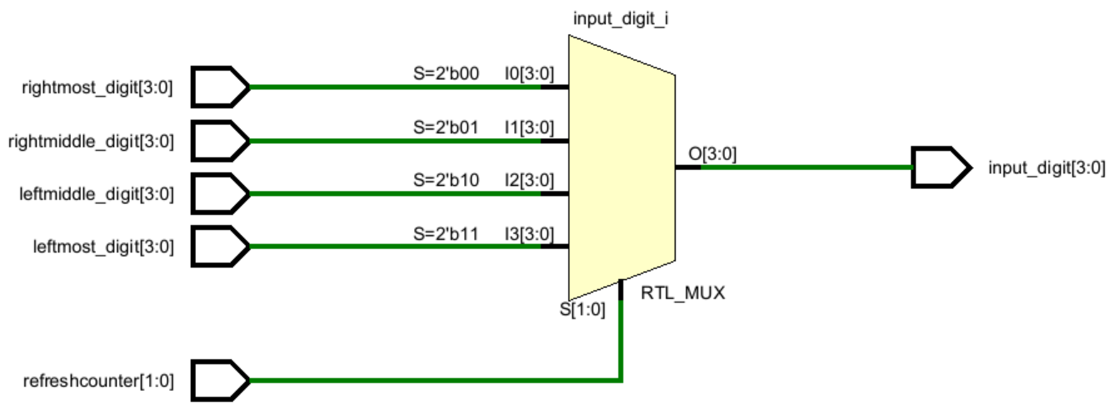

Figure 1.2.2: Structural Model obtained from synthesis of the BCD control circuit

Source Code

```
module BCD_control(

    input [3:0] rightmost_digit,
    input [3:0] rightmiddle_digit,
    input [3:0] leftmiddle_digit,
    input [3:0] leftmost_digit,
    input [1:0] refreshcounter,
    output reg[3:0] input_digit = 0
    );

    always @(refreshcounter)
    begin
        case(refreshcounter)
        2'd0:
            input_digit = rightmost_digit; // values of the digits
        2'd1:
            input_digit = rightmiddle_digit;
        2'd2:
            input_digit = leftmiddle_digit;
        2'd3:
            input_digit = leftmost_digit; // (leftmost digit)
        endcase
 end

endmodule
```

BCD to Seg circuit

Black Box

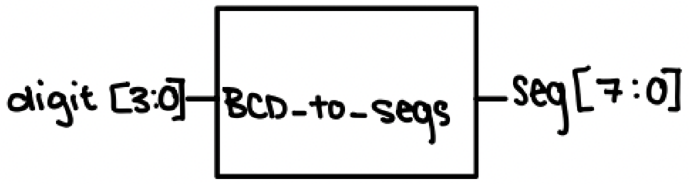

Figure 1.2.3: Black Box Diagram modeling the BCD to Seg circuit

Schematic

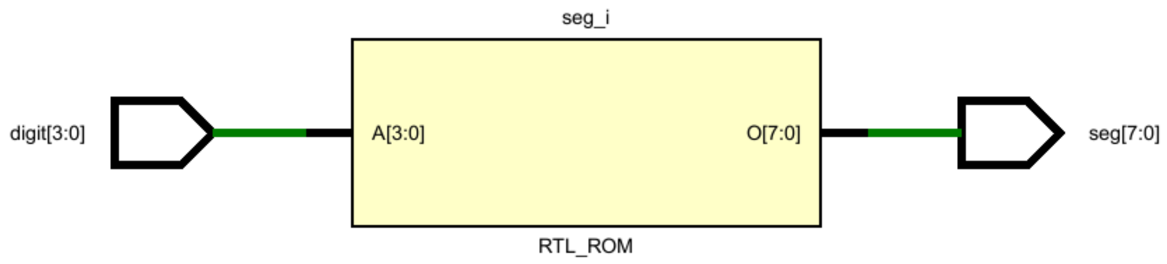

Figure 1.2.4: Structural Model obtained from synthesis of the BCD to Segs circuit

Source Code

```
module BCD_to_segs(
   input [3:0] digit,
   output reg[7:0] seg = 0
   );

   always @(digit)
   begin
      case(digit)
         4'b0000:
             seg = 8'b11000000;
         4'b0001:
             seg = 8'b11111001;
         4'b0010:
             seg = 8'b10100100;
         4'b0011:
             seg = 8'b10110000;
         4'b0100:
             seg = 8'b10011001;
         4'b0101:
             seg = 8'b10010010;
         4'b0110:
             seg = 8'b10000010;
         4'b0111:
             seg = 8'b11111000;
         4'b1000:
             seg = 8'b10000000;
         4'b1001:
             seg = 8'b10010000;
         default:
             seg = 8'b11000000;
         endcase
  end

endmodule
```

Refresh counter circuit

Black Box

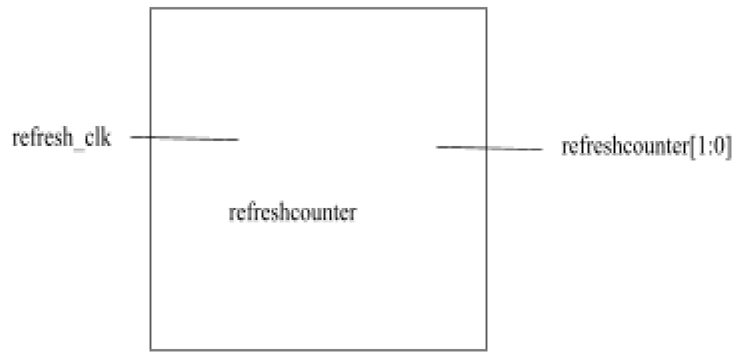

Figure 1.2.5: Black Box Diagram modeling the refresh counter circuit

Schematic

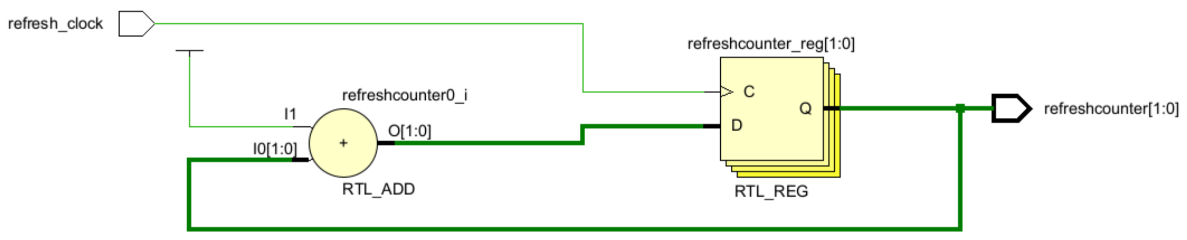

Figure 1.2.6: Structural Model obtained from synthesis of the refresh counter circuit

Source Code

```
module refreshcounter(
   input refresh_clock,
   output reg[1:0] refreshcounter = 0

   );

   always @(posedge refresh_clock ) refreshcounter <= refreshcounter +1;
endmodule
```