ECE3300L Lab 7

Group B

By Faris Khan (ID #: 012621102)

AND

Nicholas Williams (ID#:016556982)

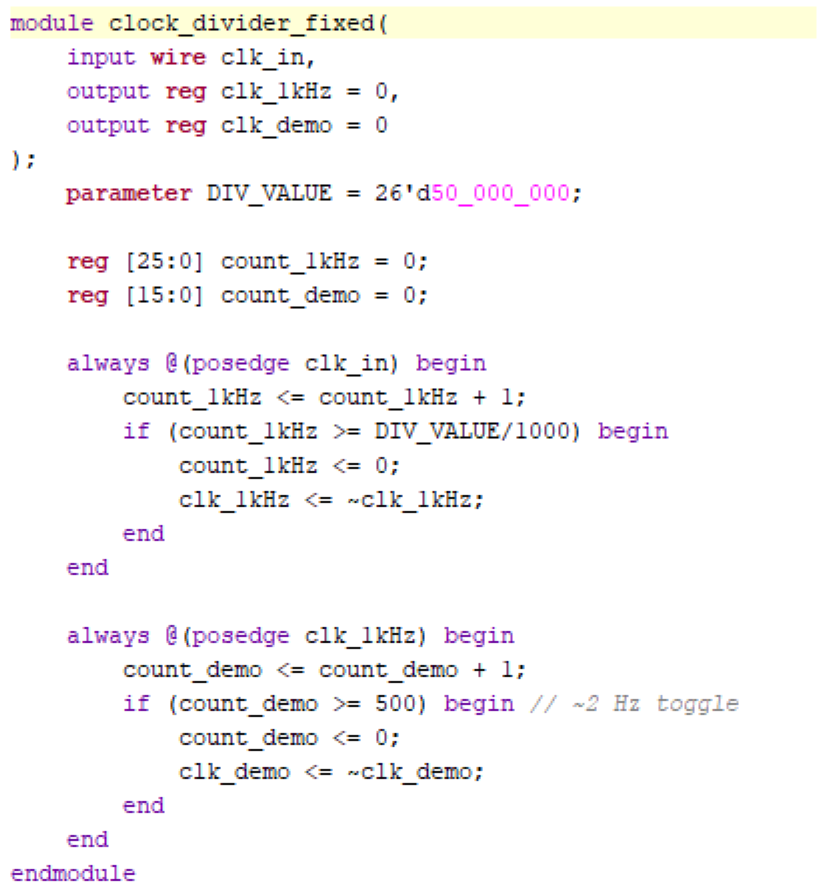
6 August 2025

Introduction

In this lab, we designed and implemented a 16-bit barrel shifter and connected it to a 4-digit 7-segment display on the Nexys A7 FPGA board. The goal was to allow the user to shift or rotate a 16-bit input word by a variable amount in either direction (left or right), using push-button controls. To make the system user-friendly and visually observable, we used debounced toggle logic to treat the push-buttons like switches and displayed the result of each operation using a scanned 7-segment display. The lab helped us understand how to build a shift/rotate unit using MUX-based logic and control it interactively using real hardware inputs. It also reinforced concepts like clock division, debouncing, and hardware display multiplexing.

Code

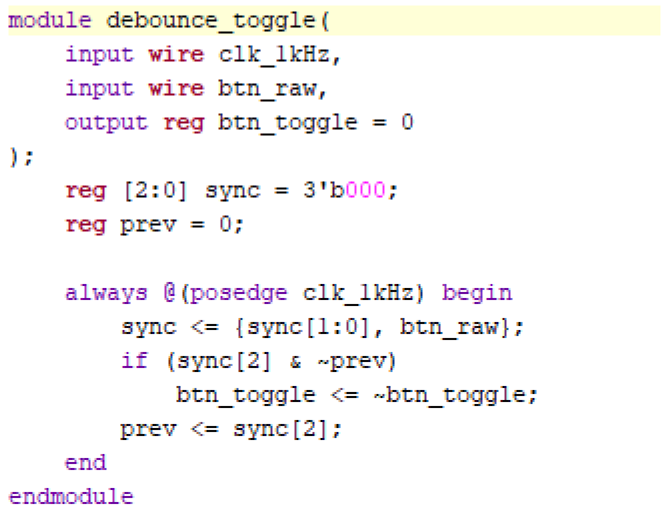
clock_divider_fixed.v



```
module clock_divider_fixed(
    input wire clk_in,
    output reg clk_1kHz = 0,
    output reg clk_demo = 0
);
    parameter DIV_VALUE = 26'd50_000_000;

    reg [25:0] count_1kHz = 0;
    reg [15:0] count_demo = 0;

    always @(posedge clk_in) begin
        count_1kHz <= count_1kHz + 1;
        if (count_1kHz >= DIV_VALUE/1000) begin
            count_1kHz <= 0;
            clk_1kHz <= ~clk_1kHz;
        end
    end

    always @(posedge clk_1kHz) begin
        count_demo <= count_demo + 1;
        if (count_demo >= 500) begin // ~2 Hz toggle
            count_demo <= 0;
            clk_demo <= ~clk_demo;
        end
    end
endmodule
```

The clock_divider_fixed module takes the 100 MHz input clock from the Nexys A7 board and slows it down to create two lower-frequency clocks. The first output, clk_1kHz, is used for debouncing button inputs and scanning the 7-segment display, so that button presses are stable and the display refreshes fast enough to look smooth. The second output, clk_demo, runs at around 2 Hz and is used to control the barrel shifter at a slow speed so we can clearly see the shifting or rotating changes on the display. This module helped make the system easier to test and observe during the demo.

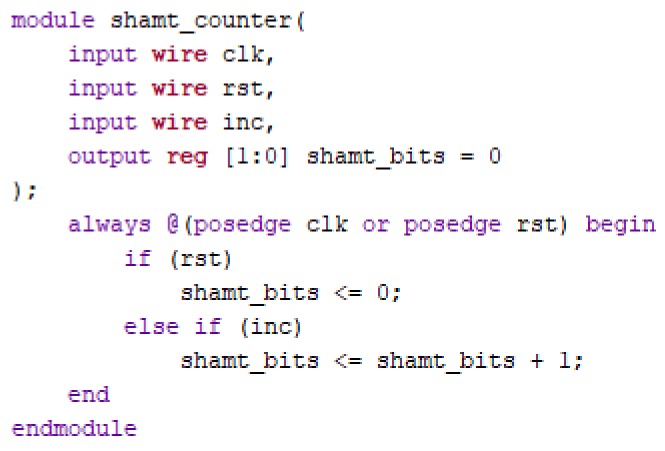
debounce_toggle.v



```
module debounce_toggle(
    input wire clk_1kHz,
    input wire btn_raw,
    output reg btn_toggle = 0
);
    reg [2:0] sync = 3'b000;
    reg prev = 0;

    always @(posedge clk_1kHz) begin
        sync <= {sync[1:0], btn_raw};
        if (sync[2] & ~prev)
            btn_toggle <= ~btn_toggle;
        prev <= sync[2];
    end
endmodule
```

The debounce_toggle module is used to clean up noisy button signals. It takes in a raw button input (btn_raw) and a 1 kHz clock (clk_1kHz) and outputs a stable toggle signal (btn_toggle). Inside the module, the raw input is passed through a 3-bit shift register (sync) to help remove glitches or bouncing effects that happen when physical buttons are pressed. When a clean rising edge is detected, the module toggles the output state.

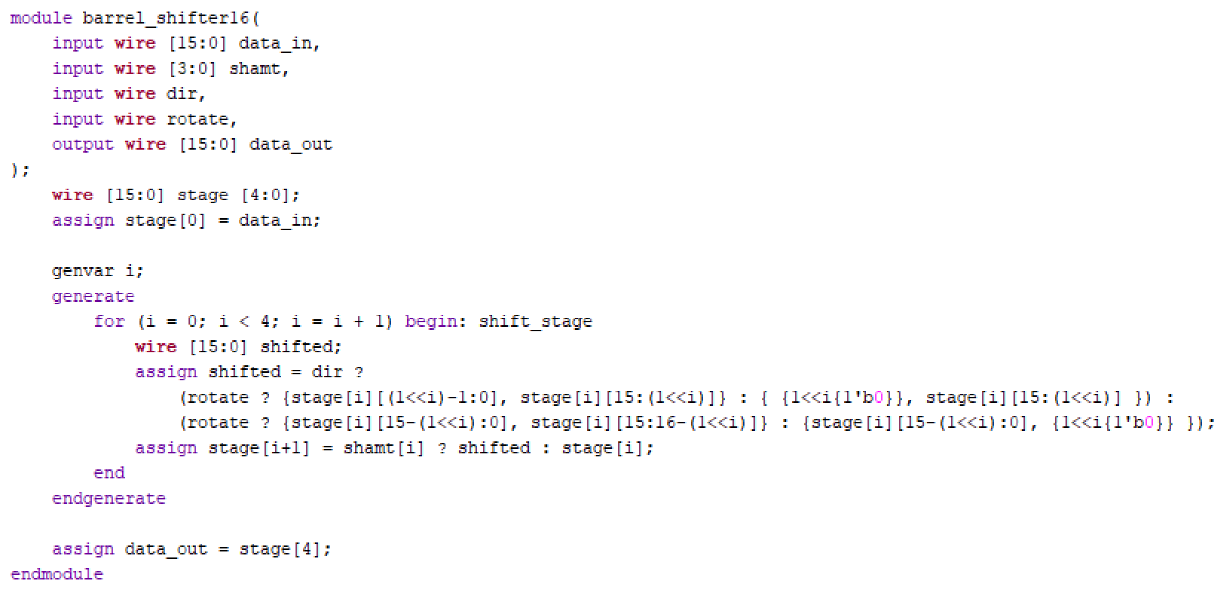
shamt_counter.v



```
module shamt_counter(
    input wire clk,
    input wire rst,
    input wire inc,
    output reg [1:0] shamt_bits = 0
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            shamt_bits <= 0;
        else if (inc)
            shamt_bits <= shamt_bits + 1;
    end
endmodule
```

The shamt_counter module is a simple 2-bit counter used to control the upper two bits of the shift amount (SHAMT[3:2]). It takes a clock, a reset signal (rst), and an increment signal (inc). On every rising edge of the clock, if the reset is high, the counter resets to zero. Otherwise, if inc is high, it increases the count by one. This module was driven by the center button in our lab, which let us cycle through different shift amounts.

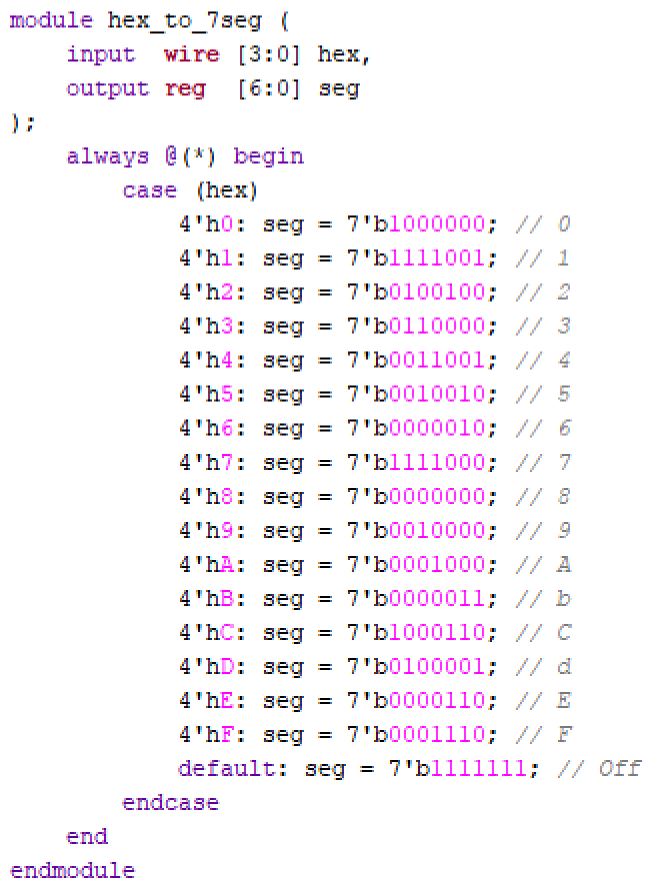
barrel_shifter16.v



```verilog
module barrel_shifter16(
    input wire [15:0] data_in,
    input wire [3:0] shamt,
    input wire dir,
    input wire rotate,
    output wire [15:0] data_out
);
    wire [15:0] stage [4:0];
    assign stage[0] = data_in;

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin: shift_stage
            wire [15:0] shifted;
            assign shifted = dir ?
                (rotate ? {stage[i][(1<<i)-1:0], stage[i][15:(1<<i)]} : { {1<<i{1'b0}}, stage[i][15:(1<<i)] }) :
                (rotate ? {stage[i][15-(1<<i):0], stage[i][15:16-(1<<i)]} : {stage[i][15-(1<<i):0], {1<<i{1'b0}} });
            assign stage[i+1] = shamt[i] ? shifted : stage[i];
        end
    endgenerate

    assign data_out = stage[4];
endmodule
```

The barrel_shifter16 module performs 16-bit shifting and rotating based on user-selected controls. It takes a 16-bit input (data_in), a 4-bit shift amount (shamt), a direction (dir), and a rotate control (rotate). The module uses a 4-stage shift tree, where each stage can optionally shift the input by 1, 2, 4, or 8 bits depending on the corresponding bit in shamt. If dir is 0, the data is shifted or rotated to the left; if it's 1, it shifts or rotates to the right. When rotate is 0, logical shifts are performed (with 0s filling in), and when it's 1, the bits wrap around.

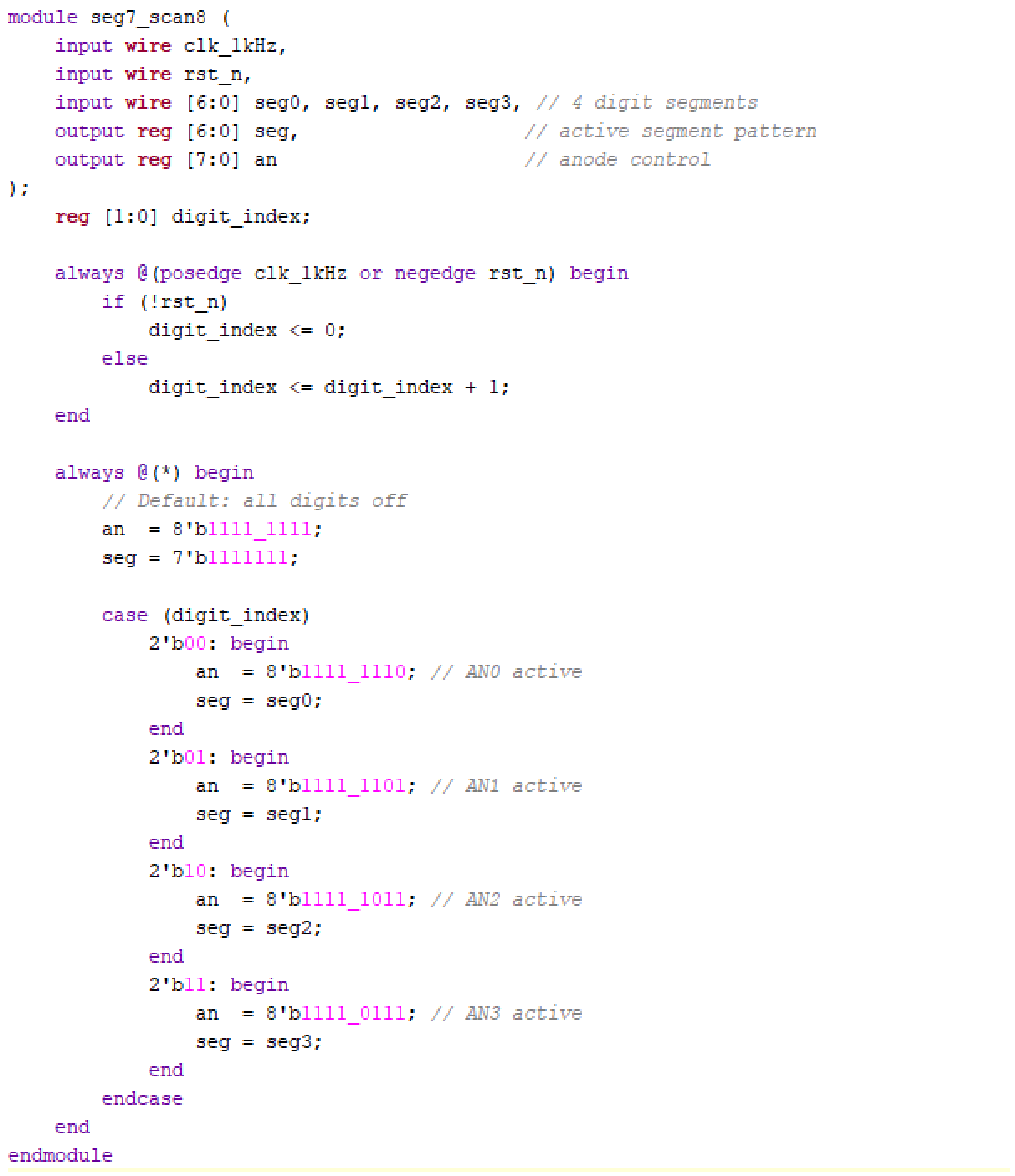
hex_to_7seg.v



```
module hex_to_7seg (
    input  wire [3:0] hex,
    output reg  [6:0] seg
);
    always @(*) begin
        case (hex)
            4'h0: seg = 7'b1000000; // 0
            4'h1: seg = 7'b1111001; // 1
            4'h2: seg = 7'b0100100; // 2
            4'h3: seg = 7'b0110000; // 3
            4'h4: seg = 7'b0011001; // 4
            4'h5: seg = 7'b0010010; // 5
            4'h6: seg = 7'b0000010; // 6
            4'h7: seg = 7'b1111000; // 7
            4'h8: seg = 7'b0000000; // 8
            4'h9: seg = 7'b0010000; // 9
            4'hA: seg = 7'b0001000; // A
            4'hB: seg = 7'b0000011; // b
            4'hC: seg = 7'b1000110; // C
            4'hD: seg = 7'b0100001; // d
            4'hE: seg = 7'b0000110; // E
            4'hF: seg = 7'b0001110; // F
            default: seg = 7'b1111111; // Off
        endcase
    end
endmodule
```

The hex_to_7seg module converts a 4-bit hexadecimal value into the corresponding 7-segment display pattern. It takes a 4-bit input (hex) and outputs a 7-bit signal (seg) that determines which segments should be lit to show numbers 0–9 and letters A–F. Each segment pattern is hardcoded using a case statement, so the correct digit or letter appears on the display. This module was used in our lab to show each 4-bit nibble of the 16-bit barrel shifter output on the 7-segment display. It allowed us to visually verify the result of every shift or rotate operation in real time.

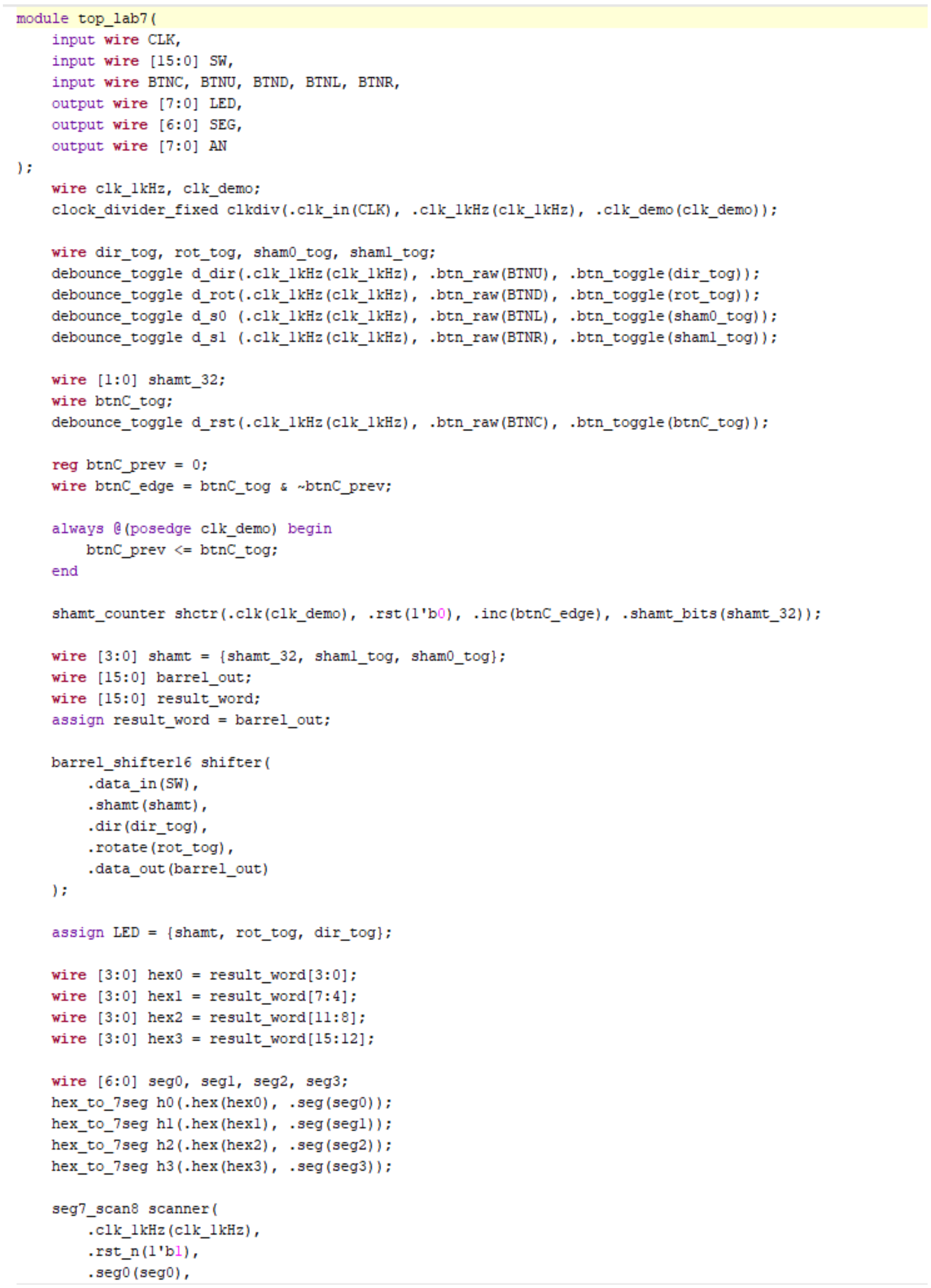
seg7_scan8.v



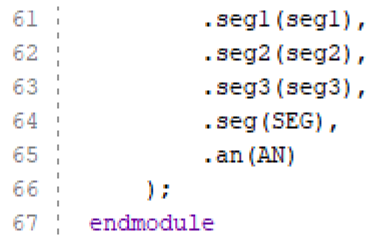
```verilog
module seg7_scan8 (
    input wire clk_1kHz,
    input wire rst_n,
    input wire [6:0] seg0, seg1, seg2, seg3, // 4 digit segments
    output reg [6:0] seg,                   // active segment pattern
    output reg [7:0] an                     // anode control
);
    reg [1:0] digit_index;

    always @(posedge clk_1kHz or negedge rst_n) begin
        if (!rst_n)
            digit_index <= 0;
        else
            digit_index <= digit_index + 1;
    end

    always @(*) begin
        // Default: all digits off
        an  = 8'b1111_1111;
        seg = 7'b1111111;

        case (digit_index)
            2'b00: begin
                an  = 8'b1111_1110; // AN0 active
                seg = seg0;
            end
            2'b01: begin
                an  = 8'b1111_1101; // AN1 active
                seg = seg1;
            end
            2'b10: begin
                an  = 8'b1111_1011; // AN2 active
                seg = seg2;
            end
            2'b11: begin
                an  = 8'b1111_0111; // AN3 active
                seg = seg3;
            end
        endcase
    end
endmodule
```

The seg7_scan8 module is used to control four digits of the 7-segment display by rapidly cycling through them one at a time. It takes in a 1 kHz clock (clk_1kHz), an active-low reset (rst_n), and four pre-decoded 7-segment values (seg0 to seg3). The module uses a 2-bit counter (digit_index) to switch between digits every clock cycle. At any given time, only one digit is active by setting one bit low in the AN output, while the rest remain off.

<u>**top_lab7.v**</u>



```
module top_lab7(
    input wire CLK,
    input wire [15:0] SW,
    input wire BTNC, BTNU, BTND, BTNL, BTNR,
    output wire [7:0] LED,
    output wire [6:0] SEG,
    output wire [7:0] AN
);
    wire clk_1kHz, clk_demo;
    clock_divider_fixed clkdiv(.clk_in(CLK), .clk_1kHz(clk_1kHz), .clk_demo(clk_demo));

    wire dir_tog, rot_tog, sham0_tog, sham1_tog;
    debounce_toggle d_dir(.clk_1kHz(clk_1kHz), .btn_raw(BTNU), .btn_toggle(dir_tog));
    debounce_toggle d_rot(.clk_1kHz(clk_1kHz), .btn_raw(BTND), .btn_toggle(rot_tog));
    debounce_toggle d_s0 (.clk_1kHz(clk_1kHz), .btn_raw(BTNL), .btn_toggle(sham0_tog));
    debounce_toggle d_s1 (.clk_1kHz(clk_1kHz), .btn_raw(BTNR), .btn_toggle(sham1_tog));

    wire [1:0] shamt_32;
    wire btnC_tog;
    debounce_toggle d_rst(.clk_1kHz(clk_1kHz), .btn_raw(BTNC), .btn_toggle(btnC_tog));

    reg btnC_prev = 0;
    wire btnC_edge = btnC_tog & ~btnC_prev;

    always @(posedge clk_demo) begin
        btnC_prev <= btnC_tog;
    end

    shamt_counter shctr(.clk(clk_demo), .rst(1'b0), .inc(btnC_edge), .shamt_bits(shamt_32));

    wire [3:0] shamt = {shamt_32, sham1_tog, sham0_tog};
    wire [15:0] barrel_out;
    wire [15:0] result_word;
    assign result_word = barrel_out;

    barrel_shifter16 shifter(
        .data_in(SW),
        .shamt(shamt),
        .dir(dir_tog),
        .rotate(rot_tog),
        .data_out(barrel_out)
    );

    assign LED = {shamt, rot_tog, dir_tog};

    wire [3:0] hex0 = result_word[3:0];
    wire [3:0] hex1 = result_word[7:4];
    wire [3:0] hex2 = result_word[11:8];
    wire [3:0] hex3 = result_word[15:12];

    wire [6:0] seg0, seg1, seg2, seg3;
    hex_to_7seg h0(.hex(hex0), .seg(seg0));
    hex_to_7seg h1(.hex(hex1), .seg(seg1));
    hex_to_7seg h2(.hex(hex2), .seg(seg2));
    hex_to_7seg h3(.hex(hex3), .seg(seg3));

    seg7_scan8 scanner(
        .clk_1kHz(clk_1kHz),
        .rst_n(1'b1),
        .seg0(seg0),
```



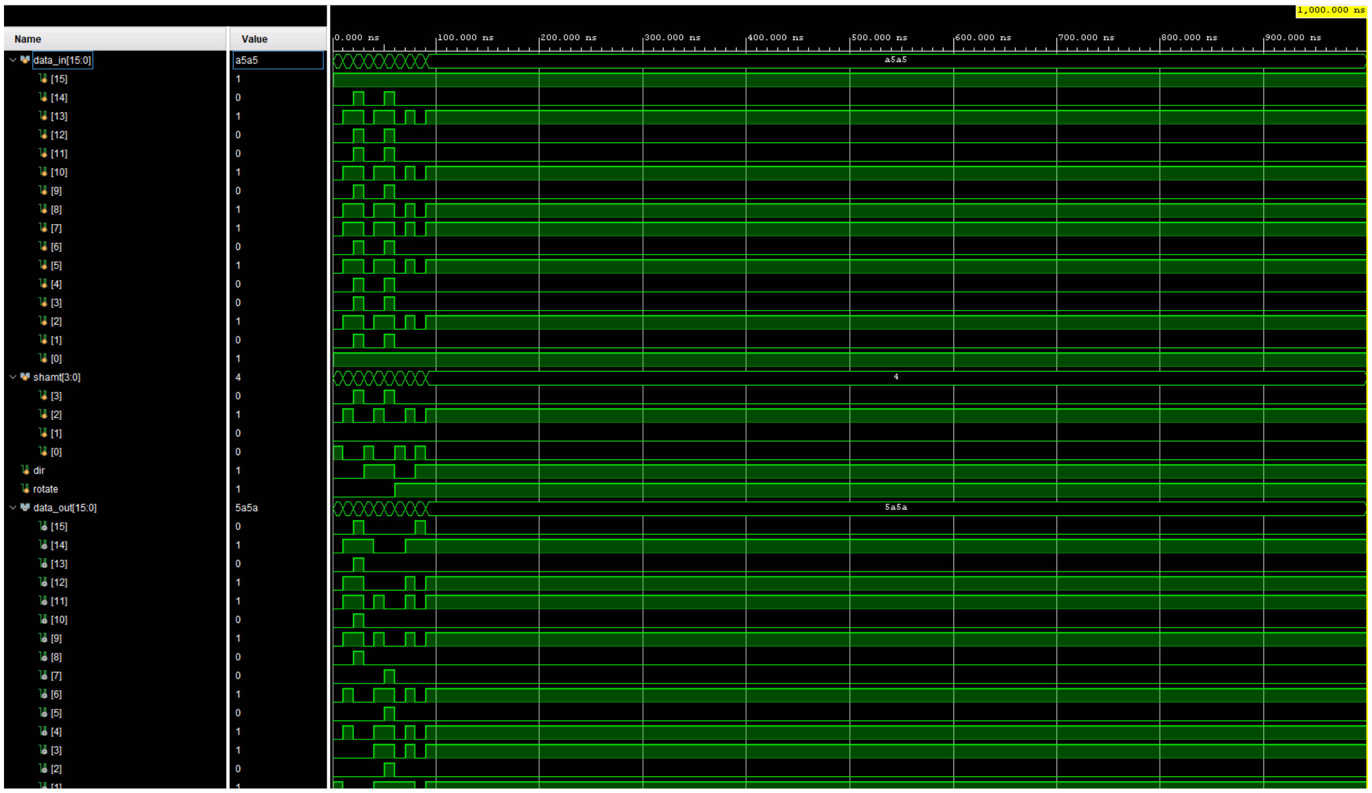
```
        .seg1(seg1),
        .seg2(seg2),
        .seg3(seg3),
        .seg(SEG),
        .an(AN)
    );
endmodule
```

The top_lab7 module connects all the other components together to implement the full functionality of the lab on the Nexys A7 FPGA. It takes in the 100 MHz system clock, the 16-bit input word from the switches, and the five on-board push-buttons. It generates the control signals for the barrel shifter using debounced toggle modules for DIR, ROT, and SHAMT[1:0], while SHAMT[3:2] is incremented through a debounced and edge-detected BTNC button. The shift result is calculated using the barrel_shifter16 module and displayed on four 7-segment digits. The hex_to_7seg modules convert each nibble of the result into the appropriate 7-segment pattern, and the seg7_scan8 module handles digit multiplexing. The LED output is used to show the current values of the shift amount, direction, and rotation mode for debugging. Overall, this top-level module ties everything together and ensures that user inputs are correctly processed and visually displayed during the lab dem

Testbenches

barrel_shifter16_tb.v

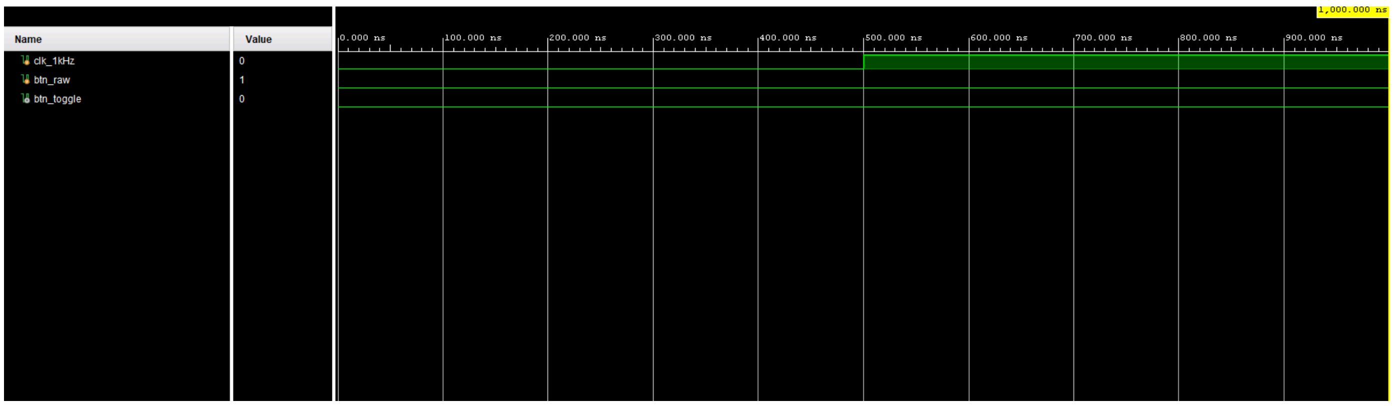
```
module barrel_shifter16_tb;

    reg [15:0] data_in;
    reg [3:0]  shamt;
    reg        dir;
    reg        rotate;
    wire [15:0] data_out;

    barrel_shifter16 uut (
        .data_in(data_in),
        .shamt(shamt),
        .dir(dir),
        .rotate(rotate),
        .data_out(data_out)
    );

    task run_test(
        input [15:0] word,
        input [3:0] shift_amount,
        input dir_val,
        input rotate_val
    );
        begin
            data_in = word;
            shamt   = shift_amount;
            dir     = dir_val;
            rotate  = rotate_val;
            #10;
            $display("data_in = %h | shamt = %0d | dir = %s | rotate = %s | data_out = %h",
                     data_in, shamt,
                     dir ? "Right" : "Left",
                     rotate ? "Rotate" : "Logical",
                     data_out);
        end
    endtask

    initial begin
        $display("Starting barrel_shifter16_tb\n");

        // Logical Left
        run_test(16'h8001, 4'd1, 0, 0);
        run_test(16'hA5A5, 4'd4, 0, 0);
        run_test(16'hFFFF, 4'd8, 0, 0);

        // Logical Right
        run_test(16'h8001, 4'd1, 1, 0);
        run_test(16'hA5A5, 4'd4, 1, 0);
        run_test(16'hFFFF, 4'd8, 1, 0);

        // Rotate Left
        run_test(16'h8001, 4'd1, 0, 1);
        run_test(16'hA5A5, 4'd4, 0, 1);

        // Rotate Right
        run_test(16'h8001, 4'd1, 1, 1);
        run_test(16'hA5A5, 4'd4, 1, 1);

        $display("\nbarrel_shifter16_tb completed.");
    end
```



```
Starting barrel_shifter16_tb

data_in = 8001 | shamt = 1 | dir =  Left | rotate = Logical | data_out = 0002
data_in = a5a5 | shamt = 4 | dir =  Left | rotate = Logical | data_out = 5a50
data_in = ffff | shamt = 8 | dir =  Left | rotate = Logical | data_out = ff00
data_in = 8001 | shamt = 1 | dir = Right | rotate = Logical | data_out = 4000
data_in = a5a5 | shamt = 4 | dir = Right | rotate = Logical | data_out = 0a5a
data_in = ffff | shamt = 8 | dir = Right | rotate = Logical | data_out = 00ff
data_in = 8001 | shamt = 1 | dir =  Left | rotate =  Rotate | data_out = 0003
data_in = a5a5 | shamt = 4 | dir =  Left | rotate =  Rotate | data_out = 5a5a
data_in = 8001 | shamt = 1 | dir = Right | rotate =  Rotate | data_out = c000
data_in = a5a5 | shamt = 4 | dir = Right | rotate =  Rotate | data_out = 5a5a
```

debounce_toggle_tb.v

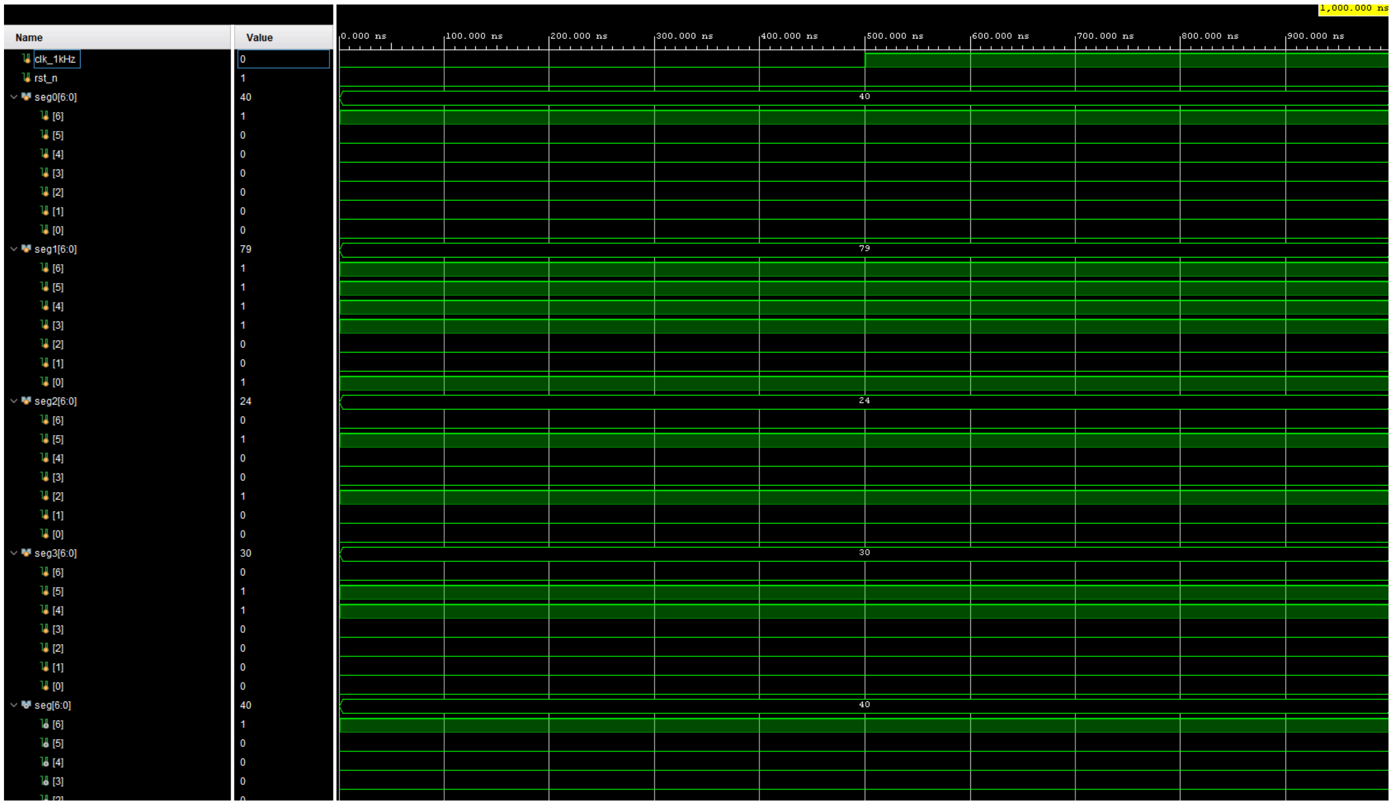
```
`timescale 1ns / 1ps

module debounce_toggle_tb;
    reg clk_1kHz = 0;
    reg btn_raw = 0;
    wire btn_toggle;


    debounce_toggle uut (
        .clk_1kHz(clk_1kHz),
        .btn_raw(btn_raw),
        .btn_toggle(btn_toggle)
    );


    always #500 clk_1kHz = ~clk_1kHz;

    // Display transitions
    initial begin
        $display("Testing debounce_toggle");
        $monitor("time=%0t btn_raw=%b btn_toggle=%b", $time, btn_raw, btn_toggle);

        #1000;

        // First press with bouncing
        $display("\n-- First press with glitches --");
        btn_raw = 1; #100;
        btn_raw = 0; #50;
        btn_raw = 1; #30;
        btn_raw = 0; #20;
        btn_raw = 1; #10000;
        btn_raw = 0; #1000;

        $display("\nEnd of debounce test. btn_toggle should change only twice.");
        #1000;
        $finish;
    end
endmodule
```



```
Testing debounce_toggle
time=0 btn_raw=0 btn_toggle=0

-- First press with glitches --
time=1000000 btn_raw=1 btn_toggle=0
```

seg7_scan8_tb.v

```
`timescale 1ns / 1ps
module seg7_scan8_tb;
    reg clk_1kHz = 0;
    reg rst_n = 1;
    reg [6:0] seg0 = 7'b1000000; // 0
    reg [6:0] seg1 = 7'b1111001; // 1
    reg [6:0] seg2 = 7'b0100100; // 2
    reg [6:0] seg3 = 7'b0110000; // 3
    wire [6:0] seg;
    wire [7:0] an;

    seg7_scan8 uut(
        .clk_1kHz(clk_1kHz),
        .rst_n(rst_n),
        .seg0(seg0),
        .seg1(seg1),
        .seg2(seg2),
        .seg3(seg3),
        .seg(seg),
        .an(an)
    );

    always #500 clk_1kHz = ~clk_1kHz;

    initial begin
        $display("Testing seg7_scan8 4-digit scan cycle");
        $monitor("time=%0t an=%b seg=%b", $time, an, seg);
        rst_n = 0; #1000;
        rst_n = 1;
        #8000;
        $finish;
    end
endmodule
```

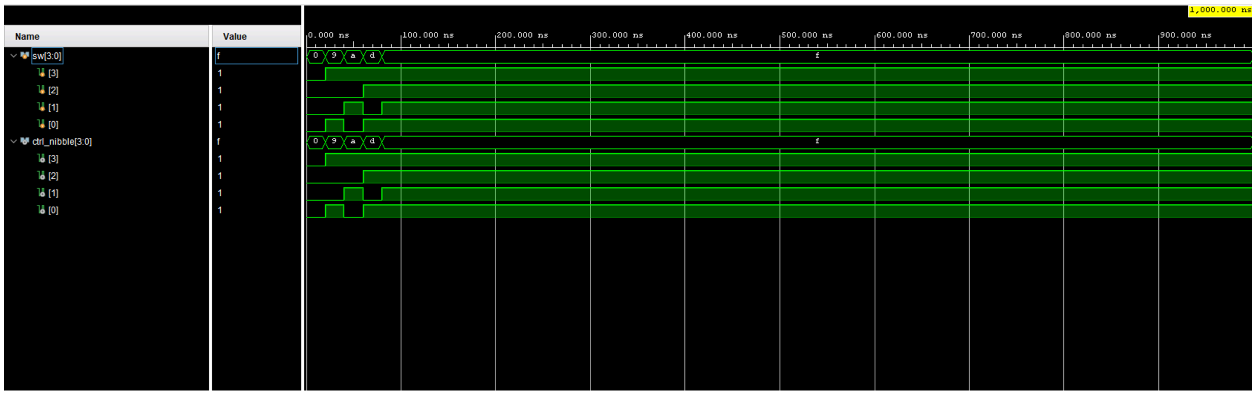


```
Testing seg7_scan8 4-digit scan cycle
time=0 an=1111110 seg=1000000
```

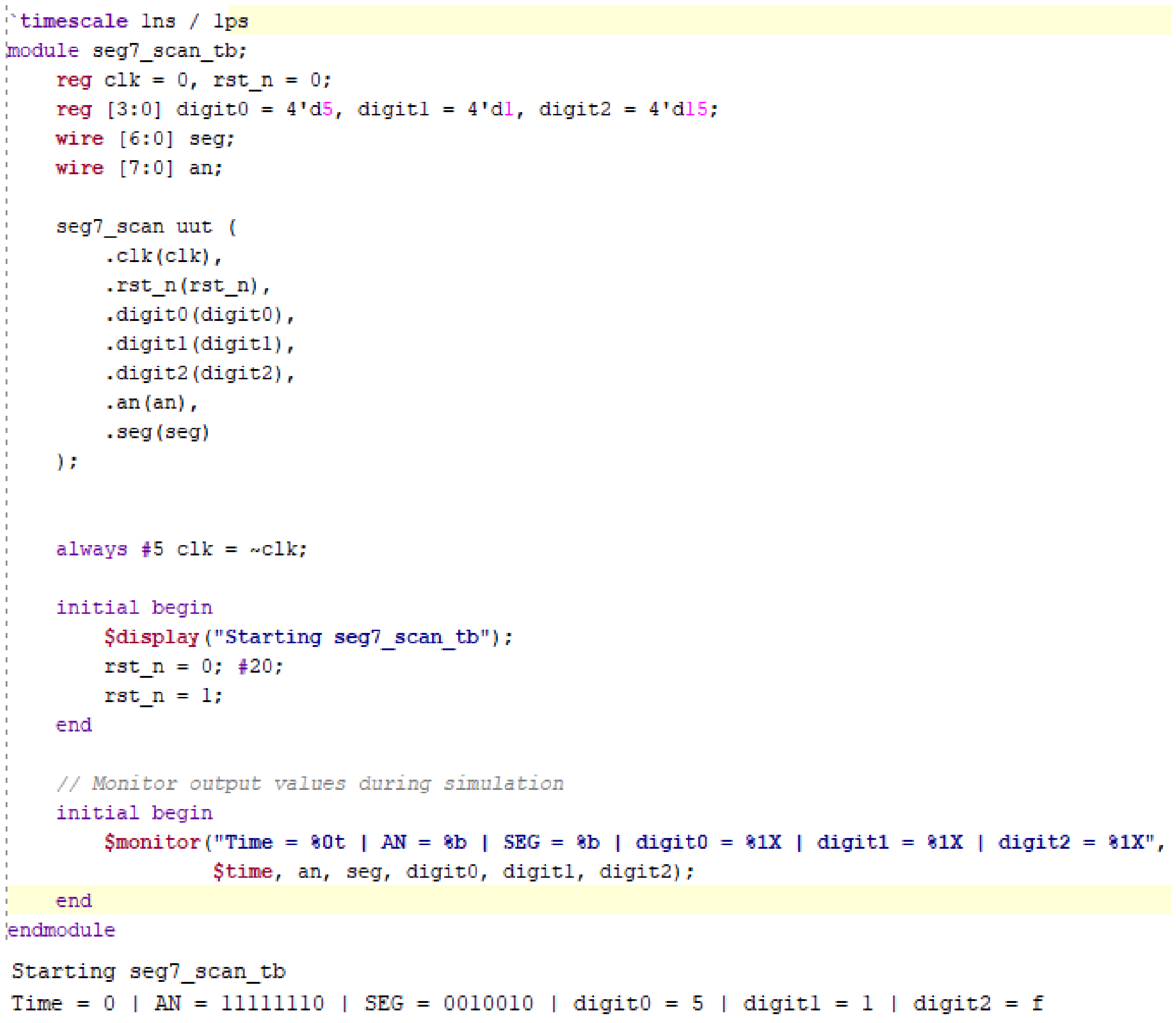
control_decoder_tb.v

```verilog
`timescale 1ns / 1ps
module control_decoder_tb;
    reg [3:0] sw;
    wire [3:0] ctrl_nibble;

    control_decoder uut (
        .sw(sw),
        .ctrl_nibble(ctrl_nibble)
    );

    initial begin
        $display("Starting control_decoder_tb");
        $monitor("SW = %b (%1X) -> ctrl_nibble = %b (%1X)", sw, sw, ctrl_nibble, ctrl_nibble);

        sw = 4'b0000; #20;  // Expect 0
        sw = 4'b1001; #20;  // Expect 9
        sw = 4'b1010; #20;  // Expect A
        sw = 4'b1101; #20;  // Expect D
        sw = 4'b1111; #20;  // Expect F
    end
endmodule
```

```
Starting control_decoder_tb
SW = 0000 (0) -> ctrl_nibble = 0000 (0)
SW = 1001 (9) -> ctrl_nibble = 1001 (9)
SW = 1010 (a) -> ctrl_nibble = 1010 (a)
SW = 1101 (d) -> ctrl_nibble = 1101 (d)
SW = 1111 (f) -> ctrl_nibble = 1111 (f)
```

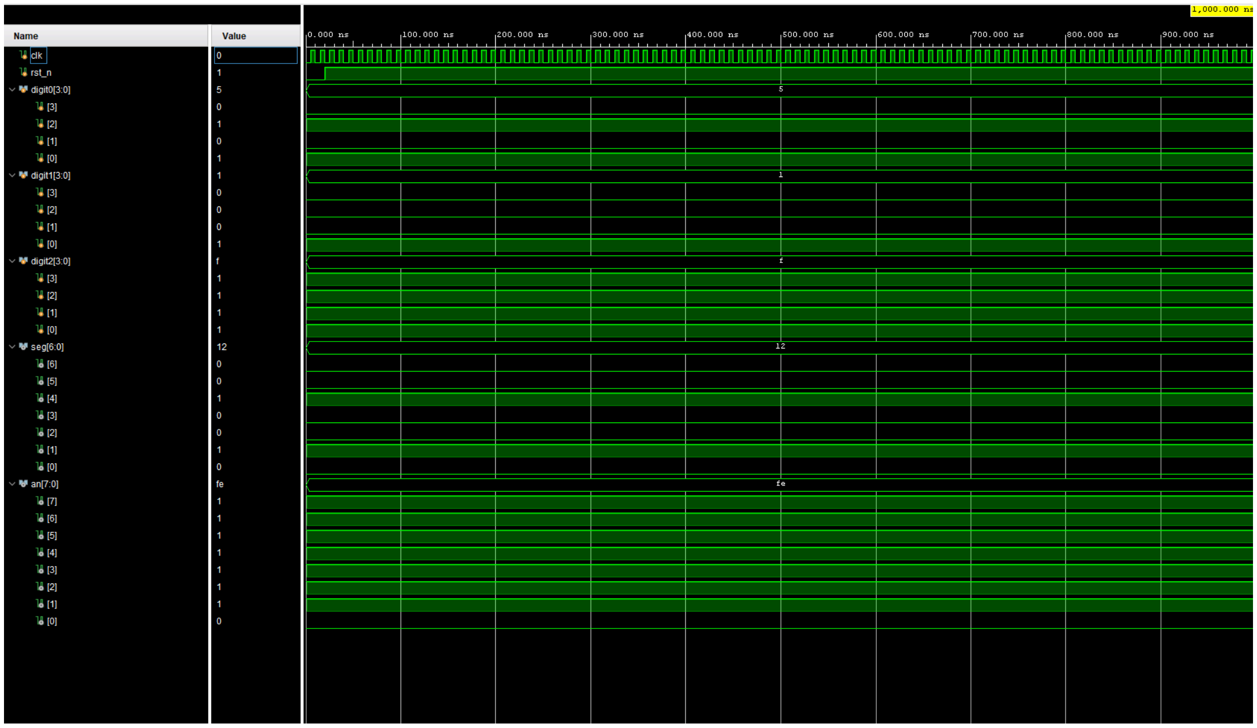


tb_seg7_scan.v



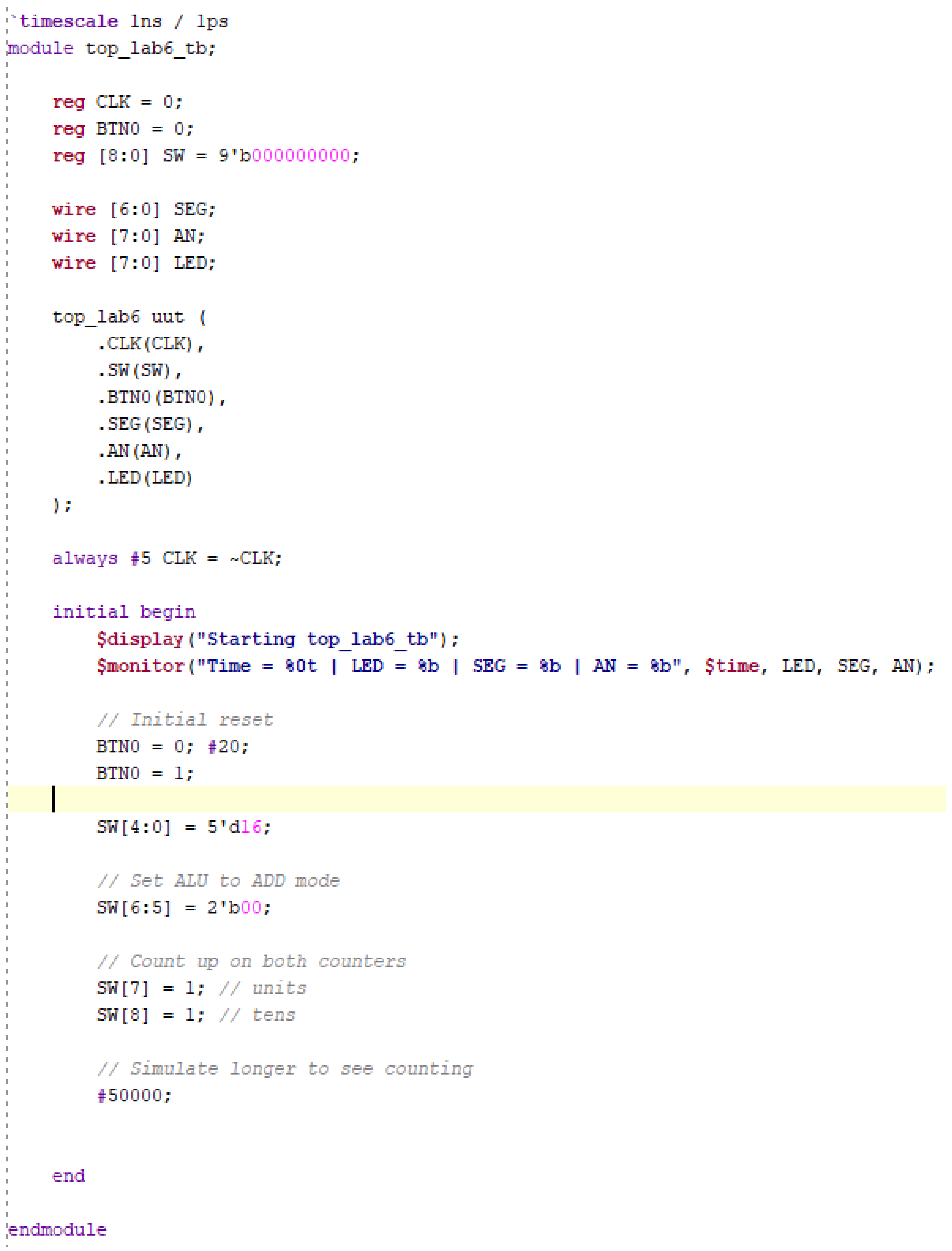
```verilog
`timescale 1ns / 1ps
module seg7_scan_tb;
    reg clk = 0, rst_n = 0;
    reg [3:0] digit0 = 4'd5, digit1 = 4'd1, digit2 = 4'd15;
    wire [6:0] seg;
    wire [7:0] an;

    seg7_scan uut (
        .clk(clk),
        .rst_n(rst_n),
        .digit0(digit0),
        .digit1(digit1),
        .digit2(digit2),
        .an(an),
        .seg(seg)
    );


    always #5 clk = ~clk;

    initial begin
        $display("Starting seg7_scan_tb");
        rst_n = 0; #20;
        rst_n = 1;
    end

    // Monitor output values during simulation
    initial begin
        $monitor("Time = %0t | AN = %b | SEG = %b | digit0 = %1X | digit1 = %1X | digit2 = %1X",
                 $time, an, seg, digit0, digit1, digit2);
    end
endmodule
```

```
Starting seg7_scan_tb
Time = 0 | AN = 11111110 | SEG = 0010010 | digit0 = 5 | digit1 = 1 | digit2 = f
```



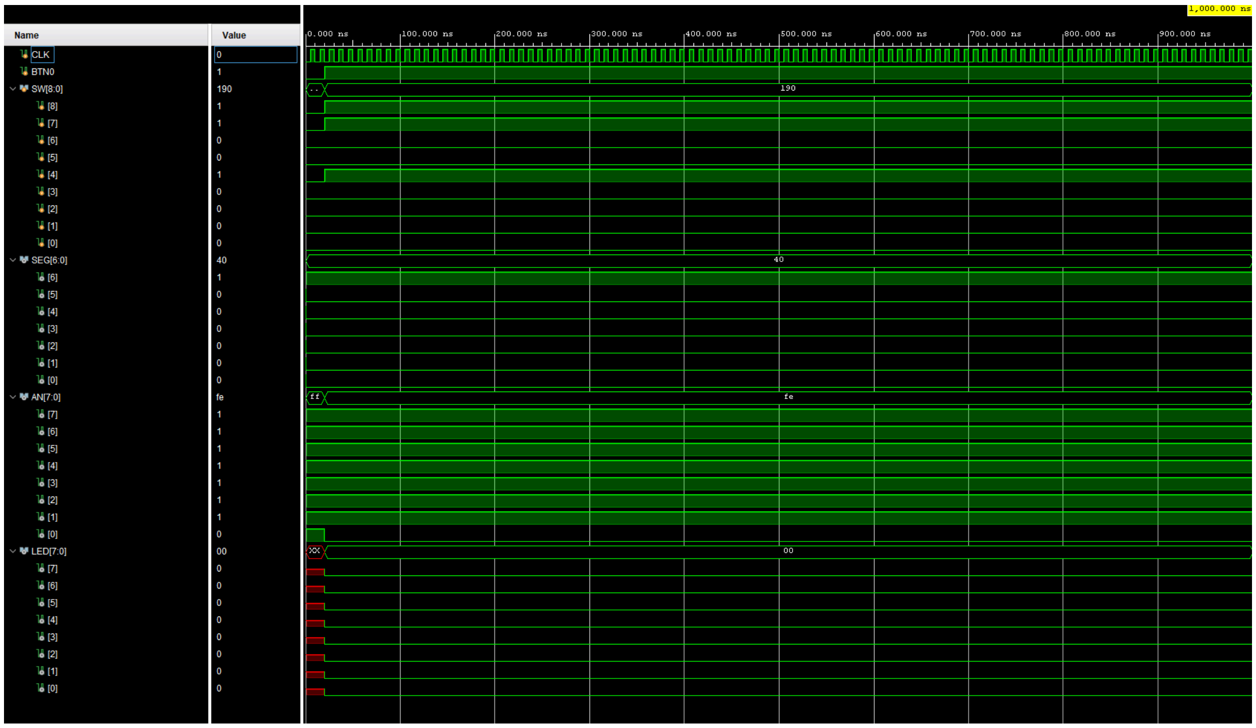
Name	Value
clk	0
rst_n	1
digit0[3:0]	5
[3]	0
[2]	1
[1]	0
[0]	1
digit1[3:0]	1
[3]	0
[2]	0
[1]	0
[0]	1
digit2[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1
seg[6:0]	12
[6]	0
[5]	0
[4]	1
[3]	0
[2]	0
[1]	1
[0]	0
an[7:0]	fe
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	1
[1]	1
[0]	0

top_lab6_tb.v



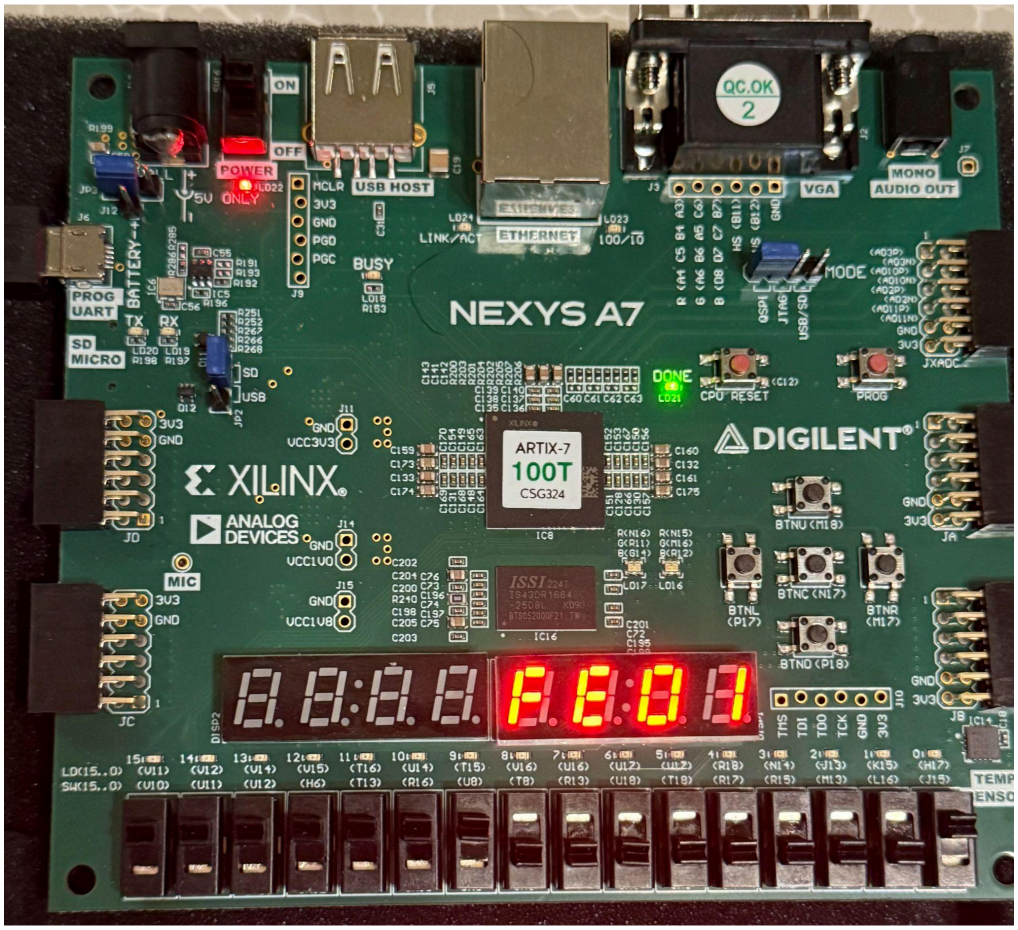
```
`timescale 1ns / 1ps
module top_lab6_tb;

    reg CLK = 0;
    reg BTN0 = 0;
    reg [8:0] SW = 9'b000000000;

    wire [6:0] SEG;
    wire [7:0] AN;
    wire [7:0] LED;

    top_lab6 uut (
        .CLK(CLK),
        .SW(SW),
        .BTN0(BTN0),
        .SEG(SEG),
        .AN(AN),
        .LED(LED)
    );

    always #5 CLK = ~CLK;

    initial begin
        $display("Starting top_lab6_tb");
        $monitor("Time = %0t | LED = %b | SEG = %b | AN = %b", $time, LED, SEG, AN);

        // Initial reset
        BTN0 = 0; #20;
        BTN0 = 1;

        SW[4:0] = 5'd16;

        // Set ALU to ADD mode
        SW[6:5] = 2'b00;

        // Count up on both counters
        SW[7] = 1; // units
        SW[8] = 1; // tens

        // Simulate longer to see counting
        #50000;


    end

endmodule
```

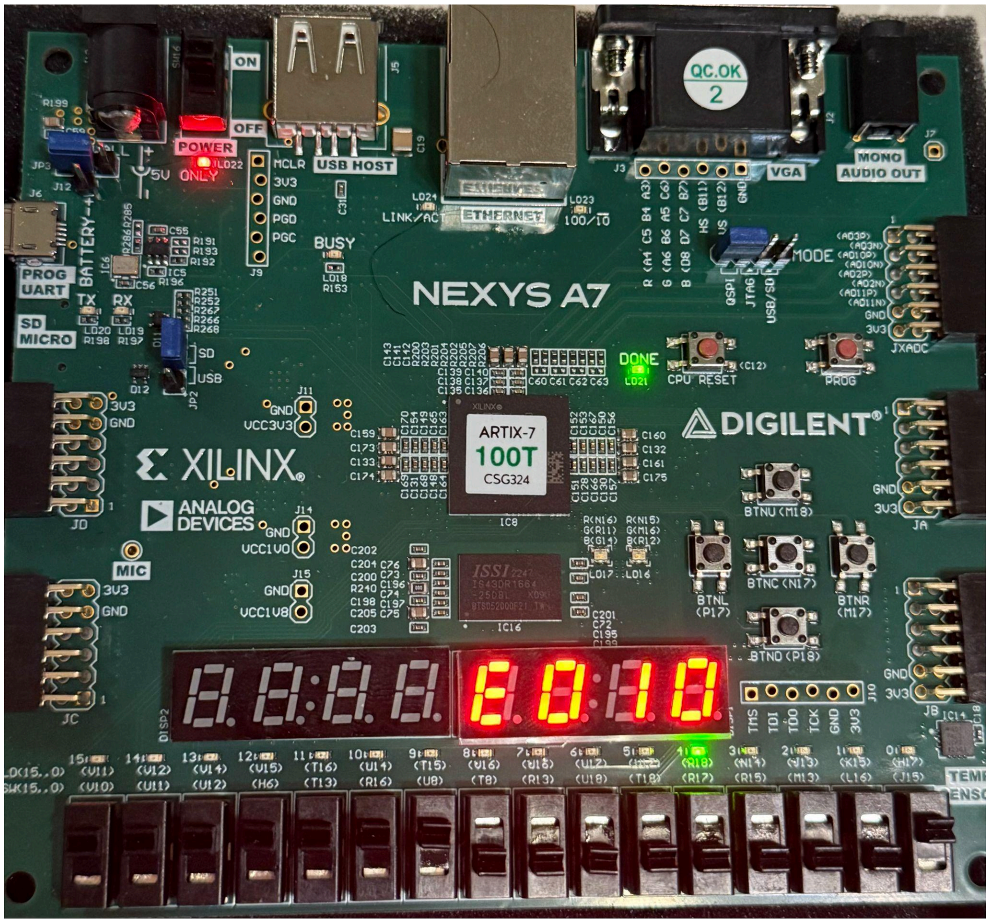
```
Starting top_lab6_tb
Time = 0 | LED = xxxxxxxx | SEG = 1000000 | AN = 11111111
Time = 20000 | LED = 00000000 | SEG = 1000000 | AN = 11111110
```



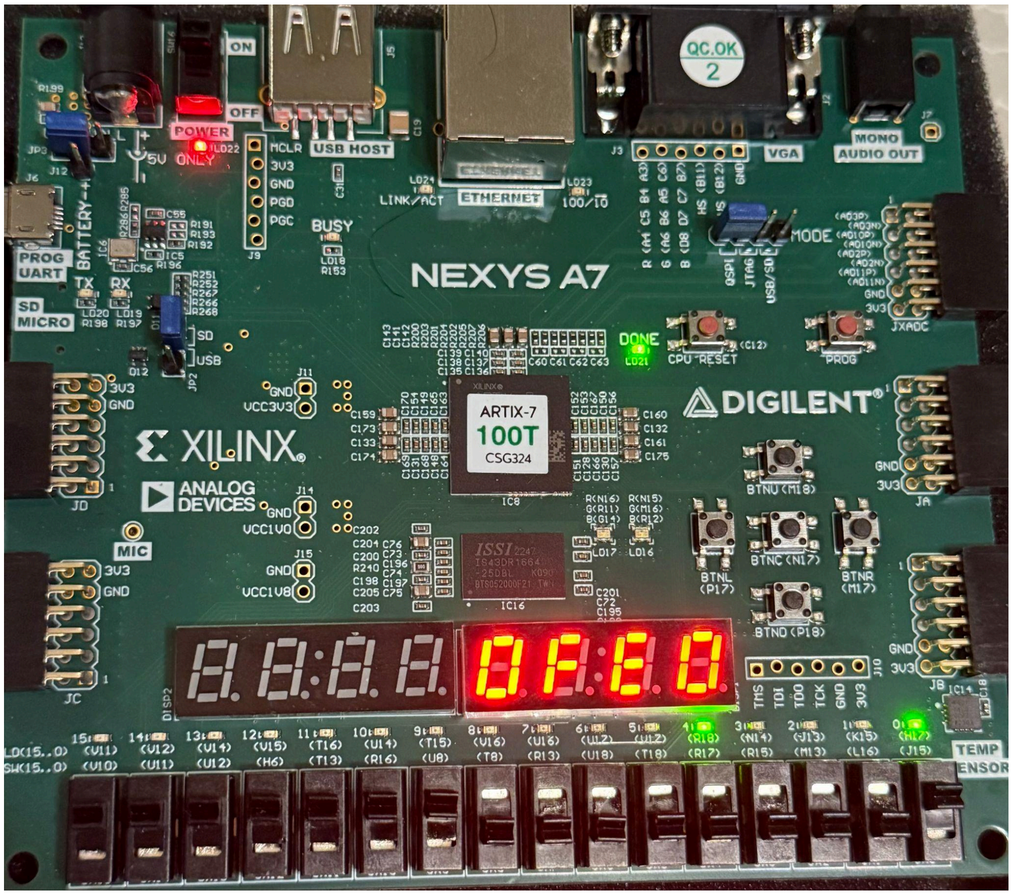
Name	Value
CLK	0
BTN0	1
SW[8:0]	190
[8]	1
[7]	1
[6]	0
[5]	0
[4]	1
[3]	0
[2]	0
[1]	0
[0]	0
SEG[6:0]	40
[6]	1
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0
[0]	0
AN[7:0]	fe
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	1
[1]	1
[0]	0
LED[7:0]	00
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0
[0]	0

Hardware Results

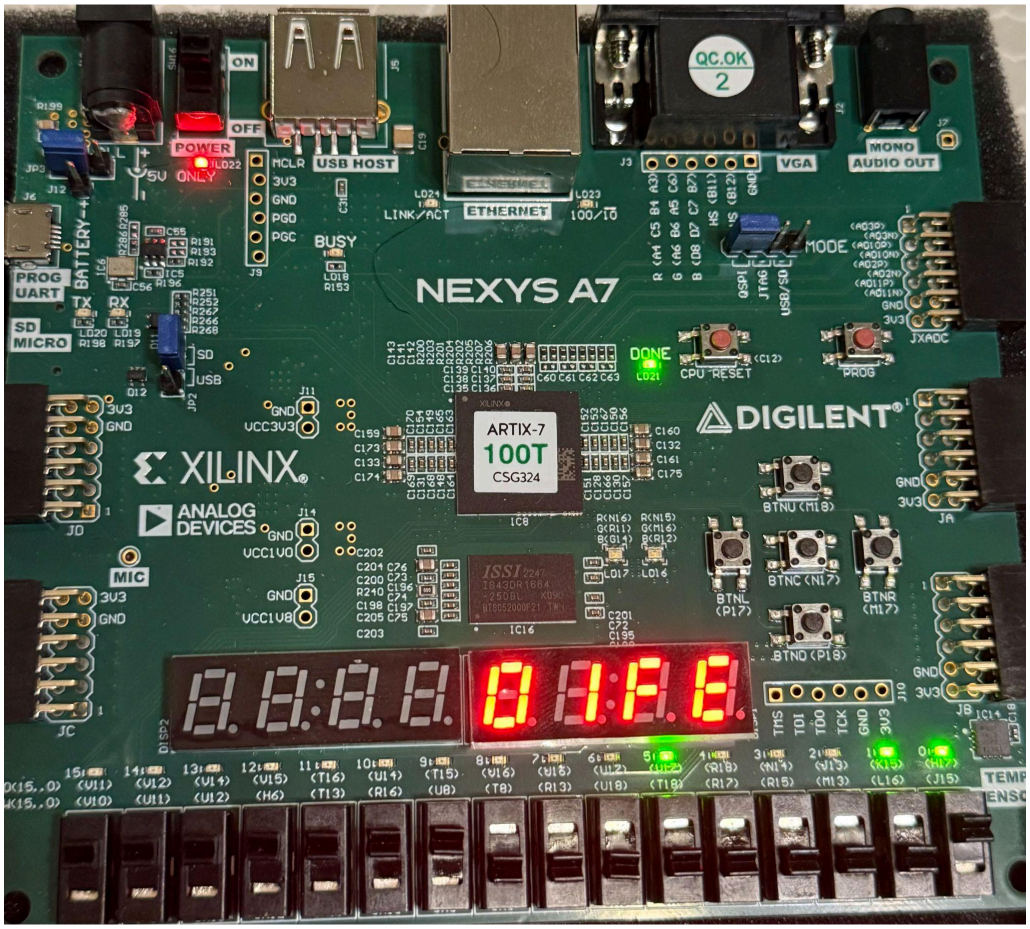

Default output



Increment shamt by 1 which shifts left

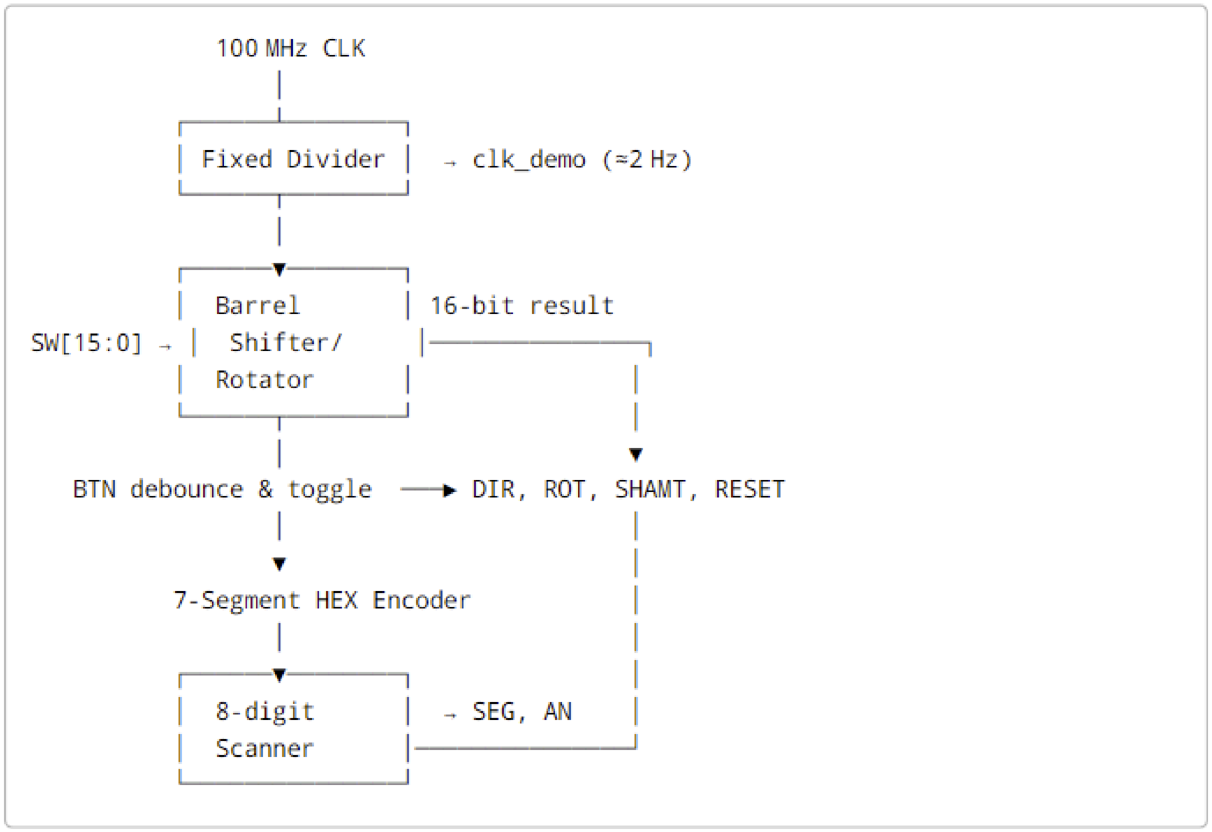


dir = 1 so it bit shift direction changes to right



dir = 1 and rot = 1 so it wraps instead of inserting 0s when shifting leading us eventually to original output

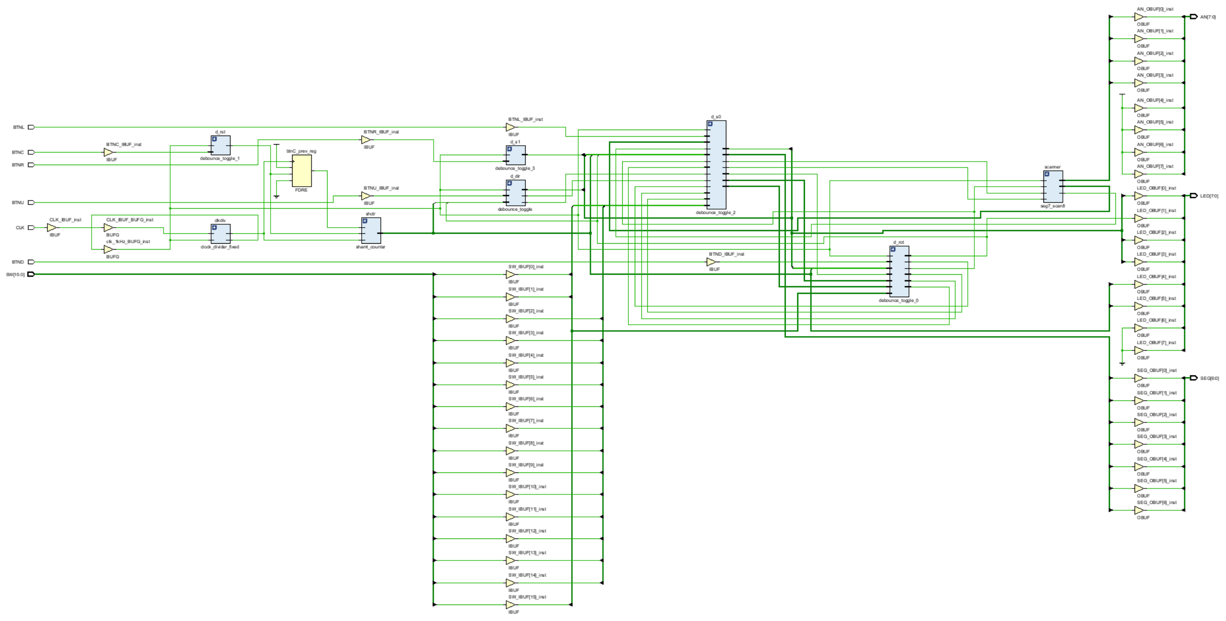
Block Diagram



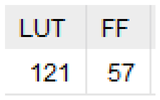
Provided by the professor, this is the diagram that represents the top module.

Screenshots

Schematic

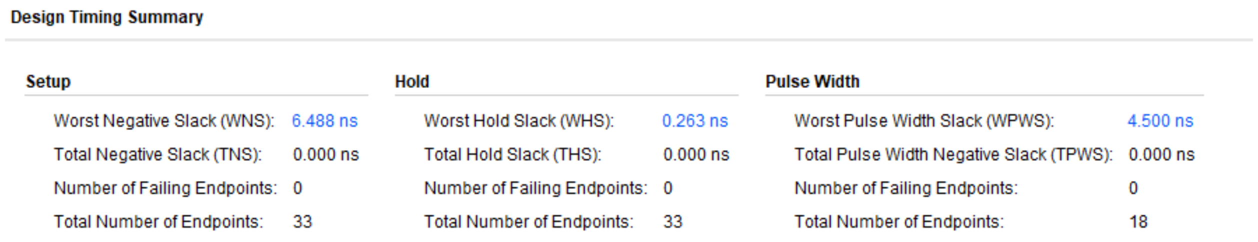


LUT & FF



LUT	FF
121	57

Timing Summary



Design Timing Summary

Setup

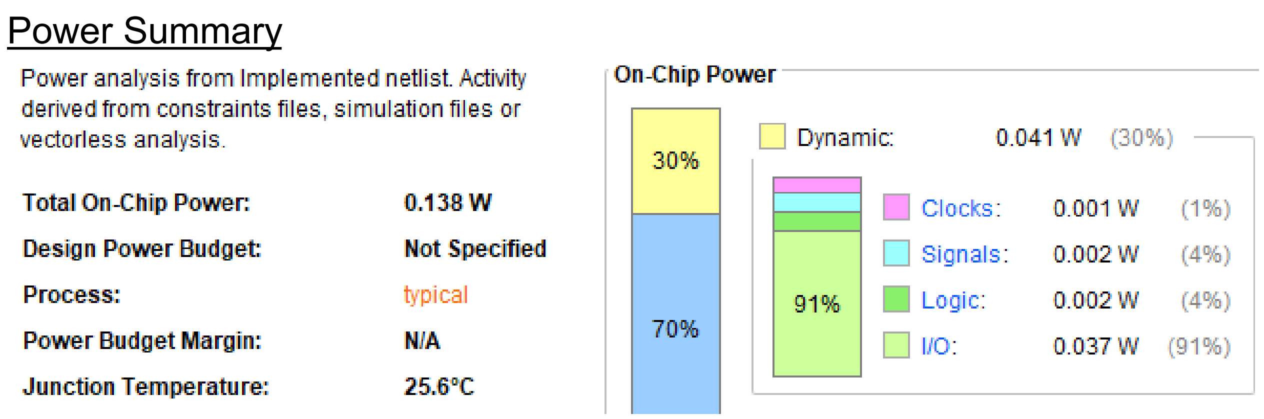
Worst Negative Slack (WNS): 6.488 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 33

Hold

Worst Hold Slack (WHS): 0.263 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 33

Pulse Width

Worst Pulse Width Slack (WPWS): 4.500 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 18



Power Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.138 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**

On-Chip Power

Dynamic:		0.041 W	(30%)
	Clocks:	0.001 W	(1%)
	Signals:	0.002 W	(4%)
	Logic:	0.002 W	(4%)
	I/O:	0.037 W	(91%)

Conclusion

This lab gave us hands-on experience building and testing a fully functional 16-bit barrel shifter system on the Nexys A7 FPGA board. We successfully implemented logical and rotational shift operations in both directions, controlled entirely through debounced push-buttons. By combining modular Verilog design with clock division and display multiplexing, we were able to show shifting behavior on a 4-digit 7-segment display. We reinforced key concepts like MUX-based shifting, edge detection, button debouncing, and display scanning.

Contributions

Faris (50%) - Some source code, helped with lab report, demo

Nicholas (50%) - Some of the source code, testbench and simulations helped with report