BCD Up/Down Counter on 7-Segment Display

3300L

Dia Agrawal and Robert Lainez Torres

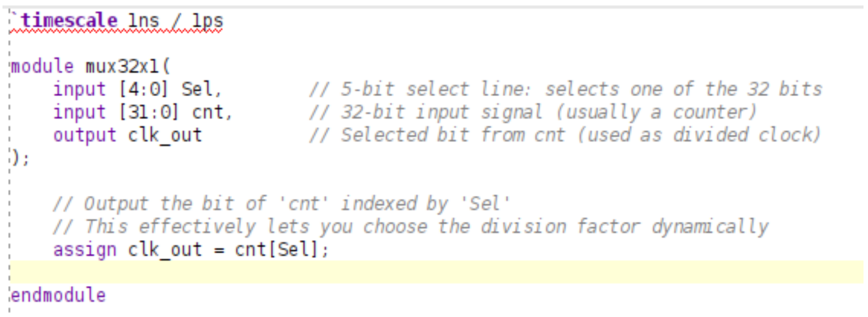
Design Overview

The goal of this design was to create a functional two-digit BCD up/down counter that could be displayed using a time-multiplexed 7-segment display. A 32-bit clock divider continuously increments from the board clock, and a 32×1 multiplexer selects a specific bit based on SW[4:0] to create a configurable slower clock signal. This selected bit is edge-detected to generate a clean one-shot tick pulse, which acts as the enable signal for the units digit of the BCD counter. The direction of counting is controlled using BTN1, and reset is handled through BTN0. Two bcd_digit_counter modules are cascaded so that the tens digit only increments or decrements when the units digit rolls over. The result is passed to a seg7_scan module, which uses an internal counter to rapidly alternate between the two digits, creating the illusion of a static display. The design was validated through simulation of each individual module, and verified on hardware using the Nexys A7.

Mux32x1

Source Code

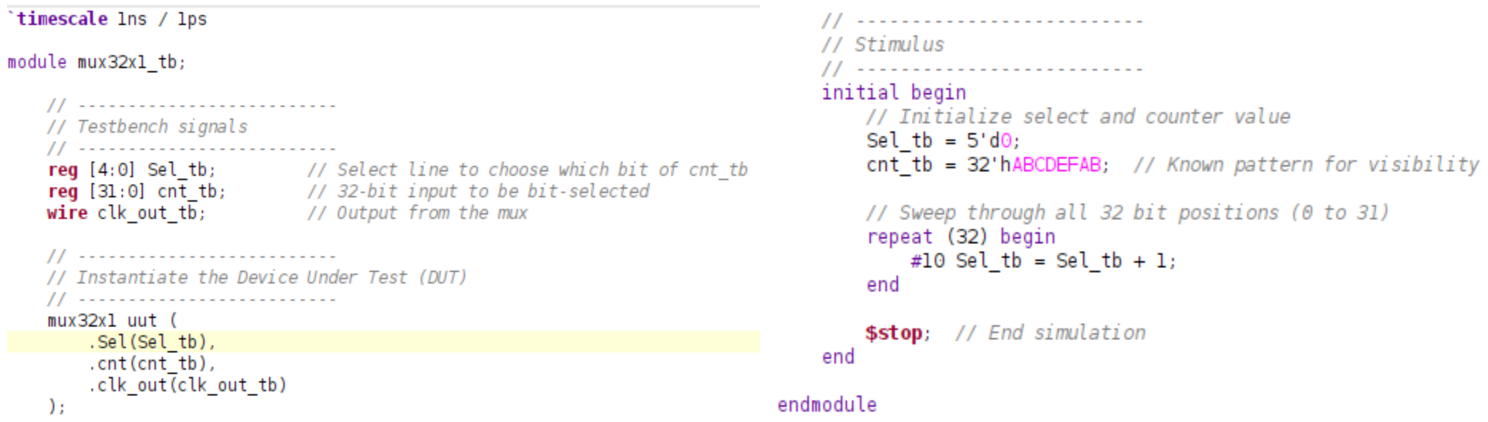
The Mux32x1 module takes in a 32-bit counter and a 5-bit select signal (Sel), and outputs one specific bit from the counter as clk_out. This allows dynamic frequency selection by letting the user choose which bit of the free-running counter to use.



```
`timescale 1ns / 1ps

module mux32x1(
    input [4:0] Sel,        // 5-bit select line: selects one of the 32 bits
    input [31:0] cnt,       // 32-bit input signal (usually a counter)
    output clk_out          // Selected bit from cnt (used as divided clock)
);

    // Output the bit of 'cnt' indexed by 'Sel'
    // This effectively lets you choose the division factor dynamically
    assign clk_out = cnt[Sel];

endmodule
```

Testbench Code

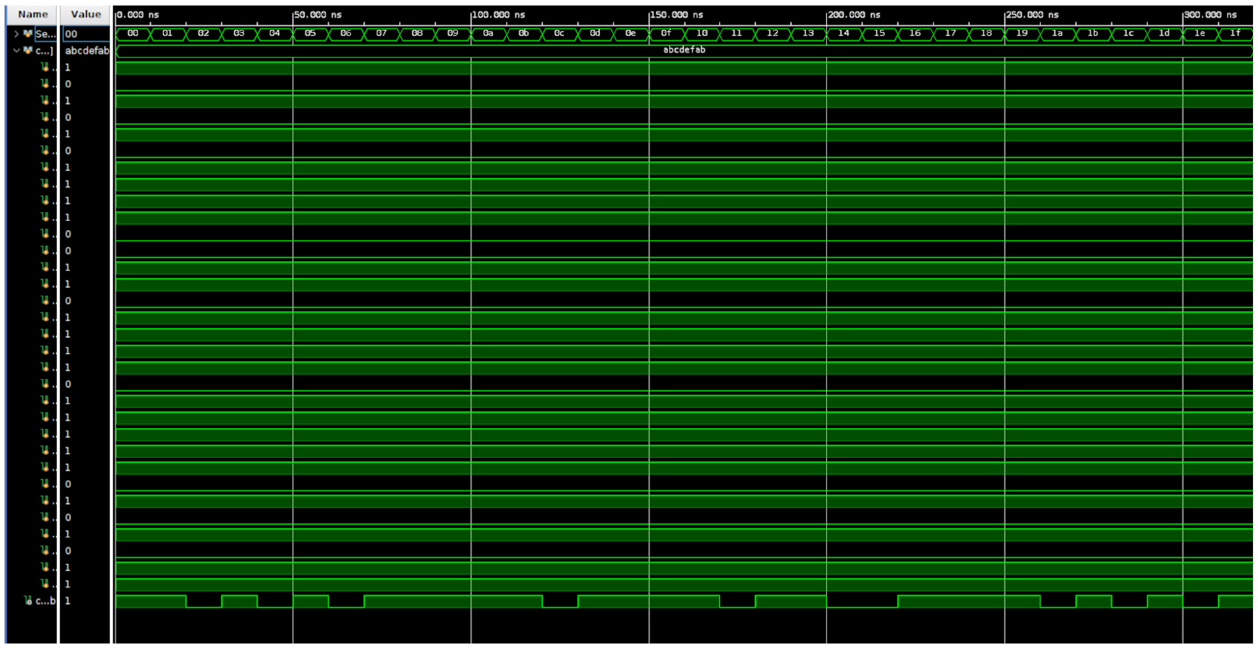
The testbench initializes the counter with a known pattern and increments the Sel signal over time to verify that the output clk_out changes according to the selected bit. This confirms that the mux is correctly selecting cnt[Sel].



```
`timescale 1ns / 1ps

module mux32x1_tb;

    // --------------------------
    // Testbench signals
    // --------------------------
    reg [4:0] Sel_tb;         // Select line to choose which bit of cnt_tb
    reg [31:0] cnt_tb;        // 32-bit input to be bit-selected
    wire clk_out_tb;          // Output from the mux

    // --------------------------
    // Instantiate the Device Under Test (DUT)
    // --------------------------
    mux32x1 uut (
        .Sel(Sel_tb),
        .cnt(cnt_tb),
        .clk_out(clk_out_tb)
    );

    // --------------------------
    // Stimulus
    // --------------------------
    initial begin
        // Initialize select and counter value
        Sel_tb = 5'd0;
        cnt_tb = 32'hABCDEFAB;  // Known pattern for visibility

        // Sweep through all 32 bit positions (0 to 31)
        repeat (32) begin
            #10 Sel_tb = Sel_tb + 1;
        end

        $stop;  // End simulation
    end

endmodule
```

Behavioral Simulation

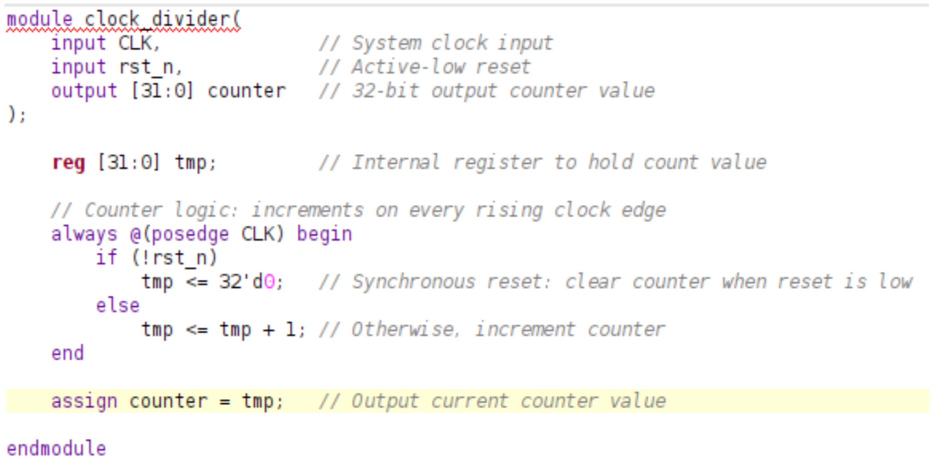
The simulation waveform showed clk_out toggling at different frequencies based on Sel_tb, proving that the MUX is behaving correctly and matching the selected bit from the counter.



Clock_divider

Source Code:

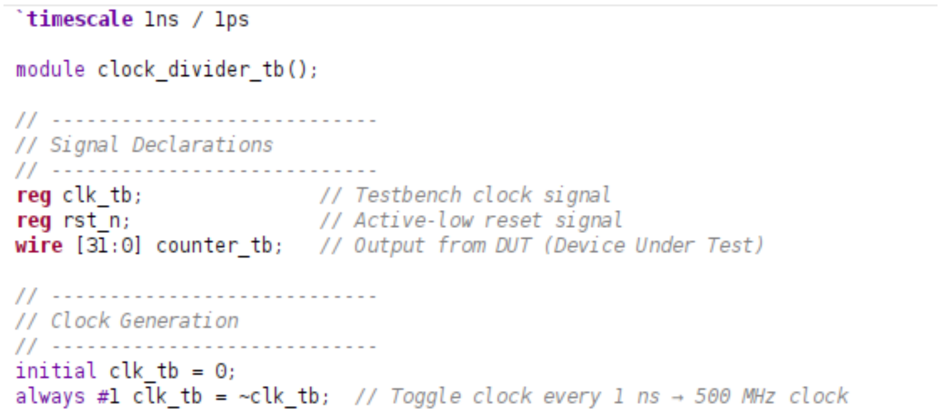
The clock_divider module is a simple 32-bit counter that increments on every positive clock edge. It serves as the frequency base for the mux and lets us create a wide range of slower clocks just by selecting one of the 32 bits.



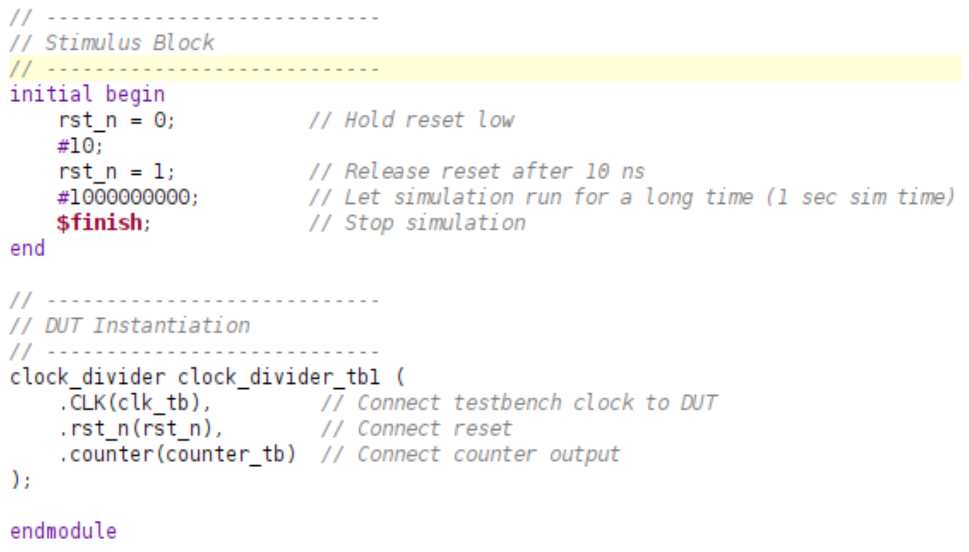
```
module clock_divider(
    input CLK,              // System clock input
    input rst_n,            // Active-low reset
    output [31:0] counter   // 32-bit output counter value
);

    reg [31:0] tmp;         // Internal register to hold count value

    // Counter logic: increments on every rising clock edge
    always @(posedge CLK) begin
        if (!rst_n)
            tmp <= 32'd0;   // Synchronous reset: clear counter when reset is low
        else
            tmp <= tmp + 1; // Otherwise, increment counter
    end

    assign counter = tmp;   // Output current counter value

endmodule
```

TestBench Code

The testbench toggles the clock and observes the value of the counter over time. A long simulation period ensures we can confirm that each bit of the counter toggles at a predictable rate (bit 0 = fastest, bit 31 = slowest).



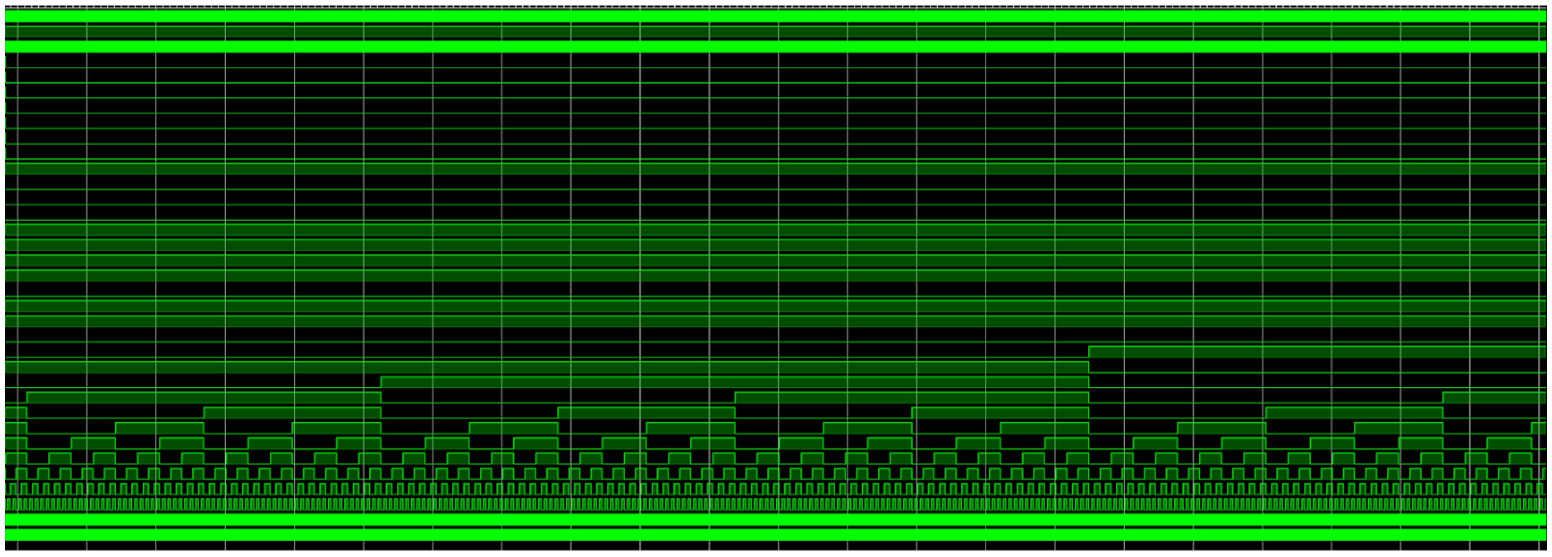
```
`timescale 1ns / 1ps

module clock_divider_tb();

// ---------------------------
// Signal Declarations
// ---------------------------
reg clk_tb;                 // Testbench clock signal
reg rst_n;                  // Active-low reset signal
wire [31:0] counter_tb;     // Output from DUT (Device Under Test)

// ---------------------------
// Clock Generation
// ---------------------------
initial clk_tb = 0;
always #1 clk_tb = ~clk_tb;  // Toggle clock every 1 ns → 500 MHz clock
```



```
// ----------------------------
// Stimulus Block
// ----------------------------
initial begin
    rst_n = 0;           // Hold reset low
    #10;
    rst_n = 1;           // Release reset after 10 ns
    #1000000000;         // Let simulation run for a long time (1 sec sim time)
    $finish;             // Stop simulation
end

// ----------------------------
// DUT Instantiation
// ----------------------------
clock_divider clock_divider_tb1 (
    .CLK(clk_tb),         // Connect testbench clock to DUT
    .rst_n(rst_n),        // Connect reset
    .counter(counter_tb)  // Connect counter output
);

endmodule
```

Behavioral Simulation

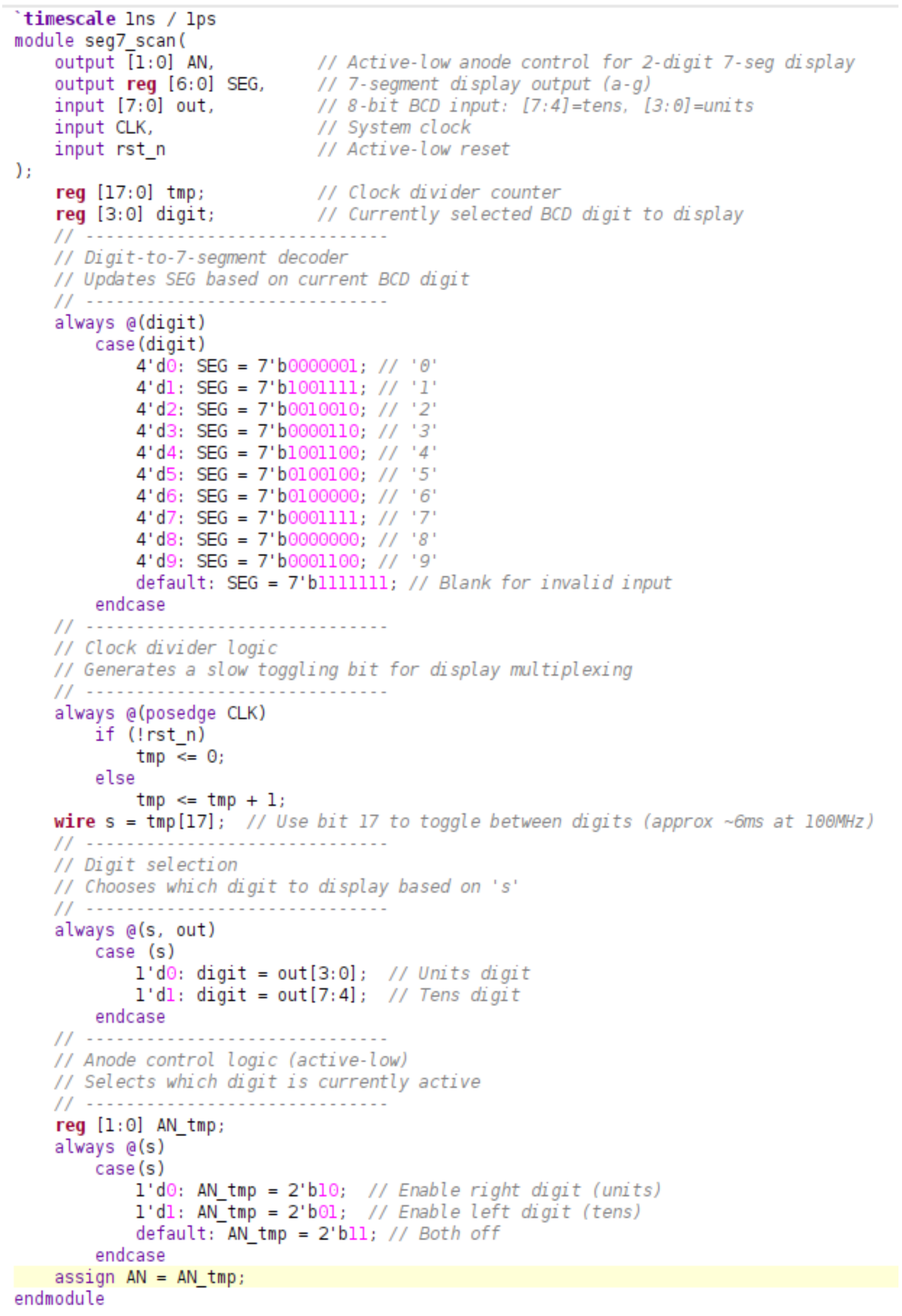
The waveform clearly showed the counter incrementing and confirmed the binary rollover pattern. Higher bits toggled slower as expected, verifying the timing behavior of the divider.



<u>Seg7_scan</u>

Source Code

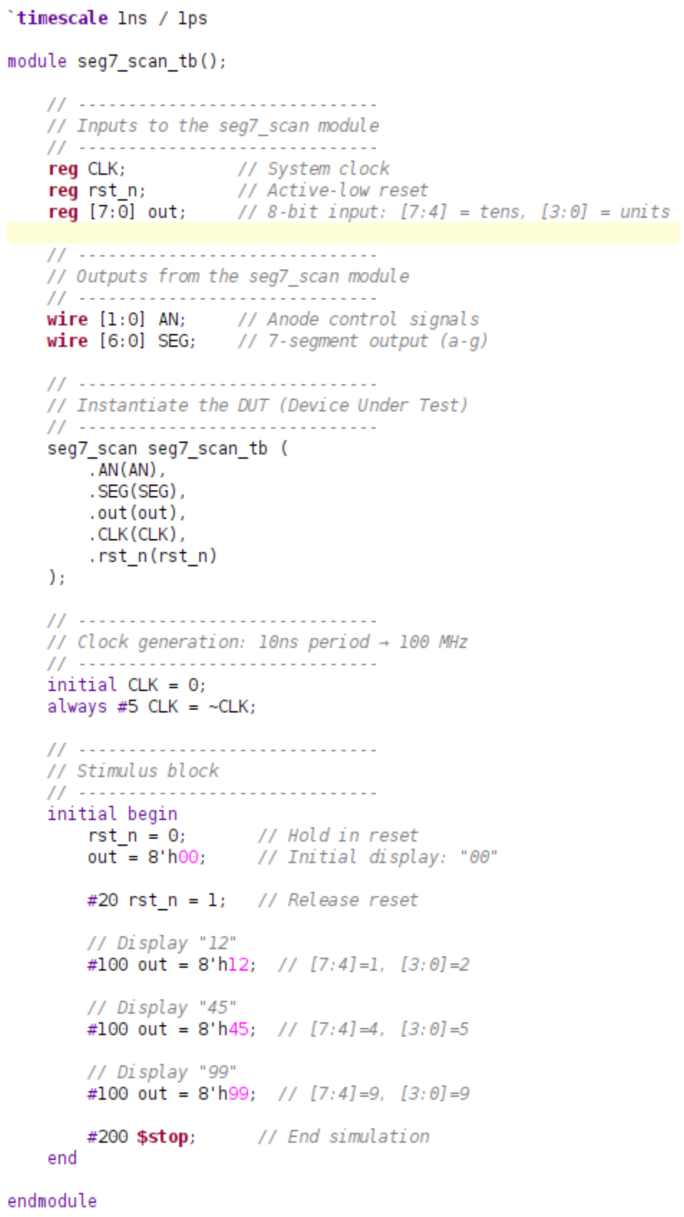
The seg7_scan module takes an 8-bit BCD value and drives a 2-digit 7-segment display by scanning between the digits using a slower internal clock. It splits the value into two 4-bit nibbles and maps each to its corresponding 7-segment code.



```
`timescale 1ns / 1ps
module seg7_scan(
    output [1:0] AN,          // Active-low anode control for 2-digit 7-seg display
    output reg [6:0] SEG,     // 7-segment display output (a-g)
    input [7:0] out,          // 8-bit BCD input: [7:4]=tens, [3:0]=units
    input CLK,                // System clock
    input rst_n               // Active-low reset
);
    reg [17:0] tmp;           // Clock divider counter
    reg [3:0] digit;          // Currently selected BCD digit to display
    // ------------------------------
    // Digit-to-7-segment decoder
    // Updates SEG based on current BCD digit
    // ------------------------------
    always @(digit)
        case(digit)
            4'd0: SEG = 7'b0000001; // '0'
            4'd1: SEG = 7'b1001111; // '1'
            4'd2: SEG = 7'b0010010; // '2'
            4'd3: SEG = 7'b0000110; // '3'
            4'd4: SEG = 7'b1001100; // '4'
            4'd5: SEG = 7'b0100100; // '5'
            4'd6: SEG = 7'b0100000; // '6'
            4'd7: SEG = 7'b0001111; // '7'
            4'd8: SEG = 7'b0000000; // '8'
            4'd9: SEG = 7'b0001100; // '9'
            default: SEG = 7'b1111111; // Blank for invalid input
        endcase
    // ------------------------------
    // Clock divider logic
    // Generates a slow toggling bit for display multiplexing
    // ------------------------------
    always @(posedge CLK)
        if (!rst_n)
            tmp <= 0;
        else
            tmp <= tmp + 1;
    wire s = tmp[17];  // Use bit 17 to toggle between digits (approx ~6ms at 100MHz)
    // ------------------------------
    // Digit selection
    // Chooses which digit to display based on 's'
    // ------------------------------
    always @(s, out)
        case (s)
            1'd0: digit = out[3:0];  // Units digit
            1'd1: digit = out[7:4];  // Tens digit
        endcase
    // ------------------------------
    // Anode control logic (active-low)
    // Selects which digit is currently active
    // ------------------------------
    reg [1:0] AN_tmp;
    always @(s)
        case(s)
            1'd0: AN_tmp = 2'b10;  // Enable right digit (units)
            1'd1: AN_tmp = 2'b01;  // Enable left digit (tens)
            default: AN_tmp = 2'b11; // Both off
        endcase
    assign AN = AN_tmp;
endmodule
```

TestBench Code

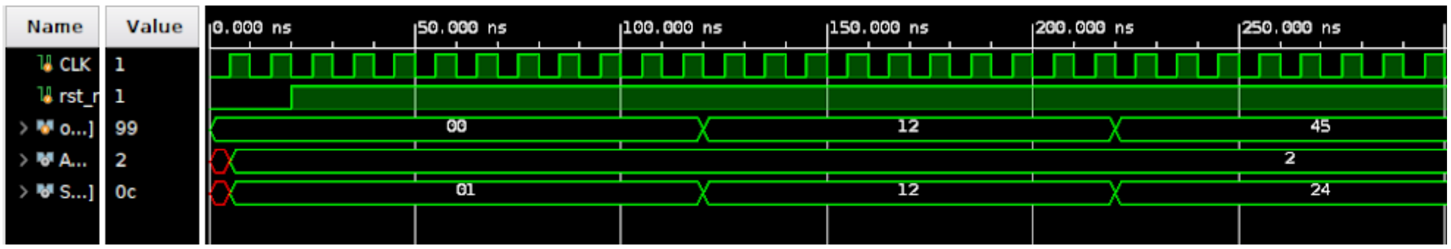
The testbench supplies known BCD values to the out input and verifies that the correct 7-segment encoding appears on SEG while AN toggles between digits. It tests combinations like 12, 45, and 99.

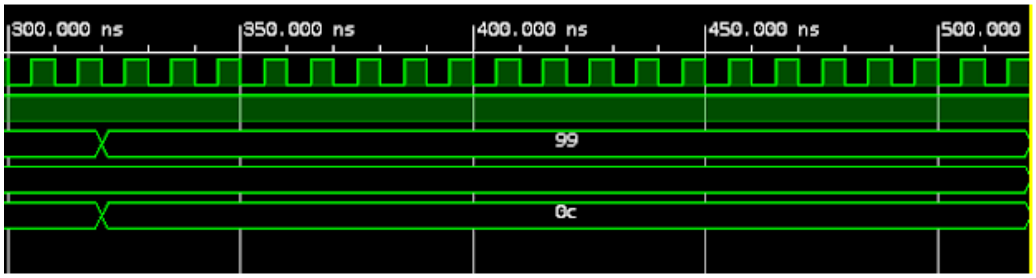


```
`timescale 1ns / 1ps

module seg7_scan_tb();

    // ----------------------------
    // Inputs to the seg7_scan module
    // ----------------------------
    reg CLK;            // System clock
    reg rst_n;          // Active-low reset
    reg [7:0] out;      // 8-bit input: [7:4] = tens, [3:0] = units

    // ----------------------------
    // Outputs from the seg7_scan module
    // ----------------------------
    wire [1:0] AN;      // Anode control signals
    wire [6:0] SEG;     // 7-segment output (a-g)

    // ----------------------------
    // Instantiate the DUT (Device Under Test)
    // ----------------------------
    seg7_scan seg7_scan_tb (
        .AN(AN),
        .SEG(SEG),
        .out(out),
        .CLK(CLK),
        .rst_n(rst_n)
    );

    // ----------------------------
    // Clock generation: 10ns period → 100 MHz
    // ----------------------------
    initial CLK = 0;
    always #5 CLK = ~CLK;

    // ----------------------------
    // Stimulus block
    // ----------------------------
    initial begin
        rst_n = 0;       // Hold in reset
        out = 8'h00;     // Initial display: "00"

        #20 rst_n = 1;   // Release reset

        // Display "12"
        #100 out = 8'h12;  // [7:4]=1, [3:0]=2

        // Display "45"
        #100 out = 8'h45;  // [7:4]=4, [3:0]=5

        // Display "99"
        #100 out = 8'h99;  // [7:4]=9, [3:0]=9

        #200 $stop;      // End simulation
    end

endmodule
```

Behavioral Simulation

The simulation showed correct toggling between digits with valid segment outputs for each value. The anode select alternated at a steady rate, confirming the scanning behavior was working.

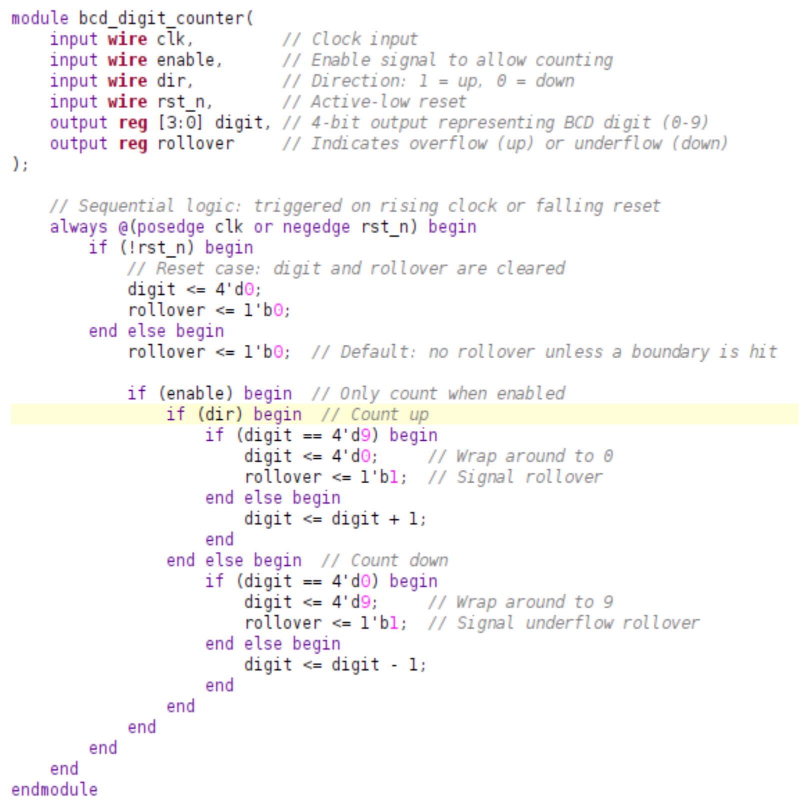




Bcd_up_down_counter

Source Code

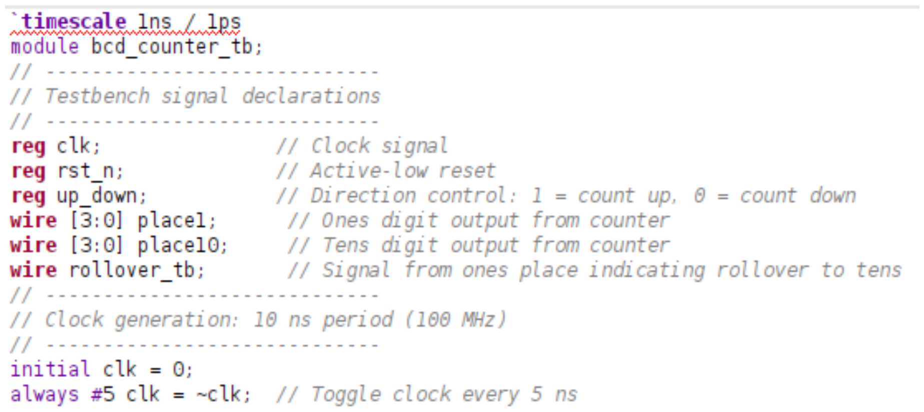
The simulation showed correct toggling between digits with valid segment outputs for each value. The anode select alternated at a steady rate, confirming the scanning behavior was working.



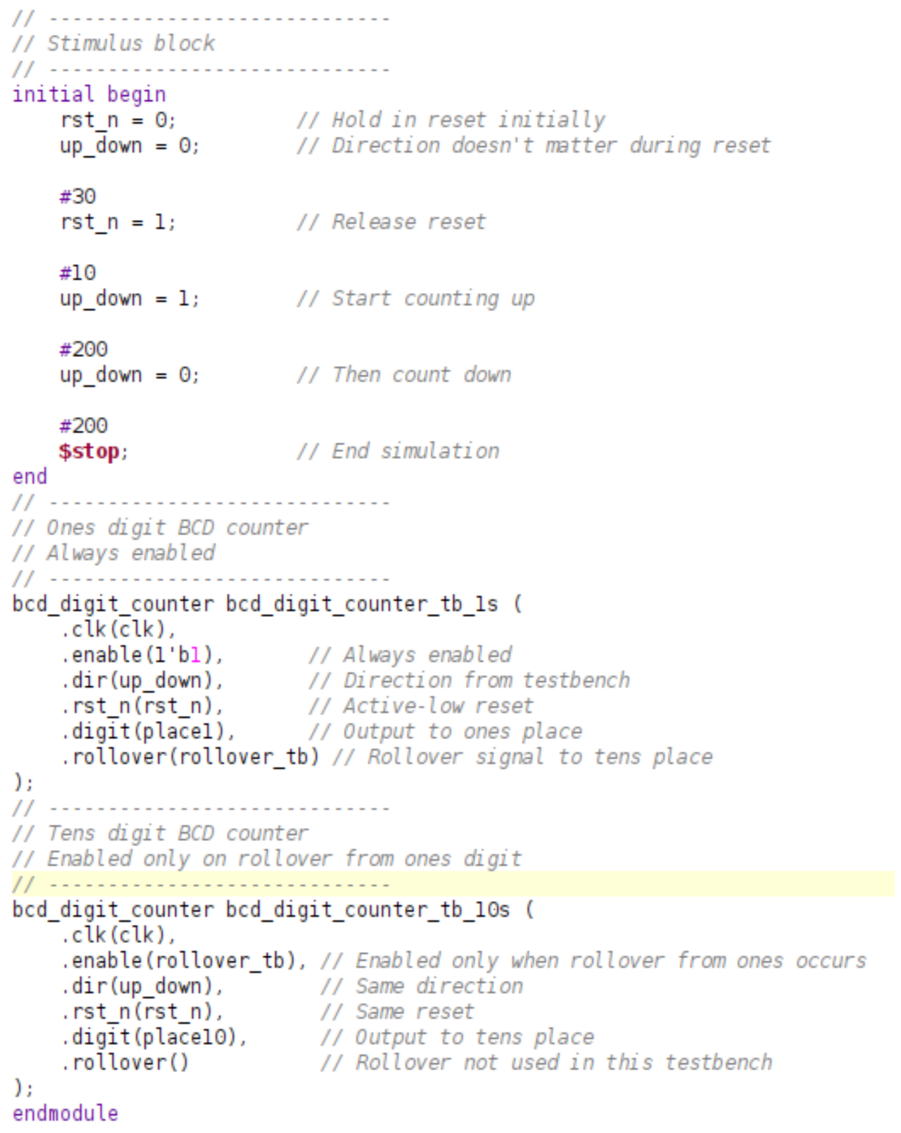
```
module bcd_digit_counter(
    input wire clk,          // Clock input
    input wire enable,       // Enable signal to allow counting
    input wire dir,          // Direction: 1 = up, 0 = down
    input wire rst_n,        // Active-low reset
    output reg [3:0] digit, // 4-bit output representing BCD digit (0-9)
    output reg rollover      // Indicates overflow (up) or underflow (down)
);

    // Sequential logic: triggered on rising clock or falling reset
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // Reset case: digit and rollover are cleared
            digit <= 4'd0;
            rollover <= 1'b0;
        end else begin
            rollover <= 1'b0;  // Default: no rollover unless a boundary is hit

            if (enable) begin  // Only count when enabled
                if (dir) begin  // Count up
                    if (digit == 4'd9) begin
                        digit <= 4'd0;     // Wrap around to 0
                        rollover <= 1'b1;  // Signal rollover
                    end else begin
                        digit <= digit + 1;
                    end
                end else begin  // Count down
                    if (digit == 4'd0) begin
                        digit <= 4'd9;     // Wrap around to 9
                        rollover <= 1'b1;  // Signal underflow rollover
                    end else begin
                        digit <= digit - 1;
                    end
                end
            end
        end
    end
endmodule
```

Test Bench Code

The testbench runs the counter with both up and down directions, verifies correct rollover at 0 and 9, and confirms that reset clears the value. It also tests the rollover signal for cascading.



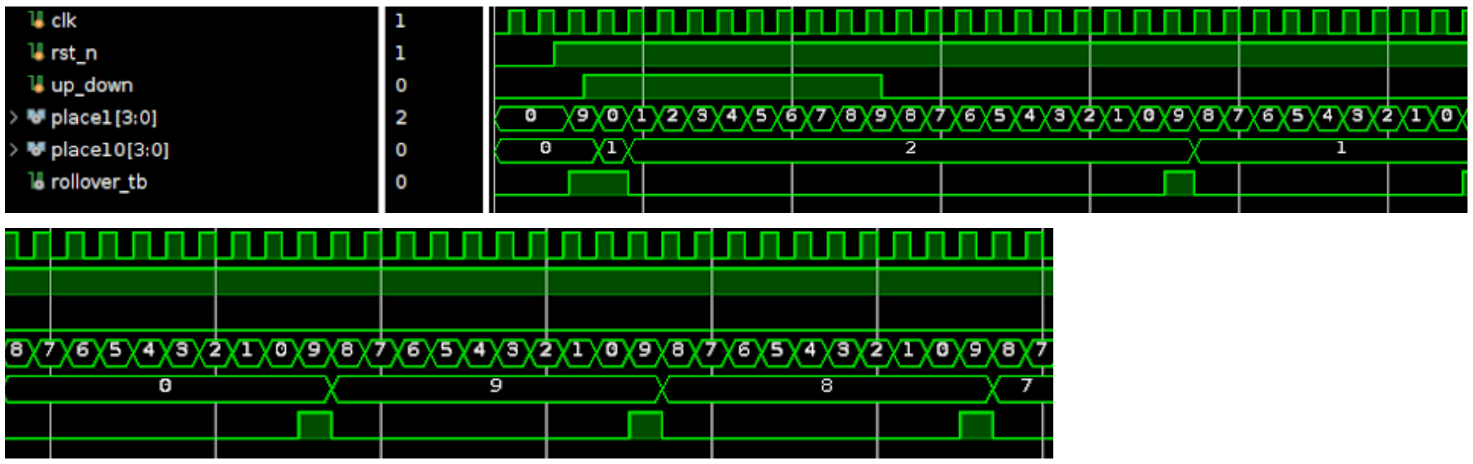
```
`timescale 1ns / 1ps
module bcd_counter_tb;
// ----------------------------
// Testbench signal declarations
// ----------------------------
reg clk;                 // Clock signal
reg rst_n;               // Active-low reset
reg up_down;             // Direction control: 1 = count up, 0 = count down
wire [3:0] place1;        // Ones digit output from counter
wire [3:0] place10;       // Tens digit output from counter
wire rollover_tb;         // Signal from ones place indicating rollover to tens
// ----------------------------
// Clock generation: 10 ns period (100 MHz)
// ----------------------------
initial clk = 0;
always #5 clk = ~clk;  // Toggle clock every 5 ns
```

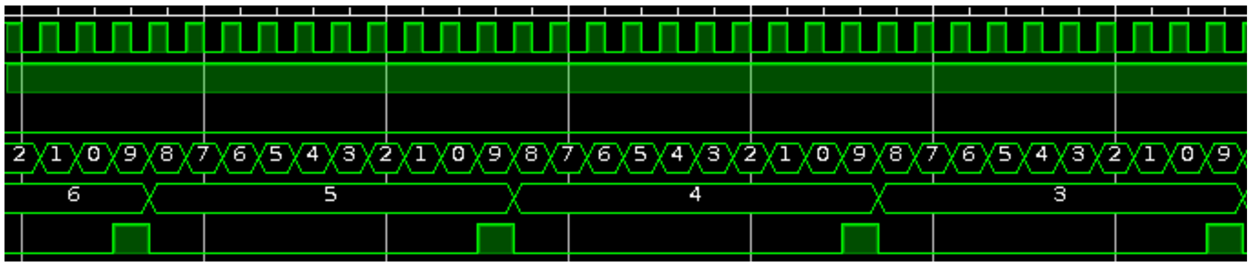


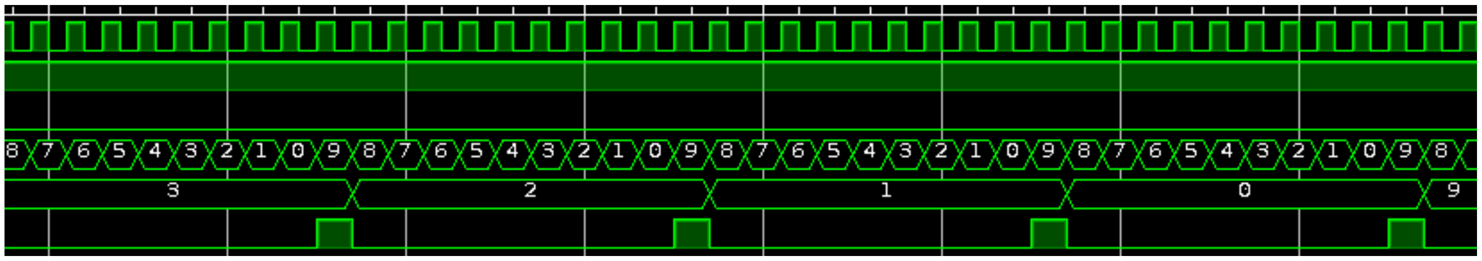
```
// ----------------------------
// Stimulus block
// ----------------------------
initial begin
    rst_n = 0;          // Hold in reset initially
    up_down = 0;        // Direction doesn't matter during reset

    #30
    rst_n = 1;          // Release reset

    #10
    up_down = 1;        // Start counting up

    #200
    up_down = 0;        // Then count down

    #200
    $stop;              // End simulation
end
// ----------------------------
// Ones digit BCD counter
// Always enabled
// ----------------------------
bcd_digit_counter bcd_digit_counter_tb_1s (
    .clk(clk),
    .enable(1'b1),       // Always enabled
    .dir(up_down),       // Direction from testbench
    .rst_n(rst_n),       // Active-low reset
    .digit(place1),      // Output to ones place
    .rollover(rollover_tb) // Rollover signal to tens place
);
// ----------------------------
// Tens digit BCD counter
// Enabled only on rollover from ones digit
// ----------------------------
bcd_digit_counter bcd_digit_counter_tb_10s (
    .clk(clk),
    .enable(rollover_tb), // Enabled only when rollover from ones occurs
    .dir(up_down),        // Same direction
    .rst_n(rst_n),        // Same reset
    .digit(place10),      // Output to tens place
    .rollover()           // Rollover not used in this testbench
);
endmodule
```

Behavioral Simulation

The output waveform showed accurate BCD counting, proper rollover, and reset behavior. Cascading was also validated by watching the tens digit increment when the units digit rolled from 9→0 or 0→9.

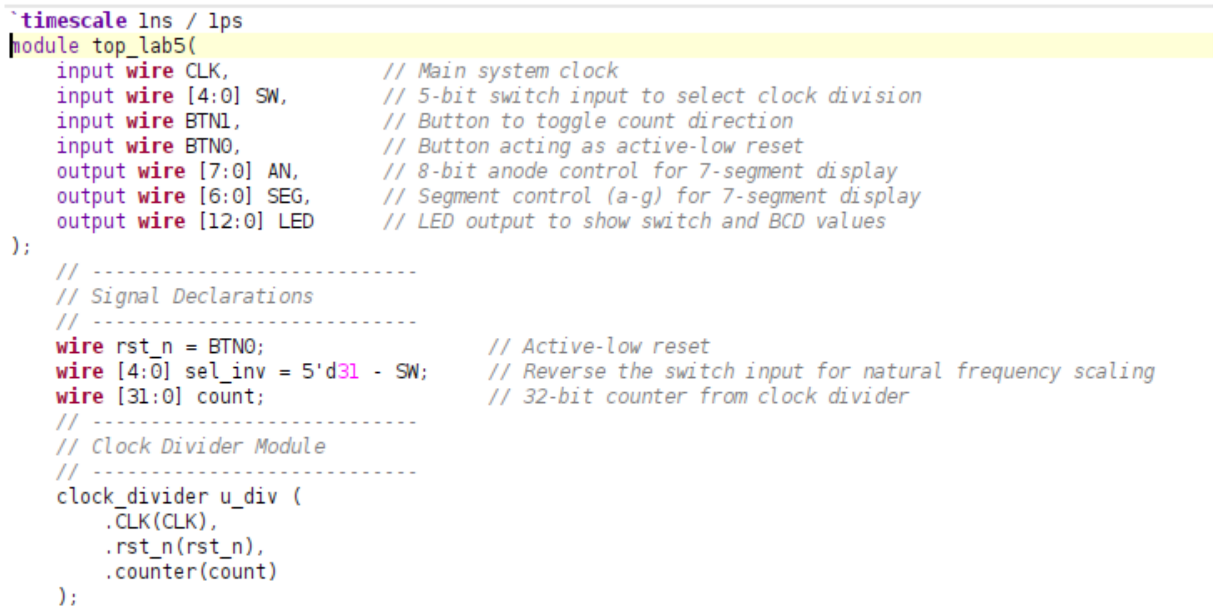






Top_lab5

The top_lab5 module connects all submodules to create a fully functional two-digit BCD up/down counter system. It takes the main 100 MHz CLK input and passes it into a clock_divider, which increments a 32-bit counter. A mux32x1 module selects one of the 32 bits from this counter based on SW[4:0], effectively setting the clock division rate. Edge detection logic is used to convert this divided clock (clk_div) into a single-cycle tick pulse. A separate edge detector on BTN1 is used to toggle the counting direction (direction). Two bcd_digit_counter modules are cascaded: the units digit updates on every tick, and the tens digit updates only when the units counter rolls over. The combined 8-bit output is passed to the seg7_scan module, which handles 2-digit time-multiplexing on the 7-segment display. Additional assignments send the switch value and the two BCD digits to the LED output for real-time debugging. This module manages direction toggling, speed control, display, and reset, making it the top-level integration of all core functionality.

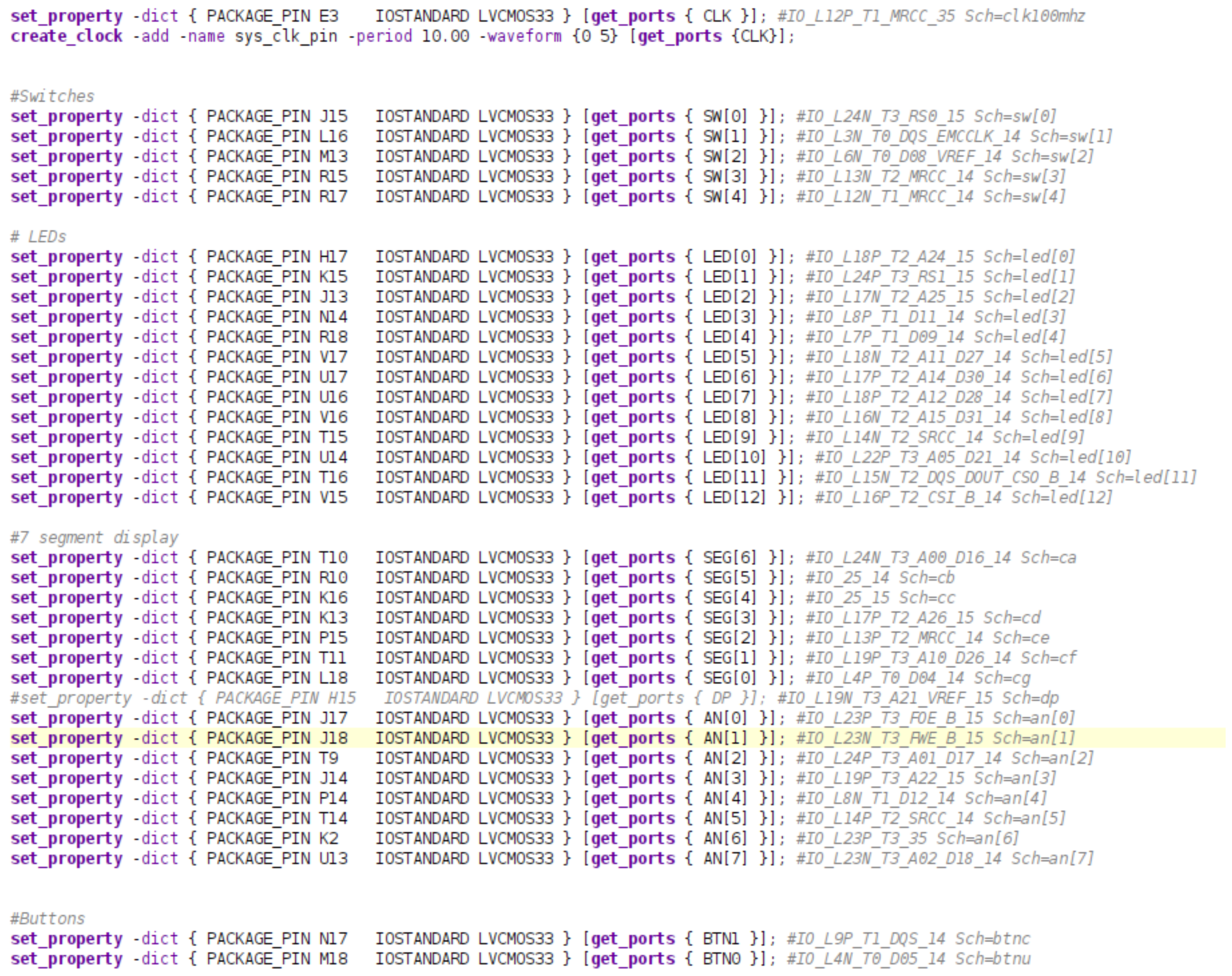


```
`timescale 1ns / 1ps
module top_lab5(
    input wire CLK,             // Main system clock
    input wire [4:0] SW,        // 5-bit switch input to select clock division
    input wire BTN1,            // Button to toggle count direction
    input wire BTN0,            // Button acting as active-low reset
    output wire [7:0] AN,       // 8-bit anode control for 7-segment display
    output wire [6:0] SEG,      // Segment control (a-g) for 7-segment display
    output wire [12:0] LED      // LED output to show switch and BCD values
);
    // ---------------------------
    // Signal Declarations
    // ---------------------------
    wire rst_n = BTN0;                  // Active-low reset
    wire [4:0] sel_inv = 5'd31 - SW;    // Reverse the switch input for natural frequency scaling
    wire [31:0] count;                  // 32-bit counter from clock divider
    // ---------------------------
    // Clock Divider Module
    // ---------------------------
    clock_divider u_div (
        .CLK(CLK),
        .rst_n(rst_n),
        .counter(count)
    );
```

```verilog
    // ----------------------------
    // MUX32x1: Select a bit of counter as divided clock
    // ----------------------------
    wire clk_div;
    mux32x1 u_mux (
        .Sel(sel_inv),
        .cnt(count),
        .clk_out(clk_div)
    );
    // ----------------------------
    // Edge Detection for clk_div (tick signal)
    // Only generates one pulse per clock_div rising edge
    // ----------------------------
    reg div_prev;
    always @(posedge CLK)
        div_prev <= rst_n ? clk_div : 1'b0;

    wire tick = clk_div & ~div_prev;  // One-shot pulse on clk_div rising edge
    // ----------------------------
    // Edge Detection for BTN1 (direction toggle)
    // ----------------------------
    reg btn1_prev;
    always @(posedge CLK)
        btn1_prev <= rst_n ? BTN1 : 1'b0;

    wire btn1_edge = BTN1 & ~btn1_prev;  // One-shot pulse on BTN1 press
    // ----------------------------
    // Direction toggle logic
    // Toggles direction on each BTN1 press
    // ----------------------------
    reg direction;
    always @(posedge CLK) begin
        if (!rst_n)
            direction <= 1'b1;          // Default count up
        else if (btn1_edge)
            direction <= ~direction;    // Toggle direction on BTN1 edge
    end

    // ---------------------------
    // Two-digit BCD counter using cascading
    // ---------------------------
    wire [3:0] units, tens;      // Ones and tens BCD digits
    wire unit_roll;              // Rollover signal from units to tens digit

    bcd_digit_counter u0 (
        .clk(CLK),
        .enable(tick),           // Enabled by divided tick
        .dir(direction),
        .rst_n(rst_n),
        .digit(units),
        .rollover(unit_roll)
    );
    bcd_digit_counter u1 (
        .clk(CLK),
        .enable(unit_roll),      // Enabled only when units rolls over
        .dir(direction),
        .rst_n(rst_n),
        .digit(tens),
        .rollover()              // Not used
    );
    wire [7:0] out_bcd = {tens, units};  // Combine BCD digits into one byte
    // ---------------------------
    // 7-Segment Display Driver
    // Scans and displays the two BCD digits
    // ---------------------------
    wire [1:0] an_active;
    seg7_scan u_seg (
        .AN(an_active),
        .SEG(SEG),
        .out(out_bcd),
        .CLK(CLK),
        .rst_n(rst_n)
    );
    assign AN = {6'b111111, an_active};  // Only enable the last two digits (others off)

    // ---------------------------
    // LED Outputs
    // ---------------------------
    assign LED[4:0] = SW;         // Display switch state
    assign LED[8:5] = units;      // Show units digit
    assign LED[12:9] = tens;      // Show tens digit
endmodule
```

Constraint File

The constraint file (.xdc) maps each logical input/output in the Verilog design to the physical pins on the Nexys A7 FPGA board. It includes pin assignments for CLK, BTN0, BTN1, SW[4:0], SEG[6:0], AN[1:0], and LED[12:0]. Proper configuration of the constraint file is essential to ensure that inputs like buttons and switches interact with the design correctly and that outputs are displayed on the correct 7-segment and LED pins during hardware testing.



```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK}];


#Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]

# LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]

#7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { SEG[6] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { SEG[5] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { SEG[4] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { SEG[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { SEG[2] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { SEG[1] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { SEG[0] }]; #IO_L4P_T0_D04_14 Sch=cg
#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


#Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTN1 }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTN0 }]; #IO_L4N_T0_D05_14 Sch=btnu
```

Team Contributions

This lab required more individual work than previous ones since we had to write all the source code modules from scratch and create four separate testbenches, rather than just one. Both of us contributed throughout the design, simulation, debugging, and documentation process.

Robert's Contributions:

I wrote my own versions of both the testbench and the top-level module. I used my testbench for the final behavioral simulation and helped debug and merge our top-level code. I gathered the simulation waveforms and added them to the report. I also wrote the sections explaining the testbench code and the constraint file.

Dia's Contributions:

I also created my own versions of the testbench and top-level modules and worked closely with Robert to debug and finalize the top-level code. I created the video demo for our final design and grabbed the utilization report from Vivado. For the lab report, I wrote the explanation sections for the driver modules and the top-level design.

We both contributed to understanding and explaining the code and collaborated equally to complete the lab.