ELEC 2607 Switching Circuits Lab 4: Seven-Segment Display Counter

Name: Dumany Lombe

Student Number: 101316658

Lab Number: Lab 4

Section: L4

1.0 Introduction

The goal of this lab is to design and implement a 4-digit counter using the Nexys A7 FPGA board and a seven-segment display. This counter is capable of incrementing, decrementing, loading values from switches, and resetting to zero. Seven-segment displays are widely used in digital clocks, meters, and embedded systems to present numerical data clearly. The design uses Verilog for hardware description and addresses issues such as push-button bouncing to ensure stable signal processing.

This report will discuss the specifications, design approach, implementation details, and testing results. It will also include a detailed analysis of the system's performance and the challenges encountered during the experiment.

2.0 Specifications

The system requirements for the seven-segment display counter are as follows:

- **Inputs**:
  - 16 switches for loading the initial value (in 4-bit BCD format for each digit).
  - Four push buttons for increment, decrement, load, and reset operations.
- **Outputs**:
  - Seven signals (CA–CG) to control the individual segments of the display.
  - Eight anode signals (AN0–AN7) to activate each digit in sequence.
- **Functionality**:
  - **Load**: The current values on the switches are loaded into the registers.
  - **Increment**: Increases the displayed number by 1, propagating overflow as necessary.
  - **Decrement**: Decreases the displayed number by 1, propagating underflow as necessary.
  - **Reset**: Clears all registers, setting the display to ‘0000’.
- **Clock**: A 100MHz input clock is used, divided to provide a suitable refresh rate.

3.0 Design

3.1 Design Overview

The design involves using Verilog to create modules that manage the seven-segment display and handle input processing. The counter is implemented with four 4-bit registers, each representing one digit of the display. The system is controlled using a finite state machine to handle button presses and display updates.

3.2 Block Diagram

A simplified block diagram of the system is provided, showing the interactions between the clock divider, seven-segment controller, and button handling logic.

3.3 Verilog Code

The code consists of multiple modules, including the seven_segment_controller and the binary_to_seven_segment converter.

4.0 Verilog Code

4.1 `seven_segment_controller`

verilog

```
`timescale 1ns / 1ps


module seven_segment_controller(
    input wire      CLK100MHZ,

    // Button inputs (Adding `load` and `reset` functionality)
    input wire      BTNU,       // Increment value
    input wire      BTND,     // Decrement value
    input wire      BTNR,         // Reset registers
    input wire      BTNL,          // Load from switches
    input [15:0]    SW,            // Switch inputs for manual
override
    output CA,
    output CB,
    output CC,
    output CD,
```

```
    output CE,
    output CF,
    output CG,
    output [7:0]    AN           // Active low anodes
);

    // Internal registers
    reg [7:0]   segment_state;      // Selects which segment to
refresh
    reg [31:0]  segment_counter;    // Clock division for refresh
rate
    reg [3:0]   routed_vals;        // BCD to be displayed
    reg [3:0]   registers [0:3];    // Four 4-bit registers
    wire [6:0]  cat_out;            // Cathode outputs to produce
the BCD

    reg slowCLK;
    reg[25:0] count = 0;
    always @ (posedge CLK100MHZ)
    begin
        count <= count +1 ;
        if (count  == 50_000_000)
        begin
            count <= 0;
            slowCLK = ~slowCLK;
        end
    end


    // Conversion from BCD to cathode assert bits
    binary_to_seven_segment my_converter(
        .bin_in (routed_vals),
        .hex_out (cat_out)
    );

    assign CA = cat_out[6];
    assign CB = cat_out[5];
    assign CC = cat_out[4];
    assign CD = cat_out[3];
    assign CE = cat_out[2];
    assign CF = cat_out[1];
```

```
assign CG = cat_out[0];

assign AN = ~segment_state;

// Initial states
initial begin
    segment_state = 8'b00000001;
    segment_counter = 32'd0;
    registers[0] = 4'b0000;
    registers[1] = 4'b0000;
    registers[2] = 4'b0000;
    registers[3] = 4'b0000;
end

// Refresh segment display
always @(posedge CLK100MHZ) begin
    if (segment_counter >= 32'd100_000) begin
        segment_counter <= 32'd0;
        segment_state <= {segment_state[6:0], segment_state[7]};
    end else begin
        segment_counter <= segment_counter + 1;
    end
end

// Logic for buttons and load
always @(posedge slowCLK) begin
    if (BTNR) begin
        registers[0] <= 4'b0000;
        registers[1] <= 4'b0000;
        registers[2] <= 4'b0000;
        registers[3] <= 4'b0000;
    end else if (BTNL) begin

        registers[0] <= SW[3:0];
        registers[1] <= SW[7:4];
        registers[2] <= SW[11:8];
        registers[3] <= SW[15:12];
    end else if (BTNU) begin
        if (registers[0] < 4'hF)
            registers[0] <= registers[0] + 1;
        else begin
```

```
            registers[0] <= 4'h0;
            if (registers[1] < 4'hF)
                registers[1] <= registers[1] + 1;
            else begin
                registers[1] <= 4'h0;
                if (registers[2] < 4'hF)
                    registers[2] <= registers[2] + 1;
                else begin
                    registers[2] <= 4'h0;
                    if (registers[3] < 4'hF)
                        registers[3] <= registers[3] + 1;
                    else
                        registers[3] <= 4'h0;
                end
            end
        end
    end else if (BTND) begin
        if (registers[0] > 4'h0)
            registers[0] <= registers[0] - 1;
        else begin
            registers[0] <= 4'hF;
            if (registers[1] > 4'h0)
                registers[1] <= registers[1] - 1;
            else begin
                registers[1] <= 4'hF;
                if (registers[2] > 4'h0)
                    registers[2] <= registers[2] - 1;
                else begin
                    registers[2] <= 4'hF;
                    if (registers[3] > 4'h0)
                        registers[3] <= registers[3] - 1;
                    else
                        registers[3] <= 4'hF;
                end
            end
        end
    end
end

// Segment state management
always @(posedge CLK100MHZ) begin
```

```
        case(segment_state)
            8'b00000001: routed_vals = registers[0];
            8'b00000010: routed_vals = registers[1];
            8'b00000100: routed_vals = registers[2];
            8'b00001000: routed_vals = registers[3];
            8'b00010000: routed_vals = 4'b0000;
            8'b00100000: routed_vals = 4'b0000;
            8'b01000000: routed_vals = 4'b0000;
            8'b10000000: routed_vals = 4'b0000;
            default: routed_vals = 4'b0000;
        endcase
    end
Endmodule
```

The seven_segment_controller module efficiently handles clock division, button operations, and display refreshing to manage a four-digit counter system. The high-frequency 100MHz clock signal from the FPGA is divided using a 26-bit counter, producing a slower clock, slowCLK, essential for human-readable display updates. Button presses are managed to perform increment, decrement, load, and reset operations: pressing the reset button (BTNR) clears all registers to zero, the load button (BTNL) transfers binary-coded decimal (BCD) values from 16 switches to the registers, the increment button (BTNU) increases the least significant register value and propagates overflow if necessary, and the decrement button (BTND) decreases the least significant register, cascading underflow as needed. The display uses a multiplexing approach, with the segment_state register cycling through each digit to activate them sequentially, while the segment_counter ensures rapid cycling to create the illusion of simultaneous illumination. The binary_to_seven_segment module converts 4-bit BCD values into segment control signals, mapping numerical values to LED patterns. Finally, the segment_state is updated to ensure smooth digit transitions and prevent flickering, providing a stable and accurate display output.

4.2 `binary_to_seven_segment`

verilog

```
`timescale 1ns / 1ps

module binary_to_seven_segment(
    input [3:0] bin_in,
    output reg [6:0] hex_out
    );
```

```
    always@* begin
        case(bin_in) // CG, CF, CE, CD, CC, CB, CA
            4'b0000: hex_out = 7'b0000001; // '0' in 7-segment
            4'b0001: hex_out = 7'b1001111; // '1' in 7-segment
            4'b0010: hex_out = 7'b0010010; // '2' in 7-segment
            4'b0011: hex_out = 7'b0000110; // '3' in 7-segment
            4'b0100: hex_out = 7'b1001100; // '4' in 7-segment
            4'b0101: hex_out = 7'b0100100; // '5' in 7-segment
            4'b0110: hex_out = 7'b0100000; // '6' in 7-segment
            4'b0111: hex_out = 7'b0001111; // '7' in 7-segment
            4'b1000: hex_out = 7'b0000000; // '8' in 7-segment

            4'b1001: hex_out = 7'b0000100; // '9' in 7-segment
            4'b1010: hex_out = 7'b0001000; // 'A' in 7-segment
            4'b1011: hex_out = 7'b1100000; // 'B' in 7-segment
            4'b1100: hex_out = 7'b0110001; // 'C' in 7-segment
            4'b1101: hex_out = 7'b1000010; // 'D' in 7-segment
            4'b1110: hex_out = 7'b0110000; // 'E' in 7-segment
            4'b1111: hex_out = 7'b0111000; // 'F' in 7-segment

        endcase
    end

endmodule
```

The binary_to_seven_segment module is a crucial component that translates 4-bit binary-coded decimal (BCD) inputs into the appropriate patterns for a seven-segment display. This conversion is necessary because the display requires specific segment activation signals to represent hexadecimal values (0 to F). The module uses an always block with a case statement to map each 4-bit input to a 7-bit output, where each bit controls one segment of the display. Active-low logic is used, meaning segments are illuminated when their corresponding output bit is '0'. By abstracting the conversion process, the module simplifies the main controller's responsibilities, making the overall design more modular and efficient. This functionality is essential for accurately displaying numeric and letter data in the lab's counter system, contributing to a reliable and visually clear output.

5.0 Implementation and Testing

5.1 Implementation

The design was synthesized and loaded onto the Nexys A7 board. The Verilog code was compiled using Xilinx Vivado, and the bitstream was successfully programmed onto the FPGA.

5.2 Testing Procedures

- **Individual Tests**: Each button was tested for correct operation, and the display was observed to ensure proper numeral representation.
- **Full System Test**: The entire counter system was tested, verifying increment, decrement, load, and reset functionalities.
- **Debouncing**: The system was monitored to ensure that debouncing logic eliminated unwanted glitches from button presses.

5.3 Results

- The counter operated smoothly, with no visible flickering on the display.
- Debouncing logic was effective, and all button operations worked as expected.
- The load and reset functions correctly initialized and cleared the registers.

6.0 Conclusion

The lab successfully demonstrated the design and implementation of a seven-segment display controller using Verilog. The objectives were met, and the system performed reliably during testing. The clock divider and debouncing mechanisms were crucial for stable operation. Potential improvements include optimizing the clock divider for better performance and exploring additional features, such as decimal-only counting.

Appendices

- **Simulation Results**: Annotated waveforms from the simulation.
- **Timing Diagrams**: Illustrations of key operations.