Department of Electrical and Computer Engineering

Author: Zainab Hussein

Title: Scrolling LED Matrix

Date: 3/5/2017

Time spent: 10 Hours

System Verilog files:

1. Top-level

```
module toplevel (
    input logic         CLK100MHZ,
    input logic         BTNU, BTNC, //reset
    input logic  [2:0]  SW, //color control
    output logic        JA1, JA2, JA3, JA7, JA8, JA9,
    output logic        JB1, JB2, JB3, JB7, JB8, JB9

    );

    //internal signals
    logic [2:0] row_cnt, col_no, ledRow;
    logic [4:0] col_cnt, scol_cnt;
    logic threeNs_clk, clk_shift ;
    logic lat, OE, en_row, en_col, en_wait;
    logic [2:0] cnt;
    logic btnu_dbn, btnu_pls;
    logic [11:0] scroll_cnt;
    logic clk, reset;
    logic [31:0] DI, character;
    logic [9:0] rdAddr, charAddr;
    logic [3:0] we;
    logic rd_en, wr_en;
    logic [8:0] wrAddr;
    logic [3:0] DO_bot, DO_top, number;
    logic [31:0] color_control;

    assign color_control = {8{1'b0, SW}};
    assign scol_cnt = col_cnt+1;
    assign ledRow = row_cnt+1;
    assign rdAddr = {scroll_cnt, ledRow};
    assign charAddr = {scroll_cnt, col_no};
    assign DI = character & color_control ; //control serving number and
color

    parameter BAUD = 3000000;  // desired frequency in Hz 1/300ns
    parameter scroll = 10;   //scroll specified at 100ms-200ms
```

```
    // add SystemVerilog code & module instantiations here
    Pulse_debounce U_DBN0( .clk(threeNs_clk), .button_in(BTNU),
.button_out(btnu_dbn), .pulse(btnu_pls) );
    Pulse_debounce U_DBN1( .clk(threeNs_clk), .button_in(BTNC),
.button_out(btnc_dbn), .pulse(pb) );
    clkdiv #( .DIVFREQ(BAUD)) hold300ns(.clk(CLK100MHZ),
.reset(btnu_pls), .sclk(threeNs_clk) );
    clkdiv #( .DIVFREQ(scroll)) scrollClk(.clk(CLK100MHZ),
.reset(btnu_pls), .sclk(scrClk) );
    col_counter U_col( .clk(threeNs_clk) , .reset(btnu_pls),
.enb(en_col),  .q(col_cnt) );
    fsm U_fsm(  .clk(CLK100MHZ), .reset(btnu_pls), .en_col(en_col),
.en_wait(en_wait), .en_row(en_row),
                .OE(OE), .lat(lat), .clk_shift(clk_shift) );
    row_counter U_row( .clk(threeNs_clk), .reset(btnu_pls), .enb(en_row),
.q(row_cnt) );

    BOTTOM_BRAM U_bot( .clk(threeNs_clk), .reset(btnu_pls), .DI(DI),
.rdAddr(rdAddr), .we(we), .rd_en(1'b1),
                        .wr_en(1'b1), .wrAddr(wrAddr), .DO(DO_bot) );
    TOP_BRAM U_top( .clk(threeNs_clk), .reset(btnu_pls), .DI(1'b1),
.rdAddr(rdAddr), .we(4'd0), .rd_en(1'b1),
                    .wr_en(1'b0), .wrAddr(12'd0), .DO(DO_top) );
    Scroller U_scroll( .clk(scrClk), .reset(btnu_pls),
.col_cnt(scol_cnt), .colScroll(scroll_cnt) );
    Character_creator U_create( .clk(CLK100MHZ), .reset(btnu_pls),
.number(number), .rdAddr(charAddr), .we(we),
                                .rd_en(1'b1), .wr_en(1'b1),
.wrAddr(wrAddr), .character(character)  );
    Number_FSM U_Num( .clk(threeNs_clk), .pb(pb), .reset(btnu_pls),
.col_no(col_no), .we(we), .wraddr(wrAddr)  );
    Number_counter U_counter( .clk(threeNs_clk), .reset(btnu_pls),
.enb(pb), .q(number) );

    //RGB color assignment
        //top
        assign JA1 = DO_top[2]; //R0
        assign JA2 = DO_top[1]; //G0
        assign JA3 = DO_top[0]; //B0

        //bot
        assign JA7 = DO_bot[2]; //R1
        assign JA8 = DO_bot[1]; //G1
        assign JA9 = DO_bot[0]; //B1

        //abc
        assign JB1 = row_cnt[0]; //A
        assign JB2 = row_cnt[1]; //B
        assign JB3 = row_cnt[2]; //C
```

```
        assign JB7 = clk_shift; //CLK
        assign JB8 = OE; //OE
        assign JB9 = lat; //LAT

endmodule
```

2. Bottom_BRAM

```
module BOTTOM_BRAM(
    // signals
    input logic clk, reset,
    input logic [31:0] DI,
    input logic [9:0] rdAddr,
    input logic [3:0] we,
    input logic rd_en, wr_en,
    input logic [8:0] wrAddr,
    output logic [3:0] DO
    );

    BRAM_SDP_MACRO #(
        .BRAM_SIZE("18Kb"), // Target BRAM, "18Kb" or "36Kb"
        .DEVICE("7SERIES"), // Target
        .WRITE_WIDTH(32), // Valid values are 172 (3772 only valid when
BRAM_SIZE="36Kb")
        .READ_WIDTH(4), // Valid values are 172 (3772 only valid when
BRAM_SIZE="36Kb")
        // Now Serving Message

.INIT_00(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_01(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_02(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_03(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_04(256'h10006000_01220000_00000000_11226644_00200000_00026000_00000
600_11226644),

.INIT_05(256'h10000000_01220000_10000000_01226000_00000000_01220000_10006
000_10006000),

.INIT_06(256'h10020004_10020004_10006640_00000000_00000000_00000000_00000
000_01226000),

.INIT_07(256'h00220000_10206000_10206000_10206000_01220000_00000000_01200
004_10020004),
```

```
.INIT_08(256'h00226000_00000000_00020000_00006000_00006000_00020000_11226
000_00000000),

.INIT_09(256'h11226000_00000000_11226040_00000000_00226000_01000000_10000
000_01000000),

.INIT_0A(256'h10020040_10020040_00006600_00000000_11226000_00000600_00000
600_00006000),

.INIT_0B(256'h0000000000000000000000000000000000000000000000000_01226600_1
0020040))


    BRAM_SDP_MACRO_three (
        .DO(DO), // Output read data port, width defined by READ_WIDTH
parameter
        .DI(DI), // Input write data port, width defined by WRITE_WIDTH
parameter
        .RDADDR(rdAddr), // Input read address, width defined by read
port depth
        .RDCLK(clk), // 1bit input read clock
        .RDEN(rd_en), // 1bit input read port enable
        .RST(reset), // 1bit input reset
        .WE(we), // Input write enable, width defined by write port depth
        .WRADDR(wrAddr), // Input write address, width defined by write
port depth
        .WRCLK(clk), // 1bit input write clock
        .WREN(wr_en) // 1bit input write port enable
    );

endmodule
```

3. Top_BRAM

```
module TOP_BRAM(
    // signals
    input logic clk, reset,
    input logic  DI,
    input logic [9:0] rdAddr,
    input logic [3:0] we,
    input logic rd_en, wr_en,
    input logic [8:0] wrAddr,
    output logic [3:0] DO
    );

    BRAM_SDP_MACRO #(
        .BRAM_SIZE("18Kb"), // Target BRAM, "18Kb" or "36Kb"
        .DEVICE("7SERIES"), // Target
        .WRITE_WIDTH(32), // Valid values are 172 (3772 only valid when
BRAM_SIZE="36Kb")
        .READ_WIDTH(4), // Valid values are 172 (3772 only valid when
```

```
BRAM_SIZE="36Kb")
        // ECE212 Message

.INIT_00(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_01(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_02(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_03(256'h0000000000000000000000000000000000000000000000000000000000
00000),

.INIT_04(256'h10000004_01226640_00000000_10000004_10006004_10006004_10006
004_11226644),

.INIT_05(256'h10006004_10006004_10006004_11226644_00000000_01000040_10000
004_10000004),

.INIT_06(256'h10020004_10200004_11000040_00000000_00000000_00000000_00000
000_10000004),

.INIT_07(256'h11000040_00000000_10000000_11226644_10000040_00000000_10000
640_10006004),

.INIT_08(256'h00000000_00000000_00000000_00000000_10000640_10006004_10020
004_10200004))


    BRAM_SDP_MACRO_two (
        .DO(DO), // Output read data port, width defined by READ_WIDTH
parameter
        .DI(DI), // Input write data port, width defined by WRITE_WIDTH
parameter
        .RDADDR(rdAddr), // Input read address, width defined by read
port depth
        .RDCLK(clk), // 1bit input read clock
        .RDEN(rd_en), // 1bit input read port enable
        .RST(reset), // 1bit input reset
        .WE(we), // Input write enable, width defined by write port depth
        .WRADDR(wrAddr), // Input write address, width defined by write
port depth
        .WRCLK(clk), // 1bit input write clock
        .WREN(wr_en) // 1bit input write port enable
    );

endmodule
```

4. Character_creator

```
module Character_creator(
    // signals
    input logic clk, reset,
    input logic [3:0] number,
    input logic [9:0] rdAddr,
    input logic [3:0] we,
    input logic rd_en, wr_en,
    input logic [8:0] wrAddr,
    output logic [31:0] character
    );
    //
    BRAM_SDP_MACRO #(
        .BRAM_SIZE("18Kb"), // Target BRAM, "18Kb" or "36Kb"
        .DEVICE("7SERIES"), // Target
        .WRITE_WIDTH(32), // Valid values are 172 (3772 only valid when
BRAM_SIZE="36Kb")
        .READ_WIDTH(4), // Valid values are 172 (3772 only valid when
BRAM_SIZE="36Kb")

.INIT_00(256'h0000000000000000000000_07777770_70000007_70000007_7000000
7_07777770),       //character 0, 5 32_bit cols

.INIT_01(256'h0000000000000000000000_00000000_70000000_77777777_7000007
0_00000000),        //character 1, 5 32_bit cols

.INIT_02(256'h0000000000000000000000_70000770_70007007_70070007_7070000
7_77000070),       //character 2, 5 32_bit cols

.INIT_03(256'h0000000000000000000000_07770770_70007007_70007007_7000000
7_07000070),       //character 3, 5 32_bit cols

.INIT_04(256'h0000000000000000000000_77777777_00007000_00007000_0000700
0_00007777),       //character 4, 5 32_bit cols

.INIT_05(256'h0000000000000000000000_07770007_70007007_70007007_7000700
7_07007777),       //character 5, 5 32_bit cols

.INIT_06(256'h0000000000000000000000_07770000_70007007_70007070_7000770
0_07777000),       //character 6, 5 32_bit cols

.INIT_07(256'h0000000000000000000000_00000077_00007707_00770007_7700000
7_00000007),        //character 7, 5 32_bit cols

.INIT_08(256'h0000000000000000000000_07770770_70007007_70007007_7000700
7_07770770),       //character 8, 5 32_bit cols

.INIT_09(256'h0000000000000000000000_77777770_00007007_00007007_0000700
7_00000770))       //character 9, 5 32_bit cols

    BRAM_SDP_MACRO_one (
```

```
        .DO(character), // Output read data port, width defined by
READ_WIDTH parameter
        .DI(number), // Input write data port, width defined by
WRITE_WIDTH parameter
        .RDADDR(rdAddr), // Input read address, width defined by read
port depth
        .RDCLK(clk), // 1bit input read clock
        .RDEN(rd_en), // 1bit input read port enable
        .RST(reset), // 1bit input reset
        .WE(we), // Input write enable, width defined by write port depth
        .WRADDR(wrAddr), // Input write address, width defined by write
port depth
        .WRCLK(clk), // 1bit input write clock
        .WREN(wr_en) // 1bit input write port enable
    );

endmodule
```

5. Lab2_FSM

```
module fsm(
    input logic clk, reset,
    output logic  en_col, en_wait, en_row, OE, lat, clk_shift
    );

  //logic
  logic [4:0] col_cnt;
  logic threeNs_clk;
  logic [8:0] wait_cnt;

  parameter BAUD = 3000000;  // desired frequency in Hz 1/300ns

   //instantiate modules
  clkdiv #(.DIVFREQ(BAUD)) hold300ns(.clk(clk), .reset(reset),
.sclk(threeNs_clk));
  col_counter U_col( .clk(threeNs_clk) , .reset(reset), .enb(en_col),
.q(col_cnt) );
  wait_counter U_wait( .clk(threeNs_clk), .reset(reset), .enb(en_wait),
.q(wait_cnt) );


  typedef enum logic [2:0] {
        S0 = 3'b000,
        S1 = 3'b001,
        S2 = 3'b010,
        S3 = 3'b011,
        S4 = 3'b100,
        WAIT = 3'b101
        } states_t;

        states_t state, next;
```

```
        always_ff @(posedge threeNs_clk) //ensure each control signal
held for 200ns
         if (reset)  state <= S0;
         else state <= next;

        always_comb begin
            next = S0;
            clk_shift = 0;
            lat = 0;
            OE = 0;
            en_row = 0;
            en_col = 0;
            en_wait = 0;
            case( state )

            S0: //default state
                begin
                  next = S1;
                  lat = 0;
                  OE = 0;   //high
                  en_row = 0;
                  en_col = 1;
                  clk_shift = 0;
                  en_wait = 0;
                end
            S1: //fill all columns
                begin
                    lat = 0;
                    OE = 0;   //high
                    en_row = 0;
                    en_col = 0;
                    clk_shift = 1;
                    en_wait = 0;

                    if (col_cnt == 5'd31 )
                        begin next = S2; end
                    else
                        begin next = S0; end
                end

            S2: //turn off display
                begin
                    next = S3;
                    lat = 0;
                    OE = 1;   //low
                    en_row = 0;
                    en_col = 0;
                    clk_shift = 0;
                    en_wait = 0;
                end
```

```
            S3: // enable row to update ABC
                begin
                    next = S4;
                    lat = 0;
                    OE = 1;   //low
                    en_row = 1;
                    en_col = 0;
                    clk_shift = 0;
                    en_wait = 0;
                end
            S4: //shift values to latch and turn display off while
shifting
                begin
                    next = WAIT;
                    lat = 1;
                    OE = 1;   //low
                    en_row = 0;
                    en_col = 0;
                    clk_shift = 0;
                    en_wait = 0;
                end
            WAIT:
             //wait time for LED's to fully turn on, 100 times/s per row,
thus count till o.o1s
             //(using a counter that is enabled with a 300ns clock (100
times/sec)
             //chose 300 times
                begin
                    lat = 0;
                    OE = 0;   //high
                    en_row = 0;
                    en_col = 0;
                    clk_shift = 0;
                    en_wait = 1;

                    if (wait_cnt < 9'd300 )
                        begin next = WAIT; end
                    else
                        begin next = S0; end
                end

            default:
                begin
                    next = S0;
                    lat = 0;
                    OE = 0;   //high
                    en_row = 0;
                    en_col = 0;
                    clk_shift = 0;
                    en_wait = 0;
```

```
                end
            endcase
        end
endmodule
```

6. Number_FSM

```
module Number_FSM(

    //signals
    input logic clk, pb, reset,
    output logic [2:0] col_no,
    output logic [3:0] we,
    output logic [8:0] wraddr
    );

    //logic
    logic colStart = 7'd96;

    //fsm logic
    typedef enum logic [2:0] {
        idle = 3'b000,
        col_1 = 3'b001,
        col_2 = 3'b010,
        col_3 = 3'b011,
        col_4 = 3'b100,
        col_5 = 3'b101
    } states_t;

    states_t state, next;

    always_ff @( posedge clk ) //ensure each control signal held for
300ns
     if( reset )  state <= idle;
     else state <= next;

    always_comb begin
        next = idle;
        case( state )

        idle:
            begin
                col_no = 3'b000;
                wraddr = 9'd0;
                we = 4'b0000;
                if( pb )
                    begin next = col_1; end
                else
                    begin next = idle; end
            end
        col_1:
            begin
```

```
                col_no = 3'b001;
                wraddr = colStart;
                we = 4'b1111;
                next = col_2;
            end
        col_2:
            begin
                col_no = 3'b010;
                wraddr = colStart + 1;
                we = 4'b1111;
                next = col_3;
            end
        col_3:
            begin
                col_no = 3'b011;
                wraddr = colStart + 2;
                we = 4'b1111;
                next = col_4;
            end
        col_4:
            begin
                col_no = 3'b100;
                wraddr = colStart + 3;
                we = 4'b1111;
                next = col_5;
            end

        col_5:
            begin
                col_no = 3'b101;
                wraddr = colStart + 4;
                we = 4'b1111;
                next = idle;
            end

        default:
            begin
                col_no = 3'b000;
                wraddr = 9'd0;
                we = 4'b0000;
                next = idle;
            end
        endcase
    end
endmodule
```

7. Number counter

```
module Number_counter(
    input logic clk, reset, enb,
    output logic [3:0] q
    );
```

```
    always_ff@( posedge clk )
        begin
            if( reset )
                q <= 4'd0;
            else if( enb && q < 9 )
                q <= q + 1;
            else
                q <= q;
        end

endmodule
```

8. Debouncer_Pulse

```
module Pulse_debounce(
                input logic clk,    //should be synchronized
                input logic  button_in,
                output logic button_out,
                output logic pulse);

   parameter DEBOUNCE_TIME_MS = 5;
   parameter CLKFREQ = 100_000_000;
   parameter WAIT_COUNT = DEBOUNCE_TIME_MS*(CLKFREQ/1000);

   // States for button debouncing
   logic               button_state, button_state_next;
   // Counter for debouncing
   logic [26:0]        count_reg, count_next;

   // Counter and button state register
   always_ff @(posedge clk)
     begin
      button_state <= button_state_next;
      count_reg <= count_next;
     end

   // Next-state / output logic
   always_comb
     begin
      // Defaults
      button_state_next = button_state;
      count_next = count_reg;
      pulse = 1'b0;

      // Does the button input match the stored button state?
      if (button_in == button_state)
        // Yes, so just reset the counter
        count_next = 0;
      else if (count_reg == WAIT_COUNT-1)
        begin
           // No, so if the counter is done, transition to the other
```

```
state
           button_state_next = ~button_state;
           count_next = 0;
           // Generate a pulse if going from 0=>1
           pulse = ~button_state;
         end
       else
         // Have not reached wait count yet, so increment counter.
         count_next = count_reg + 1;
     end // always_comb

   assign button_out = button_state;

endmodule // debounce
```

9. Scroller

```
module Scroller(
    // signals
    input logic clk, reset,
    input logic [4:0] col_cnt,
    output logic [11:0] colScroll
    );

    // logic
    logic [6:0] refresh_cnt;

    // column refresh counter
    always_ff@( posedge clk )
    begin
        if( reset )
            refresh_cnt <= 7'd0;
        else
            refresh_cnt <= refresh_cnt + 1;
    end

    // scroll columns
    // add blank columns to to programmed columns
    assign colScroll = col_cnt + refresh_cnt;

endmodule
```

10. ClkShift counter

```
module clkShift_counter(
    input logic clk,
    input logic reset,
    input logic enb,
    output logic [4:0] q);

    always_ff @(posedge clk)
        if (reset) q <= 5'd0;
        else if (enb) q <= q + 1;
```

```
endmodule
```

11. Clkdiv

```
module clkdiv(input logic clk, input logic reset, output logic sclk);
   parameter DIVFREQ = 100;  // desired frequency in Hz (change as
needed)
   parameter DIVBITS = 26;   // enough bits to divide 100MHz down to 1 Hz
   parameter CLKFREQ = 100_000_000;
   parameter DIVAMT = (CLKFREQ / DIVFREQ) / 2;

   logic [DIVBITS-1:0] q;

   always_ff @(posedge clk)
     if (reset)
       begin
           q <= 0;
           sclk <= 0;
       end
     else if (q == DIVAMT-1)
        begin
            q <= 0;
            sclk <= ~sclk;
        end
     else q <= q + 1;

endmodule // clkdiv
```

12. Col_counter

```
module col_counter(
    input logic clk,
    input logic reset,
    input logic enb,
    output logic [4:0] q);

    always_ff @(posedge clk)
        if ( reset ) q <= 5'd0;
        else if (enb) q <= q + 1;

endmodule
```

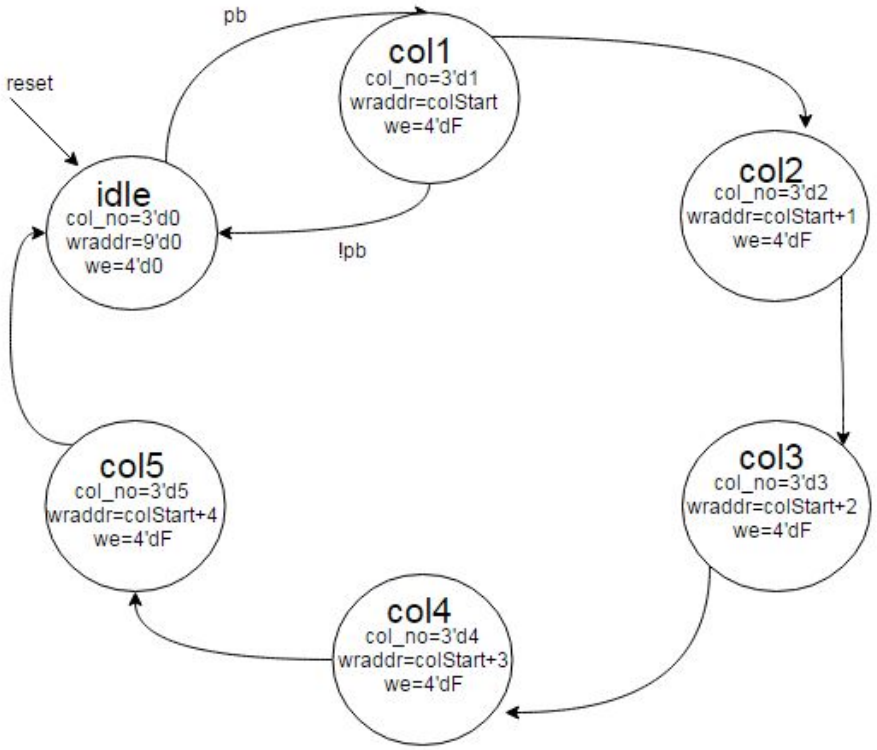
13. Row_counter

```
module row_counter(
    input logic clk,
    input logic reset,
    input logic enb,
    output logic [2:0] q);

    always_ff @(posedge clk)
        if ( reset ) q <= 3'd0;
        else if ( enb ) q <= q + 1;

endmodule
```

14. Wait_counter

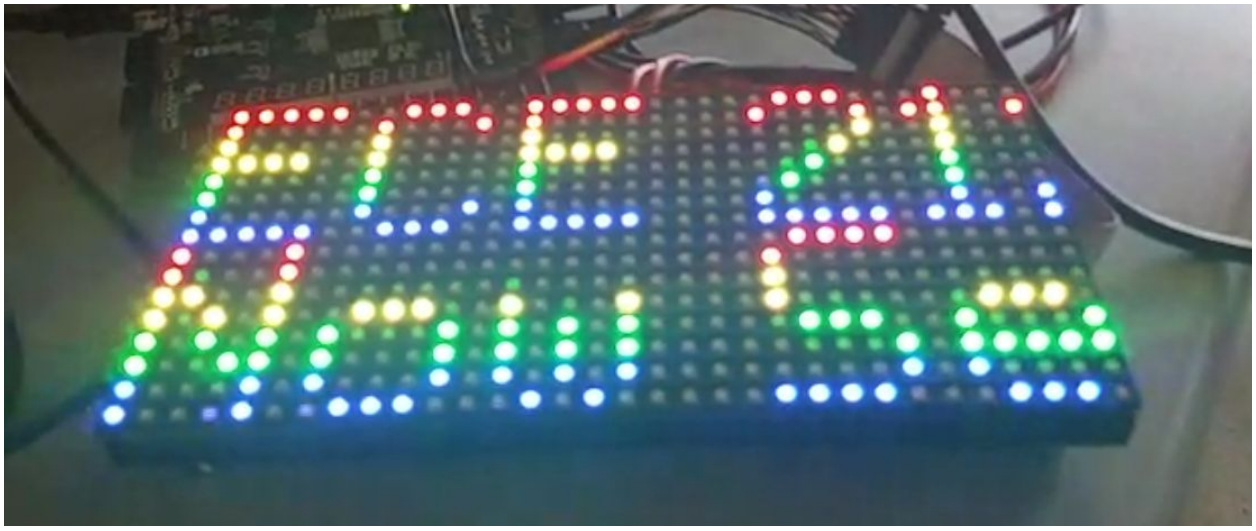
```
module wait_counter(
    input logic clk,
    input logic reset,
    input logic enb,
    output logic [8:0] q );

    always_ff @(posedge clk)
        if ( reset ) q <= 9'd0;
        else if ( enb ) q <= q + 1;

endmodule
```

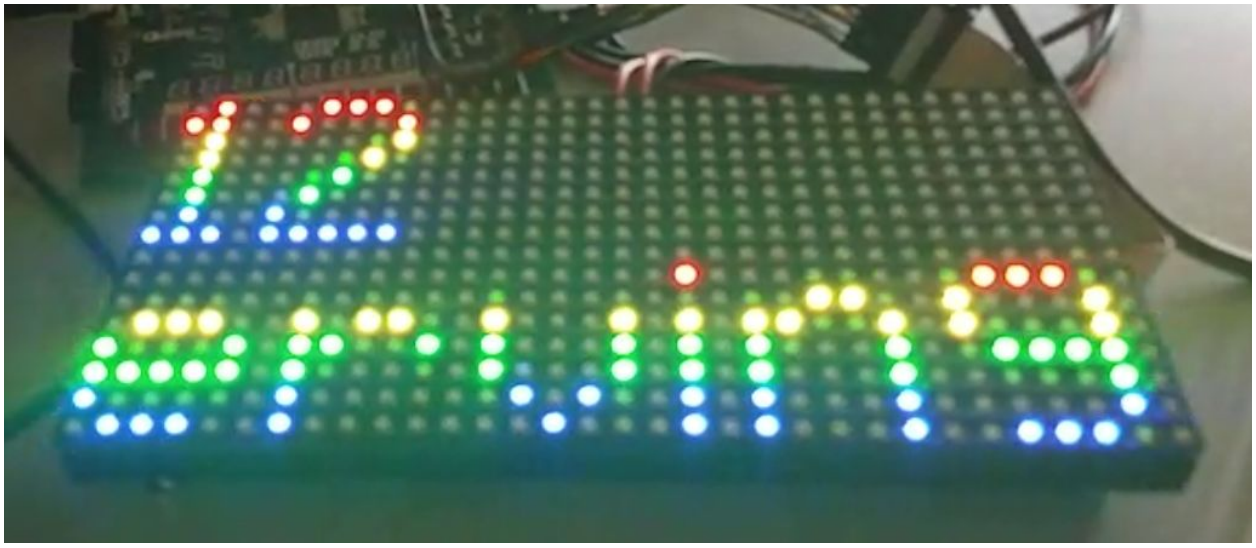
Number FSM diagrams



4. Scrolling frequency used is 10 Hz, i did not calculate, rather varied between 10 to 20 Hz suggested by the 100-200ms in the lab manual.

5. Pictures of the LED display for the final implementation.





6. This lab was very robust in terms of implementation of the scrolling LED matrix. One of the biggest challenges was to implement the character creator for the number that should come after the “Now Serving” message. The initial design idea for that was to use many muxes to produce each character from the 5 columns. This would need 10 muxes for character 0-9. However, to minimize the amount of logic elements I opted for a BRAM to store all the values, but eventually this was my problem on the LED. I have just one row that does not change, but the colors follow the switch RGB control.

7. The lab extension is definitely very appreciated and was instrumental in implementing the lab. It would be great to have more brainstorm session with the instructor, and when it comes to the addressing of the bottom and top BRAM. A proper working lab 2 also saved a lot of time for this lab.

HIGH LEVEL DIAGRAM