ECE241 Digital Systems Final Project Report

Project: The Typist

Teaching Assistant: David Han

Team Members & Student Number:

Lisa Yoneyama 1001085381

Kung Lim Siew Jin 1001511287

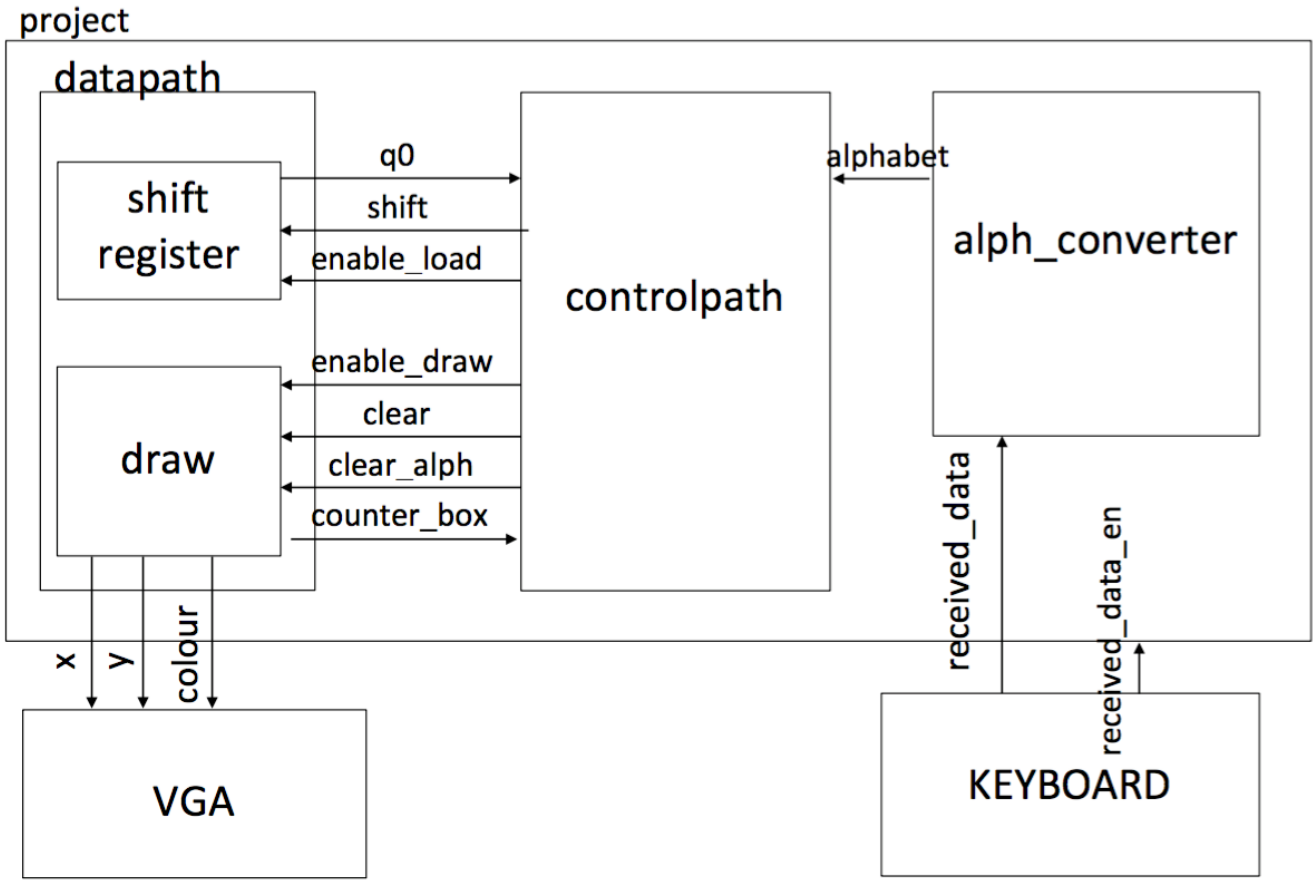
1. Introduction

For the final project in the ECE241 Digital Systems course, we designed a circuit to implement a typing game that is projected onto a monitor via VGA. The user would then type the letters on the screen with a PS2 keyboard to “delete” or shift characters until all the characters have been “deleted”.

We had several goals when we started working on the project. The milestones listed are in chronological order of completion:

- Drawing a single letter on the monitor
- Drawing an entire sentence on the monitor
- Shifting the characters on the screen via FPGA KEY press
- Shifting the characters via PS2 keyboard
- Drawing the background to suit the game

2. The Design



Project - This module contains two main modules: the control path and the data path. It is also where the VGA and Keyboard inputs/outputs are derived.

Datapath - The datapath consists of two modules: the shift register module and the draw module. It takes shift, enable_load, enable_draw, clear, clear_alph and counter_box as inputs from the control path.

Controlpath - The control path module consists of 28 different states: a reset state, register state, a state for each letter on the screen (twenty-four letters), a wait state and a shift state. The reset state restarts the game. The register state is a delay state that allows the letters from the shift register to register in the twenty-four D-Flip Flops for each letter on the screen. The draw states will draw the twenty-four letters on the screen from left to right, each one going to the next state when the previous letter is finished drawing and counter_box is incremented. Once all the letters are on the screen the control path will go to the wait state and stay in the wait state until the user inputs the first letter on the screen correctly via keyboard. The shift state will shift the shift register exactly one time so that the second letter is now on the very left of the screen.

Alph_converter - The alph_converter module takes input from the PS2 Keyboard and converts the hex numbers to binary parameters we have assigned to each letter.

Shiftregister - The shift register module contains the hard coded sentences, each letter registered in a D-Flip flop.

Draw - The draw module draws a single letter on the screen. It reads the colour stored in the ROM memory and draws each pixel while counter_alphabet is incrementing to thirty. The start point for each letter is taken in as an input from the datapath module and changes when counter_box is incremented.

3. Report on Success

Our final project performed as planned except for a minor bug, which causes the first pixel out of 30 pixels drawn for each character to be overwritten by a differently specified colour. We speculate this to be a timing issue due to the complexity of the circuit, as some variations of the code did not have this anomaly. All the characters shifted to the left when the leftmost character on the monitor was “deleted” while the rightmost character was drawn onto the monitor. The characters displayed on the monitor are preloaded into a series of shift registers. Although we only hard coded 2 sentences into the shift registers, we could have added many more. The game starts automatically and is reset to the image shown below when KEY[0] is pressed.

4. What We Would Do Differently

Some of the challenges we faced while working on this project was time management which resulted in a mad rush to complete the project before the deadline. Although we mostly worked together which avoided the problem of putting separate modules together, it was highly inefficient and therefore, more time consuming than it had to be. We spent the majority of our time on this project debugging instead of writing code or figuring out game logic, thus wasting plenty of time for compilation to happen as ModelSIM was a less than ideal method of debugging our circuit due to usage of a VGA and monitor.

We would have broken down the parts suggested as per our TA's recommendation, as some parts of the circuit could have been done separately and put together easily. We would allocate more time to debugging our code.

We also attempted to create a timer that started when the user began typing and stopped when the game was over. However, by the time we finished the main components of the game we realized that we would have to change a large portion of the existing code in order to make this timer. In the end, we decided against making this change because we simply did not have enough time.

We would have planned all the components of the game beforehand so as each part of the code will connect smoothly with the rest of the code.

Appendix

```
/*****************************************************************************

 Module:      ECE241 Typing Project
 Description:
 This module was made by Siew Jin Kung Lim & Lisa Yoneyama.

*****************************************************************************/

module project(
      CLOCK_50,                  //  On Board 50 MHz
      // Your inputs and outputs here
      KEY,
      SW,
                  PS2_CLK,
                  PS2_DAT,
                  LEDR,
                  HEX0,
                  HEX1,
                  HEX2,
                  HEX3,
                  HEX4,
                  HEX5,
      // The ports below are for the VGA output.  Do not change.
      VGA_CLK,                   //  VGA Clock
      VGA_HS,                    //  VGA H_SYNC
      VGA_VS,                    //  VGA V_SYNC
      VGA_BLANK,                 //  VGA BLANK
      VGA_SYNC,                  //  VGA SYNC
      VGA_R,                     //  VGA Red[9:0]
      VGA_G,                     //  VGA Green[9:0]
      VGA_B                      //  VGA Blue[9:0]
  );

/*****************************************************************************
*                    Parameter Declarations                 *
*****************************************************************************/


/*****************************************************************************
*                    Port Declarations                      *
*****************************************************************************/
          input          CLOCK_50;            //  50 MHz

  // Declare your inputs and outputs here
          input   [3:0]   KEY;
          input   [9:0]   SW;
          inout PS2_CLK, PS2_DAT;
          output [9:0]LEDR;
```

```
        output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;

  // Do not change the following outputs
  output        VGA_CLK;            //  VGA Clock
  output        VGA_HS;             //  VGA H_SYNC
  output        VGA_VS;             //  VGA V_SYNC
  output        VGA_BLANK;          //  VGA BLANK
  output        VGA_SYNC;           //  VGA SYNC
  output  [9:0]   VGA_R;            //  VGA Red[9:0]
  output  [9:0]   VGA_G;            //  VGA Green[9:0]
  output  [9:0]   VGA_B;            //  VGA Blue[9:0]


  wire resetn;
  assign resetn = KEY[0];

  // Create the colour, x, y and writeEn wires that are inputs to the controller.

  wire [2:0] colour;
  wire [7:0] x;
  wire [6:0] y;
  wire writeEn;

        wire shift, clear;

        wire [7:0] received_data;
        wire received_data_en;
        wire [5:0]q0;
        wire[5:0]d0;
        wire[5:0] alphabet;
        wire [2:0] y_cord;
        wire clear_alph, enable_draw;
        wire [4:0]counter_alphabet;
        wire [4:0] counter_box;
        wire [14:0] counter_black;
        wire enable_load;
        wire [3:0]time_counter;

        reg [7:0]last_data_received;

  // Create an Instance of a VGA controller - there can be only one!
  // Define the number of colours as well as the initial background
  // image file (.MIF) for the controller.

/****************************************************************************
*                       Internal Modules                       *
****************************************************************************/
// Instance of VGA
  vga_adapter VGA(
        .resetn(resetn),
```

```
        .clock(CLOCK_50),
        .colour(colour),
        .x(x),
        .y(y),
        .plot(writeEn),
        /* Signals for the DAC to drive the monitor. */
        .VGA_R(VGA_R),
        .VGA_G(VGA_G),
        .VGA_B(VGA_B),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .VGA_BLANK(VGA_BLANK),
        .VGA_SYNC(VGA_SYNC),
        .VGA_CLK(VGA_CLK));
    defparam VGA.RESOLUTION = "160x120";
    defparam VGA.MONOCHROME = "FALSE";
    defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
    defparam VGA.BACKGROUND_IMAGE = "Background.mif";

        //Instance of keyboard
        PS2_Controller KEYBOARD (
                                .CLOCK_50(CLOCK_50),
                                .reset(~resetn),
                                //.the_command(),
                                //.send_command(),
                                .PS2_CLK(PS2_CLK),
                                .PS2_DAT(PS2_DAT),
                                //.command_was_sent(),
                                //.error_communication_timed_out,
                                .received_data(received_data),
                                .received_data_en(received_data_en)

        );

        // keyboard conversion
        alph_converter u0(alphabet, last_data_received);

  // Instanciate datapath
        datapath u1 (x, y, colour, counter_box, counter_black, q0, d0, enable_draw, writeEn, shift, clear,
CLOCK_50, clear_alph, enable_load);

  // Instanciate FSM control
        controlpath u2(clear_alph, LEDR, enable_draw, enable_load, shift, writeEn, clear, alphabet,
received_data_en, q0, CLOCK_50, SW[0], counter_box, SW[1], SW[2], KEY[1], counter_black,
KEY[2], resetn, HEX2, HEX3, HEX4, HEX5);

        time_counter u3(time_counter, CLOCK_50);

        seg7 u4 (HEX0, time_counter);
```

```
/*****************************************************************************
*                    Sequential logic                                       *
*****************************************************************************/
        always @(posedge CLOCK_50)//clk
        begin
                if (received_data_en == 1'b1)
                        last_data_received <= received_data;
                //else
                        //last_data_received <= 7'b0;
        end

endmodule

//datapath module
module datapath (x, y, colour, counter_box, counter_black, q0, d0, enable_draw, writeEn, shift, clear, clk,
clear_alph, enable_load);
        input shift, writeEn, clear, clk, enable_draw, clear_alph;
        input enable_load;
        output [4:0]counter_box;
        output [14:0] counter_black;
        output [5:0]q0;
        output [5:0]d0;
        output [2:0]colour;
        output [7:0]x;
        output [6:0]y;

        reg [5:0] letter;
        wire [4:0]counter_box;
        wire [4:0]counter_box_1;
        reg [7:0]x_start;
        reg [6:0]y_start;
        wire [2:0] x_cord, y_cord;



        wire [5:0]q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18, q19, q20,
q21,
        q22, q23, q24, q25, q26, q27, q28, q29;

        parameter A = 6'b0, B = 6'b000001, C = 6'b000010, D = 6'b000011, E = 6'b000100, F =
6'b000101, G = 6'b000110,
        H = 6'b000111, I = 6'b001000, J = 6'b001001, K = 6'b001010, L = 6'b001011, M = 6'b001100, N
= 6'b001101,
        O = 6'b001110, P = 6'b001111, Q = 6'b010000, R = 6'b010001, S = 6'b010010, T = 6'b010011, U
= 6'b010100,
        V = 6'b010101, W = 6'b010110, X = 6'b010111, Y = 6'b011000, Z = 6'b011001, SPACE =
6'b011010, PERIOD = 6'b011011;

/*****************************************************************************
*                    Sequential logic                                       *
```

```
**************************************************************************************/
	always @ (*)
		begin
			if (counter_box == 5'd0)
				begin
					letter = q0;
					x_start = 8'd9;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd1)
				begin
					letter = q1;
					x_start = 8'd15;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd2)
				begin
					letter = q2;
					x_start = 8'd21;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd3)
				begin
					letter = q3;
					x_start = 8'd27;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd4)
				begin
					letter = q4;
					x_start = 8'd33;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd5)
				begin
					letter = q5;
					x_start = 8'd39;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd6)
				begin
					letter = q6;
					x_start = 8'd45;
					y_start = 7'd80;
				end
			else if (counter_box == 5'd7)
				begin
					letter = q7;
					x_start = 8'd51;
					y_start = 7'd80;
				end
```

```
else if (counter_box == 5'd8)
        begin
                letter = q8;
                x_start = 8'd57;
                y_start = 7'd80;
        end
else if (counter_box == 5'd9)
        begin
                letter = q9;
                x_start = 8'd63;
                y_start = 7'd80;
        end
else if (counter_box == 5'd10)
        begin
                letter = q10;
                x_start = 8'd69;
                y_start = 7'd80;
        end
else if (counter_box == 5'd11)
        begin
                letter = q11;
                x_start = 8'd75;
                y_start = 7'd80;
        end
else if (counter_box == 5'd12)
        begin
                letter = q12;
                x_start = 8'd81;
                y_start = 7'd80;
        end
else if (counter_box == 5'd13)
        begin
                letter = q13;
                x_start = 8'd87;
                y_start = 7'd80;
        end
else if (counter_box == 5'd14)
        begin
                letter = q14;
                x_start = 8'd93;
                y_start = 7'd80;
        end
else if (counter_box == 5'd15)
        begin
                letter = q15;
                x_start = 8'd99;
                y_start = 7'd80;
        end
else if (counter_box == 5'd16)
        begin
                letter = q16;
```

```
                    x_start = 8'd105;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd17)
            begin
                    letter = q17;
                    x_start = 8'd111;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd18)
            begin
                    letter = q18;
                    x_start = 8'd117;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd19)
            begin
                    letter = q19;
                    x_start = 8'd123;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd20)
            begin
                    letter = q20;
                    x_start = 8'd129;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd21)
            begin
                    letter = q21;
                    x_start = 8'd135;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd22)
            begin
                    letter = q22;
                    x_start = 8'd141;
                    y_start = 7'd80;
            end
    else if (counter_box == 5'd23)
            begin
                    letter = q23;
                    x_start = 8'd147;
                    y_start = 7'd80;
            end
    else
            begin
                    letter = SPACE;
                    x_start = 8'd0;
                    y_start = 7'd0;
end
```

```
end

	shiftRegister u1(d0, q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18, q19, q20, q21, q22, q23, q24, q25, q26, q27, q28, q29, clk, shift, enable_load);

	draw draw0 (x, y, colour, counter_alphabet, counter_box, counter_black, letter, x_start, y_start, enable_draw, clk, LEDR, clear_alph, clear);//12

	endmodule

/*****************************************************************************
*                    Finite State Machine(s)                    *
*****************************************************************************/

module controlpath (clear_alph, LEDR, enable_draw, enable_load, shift, writeEn, clear, alphabet, received_data_en, q0, clk, next, counter_box, asd, asdf, hi, counter_black, hello, reset, HEX2, HEX3, HEX4, HEX5);
	input clk, next, reset, received_data_en, asd, asdf, hi, hello;
	output [6:0] HEX2, HEX3, HEX4, HEX5;
	input [5:0]alphabet;
	input [5:0]q0;
	input [4:0] counter_box;
	input [14:0] counter_black;
	output reg enable_load;
	output reg enable_draw;
	//output reg [5:0]Sel;
	output reg shift, writeEn, clear, clear_alph;
	output reg [5:0]LEDR;
	//output reg [4:0] counter_box;

	reg [5:0]PresentState, NextState;
	parameter RESET_S = 6'b000000, REG_S = 6'b000001, FIRST_S = 6'b000010, SECOND_S = 6'b000011,
	THIRD_S = 6'b000100, FOURTH_S = 6'b000101, FIFTH_S = 6'b000110, SIXTH_S = 6'b000111, SEVENTH_S = 6'b001000,
	EIGHTH_S = 6'b001001, NINTH_S = 6'b001010, TENTH_S = 6'b001011, ELEVENTH_S = 6'b001100, TWELTH_S = 6'b001101,
	THIRTEENTH_S = 6'b001110, FOURTEENTH_S = 6'b001111, FIFTEENTH_S = 6'b010000, SIXTEENTH_S = 6'b010001,
	SEVENTEENTH_S = 6'b010010, EIGHTEENTH_S = 6'b010011, NINETEETH_S = 6'b010100, TWENTIETH_S = 6'b010101,
	TWENTYFIRST_S = 6'b010110, TWENTYSECOND_S = 6'b010111, TWENTYTHIRD_S = 6'b011000, TWENTYFOURTH_S = 6'b011001,
	CLEAR_S = 6'b011010, SHIFT_S = 6'b011011, WAIT_S = 6'b011100;

	seg7 u5 (HEX2, alphabet[3:0]);//hex2
	seg7 u6 (HEX3, alphabet[5:4]);//hex3
	seg7 u7 (HEX4, q0[3:0]);//hex2
	seg7 u8 (HEX5, q0[5:4]);//hex3
```

```
/*********************************************************************************
*                        Sequential logic                          *
*********************************************************************************/
        //state table
  always@(*)
  begin: state_table
     case(PresentState)
                        RESET_S:
                                begin
                                        NextState = REG_S;
                                end
                        REG_S:
                                begin
                                        NextState = FIRST_S;
                                end
                        FIRST_S:
                                if(counter_box == 5'd1)
                                        NextState = SECOND_S;
                                else
                                        NextState = FIRST_S;
                        SECOND_S:
                                if(counter_box == 5'd2)
                                        NextState = THIRD_S;
                                else
                                        NextState = SECOND_S;
                        THIRD_S:
                                if(counter_box == 5'd3)
                                        NextState = FOURTH_S;
                                else
                                        NextState = THIRD_S;
                        FOURTH_S:
                                if(counter_box == 5'd4)
                                        NextState = FIFTH_S;
                                else
                                        NextState = FOURTH_S;
                        FIFTH_S:
                                if(counter_box == 5'd5)
                                        NextState = SIXTH_S;
                                else
                                        NextState = FIFTH_S;
                        SIXTH_S:
                                if(counter_box == 5'd6)
                                        NextState = SEVENTH_S;
                                else
                                        NextState = SIXTH_S;
                        SEVENTH_S:
                                if(counter_box == 5'd7)
                                        NextState = EIGHTH_S;
                                else
                                        NextState = SEVENTH_S;
                        EIGHTH_S:
```

```
        if(counter_box == 5'd8)
                NextState = NINTH_S;
        else
                NextState = EIGHTH_S;
NINTH_S:
        if(counter_box == 5'd9)
                NextState = TENTH_S;
        else
                NextState = NINTH_S;
TENTH_S:
        if(counter_box == 5'd10)
        NextState = ELEVENTH_S;
        else
                NextState = TENTH_S;
ELEVENTH_S:
        if(counter_box == 5'd11)
        NextState = TWELTH_S;
        else
                NextState = ELEVENTH_S;
TWELTH_S:
        if(counter_box == 5'd12)
        NextState = THIRTEENTH_S;
        else
                NextState = TWELTH_S;
THIRTEENTH_S:
        if(counter_box == 5'd13)
        NextState = FOURTEENTH_S;
        else
                NextState = THIRTEENTH_S;
FOURTEENTH_S:
        if(counter_box == 5'd14)
        NextState = FIFTEENTH_S;
        else
                NextState = FOURTEENTH_S;
FIFTEENTH_S:
        if(counter_box == 5'd15)
        NextState = SIXTEENTH_S;
        else
                NextState = FIFTEENTH_S;
SIXTEENTH_S:
        if(counter_box == 5'd16)
        NextState =  SEVENTEENTH_S;
        else
                NextState = SIXTEENTH_S;
SEVENTEENTH_S:
        if(counter_box == 5'd17)
        NextState = EIGHTEENTH_S;
        else
                NextState = SEVENTEENTH_S;
EIGHTEENTH_S:
        if(counter_box == 5'd18)
```

```
        NextState = NINETEETH_S;
        else
                NextState = EIGHTEENTH_S;
NINETEETH_S:
        if(counter_box == 5'd19)
        NextState = TWENTIETH_S;
        else
                NextState = NINETEETH_S;
TWENTIETH_S:
        if(counter_box == 5'd20)
        NextState = TWENTYFIRST_S;
        else
                NextState = TWENTIETH_S;
TWENTYFIRST_S:
        if(counter_box == 5'd21)
                NextState = TWENTYSECOND_S;
                else
                NextState = TWENTYFIRST_S;
TWENTYSECOND_S:
        begin
                if(counter_box == 5'd22)
                        NextState = TWENTYTHIRD_S;
                else
                        NextState = TWENTYSECOND_S;
        end
TWENTYTHIRD_S:
        begin
                if(counter_box == 5'd23)
                        NextState = TWENTYFOURTH_S;
                else
                        NextState = TWENTYTHIRD_S;
        end
TWENTYFOURTH_S:
        begin
                if(counter_box == 5'd24)
                        NextState = WAIT_S;
        end
CLEAR_S:
        begin
                if(counter_black == 15'd720)
                        NextState = WAIT_S;
                else
                        NextState = CLEAR_S;
        end
SHIFT_S:
        begin
                NextState = FIRST_S;
        end
WAIT_S:
        begin
                if(alphabet == q0)
```

```
                                        //if(hello == 1'b0)
                                                NextState = SHIFT_S;
                                        else
                                                NextState = WAIT_S;
                                end

                                default: NextState = RESET_S;
                        endcase
                end // state_table

                //state registers
always@(posedge clk)
begin: state_FFs
   if(reset == 1'b0)
      PresentState <= RESET_S;
   else
      PresentState <= NextState;
end //state_FFs

        //output logic
always@(*)
begin: output_logic
   case(PresentState)
                        RESET_S:
                                begin
                                        shift = 0;
                                        writeEn = 1;
                                        clear = 1;
                                        clear_alph = 1;
                                        enable_draw = 0;
                                        enable_load = 1;
                                        LEDR = RESET_S;
                                end
                        REG_S:
                                begin
                                        shift = 0;
                                        writeEn = 0;
                                        clear = 1;
                                        clear_alph = 1;
                                        enable_draw = 0;
                                        enable_load = 1;
                                        LEDR = REG_S;
                                end
                        FIRST_S:
                                begin
                                        shift = 0;
                                        writeEn = 1;
                                        clear = 0;
                                        clear_alph = 0;
                                        enable_draw = 1;
```

```
                enable_load = 0;
                LEDR = FIRST_S;
        end
SECOND_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = SECOND_S;
        end
THIRD_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = THIRD_S;
        end
FOURTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = FOURTH_S;
        end
FIFTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = FIFTH_S;
        end
SIXTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
```

```
                LEDR = SIXTH_S;
        end
SEVENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = SEVENTH_S;
        end
EIGHTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = EIGHTH_S;
        end
NINTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = NINTH_S;
        end
TENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = TENTH_S;
        end
ELEVENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = ELEVENTH_S;
```

```
        end
TWELTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = TWELTH_S;
        end
THIRTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = THIRTEENTH_S;
        end
FOURTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = FOURTEENTH_S;
        end
FIFTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = FIFTEENTH_S;
        end
SIXTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = SIXTEENTH_S;
        end
```

```
SEVENTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = SEVENTEENTH_S;
        end
EIGHTEENTH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = EIGHTEENTH_S;
        end
NINETEETH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = NINETEETH_S;
        end
TWENTIETH_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = TWENTIETH_S;
        end
TWENTYFIRST_S:
        begin
                shift = 0;
                writeEn = 1;
                clear = 0;
                clear_alph = 0;
                enable_draw = 1;
                enable_load = 0;
                LEDR = TWENTYFIRST_S;
        end
TWENTYSECOND_S:
```

```
		begin
			shift = 0;
			writeEn = 1;
			clear = 0;
			clear_alph = 0;
			enable_draw = 1;
			enable_load = 0;
			LEDR = TWENTYSECOND_S;
		end
	TWENTYTHIRD_S:
		begin
			shift = 0;
			writeEn = 1;
			clear = 0;
			clear_alph = 0;
			enable_draw = 1;
			enable_load = 0;
			LEDR = TWENTYTHIRD_S;
		end
	TWENTYFOURTH_S:
		begin
			shift = 0;
			writeEn = 1;
			clear = 0;
			clear_alph = 0;
			enable_draw = 1;
			enable_load = 0;
			LEDR = TWENTYFOURTH_S;
		end
	CLEAR_S:
		begin
			shift = 0;
			writeEn = 1;
			clear = 1;
			clear_alph = 1;
			enable_draw = 0;
			enable_load = 0;
			LEDR = CLEAR_S;
		end
	SHIFT_S:
		begin
			shift = 1;
			writeEn = 0;
			clear = 0;
			clear_alph = 1;
			enable_draw = 0;
			LEDR = SHIFT_S;
			enable_load = 1;
		end
	WAIT_S:
		begin
```

```
                                        shift = 0;
                                        writeEn = 0;
                                        clear = 1;
                                        clear_alph = 1;
                                        enable_draw = 0;
                                        LEDR = WAIT_S;
                                        enable_load = 0;
                                end
                        default:
                                begin
                                        shift = 0;
                                        writeEn = 0;
                                        clear = 1;
                                        enable_draw = 0;
                                        enable_load = 0; //changed from 1 to 0
                                        LEDR = 0;
                                end
                endcase
        end
endmodule

module shiftRegister (d0, q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17,
q18, q19, q20, q21,
        q22, q23, q24, q25, q26, q27, q28, q29, clk, shift, enable_load);

        input clk, shift, enable_load;
        output [5:0] q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18,
q19, q20, q21,
        q22, q23, q24, q25, q26, q27, q28, q29;
        output [5:0]d0;
        wire [5:0] d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, d15, d16, d17, d18, d19,
d20,
        d21, d22, d23, d24, d25, d26, d27, d28, d29, d30, d31, d32, d33, d34, d35, d36, d37, d38, d39,
d40, d41, d42, d43, d44, d45, d46, d47, d48, d49, d50, d51, d52, d53, d54, d55, d56, d57, d58, d59, d60,
d61, d62, d63, d64, d65, d66,
  d67, d68, d69, d70, d71, d72, d73, d74, d75, d76, d77, d78, d79, d80, d81, d82, d83, d84, d85;
        wire [5:0] q30, q31, q32, q33, q34, q35, q36, q37, q38, q39, q40, q41, q42, q43, q44, q45, q46,
q47, q48, q49, q50,
        q51, q52, q53, q54, q55, q56, q57, q58, q59, q60, q61, q62, q63, q64, q65, q66, q67, q68, q69,
q70, q71, q72, q73, q74, q75, q76, q77, q78, q79, q80, q81, q82, q83, q84, q85;

        parameter A = 6'b0, B = 6'b000001, C = 6'b000010, D = 6'b000011, E = 6'b000100, F =
6'b000101, G = 6'b000110,
        H = 6'b000111, I = 6'b001000, J = 6'b001001, K = 6'b001010, L = 6'b001011, M = 6'b001100, N
= 6'b001101,
        O = 6'b001110, P = 6'b001111, Q = 6'b010000, R = 6'b010001, S = 6'b010010, T = 6'b010011, U
= 6'b010100,
        V = 6'b010101, W = 6'b010110, X = 6'b010111, Y = 6'b011000, Z = 6'b011001, SPACE =
6'b011010, PERIOD = 6'b011011;
```

```
mux2to1 m85 (d85, PERIOD, SPACE, shift);
D_FF dff85 (q85, clk, enable_load, d85);

        mux2to1 m84 (d84, T, q85, shift);
D_FF dff84 (q84, clk, enable_load, d84);

        mux2to1 m83 (d83, I, q84, shift);
D_FF dff83 (q83, clk, enable_load, d83);

        mux2to1 m82 (d82, SPACE, q83, shift);
D_FF dff82 (q82, clk, enable_load, d82);

        mux2to1 m81 (d81, O, q82, shift);
D_FF dff81 (q81, clk, enable_load, d81);

        mux2to1 m80 (d80, D, q81, shift);
D_FF dff80 (q80, clk, enable_load, d80);

        mux2to1 m79 (d79, SPACE, q80, shift);
D_FF dff79 (q79, clk, enable_load, d79);

        mux2to1 m78 (d78, T, q79, shift);
D_FF dff78 (q78, clk, enable_load, d78);

        mux2to1 m77 (d77, S, q78, shift);
D_FF dff77 (q77, clk, enable_load, d77);

        mux2to1 m76 (d76, U, q77, shift);
D_FF dff76 (q76, clk, enable_load, d76);

        mux2to1 m75 (d75, J, q76, shift);
D_FF dff75 (q75, clk, enable_load, d75);

        mux2to1 m74 (d74, PERIOD, q75, shift);
D_FF dff74 (q74, clk, enable_load, d74);

        mux2to1 m73 (d73, S, q74, shift);
D_FF dff73 (q73, clk, enable_load, d73);

        mux2to1 m72 (d72, M, q73, shift);
D_FF dff72 (q72, clk, enable_load, d72);

        mux2to1 m71 (d71, A, q72, shift);
D_FF dff71 (q71, clk, enable_load, d71);

        mux2to1 m70 (d70, E, q71, shift);
D_FF dff70 (q70, clk, enable_load, d70);
```

```
        mux2to1 m69 (d69, R, q70, shift);
D_FF dff69 (q69, clk, enable_load, d69);

        mux2to1 m68 (d68, D, q69, shift);
D_FF dff68 (q68, clk, enable_load, d68);

        mux2to1 m67 (d67, SPACE, q68, shift);
D_FF dff67 (q67, clk, enable_load, d67);

        mux2to1 m66 (d66, E, q67, shift);
D_FF dff66 (q66, clk, enable_load, d66);

        mux2to1 m65 (d65, B, q66, shift);
D_FF dff65 (q65, clk, enable_load, d65);

        mux2to1 m64 (d64, SPACE, q65, shift);
D_FF dff64 (q64, clk, enable_load, d64);

        mux2to1 m63 (d63, S, q64, shift);
D_FF dff63 (q63, clk, enable_load, d63);

        mux2to1 m62 (d62, M, q63, shift);
D_FF dff62 (q62, clk, enable_load, d62);

        mux2to1 m61 (d61, A, q62, shift);
D_FF dff61 (q61, clk, enable_load, d61);

        mux2to1 m60 (d60, E, q61, shift);
D_FF dff60 (q60, clk, enable_load, d60);

        mux2to1 m59 (d59, R, q60, shift);
D_FF dff59 (q59, clk, enable_load, d59);

        mux2to1 m58 (d58, D, q59, shift);
D_FF dff58 (q58, clk, enable_load, d58);

        mux2to1 m57 (d57, SPACE, q58, shift);
D_FF dff57 (q57, clk, enable_load, d57);

        mux2to1 m56 (d56, R, q57, shift);
D_FF dff56 (q56, clk, enable_load, d56);

        mux2to1 m55 (d55, U, q56, shift);
D_FF dff55 (q55, clk, enable_load, d55);

        mux2to1 m54 (d54, O, q55, shift);
D_FF dff54 (q54, clk, enable_load, d54);

        mux2to1 m53 (d53, Y, q54, shift);
```

```
D_FF dff53 (q53, clk, enable_load, d53);

        mux2to1 m52 (d52, SPACE, q53, shift);
D_FF dff52 (q52, clk, enable_load, d52);

        mux2to1 m51 (d51, T, q52, shift);
D_FF dff51 (q51, clk, enable_load, d51);

        mux2to1 m50 (d50, E, q51, shift);
D_FF dff50 (q50, clk, enable_load, d50);

        mux2to1 m49 (d49, L, q50, shift);
D_FF dff49 (q49, clk, enable_load, d49);

        mux2to1 m48 (d48, SPACE, q49, shift);
D_FF dff48 (q48, clk, enable_load, d48);

        mux2to1 m47 (d47, T, q48, shift);
D_FF dff47 (q47, clk, enable_load, d47);

        mux2to1 m46 (d46, N, q47, shift);
D_FF dff46 (q46, clk, enable_load, d46);

        mux2to1 m45 (d45, O, q46, shift);
D_FF dff45 (q45, clk, enable_load, d45);

        mux2to1 m44 (d44, D, q45, shift);
D_FF dff44 (q44, clk, enable_load, d44);

        mux2to1 m43 (d43, PERIOD, q44, shift); // input empty space for y
        D_FF dff43 (q43, clk, enable_load, d43);

        mux2to1 m42 (d42, G, q43, shift);
        D_FF dff42 (q42, clk, enable_load, d42);

        mux2to1 m41 (d41, O, q42, shift);
        D_FF dff41 (q41, clk, enable_load, d41);

        mux2to1 m40 (d40, D, q41, shift);
        D_FF dff40 (q40, clk, enable_load, d40);

        mux2to1 m39 (d39, SPACE, q40, shift);
        D_FF dff39 (q39, clk, enable_load, d39);

        mux2to1 m38 (d38, Y, q39, shift);
        D_FF dff38 (q38, clk, enable_load, d38);

        mux2to1 m37 (d37, Z, q38, shift);
        D_FF dff37 (q37, clk, enable_load, d37);
```

```
mux2to1 m36 (d36, A, q37, shift);
D_FF dff36 (q36, clk, enable_load, d36);

mux2to1 m35 (d35, L, q36, shift);
D_FF dff35 (q35, clk, enable_load, d35);

mux2to1 m34 (d34, SPACE, q35, shift);
D_FF dff34 (q34, clk, enable_load, d34);

mux2to1 m33 (d33, E, q34, shift);
D_FF dff33 (q33, clk, enable_load, d33);

mux2to1 m32 (d32, H, q33, shift);
D_FF dff32 (q32, clk, enable_load, d32);

mux2to1 m31 (d31, T, q32, shift);
D_FF dff31 (q31, clk, enable_load, d31);

mux2to1 m30 (d30, SPACE, q31, shift);
D_FF dff30 (q30, clk, enable_load, d30);

mux2to1 m29 (d29, R, q30, shift);
D_FF dff29 (q29, clk, enable_load, d29);

mux2to1 m28 (d28, E, q29, shift);
D_FF dff28 (q28, clk, enable_load, d28);

mux2to1 m27 (d27, V, q28, shift);
D_FF dff27 (q27, clk, enable_load, d27);

mux2to1 m26 (d26, O, q27, shift);
D_FF dff26 (q26, clk, enable_load, d26);

mux2to1 m25 (d25, SPACE, q26, shift);
D_FF dff25 (q25, clk, enable_load, d25);

mux2to1 m24 (d24, S, q25, shift);
D_FF dff24 (q24, clk, enable_load, d24);

mux2to1 m23 (d23, P, q24, shift);
D_FF dff23 (q23, clk, enable_load, d23);

mux2to1 m22 (d22, M, q23, shift);
D_FF dff22 (q22, clk, enable_load, d22);

mux2to1 m21 (d21, U, q22, shift);
D_FF dff21 (q21, clk, enable_load, d21);
```

```
mux2to1 m20 (d20, J, q21, shift);
D_FF dff20 (q20, clk, enable_load, d20);

mux2to1 m19 (d19, SPACE, q20, shift);
D_FF dff19 (q19, clk, enable_load, d19);

mux2to1 m18 (d18, X, q19, shift);
D_FF dff18 (q18, clk, enable_load, d18);

mux2to1 m17 (d17, O, q18, shift);
D_FF dff17 (q17, clk, enable_load, d17);

mux2to1 m16 (d16, F, q17, shift);
D_FF dff16 (q16, clk, enable_load, d16);

mux2to1 m15 (d15, SPACE, q16, shift);
D_FF dff15 (q15, clk, enable_load, d15);

mux2to1 m14 (d14, N, q15, shift);
D_FF dff14 (q14, clk, enable_load, d14);

mux2to1 m13 (d13, W, q14, shift);
D_FF dff13 (q13, clk, enable_load, d13);

mux2to1 m12 (d12, O, q13, shift);
D_FF dff12 (q12, clk, enable_load, d12);

mux2to1 m11 (d11, R, q12, shift);
D_FF dff11 (q11, clk, enable_load, d11);

mux2to1 m10 (d10, B, q11, shift);
D_FF dff10 (q10, clk, enable_load, d10);

mux2to1 m9 (d9, SPACE, q10, shift);
D_FF dff9 (q9, clk, enable_load, d9);

mux2to1 m8 (d8, K, q9, shift);
D_FF dff8 (q8, clk, enable_load, d8);

mux2to1 m7 (d7, C, q8, shift);
D_FF dff7 (q7, clk, enable_load, d7);

mux2to1 m6 (d6, I, q7, shift);
D_FF dff6 (q6, clk, enable_load, d6);

mux2to1 m5 (d5, U, q6, shift);
D_FF dff5 (q5, clk, enable_load, d5);

mux2to1 m4 (d4, Q, q5, shift);
```

```
        D_FF dff4 (q4, clk, enable_load, d4);

        mux2to1 m3 (d3, SPACE, q4, shift);
        D_FF dff3 (q3, clk, enable_load, d3);

        mux2to1 m2 (d2, E, q3, shift);
        D_FF dff2 (q2, clk, enable_load, d2);

        mux2to1 m1 (d1, H, q2, shift);
        D_FF dff1 (q1, clk, enable_load, d1);

        mux2to1 m0 (d0, T, q1, shift);
        D_FF dff0 (q0, clk, enable_load, d0);

endmodule

module mux2to1(z, x, y, s);
        input [5:0]x, y;
        input s;
        output [5:0]z;

        assign z = s? y : x;
endmodule

module D_FF (q, clk, enable, d);
  input clk, enable;
  input [5:0]d;
  output reg [5:0]q;
  always @ (posedge clk)
  begin
     if(enable)
        q <= d;
  end
endmodule


module upcount_box (clear, counter_alphabet, Clock, Q);
        input Clock, clear;
        input [4:0] counter_alphabet;
 output reg [4:0] Q;

 always @(posedge Clock)
        if(clear)
                Q <= 1'b0;
        else if(counter_alphabet == 5'd28)
                Q <= Q + 1'b1;
endmodule

module upcount_alphabet (clear, enable, Clock, Q);
        input clear, Clock, enable;
```

```
        output reg [4:0] Q;
        always @(posedge Clock)
                if (clear)
                        Q <= 0;
                else if (Q == 5'd29)
                        begin
                        Q <= 5'b0;
                        end
    else if (enable)
                        Q <= Q + 1'b1;
endmodule

module upcount_X (clear, enable, Clock, counter_alphabet, y_cord, enable_y, Q);
  input clear, enable, Clock;
        input [4:0]counter_alphabet;
        input [2:0]y_cord;
        output reg enable_y;
  output reg [2:0] Q;

  always @(posedge Clock)
        begin
                if (clear || counter_alphabet == 5'd29)
                        begin
                        Q <= 3'd0;
                        enable_y = 1'b0;
                        end

                else if(Q == 3'd5)
                        begin
                        Q <= 3'b0;
                        enable_y <= 1'b0;
                        end

                else if (Q == 3'd4)
                        begin
                        Q <= Q + 1'b1;
                        enable_y = 1'b1;
                        end

                else if (enable)
                        begin
                        enable_y <= 0;
                        Q <= Q + 1'b1;
                        end
        end

endmodule

module upcount_Y (clear, enable, Clock, counter_alphabet, Q);
        input Clock, enable, clear;
```

```
        input [4:0]counter_alphabet;
        output reg [2:0] Q;

        always @(posedge Clock)
                if (clear || counter_alphabet == 5'd29)
                        Q <= 1'b0;
                else if (Q == 3'd5)
                        Q <= 3'd0;
    else if (enable)
                        Q <= Q + 1'b1;
endmodule


module upcount_black_X (clear, enable, Clock, enable_y, Q);
  input clear, Clock, enable;
        output reg enable_y;
        output reg [7:0] Q;

  always @(posedge Clock)
     if (~clear)
                        begin
        Q <= 1'b0;
                                enable_y <= 1'b0;
                        end
                else if (Q == 15'd142)
                begin
                        Q <= 3'b0;
                        enable_y <= 1'b0;
                        end

                        else if (Q == 15'd141)
                                begin
                                        Q <= Q + 1'b1;
                                        enable_y <= 1'b1;
                                end

    else if (enable)
                        begin
        Q <= Q +1;
                                enable_y <= 1'b0;
                        end
endmodule

module upcountblack_Y (clear, enable, Clock, Q);
        input Clock, enable, clear;
        output reg [2:0] Q;

        always @(posedge Clock)
                if (~clear)
                        Q <= 1'b0;
```

```
                else if (Q == 3'd5)
                        Q = 3'd0;
    else if (enable)
                        Q = Q + 1'b1;
endmodule

module upcount_black (Clock, enable, Q);
        input Clock, enable;
        output reg [14:0] Q;
        always @(posedge Clock)
                if (Q == 14'd720)
                        Q <= 0;
    else if (enable)
                        Q <= Q + 1'b1;
endmodule


module draw (x_out, y_out, colour, counter_alphabet, counter_box, counter_black, letter, x_start, y_start,
enable_draw, clk, LEDR, clear_alph, clear);
        input enable_draw, clk;
        input [5:0]letter;
        input [7:0]x_start;
        input [6:0]y_start;
        input clear_alph;
        input clear;
        output reg [7:0] x_out;
        output reg [6:0] y_out;
        output [4:0] counter_alphabet;
        output [4:0] counter_box;
        //output [4:0] counter_box;
        output reg [2:0] colour;
        output [9:0] LEDR;
        reg [9:0] memory;

        //assign writeEn = 1'b1;
        wire [2:0] x_cord;
        wire [2:0] y_cord;
        wire enable_y;
        wire [4:0]counter_box_1;
        wire [7:0] counter_black_X;
        wire[6:0]counter_black_Y;
        wire [2:0]colour_1;

        output [14:0]counter_black;

        parameter A = 6'b0, B = 6'b000001, C = 6'b000010, D = 6'b000011, E = 6'b000100, F =
6'b000101, G = 6'b000110,
        H = 6'b000111, I = 6'b001000, J = 6'b001001, K = 6'b001010, L = 6'b001011, M = 6'b001100, N
= 6'b001101,
```

```
	O = 6'b001110, P = 6'b001111, Q = 6'b010000, R = 6'b010001, S = 6'b010010, T = 6'b010011, U = 6'b010100,
	V = 6'b010101, W = 6'b010110, X = 6'b010111, Y = 6'b011000, Z = 6'b011001, SPACE = 6'b011010, PERIOD = 6'b011011;

	upcount_X u0(clear_alph, enable_draw, clk, counter_alphabet, y_cord, enable_y, x_cord); //X coordinate to draw
	upcount_Y u1(clear_alph, enable_y, clk, counter_alphabet, y_cord); //Y coordinate to draw

	//upXYcounter(enable_draw, clear_alph, clk, x_cord, y_cord);
	//seg7 q0 (HEX0, {4'b0, x_cord});
	seg7 q1 (HEX1, {4'b0, y_cord});
	seg7 q2 (HEX2, {3'b0, counter_alphabet[3:0]});
	seg7 q3 (HEX3, {6'b0, counter_alphabet[4]});


	upcount_black_X upbx(clear, clear, clk, counter_y_en, counter_black_X);


	upcountblack_Y upy(clear, counter_y_en, clk, counter_black_Y);

	upcount_black upb(clk, clear, counter_black);

	always@(*)
	begin
		if(clear)
			begin
			x_out = 8'd9 + counter_black_X;
			y_out = 7'd80 + counter_black_Y;
			//colour = 3'b000;
			end
		else
			begin
				x_out = x_start + {5'b0, x_cord};
				y_out = y_start + {4'b0, y_cord};
			end
	end

	upcount_box upbox (clear, counter_alphabet, clk, counter_box);
	//D_FF upboxdelay (counter_box, clk, clear, counter_box_1); // probably do not need this

	//upcount u4(clear_alph, enable_draw, clk, x_cord, y_cord, counter_alphabet);
	upcount_alphabet u2(clear_alph, enable_draw, clk, counter_alphabet);

	always@(*)
	begin
		if(clear)
			colour = 3'b000;
		else
			colour = colour_1;
```

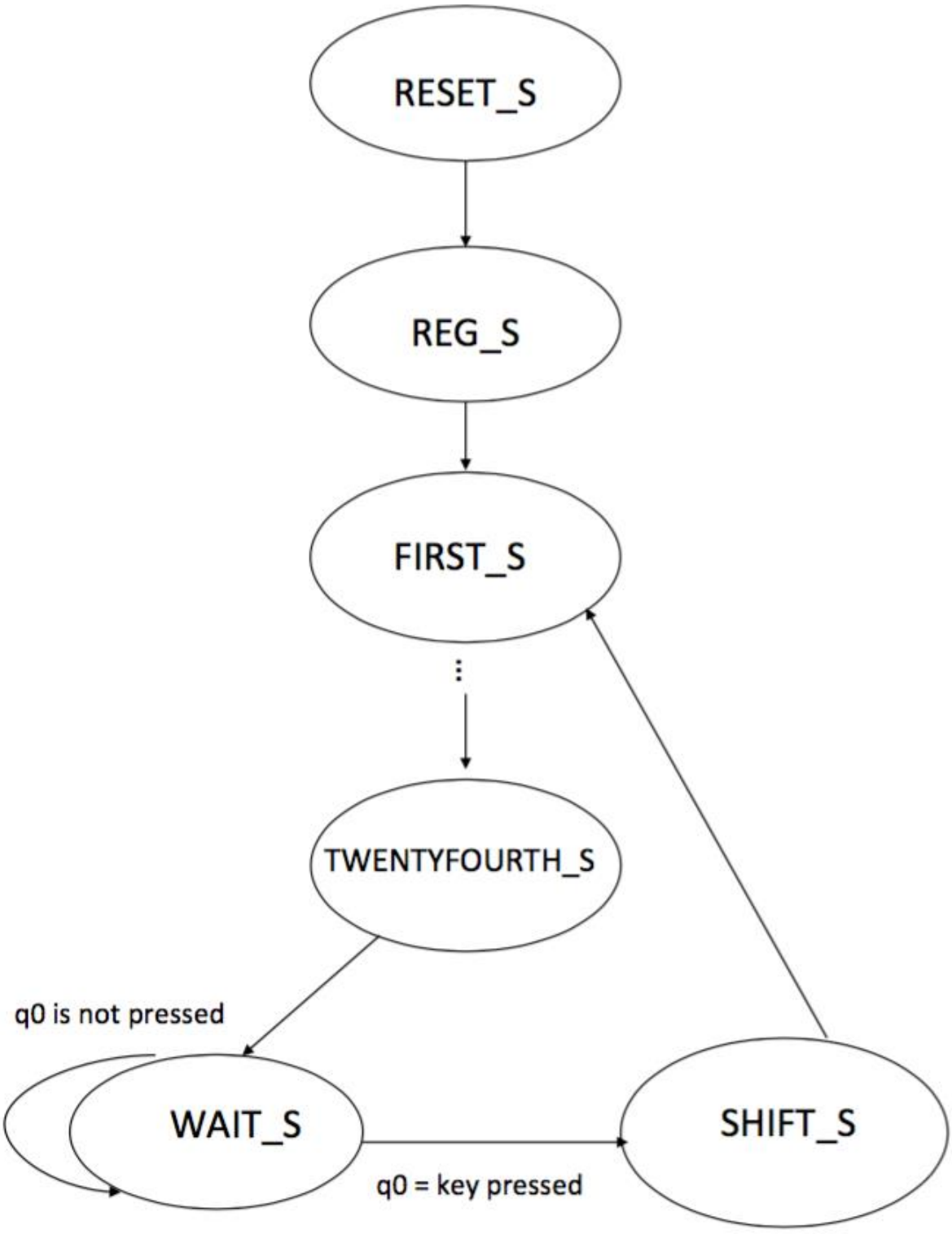
```
end
rom840x3 u3 (memory, clk, colour_1);

seg7 u9 (HEX4, {3'b0, counter_alphabet});
seg7 u10 (HEX5, {3'b0, counter_box});

always@(*)
begin
	case(letter)
		A:
			begin
				memory = {5'b0, counter_alphabet} + 10'd1;
			end
		B:
			begin
				memory = {5'b0, counter_alphabet} + 10'd31;
			end
		C:
			begin
				memory = {5'b0, counter_alphabet} + 10'd61;
			end
		D:
			begin
				memory = {5'b0, counter_alphabet} + 10'd91;
			end
		E:
			begin
				memory = {5'b0, counter_alphabet} + 10'd121;
			end
		F:
			begin
				memory = {5'b0, counter_alphabet} + 10'd151;
			end
		G:
			begin
				memory = {5'b0, counter_alphabet} + 10'd181;
			end
		H:
			begin
				memory = {5'b0, counter_alphabet} + 10'd211;
			end
		I:
			begin
				memory = {5'b0, counter_alphabet} + 10'd241;
			end
		J:
			begin
				memory = {5'b0, counter_alphabet} + 10'd271;
			end
		K:
			begin
```

```
                memory = {5'b0, counter_alphabet} + 10'd301;
            end
L:
            begin
                memory = {5'b0, counter_alphabet} + 10'd331;
            end
M:
            begin
                memory = {5'b0, counter_alphabet} + 10'd361;
            end
N:
            begin
                memory = {5'b0, counter_alphabet} + 10'd391;
            end
O:
            begin
                memory = {5'b0, counter_alphabet} + 10'd421;
            end
P:
            begin
                memory = {5'b0, counter_alphabet} + 10'd451;
            end
Q:
            begin
                memory = {5'b0, counter_alphabet} + 10'd481;
            end
R:
            begin
                memory = {5'b0, counter_alphabet} + 10'd511;
            end
S:
            begin
                memory = {5'b0, counter_alphabet} + 10'd541;
            end
T:
            begin
                memory = {5'b0, counter_alphabet} + 10'd571;
            end
U:
            begin
                memory = {5'b0, counter_alphabet} + 10'd601;
            end
V:
            begin
                memory = {5'b0, counter_alphabet} + 10'd631;
            end
W:
            begin
                memory = {5'b0, counter_alphabet} + 10'd661;
            end
X:
```

```
                        begin
                                memory = {5'b0, counter_alphabet} + 10'd691;
                        end
                Y:
                        begin
                                memory = {5'b0, counter_alphabet} + 10'd721;
                        end
                Z:
                        begin
                                memory = {5'b0, counter_alphabet} + 10'd751;
                        end
                SPACE:
                        begin
                                memory = {5'b0, counter_alphabet} + 10'd781;
                        end
                PERIOD:
                        begin
                                memory = {5'b0, counter_alphabet} + 10'd811;
                        end
                default: memory = {5'b0, counter_alphabet} + 10'd781;
        endcase
        end
endmodule

module alph_converter(alphabet, received_data);
  input [7:0]received_data;
  output reg [5:0]alphabet;

  parameter A = 6'b0, B = 6'b000001, C = 6'b000010, D = 6'b000011, E = 6'b000100, F = 6'b000101, G =
6'b000110,
  H = 6'b000111, I = 6'b001000, J = 6'b001001, K = 6'b001010, L = 6'b001011, M = 6'b001100, N =
6'b001101,
  O = 6'b001110, P = 6'b001111, Q = 6'b010000, R = 6'b010001, S = 6'b010010, T = 6'b010011, U =
6'b010100,
  V = 6'b010101, W = 6'b010110, X = 6'b010111, Y = 6'b011000, Z = 6'b011001, SPACE = 6'b011010,
PERIOD = 6'b011011;

  always@(*)
  begin
     if(received_data == 8'h1C)
        alphabet = A;
     else if (received_data == 8'h32)
        alphabet = B;
     else if (received_data == 8'h21)
        alphabet = C;
     else if (received_data == 8'h23)
        alphabet = D;
     else if (received_data == 8'h24)
        alphabet = E;
     else if (received_data == 8'h2B)
        alphabet = F;
```

```
      else if (received_data == 8'h34)
         alphabet = G;
      else if (received_data == 8'h33)
         alphabet = H;
      else if (received_data == 8'h43)
         alphabet = I;
      else if (received_data == 8'h3B)
         alphabet = J;
      else if (received_data == 8'h42)
         alphabet = K;
      else if (received_data == 8'h4B)
         alphabet = L;
      else if (received_data == 8'h3A)
         alphabet = M;
      else if (received_data == 8'h31)
         alphabet = N;
      else if (received_data == 8'h44)
         alphabet = O;
      else if (received_data == 8'h4D)
         alphabet = P;
      else if (received_data == 8'h15)
         alphabet = Q;
      else if (received_data == 8'h2D)
         alphabet = R;
      else if (received_data == 8'h1B)
         alphabet = S;
      else if (received_data == 8'h2C)
         alphabet = T;
      else if (received_data == 8'h3C)
         alphabet = U;
      else if (received_data == 8'h2A)
         alphabet = V;
      else if (received_data == 8'h1D)
         alphabet = W;
      else if (received_data == 8'h22)
         alphabet = X;
      else if (received_data == 8'h35)
         alphabet = Y;
      else if (received_data == 8'h1A)
         alphabet = Z;
      else if (received_data == 8'h29)
         alphabet = SPACE;
      else if (received_data == 8'h49)
         alphabet = PERIOD;
                              else
                                       alphabet = X;
   end
endmodule

module seg7 (HEX0, SW);
          input [7:0] SW;
```

```
	output reg [6:0] HEX0;

	always@(*)
	begin
		case(SW)
			0: HEX0 = 7'b1000000;
			1: HEX0 = 7'b1111001;
			2: HEX0 = 7'b0100100;
			3: HEX0 = 7'b0110000;
			4: HEX0 = 7'b0011001;
			5: HEX0 = 7'b0010010;
			6: HEX0 = 7'b0000010;
			7: HEX0 = 7'b1111000;
			8: HEX0 = 7'b0;
			9: HEX0 = 7'b0010000;
			10: HEX0 = 7'b0001000;
			11: HEX0 = 7'b0000011;
			12: HEX0 = 7'b1000110;
			13: HEX0 = 7'b0100001;
			14: HEX0 = 7'b0000110;
			15: HEX0 = 7'b0001110;
			default: HEX0 = 7'bx;
		endcase
	end
endmodule

module time_counter (counter, clk);
  reg [26:0]out;
  input clk;
  output reg [3:0]counter;
  always@(posedge clk)
		begin
			if(out == 27'd49999999)
				begin
					out <= 1'b0;
					counter <= counter + 1'b1;
				end
			else
				out <= out + 1'b1;

    end
endmodule
```


RESET_S
REG_S
FIRST_S
TWENTYFOURTH_S
q0 is not pressed
WAIT_S
SHIFT_S
q0 = key pressed