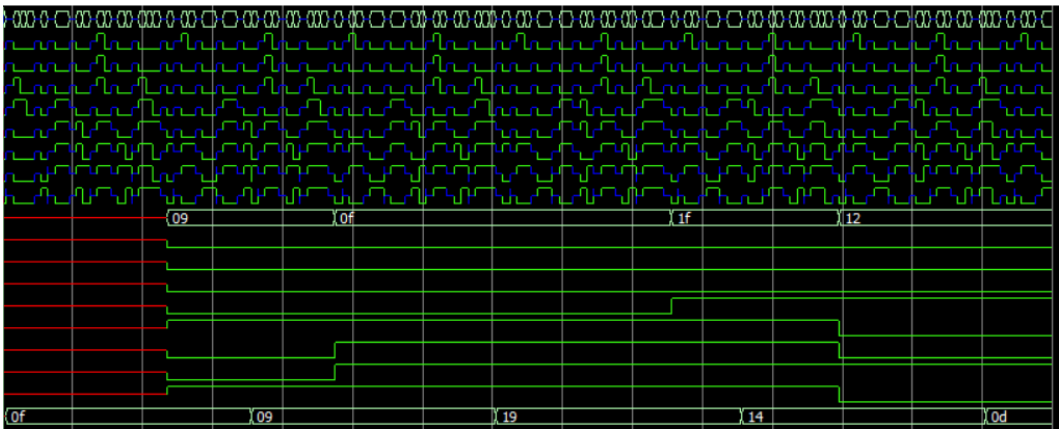
SystemVerilog Logbook
Joseph Butterworth



Contents

Logic Gates - 23/11/20

NOT gate

```
module not (output logic y,
input logic a);

always_comb
y = ~a;

endmodule
```

AND gate

```
module and2 (output logic y,
input logic a, b);

always_comb
y = a & b;

endmodule
```

OR gate

```
module or2 (output logic y,
input logic a, b);

always_comb
y = a | b;

endmodule
```

NAND gate

```
module nand2 (output logic y,
input logic a, b);

always_comb
y = ~(a & b);

endmodule
```

NOR gate

```
module nor2 (output logic y,
input logic a, b);

always_comb
y = ~(a | b);

endmodule
```

3 Input NAND gate

```
module nand2 (output logic y,
input logic a, b, c);

always_comb
y = ~(a & b & c);

endmodule
```

Full Adders - 23/11/20

A structural system replicates the function of a logic system by using simulated versions of each logic gate.

```
module full_adder (output logic sum, co,
                   input logic a, b, ci);
                         logic i, j, k, l;

xor g0 (i, a, b);
xor g1 (sum, i, ci);
and g2 (j, a, b);
and g3 (k, a, ci);
and g4 (l, b, ci);
or g5 (co, j, k, l);

endmodule
```

Two always_comb blocks can be used to simulate the different levels of logic. This is particularly useful when looking at propagation delay in the system.

```
module full_adder (output logic sum, co,
                   input logic a, b, ci);
                         logic i, j, k, l;

//level of logic 1
always_comb
    begin
    i = (a & ~b) | (~a & b);
    j = a & b;
    k = a & ci;
    l = b & ci;
    end

//level of logic 2
always_comb
    begin
    sum = (i & ~ci) | (~i & ci);
    co = j | k | l;
    end

endmodule
```

A dataflow version emulates the function of the circuit most efficiently.

```
module full_adder (output logic sum, co,
                   input logic a, b, ci);

always_comb
    begin
    sum = a ^ b ^ ci;
    co = (a & b) |
         (a & ci) |
         (b & ci);
    end

endmodule
```

Multiplexers - 23/11/20

2 Input Multiplexer

```
module mux2 (output logic y,
             input logic a, b, s);
always_comb
    if (s)
        y = b;
    else
        y = a;

endmodule
```

4 Input Multiplexer

```
module mux (input logic a, b, c, d,
            input logic s1, s0,
           output logic y);

always_comb
    if (s0)
        if (s1)
            y = d;
        else
            y = c;
    else
        if (s1)
            y = b;
        else
            y = a;

endmodule
```

8 Input Multiplexer using cases

```
module mux #(parameter N = 3) // n = no. of bits
    (input logic [(2**N)-1:0] a, [N-1:0] s,
    output logic y);

always_comb
    case(s)
        N'b111 : y = a[7]
        N'b110 : y = a[6]
        N'b101 : y = a[5]
        N'b100 : y = a[4]
        N'b011 : y = a[3]
        N'b010 : y = a[2]
        N'b001 : y = a[1]
        N'b000 : y = a[0]
        default:
    endcase
endmodule
```

n bit Multiplexer

```
module mux #(parameter N = 3) // n = no. of bits
    (input logic [(2**N)-1:0] a, [N-1:0] s,
    output logic y);

always_comb
    y = a[s]

endmodule
```

Decoders - 23/11/20

2-4 bit Decoder

```
module decoder (input logic [1:0] a,
                output logic [3:0] y);
always_comb
    case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
        3 : y = 8;
        default : y = 'x;
    endcase

endmodule
```

3-8 bit Decoder

```
 module decoder (input logic [2:0] a,
                 output logic [7:0] y);
 always_comb
     case (a)
         0 : y = 1;
         1 : y = 2;
         2 : y = 4;
         3 : y = 8;
         4 : y = 16;
         5 : y = 32;
         6 : y = 64;
         7 : y = 128;
         default : y = 'x;
     endcase

 endmodule
```

n- 2^n bit Decoder

```
module decoder2
    #(parameter N =3)
    (input logic [N-1:0] a,
    output logic [(2**N)-1:0] y );

    always_comb
        y = 1'b1 << a;

endmodule
```

Encoders - 13/01/21

4-2 bit Decoder

```
module encoder (output logic [1:0] y, output logic valid,
                input logic [3:0] a);

always_comb //procedural block, combinational logic
    begin
    valid = 1'b1; // default value
    unique casez (a)    //treat z or ? as don't care
                        //unique checks for overlapping branches
        4'b1??? : y = 2'b11;
        4'b01?? : y = 2'b10;
        4'b001? : y = 2'b01;
        4'b0001 : y = 2'b00;
        4'b0000 : begin
                y = 2'b00;
                valid = 1'b0;
                end
    default : begin // unknown in, unknown out
                y = 2'bxx;
                valid = 1'bx;
                end
    endcase
    end
endmodule
```

2^n -n bit Encoder

```
//2^n - n bit encoder
//https://stackoverflow.com/questions/29529730/priority-encoder-in-verilog

module n_encoder #(parameter N =3)
    (output logic [N-1:0] y, output logic valid,
      input logic [(2**N)-1:0] a);
    integer i;

    always @*
    begin
    y = 0; // default value if a is all 0's
    valid = 0;

    for (i=[(2**N)-1:0]; i>=0; i=i-1)
    {
        if (a[i])
        {
            y = i;
            valid=1;
        }
    }
    end
endmodule
```

Tri-State Buffer - 23/11/20

```
module three_state
    (input logic a, enable,
    output logic y);

    always_comb
        y = enable ? a : 1'bz;

endmodule
```

Test Benches - 23/11/10

4 Input Multiplexer

```
module test_decoder; // No i/o

logic a, b, c, d;
logic s1, s0;
logic y;

decoder d0 (.*);    // only works if internal
                    // signals have the same names

initial
    begin
    a, b, c, d = 0;
    #10ns
    s1 = 0;
    s0 = 0;
    #10ns
    s1 = 0;
    s0 = 1;
    #10ns
    s1 = 1;
    s0 = 0;
    #10ns
    s1 = 1;
    s0 = 1;
    a, b, c, d = 1;
```

```
    #10ns
    s1 = 0;
    s0 = 0;
    #10ns
    s1 = 0;
    s0 = 1;
    #10ns
    s1 = 1;
    s0 = 0;
    #10ns
    s1 = 1;
    s0 = 1;
    end

endmodule
```

2-4 bit Decoder

```
module test_decoder; // No i/o

logic [1:0] a;
logic [3:0] y;

decoder d0 (.*);    // only works if internal
                    // signals have the same names

initial
    begin
    #10ns a = 0;       // initial value of a is 'x
    #10ns a = 1;
    #10ns a = 2;
    #10ns a = 3;
    end

endmodule
```

3-8 bit Decoder

```
module test_decoder; // No i/o

logic [2:0] a;
logic [7:0] y;

decoder d0 (.*);    // only works if internal
                    // signals have the same names

initial
    begin
    #10ns a = 0;       // initial value of a is 'x
    #10ns a = 1;
    #10ns a = 2;
    #10ns a = 3;
    #10ns a = 4;
    #10ns a = 5;
    #10ns a = 6;
    #10ns a = 7;
    end

endmodule
```

Flip-flops - 13/01/21

D-type flip flop

```
module dff (output logic q,
            input logic d, clk);

always_ff @(posedge clk)
    q <= d;
endmodule
```

Resettable d-type flip flop

```
module dffr (output logic q,
             input logic d, clk, n_reset);

always_ff @(posedge clk,
            negedge n_reset)

    if (!n_reset)
        q <= '0;
    else
        q <= d;

endmodule
```

Settable d-type flip flop

```
module dffs (output logic q,
             input logic d, clk, set);

always_ff @(negedge clk,
            posedge set)

    if (set)
        q <= '1;
    else
        q <= d;

endmodule
```

Enabled d-type flip flop

```
module dffe (output logic q,
             input logic d, clk, enable);

    always_ff @(posedge clk)
        if (enable)
            q <= d;
endmodule
```

Enabled resettable d-type flip flops

```
module dffre (output logic q,
              input logic d, clk, enable, n_reset);

always_ff @(posedge clk,
            negedge n_reset)
    if (!n_reset)
        q <= '0;
    else if (enable)
        q <= d;
endmodule
```

Registers - 13/01/21

n-bit register

```
module dffn #(parameter N = 8)
            (output logic [N-1:0] q,
              input logic [N-1:0] d,
              input clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
    if (!n_reset)
        q <= '0;
    else
        q <= d;
endmodule
```

Enabled n-bit register

```
module dffne #(parameter N = 8)
            (output logic [N-1:0] q,
              input logic [N-1:0] d,
              input clk, enable, n_reset);

always_ff @(posedge clk,
            negedge n_reset)
    if (!n_reset)
        q <= '0;
    else if (enable)
        q <= d;
endmodule
```

Binary Counter

```
module counter #(parameter N = 8)
                (output logic [N-1:0] q,
                  input logic clk, n_reset);

always_ff @(posedge clk,
            negedge n_reset)
    if (!n_reset)
        q <= 0;
    else
        q <= q + 1;
endmodule
```

SIPO shift register

```
module sipo #(parameter N = 8)
            (output logic [N-1:0] q,
              input logic a, clk);

always_ff @(posedge clk)
    q <= {q[N-2:0], a};
endmodule
```

PISO shift register

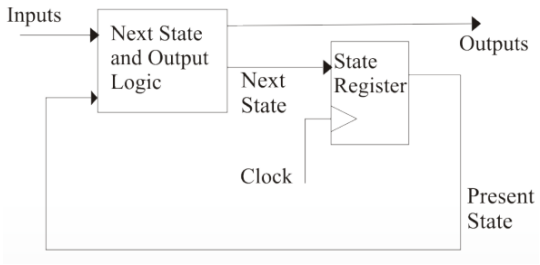
```
module piso #(parameter N = 8)
            (output logic q,
              input logic [N-1:0] a, clk);

always_ff @(posedge clk)
    q <= {a, q[N-1:1]};
endmodule
```

Universal shift register

```
module usr #(parameter N = 4)
            (output logic [N-1:0] q,
              input logic lin, rin, clk, n_reset,
              input logic [N-1:0] a,
              input logic [1:0] s);

always_ff @(posedge clk,
            negedge n_reset)
    if (!n_reset)
        q <= '0;
    else
        case (s)
            2'b11: q <= a;
            2'b01: q <= {q[N-2:0], lin};
            2'b10: q <= {rin, q[N-1:1]};
            default:;
        endcase
endmodule
```

State Machines - 13/01/21

Traffic light with one always_comb block



```
module traffic (output logic start_timer,
                            major_green, minor_green,
                input logic clock, n_reset, timed, car);

enum {G, R} present_state, next_state;

always_ff @(posedge clock, negedge n_reset)
    begin: SEQ //label
        if (!n_reset)
            present_state <= G;
        else
        present_state <= next_state;
end

always_comb
    begin: COM
    start_timer = '0;
    minor_green = '0;
    major_green = '0;
    next_state = present_state;

    unique case (present_state)
        G: begin
            major_green = '1;
            if (car)
                begin
                    start_timer = '1;
                    next_state = R;
                end
        end
        R: begin
            minor_green = '1;
            if (timed)
                next_state = G;
        end
    endcase
end
endmodule
```

Traffic light with two always_comb blocks

```
module traffic (output logic start_timer,
                                major_green, minor_green,
                input logic clock, n_reset, timed, car);

enum {G, R} state;

always_ff @(posedge clock,
            negedge n_reset)
    begin: SEQ
        if (!n_reset)
            state <= G;
        else
            unique case (state)
                G: if (car)
                    state <= R;
                R: if (timed)
                    state <= G;
            endcase
end

always_comb
    begin: OP
    start_timer = '0;
    minor_green = '0;
    major_green = '0;

    unique case (state)
        G: begin
        major_green = '1;
        if (car)
            start_timer = '1;
        end
        R: minor_green = '1;
    endcase
end
endmodule
```

Vending Machine with one always_comb block

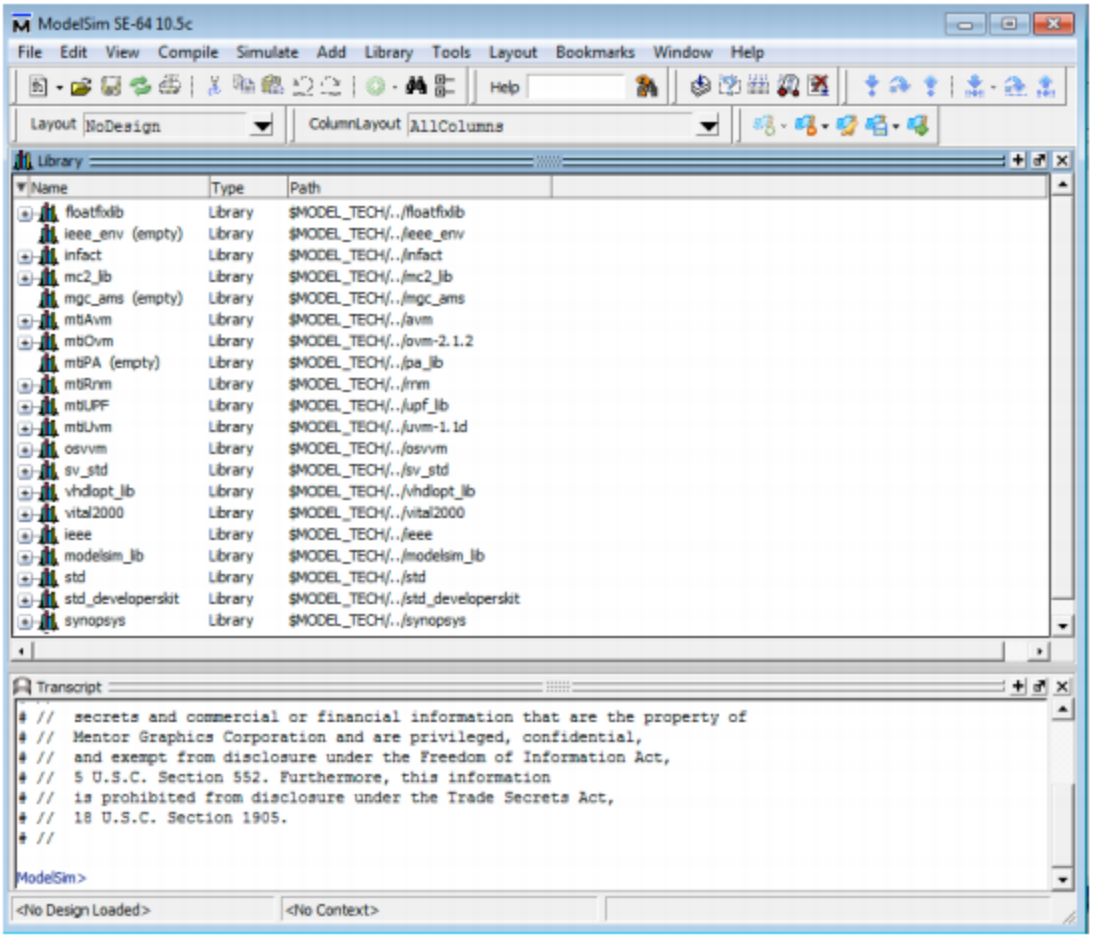
```
// vending machine

module vending(output logic ready,dispense,ret,coin
               input logic clock,n_reset,twenty,ten);

enum {A, B, C, D, F, I} present_state, next_state;

always_ff @(posedge clock, negedge n_reset)
    begin: SEQ
        if (~n_reset)
            present_state <= A;
        else
            present_state <= next_state;
    end

always_comb
    begin: COM
        ready = '0;
        dispense = '0;
        ret = '0;
        coin = '0;

        unique case (present_state)
            A : begin
                ready = '1;
                if (twenty)
                    next_state = D;
                else if (ten)
                    next_state = C;
                else
                    next_state = A;
            end
            B : begin
                dispense = '1;
                next_state = A;
            end
```

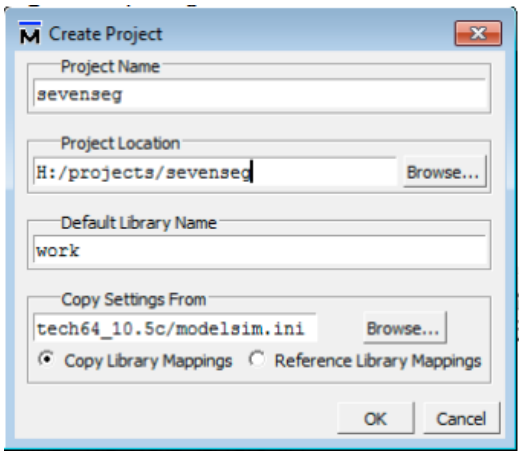
```
            C : begin
                coin = '1;
                if (twenty)
                    next_state = F;
                else if (ten)
                    next_state = D;
                else
                    next_state = C;
            end
            D : begin
                coin = '1;
                if (twenty)
                    next_state = B;
                else if (ten)
                    next_state = F;
                else
                    next_state = D;
            end
            F : begin
                coin = '1;
                if (twenty)
                    next_state = I;
                else if (ten)
                    next_state = B;
                else
                    next_state = F;
            end
            I : begin
                ret = '1;
                next_state = A;
            end
        endcase
    end
endmodule
```

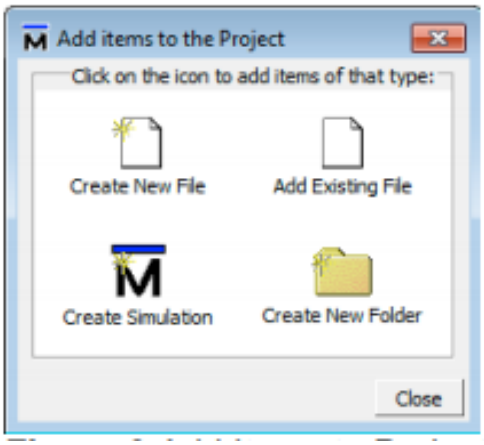
X5: Programmable Logic Devices

ModelSim Walkthrough - 23/11/20

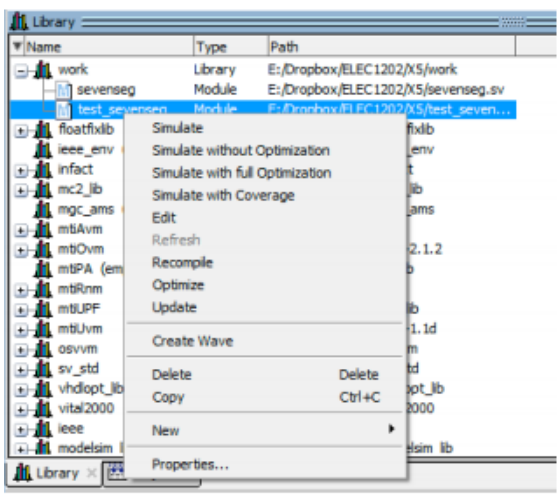
Start the ModelSim simulator from the Windows Start menu (Your School Software -> Modelsim). You will see a window like that in Fig. 1. (You will see a “splash” screen first and you may get a “Welcome” menu, which you can close.) Note that the version number at the top of the window might be different. Click on File -> New -> Project. Enter the name of the project and set the Project Location, Fig. 2. Notice that ModelSim uses forward slashes (/) in folder paths.

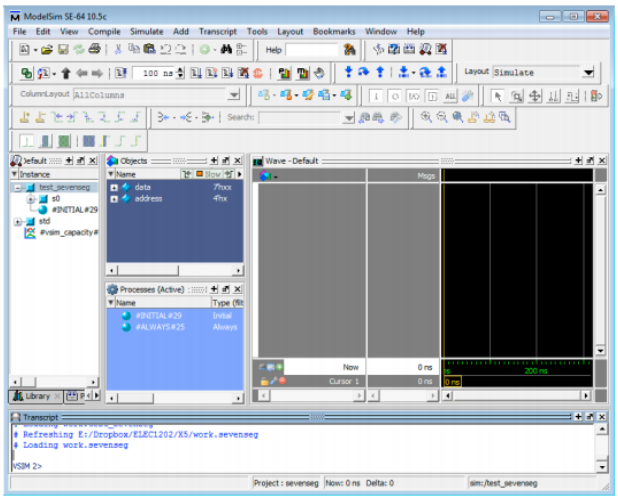




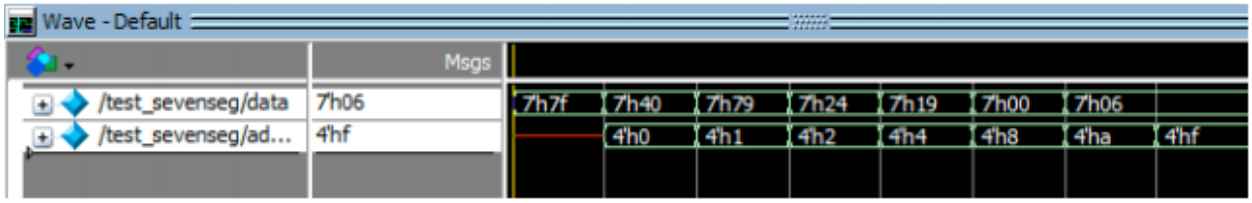


Click on Add Existing File and then select sevenseg.sv and test_sevenseg.sv using the browse button in the next window. Click on OK and Close. Compile sevenseg.sv and test_sevenseg.sv by selecting Compile -> Compile All. Any syntax errors will be listed in the Transcript window. Select the Library tab, click on the work library to open it and right-click on test_sevenseg, Fig. 4. Select Simulate. Several simulation windows will now open, Fig. 5. If the Wave window is not open, click View -> Wave. Click and drag the two signals in the Objects window to the Wave window.



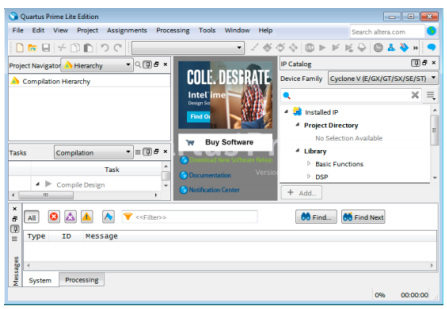


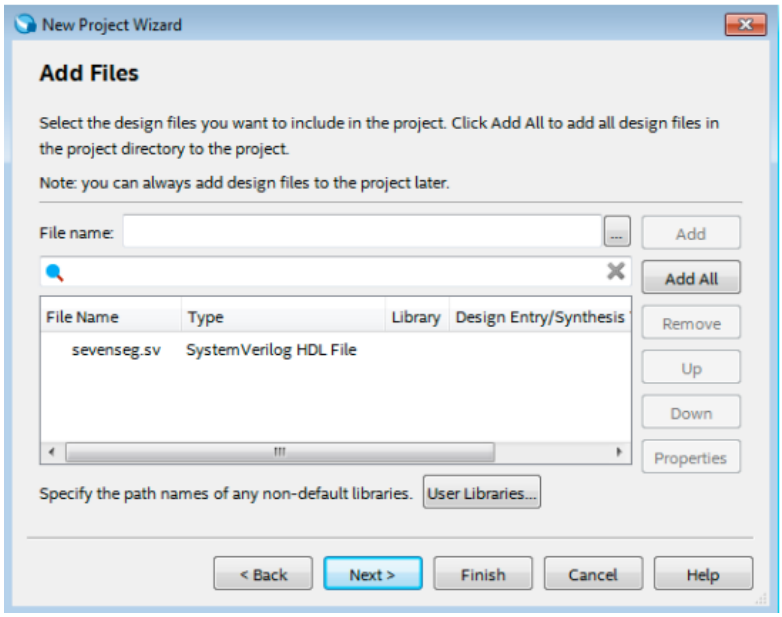
Start the simulation from Simulate -> Run -> Run -All (or use the ‘down arrow’ icon to the right of the “100 ns” box). Expand the waveform using Wave -> Zoom -> Zoom Full (or the solid magnifying glass icon). You should see a waveform like that in Fig. 6. X (unknown) values are shown with a red line. Individual bits can be displayed by clicking on the + symbol on the left hand side.



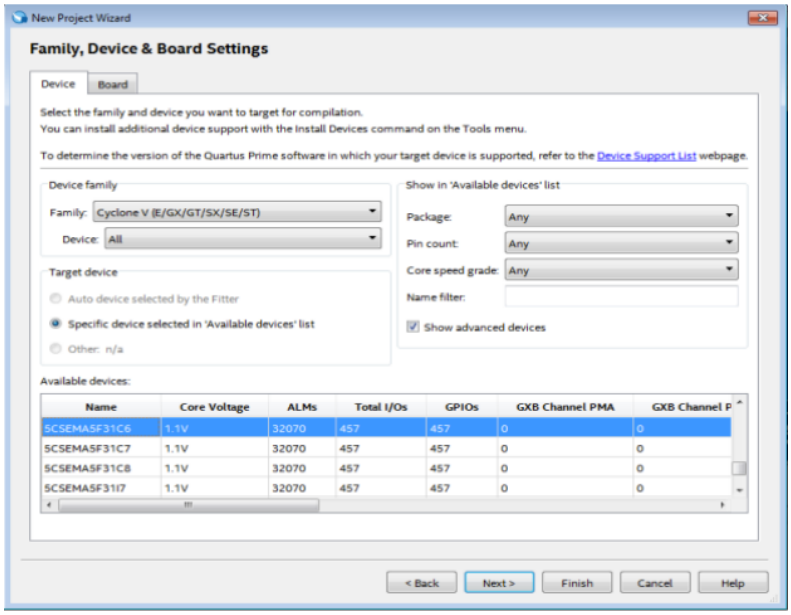
Quartus Walkthrough - 23/11/20

This tutorial is written for the DE1-SoC development board, which uses a Cyclone V FPGA. If you use a different board, you will have to modify these instructions as appropriate. Download the files sevenseg.sv and sevenseg_pins.qsf and save them to a suitable folder. If you have already done this for the simulation walk-through, use the same folder. In general, it is a good idea to use exactly the same files for simulation and synthesis. Start Quartus from the Windows Start menu (Your School Software -> Quartus Prime 16.1). You will see a window like that in Fig. 1. (You will see a “splash” screen first.) Click on File -> New Project Wizard. Enter working directory and the name of the project, Fig. 2.

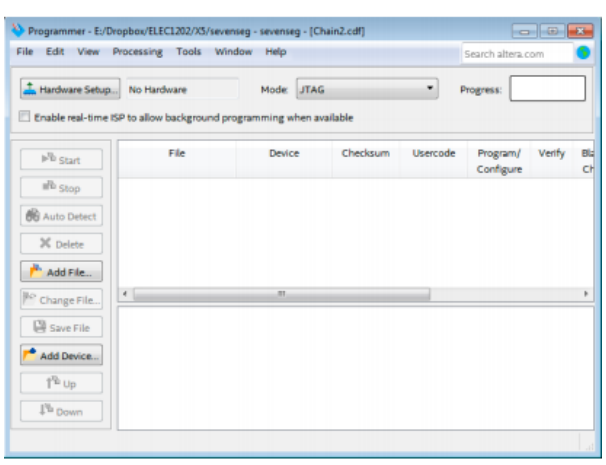


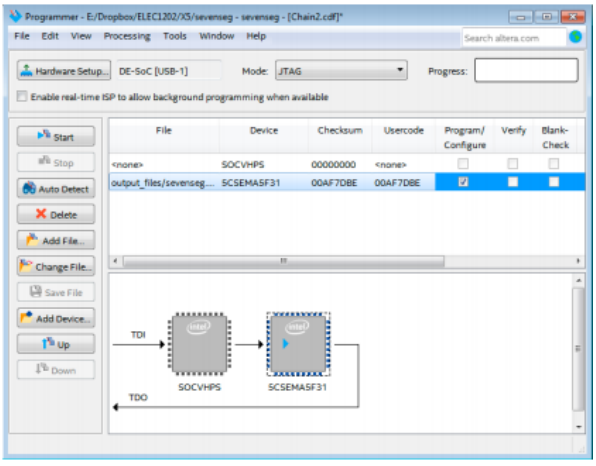


Click on the browse button (...) next to the ‘Add’ button and then select sevenseg.sv using the browse button in the next window. Click on ‘Open’ and then ‘Next >’ again. In the Family, Device & Board Settings window, on the Device tab select a Cyclone V, 5CSEMA5F31C6. It is very important that you choose the correct device. Click ‘Next >’ and ‘Next >’ again. Check the details on the Summary page and click ‘Finish’.



Compile sevenseg.sv by selecting Processing -> Start -> Start analysis and Synthesis. Any syntax errors will be listed in the Messages window. This does not do a complete placement and routing, but checks the syntax and that the design can be synthesised. This step creates a project file, sevenseg.qsf. Open sevenseg.qsf in a text editor (do not use Word or another word processing tool). Also open sevenseg_pins.qsf and paste the contents of sevenseg_pins.qsf into sevenseg.qsf. This assigns the signals in the SystemVerilog design to the physical pins of the FPGA and hence to the switches and LEDs on the board. Save the modified version of sevenseg.qsf. In Quartus, select Processing -> Start Compilation. This will do a full compilation, together with the placement and routing and may take a few minutes to complete. When the compilation view, the Task pane will have green ticks – Fig. 5. Select Tools -> Programmer. This opens a new window, Fig. 6. If 'No Hardware' is shown, click on Hardware Setup and double click on the DE-SoC device. If this is not visible, install the driver as shown in the box below. Now click on Auto Detect. Select the 2nd option, 5CSEMA5. Right click on the image 5CSEMA5F31 and select Edit -> Change File. In the dialogue box, open the folder output_files and select sevenseg.sof. Click on the Program/Configure option for the 5CSEMA5F31. The Programmer box should now look like Fig. 7.



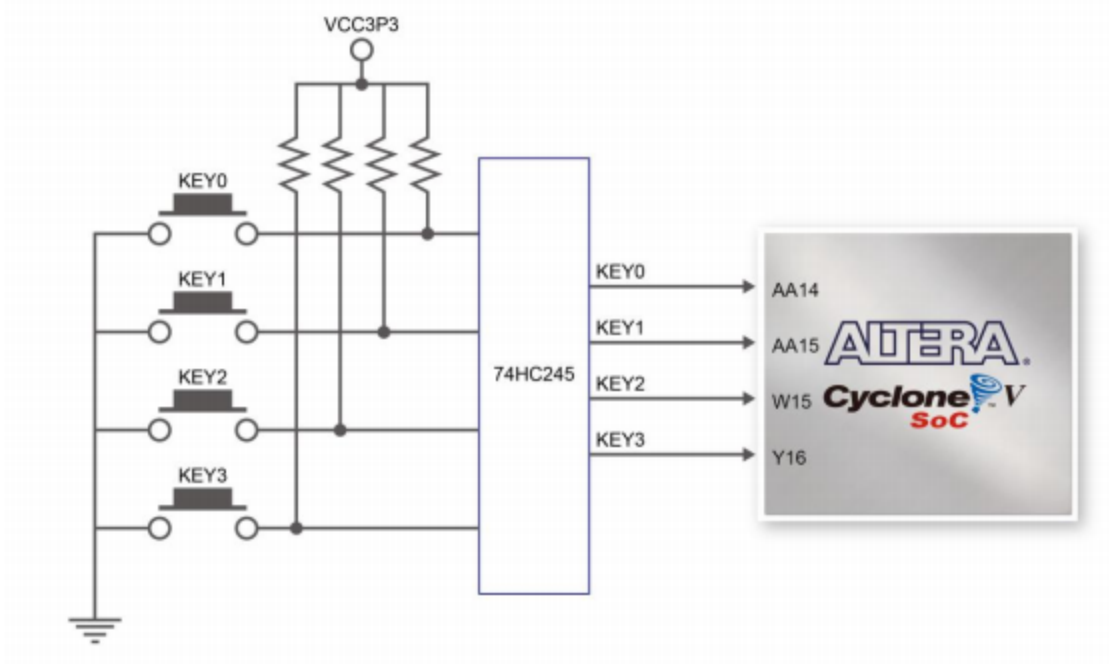


The DE1-SoC Development Board and the Intel/Altera Cyclone V - 23/11/20

1) Which FPGA pins are connected to the slide switches, SW0-SW9, to the pushbuttons, Key0-Key3, and to the 7-segment displays HEX0-HEX1? ´

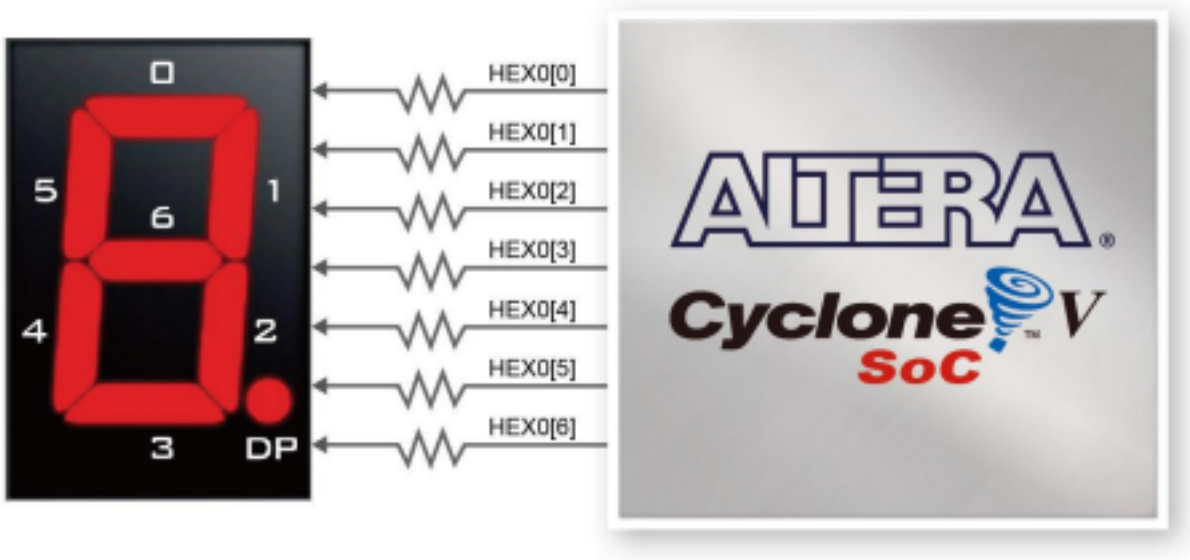
SW0-SW9

KEY0-KEY3



Each push-button switch provides a high logic level when it is not pressed, and provides a low logic level when depressed

HEX0-HEX1


0
1
2
3
4
5
6
DP
HEX0[0]
HEX0[1]
HEX0[2]
HEX0[3]
HEX0[4]
HEX0[5]
HEX0[6]
ALTERA
Cyclone V
SoC

Signal Name	FPGA Pin No.	FPGA Pin No.
HEX0[0]	PIN_AE26	Seven Segment Digit 0[0]
HEX0[1]	PIN_AE27	Seven Segment Digit 0[1]
HEX0[2]	PIN_AE28	Seven Segment Digit 0[2]
HEX0[3]	PIN_AG27	Seven Segment Digit 0[3]
HEX0[4]	PIN_AF28	Seven Segment Digit 0[4]
HEX0[5]	PIN_AG28	Seven Segment Digit 0[5]
HEX0[6]	PIN_AH28	Seven Segment Digit 0[6]
HEX1[0]	PIN_AJ29	Seven Segment Digit 1[0]
HEX1[1]	PIN_AH29	Seven Segment Digit 1[1]
HEX1[2]	PIN_AH30	Seven Segment Digit 1[2]
HEX1[3]	PIN_AG30	Seven Segment Digit 1[3]
HEX1[4]	PIN_AF29	Seven Segment Digit 1[4]
HEX1[5]	PIN_AF30	Seven Segment Digit 1[5]
HEX1[6]	PIN_AD27	Seven Segment Digit 1[6]

2) Are the 7-segment elements turned on by a logic 1 or by a logic 0? (In other words, are they active high or active low?) Are the push-buttons active high, or active low?

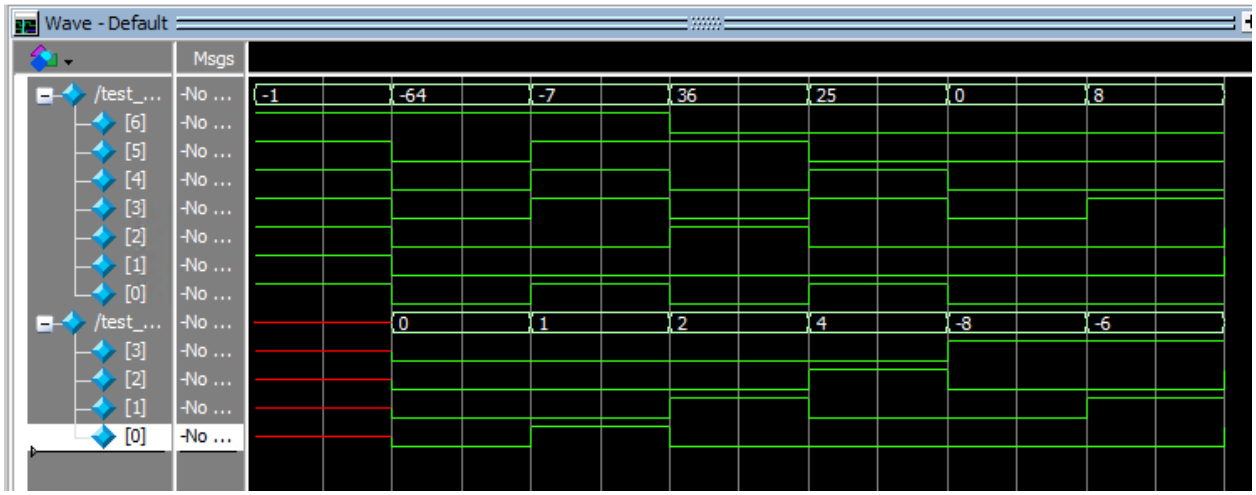
The seven segments (common anode) are connected to pins on Cyclone V SoC FPGA. Applying a low logic level to a segment will light it up and applying a high logic level turns it off.

SystemVerilog - 23/11/20

First I modified the provided code so that it would output the range of 0-15 in hexadecimal on a single seven segment display.

```
always_comb
  unique casez (address)
    4'b0000 : data = 7'b1000000;
    4'b0001 : data = 7'b1111001;
    4'b0010 : data = 7'b0100100;
    4'b0011 : data = 7'b0110000;
    4'b0100 : data = 7'b0011001;
    4'b0101 : data = 7'b0010010;
    4'b0110 : data = 7'b0000010;
    4'b0111 : data = 7'b1111000;
    4'b1000 : data = 7'b0000000;
    4'b1001 : data = 7'b0010000;
    4'b1010 : data = 7'b0001000;
    4'b1011 : data = 7'b0000011;
    4'b1100 : data = 7'b1000110;
    4'b1101 : data = 7'b0100001;
    4'b1110 : data = 7'b0000110;
    4'b1111 : data = 7'b0001110;
    default : data = 7'b1111111;
  endcase
endmodule
```

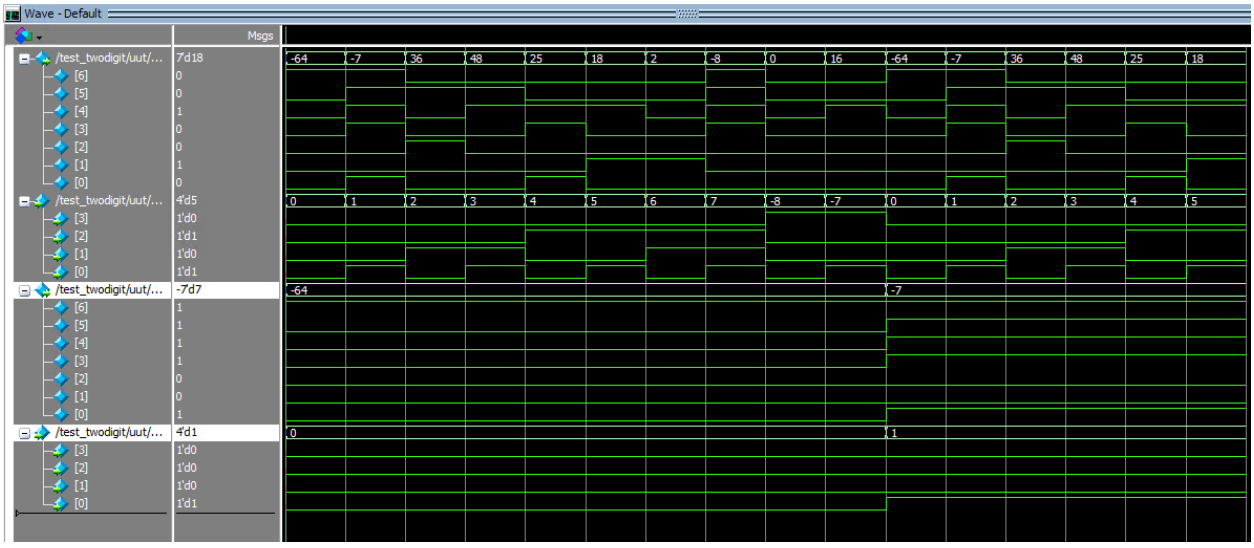
I then simulated my code on ModelSim to test if the segment is working. From my testing I found that my segment was working correctly.



I then implemented a module that runs two seven segment displays and outputs the values in decimal. I also wrote a test bench that would check the output was working properly.

```
43  //Divide the input by 10
44  module twodigit(output logic [6:0] disp0,
45                  output logic [6:0] disp1,
46                   input logic [3:0] switch);
47  reg [3:0] tens;
48  reg [3:0] units;
49  sevenseg a0 (.data(disp0), .address(units));
50  sevenseg a1 (.data(disp1), .address(tens));
51
52  always@ (switch)
53    begin
54      units = switch % 10;
55      tens = switch / 10;
56    end
57  endmodule
```

```
1   module test_twodigit;
2
3   logic [6:0] disp0;
4   logic [6:0] disp1;
5   logic [3:0] switch;
6   int i;
7
8   twodigit uut (.*);
9
10  initial begin
11    for (i = 0; i < 16; i = i + 1)
12      begin
13        switch = i;
14        #10;
15      end
16  end
17  endmodule
```



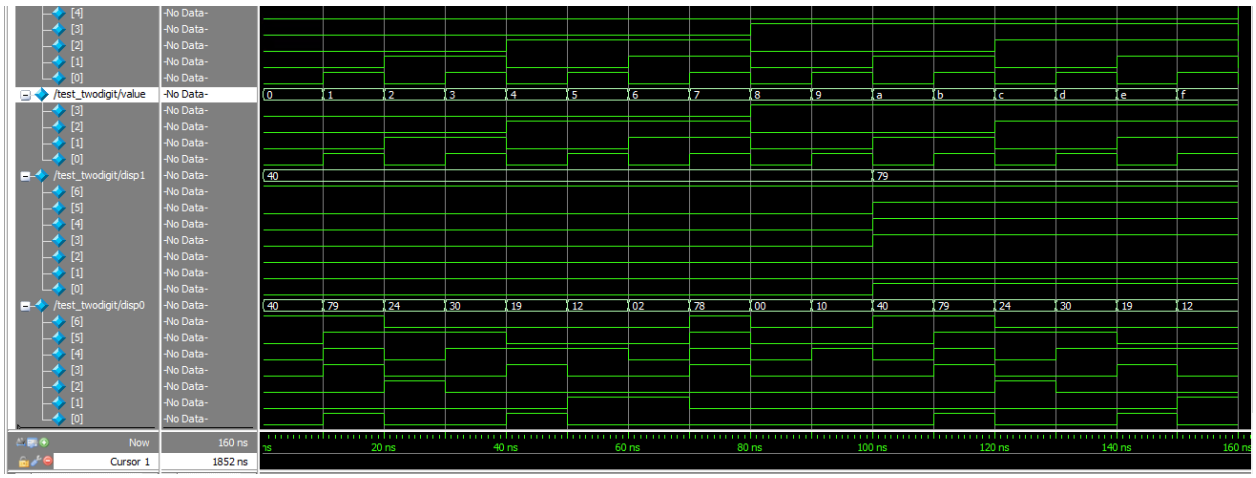
twodigit > twodigit_pins.qsf

```
1   set_location_assignment PIN_AB12 -to address[0]
2   set_location_assignment PIN_AC12 -to address[1]
3   set_location_assignment PIN_AF9  -to address[2]
4   set_location_assignment PIN_AF10 -to address[3]
5   set_location_assignment PIN_AE26 -to data0[0]
6   set_location_assignment PIN_AE27 -to data0[1]
7   set_location_assignment PIN_AE28 -to data0[2]
8   set_location_assignment PIN_AG27 -to data0[3]
9   set_location_assignment PIN_AF28 -to data0[4]
10  set_location_assignment PIN_AG28 -to data0[5]
11  set_location_assignment PIN_AH28 -to data0[6]
12  set_location_assignment PIN_AJ29 -to data1[0]
13  set_location_assignment PIN_AH29 -to data1[1]
14  set_location_assignment PIN_AH30 -to data1[2]
15  set_location_assignment PIN_AG30 -to data1[3]
16  set_location_assignment PIN_AF29 -to data1[4]
17  set_location_assignment PIN_AF30 -to data1[5]
18  set_location_assignment PIN_AD27 -to data1[6]
```

Finally I implemented a module that works as a binary counter and outputs its value to the seven segment displays. I also wrote a test bench that would check the counter and output were working properly.

```
22  //Divide the input by 10
23  module twodigit(output logic [6:0] disp0,
24                  output logic [6:0] disp1,
25                   input logic [3:0] value);
26  reg [3:0] tens;
27  reg [3:0] units;
28  sevenseg a0 (.data(disp0), .address(units));
29  sevenseg a1 (.data(disp1), .address(tens));
30
31  always@ (value)
32    begin
33      units = value % 10;
34      tens = value / 10;
35    end
36  endmodule
37
38  // simple counter for X5
39  module counter(input logic clk, reset,
40                 output logic [3:0] value);
41
42  always_ff @(negedge clk, negedge reset)
43    if (~reset)
44      value <= 0;
45    else
46      value <= value + 1;
47  endmodule
48
```

```
1   module test_counter;
2
3   logic clk, reset;
4   int i;
5
6   counter uut (.clk(clk), .reset(reset));
7
8   initial begin
9     for (i = 0; i < 16; i = i + 1)
10      begin
11        clk = 1;
12        #5;
13        clk = 0;
14        #5;
15      end
16    reset = 0;
17  end
18  endmodule
19
20  module test_twodigit;
21
22  logic [6:0] disp0;
23  logic [6:0] disp1;
24  logic [3:0] value;
25  int i;
26
27  twodigit uut (.*);
28
29  initial begin
30    for (i = 0; i < 16; i = i + 1)
31      begin
32        value = i;
33        #10;
34      end
35  end
36  endmodule
```

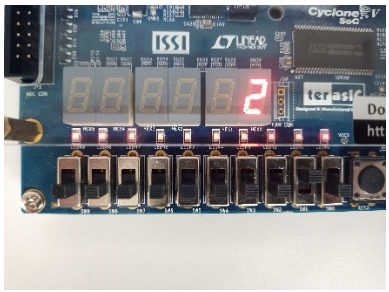
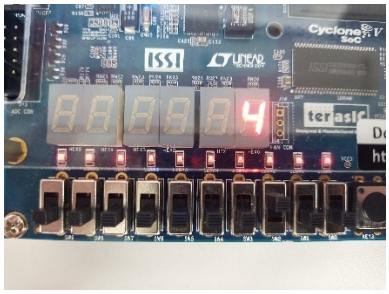
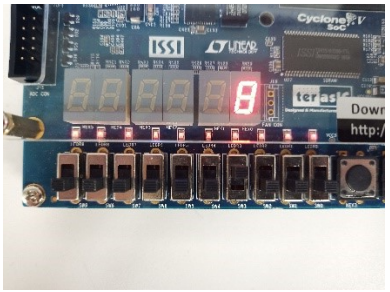
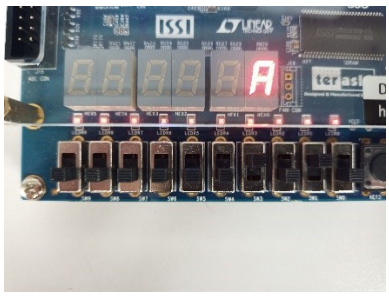
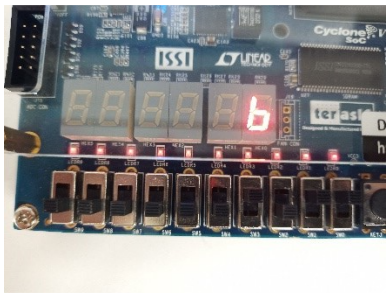
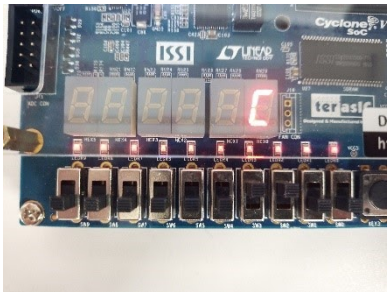
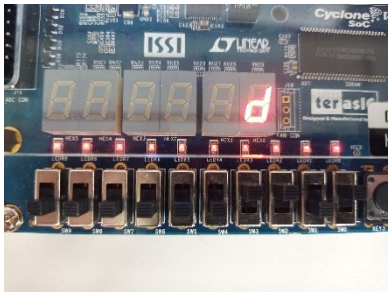
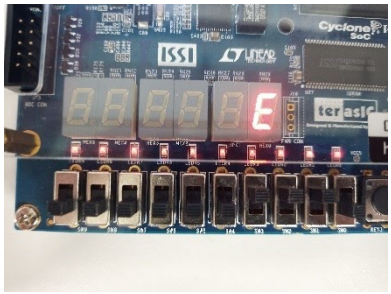
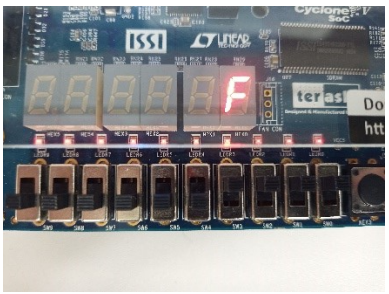


```
set_location_assignment PIN_AA14 -to reset
set_location_assignment PIN_AA15 -to clk
set_location_assignment PIN_AE26 -to data0[0]
set_location_assignment PIN_AE27 -to data0[1]
set_location_assignment PIN_AE28 -to data0[2]
set_location_assignment PIN_AG27 -to data0[3]
set_location_assignment PIN_AF28 -to data0[4]
set_location_assignment PIN_AG28 -to data0[5]
set_location_assignment PIN_AH28 -to data0[6]
set_location_assignment PIN_AJ29 -to data1[0]
set_location_assignment PIN_AH29 -to data1[1]
set_location_assignment PIN_AH30 -to data1[2]
set_location_assignment PIN_AG30 -to data1[3]
set_location_assignment PIN_AF29 -to data1[4]
set_location_assignment PIN_AF30 -to data1[5]
set_location_assignment PIN_AD27 -to data1[6]
```

sevenseg: a 7-segment display decoder - 24/11/20

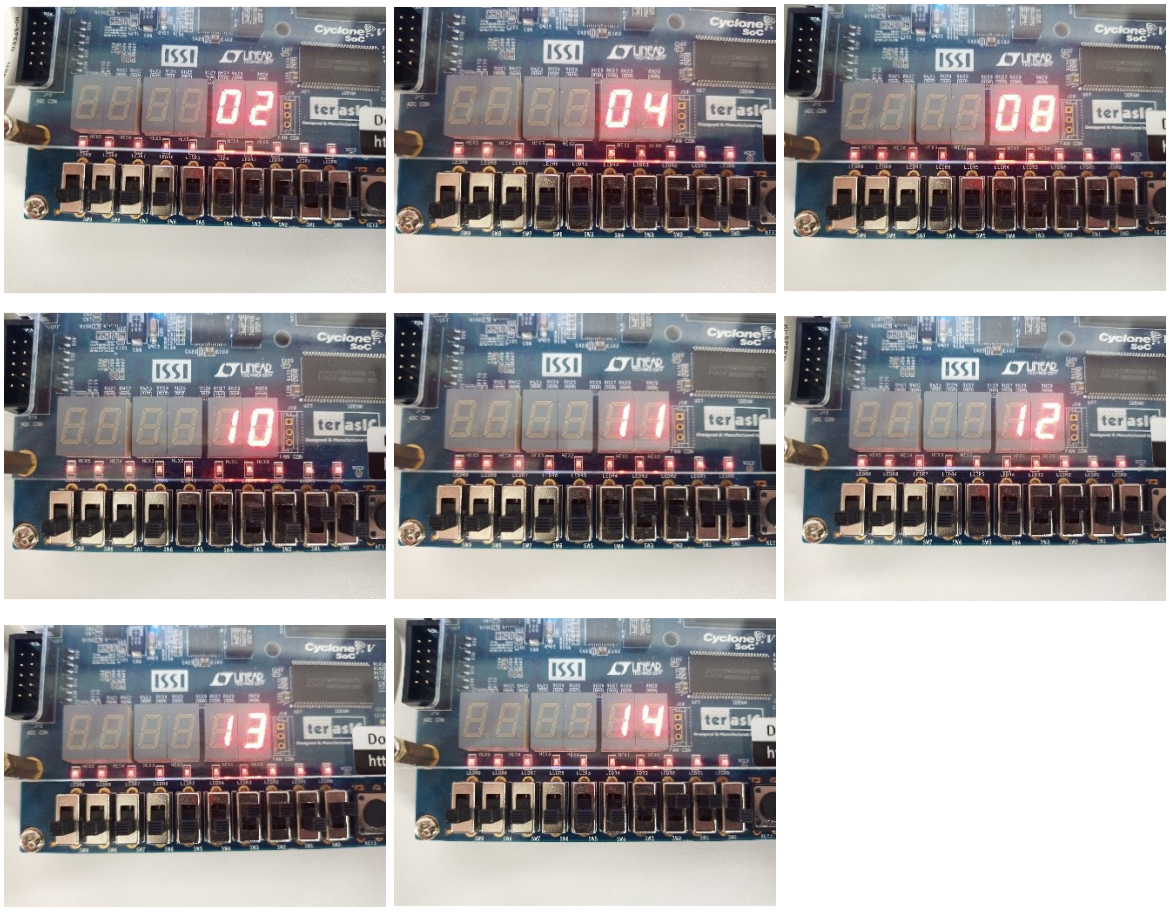
The FPGA correctly displayed values above the value of 9.

```
always_comb
  unique casez (address)
    4'b0000 : data = 7'b1000000;
    4'b0001 : data = 7'b1111001;
    4'b0010 : data = 7'b0100100;
    4'b0011 : data = 7'b0110000;
    4'b0100 : data = 7'b0011001;
    4'b0101 : data = 7'b0010010;
    4'b0110 : data = 7'b0000010;
    4'b0111 : data = 7'b1111000;
    4'b1000 : data = 7'b0000000;
    4'b1001 : data = 7'b0010000;
    4'b1010 : data = 7'b0001000;
    4'b1011 : data = 7'b0000011;
    4'b1100 : data = 7'b1000110;
    4'b1101 : data = 7'b0100001;
    4'b1110 : data = 7'b0000110;
    4'b1111 : data = 7'b0001110;
    default : data = 7'b1111111;
  endcase
endmodule
```

twodigit: two 7-segment displays - 24/11/20

The two digit program also correctly works.



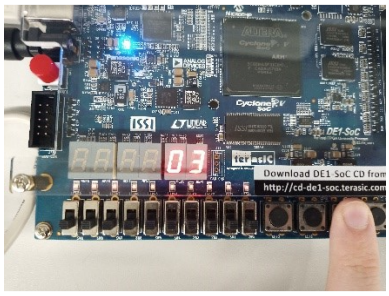
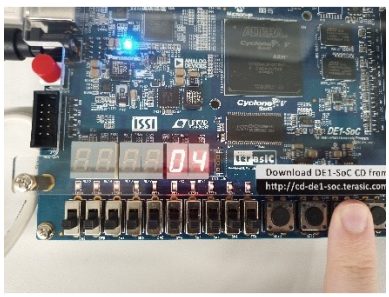
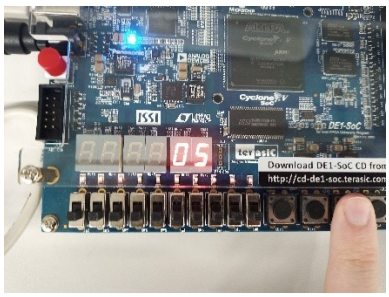
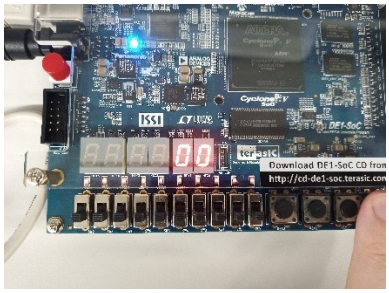
“disp1[1,2,6]” are said to be stuck to GND and VCC because these pins are always off or always on.

```
13024 Output pins are stuck at VCC or GND
13410 Pin "disp1[1]" is stuck at GND
13410 Pin "disp1[2]" is stuck at GND
13410 Pin "disp1[6]" is stuck at VCC
```

counter: a 4-bit binary counter - 24/11/20

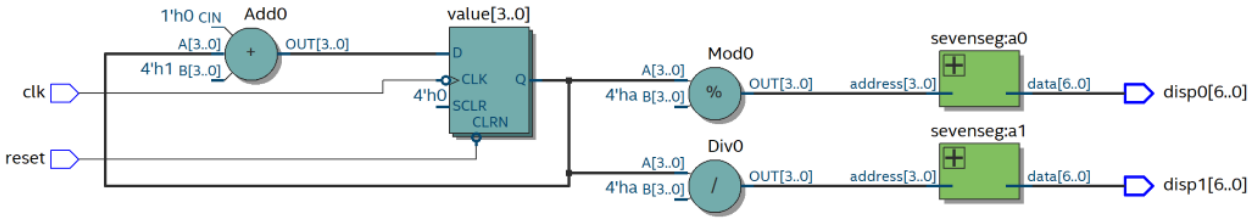
I then implemented the counter so that it would count to fifteen on the clock cycle and reset when the reset key was pressed. I had issues with the modules from my preparation not linking correctly and so I combined the counter and display arithmetic's together into one module.

```
22  // counter implementation
23  module counter2(input logic clk, reset,
24                  output logic [6:0] disp0,
25                  output logic [6:0] disp1);
26
27  reg [3:0] value;
28  reg [3:0] tens;
29  reg [3:0] units;
30
31  always_ff @(negedge clk, negedge reset)
32    if (~reset)
33      value <= 0;
34    else
35      value <= value + 1;
36
37  sevenseg a0 (.data(disp0), .address(units));
38  sevenseg a1 (.data(disp1), .address(tens));
39
40  always@ (value)
41    begin
42      units = value % 10;
43      tens = value / 10;
44    end
45  endmodule
46
```

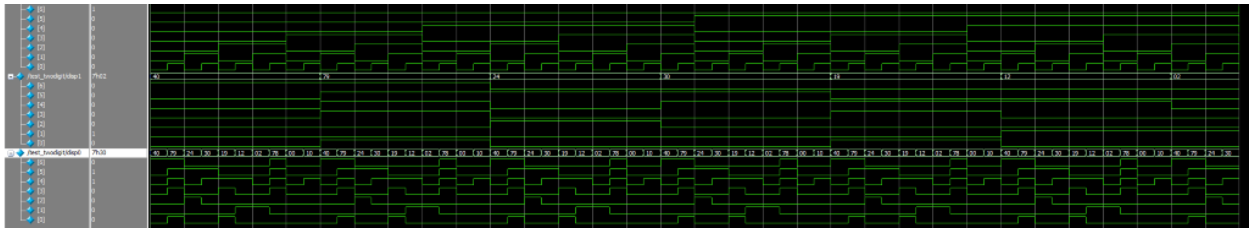
Timing Models	Final
Logic utilization (in ALMs)	11 / 32,070 (< 1 %)
Total registers	8
Total pins	16 / 457 (4 %)
Total virtual pins	0

Looking at the compilation report I found that very little of the FPGA was used with the logic elements being used as less than 1% of the total.
Using the path:
Tools -> Netlist Viewers -> RTL Viewer
I made Quartus create the circuit diagram of a conventional implementation of the device.

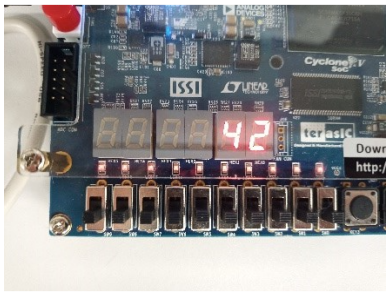


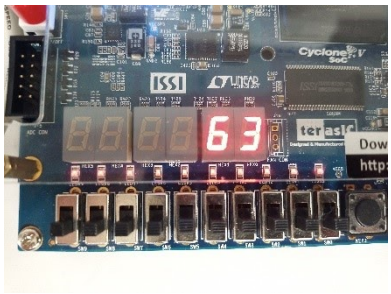
counter: a 6-bit binary counter and display - 24/11/20

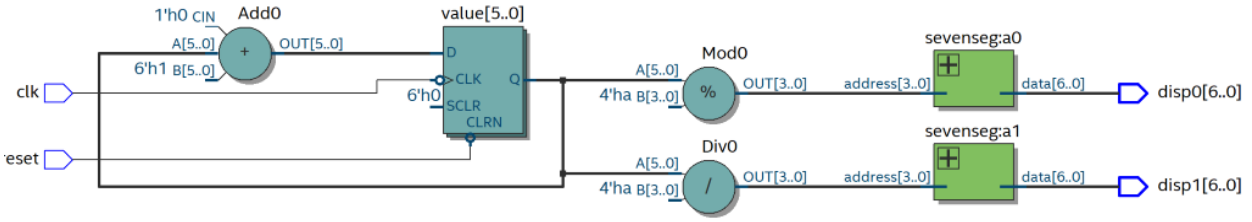
I then modified my code by increasing the size of the array value[3:0] to value[5:0] allowing it to count up to six bits. First I tested it in ModelSim, creating a testbench to do so.



Then I used Quartus to put it on the FPGA and count up to 63. I also made it create the circuit diagram for the counter.







T4: Bus Operation and Control

Understanding Tri-states - 06/12/20

The PlayBus model uses the 7- segment displays on the FPGA board to show the hexadecimal value on the data bus. This display shows ‘H’ (for ‘High Impedance’) if the bus is not driven. If two or more data sources are active simultaneously, the display shows ‘U’ (for ‘Undefined’). ‘H’ is the ‘Z’ value in SystemVerilog and ‘U’ is the ‘X’ value. It is not possible to show ‘X’ and ‘Z’ on a 7-segment display, so alternative letters are used.

A buffer takes the input and puts it on the output, like a not not gate. This can be useful in only letting data travel in one direction

Input A	Enable B	Output C
X	0	Z
0	1	0
1	1	1

```
module tristate (output logic C, input logic A, B)
always_comb

if (B == 1)
  C = A;
else
  C = Z;

endmodule
```

Enabled Registers - 06/12/20

A register is needed to hold the value and to provide enough current to drive the LED as the bus has minimal current.The input D after the active clock edge due to the propagation delay of the d type flip-flop. The enable input is asserted before the active clock edge so that an input can be detected.

```
module dffe (output logic Q, input logic D, enable, clock)
always_ff @(clock)

    if (enable)
        q <= d;
endmodule
```

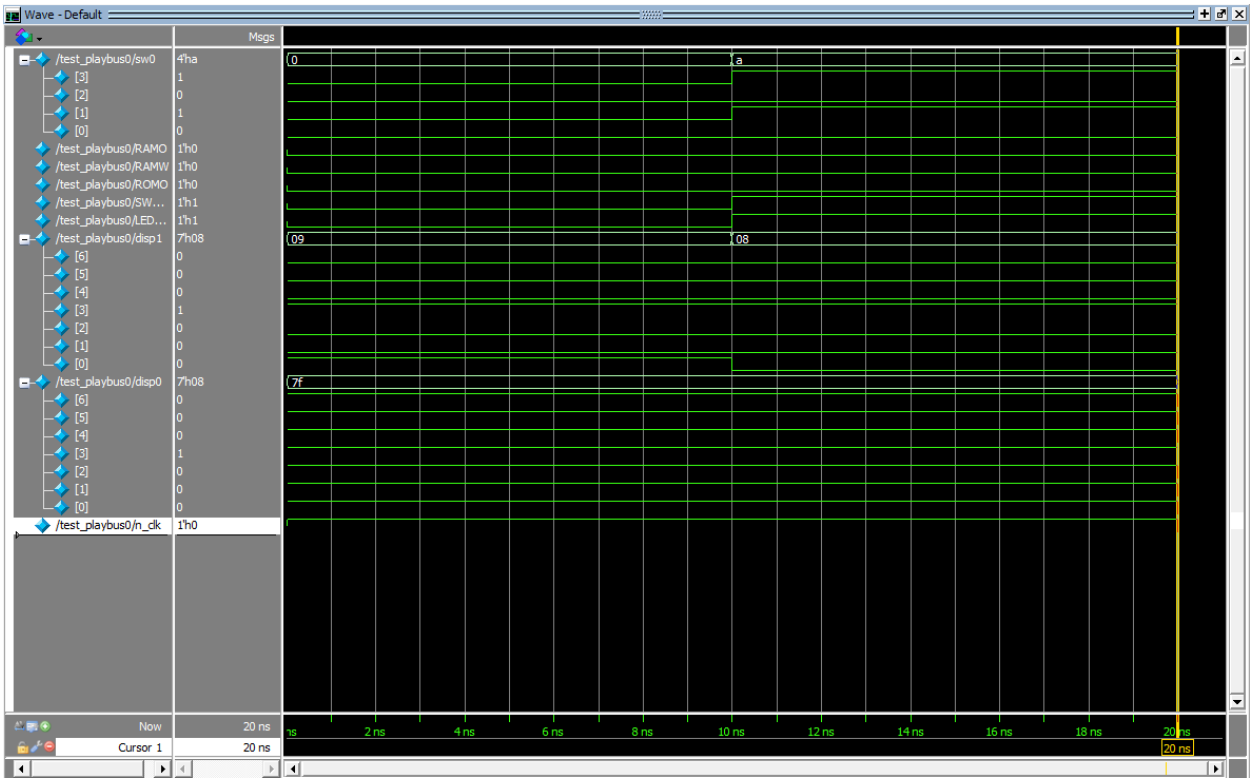
Control Signals - 06/12/20

EPROM		SRAM		Buffer		Register	
OE	Output Enable	OE	Output Enable	OE	Output Enable	OE	Output Enable
CS	Component Select	CS	Component Select	CS	Component Select	-	-
-	-	WE	Write Enable	-	-	-	-

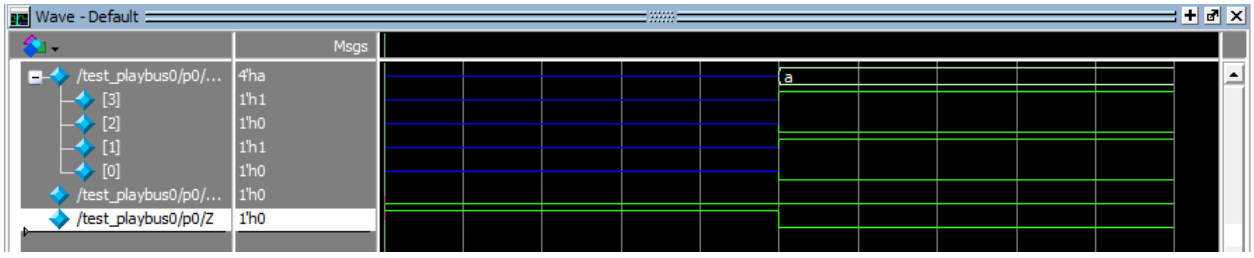
Data Source	To read from source		Data Sink	To read from sink	
SRAM	RAMO	1	SRAM	RAMO	0
	RAMW	0		RAMW	1
	ROMO	0		ROMO	0
	SWBEN	0		SWBEN	0
	LEDLTCH	0		LEDLTCH	0
Eprom	RAMO	0	Registers (LEDs)	RAMO	0
	RAMW	0		RAMW	0
	ROMO	1		ROMO	0
	SWBEN	0		SWBEN	0
	LEDLTCH	0		LEDLTCH	1
Buffer (switches)	RAMO	0			
	RAMW	0			
	ROMO	0			
	SWBEN	1			
	LEDLTCH	0			

Simulation - 06/12/20

Initially I simulated test_playbus0 with all the surface level signals. This told me very little about what was happening, as all the changes that occur can be inferred from the code of test_playbus0.



Then expanding p0 in the test section I was able to test the data bus, the contend signal and the 'z' signal. I found that when the data line is initially in a floating state when 'z' is set high and then when 'z' is set low there is data on the bus,



Controller Operation - 06/12/20

Operation	Signal	Pin Assignment	Switch
Read data from EPROM at address 5 onto bus	ROMO	AD12	5
Read data from RAM at address 5 onto bus	RAMO	AD11	4
Read data from Switches onto bus	SWBEN	AE11	6
Copy data from Switches into RAM at address 5	SWBEN, RAMW	AE11, AC9	6 and 7
Copy data from EPROM into RAM at address 5	ROMO, RAMW	AD12, AC9	5 and 7
Copy data from Switches into LEDs	SWBEN, LEDLTCH	AE11, PD10	6 and 8
Copy data from EPROM at address 5 into LEDs	ROMO, LEDLTCH	AD12, AD10	5 and 8
Copy data from RAM at address 5 into LEDs	RAMO, LEDLTCH	AD11, AD10	4 and 8

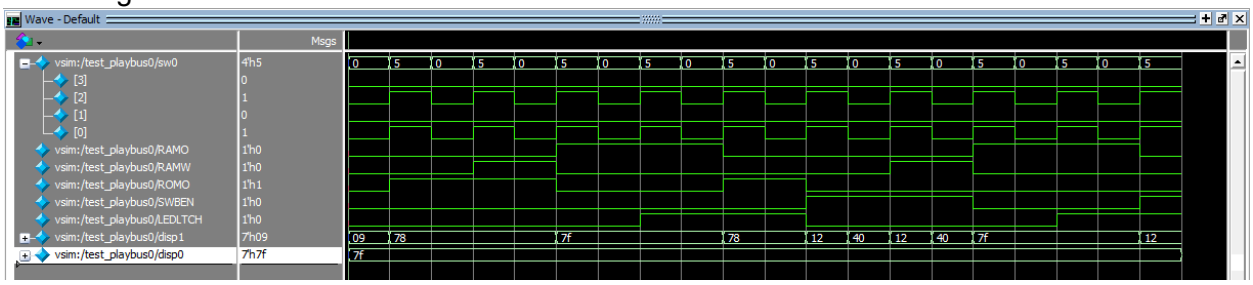
I then edited the testbench to follow the operations in this order:

- Read data from EPROM at address 5 onto bus
- Copy data from EPROM into RAM at address 5
- Read data from RAM at address 5 onto bus
- Copy data from RAM at address 5 into LEDs
- Copy data from EPROM at address 5 into LEDs
- Read data from Switches onto bus
- Copy data from Switches into RAM at address 5
- Read data from RAM at address 5 onto bus
- Copy data from RAM at address 5 into LEDs
- Copy data from Switches into LEDs

Each operation was written similar to this:

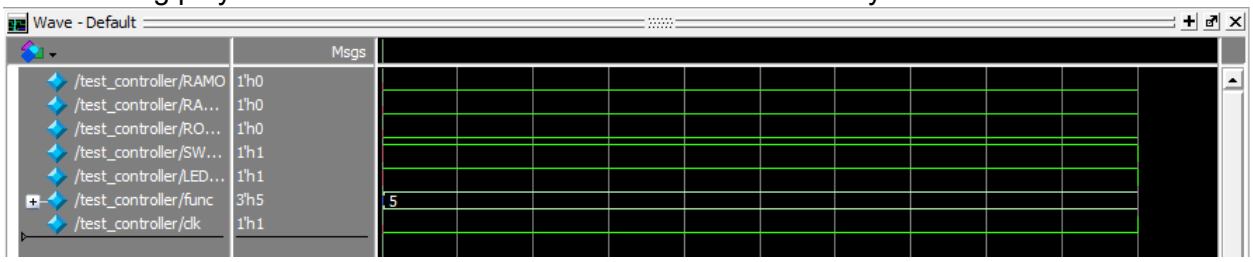
```
//Copy data from RAM at address 5 into LEDs
    n_clk = 1;
    sw0 = 4'b0000;
    #10ns sw0 = 4'b0101;
    RAMO = 1;
    RAMW = 0;
    ROMO = 0;
    SWBEN = 0;
    LEDLTCH = 1;
    #10ns n_clk = 0;
```

Using modelSim to verify that the functions operated correctly. I tested that each of the pins that should be on were on and the pins that should be off were off. I also looked at the output of seven segment to make sure the results were coherent



- Modelsim displays logical X as a red line and logical Z as a blue line.
- The bus will be in state Z when switches are 0 or not 1.
- When more than one switch goes 1 the bus will be in.
- The contend will prevent 2 sources from writing to the bus at the same time.

When testing playbus2 I used the controller testbench and found my value to be 5.



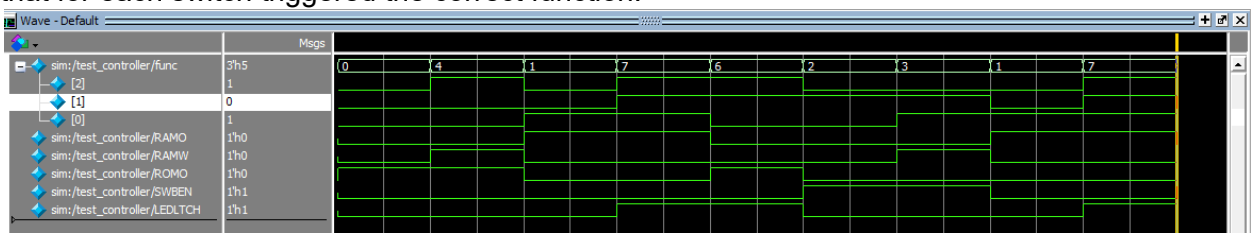
I then modified the controller so that it would implement all eight of the functions.

```
case (present_state)
  start: begin
        if ((func == 1) || (func == 7))
           begin
           RAMO = '1;
           end

        if ((func == 3) || (func == 4))
           begin
          RAMW = '1;
          next_state = xfer;
           end;

        if ((func == 0) || (func == 4) || (func == 6))
           begin
           ROMO = '1;
           end

        if ((func == 2) || (func == 3) || (func == 5))
           begin
           SWBEN = '1;
           end;

        if ((func == 5) || (func == 6) || (func == 7))
           begin
           LEDLTCH = '1;
           next_state = xfer;
           end;
         end
  xfer : next_state = start;
  endcase
```

I also modified the testbench so it would run through each of the functions in the same order as the first testbench.

```
initial
  begin
  clk = 0;
  func = 0;
  #10ns clk = 1;
  clk = 0;
  func = 4;
  #10ns clk = 1;
  clk = 0;
  func = 1;
  #10ns clk = 1;
  clk = 0;
  func = 7;
  #10ns clk = 1;
  clk = 0;
  func = 6;
  #10ns clk = 1;
  clk = 0;
  func = 2;
  #10ns clk = 1;
  clk = 0;
  func = 3;
  #10ns clk = 1;
  clk = 0;
  func = 1;
  #10ns clk = 1;
  clk = 0;
  func = 7;
  #10ns clk = 1;
  clk = 0;
  func = 5;
  end

endmodule
```

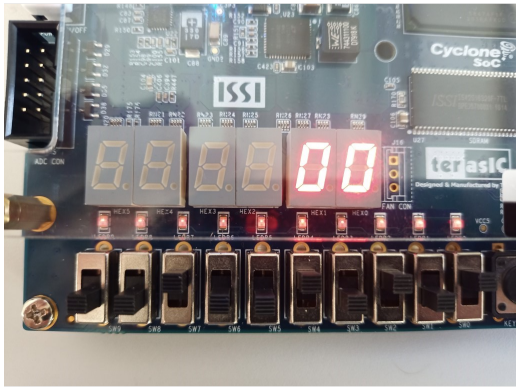
Testing in modelSim I looked at the RAMO, RAMW, ROMO, SWBEN, and LEDLTCH and found that for each switch triggered the correct function.

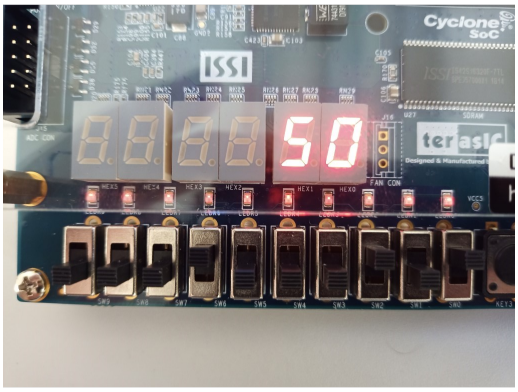


Bus Operation - 08/12/20

I then edited the testbench to follow the operations in this order:
Read data from EPROM at address 5 onto bus (5)
Copy data from EPROM into RAM at address 5 (5,7)
Read data from RAM at address 5 onto bus (4)
Copy data from RAM at address 5 into LEDs (4,8)
Copy data from EPROM at address 5 into LEDs (5,8)
Read data from Switches onto bus (6)
Copy data from Switches into RAM at address 5 (6,7)
Read data from RAM at address 5 onto bus (4)
Copy data from RAM at address 5 into LEDs (4,8)
Copy data from Switches into LEDs (6,8)

Using the same sequence as in my prep I verified that the FPGA worked as it gave the same output on the segment as the modelSim simulation.





The FPGA system determines the data bus is in the ‘Z’ state when the value on the data bus is between the voltage thresholds for 0 and 1.
It is possible to design logic that would detect ‘Z’ or ‘X’ states and show them on a 7-segment display, as the playbus0 code does this. Hence, these states are “real”.

Controller Operation - 08/12/20

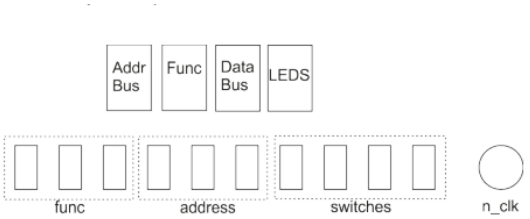


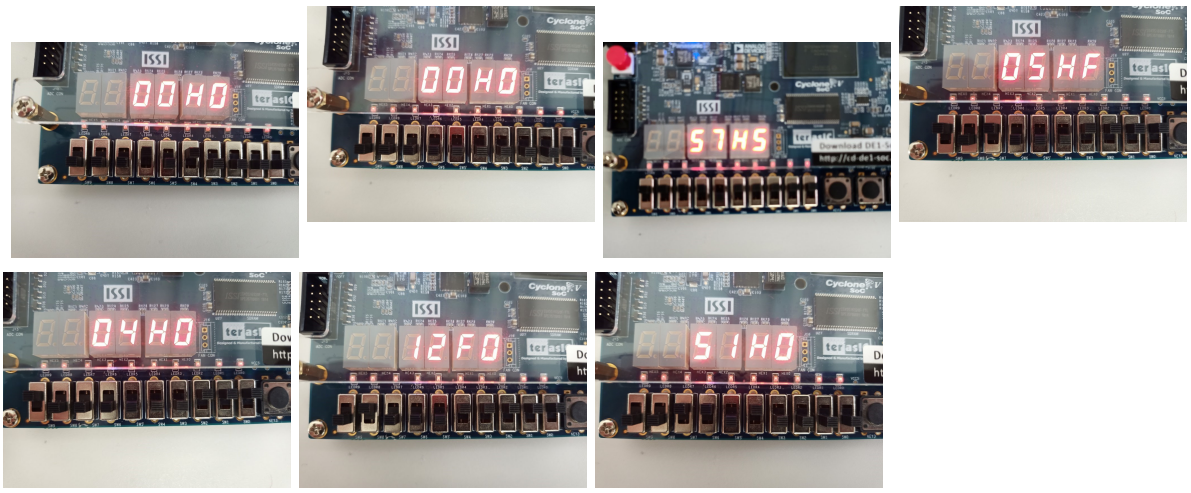
FIGURE 4: LED and switch functions for PlayBus1

Number	Operation
0	Read data from EPROM at address onto bus
1	Read data from RAM at address onto bus
2	Read data from Switches onto bus
3	Copy data from Switches into RAM at address
4	Copy data from EPROM into RAM at address
5	Copy data from Switches into LEDs
6	Copy data from EPROM at address into LEDs
7	Copy data from RAM at address into LEDs

TABLE 4 – PlayBus1 Functions

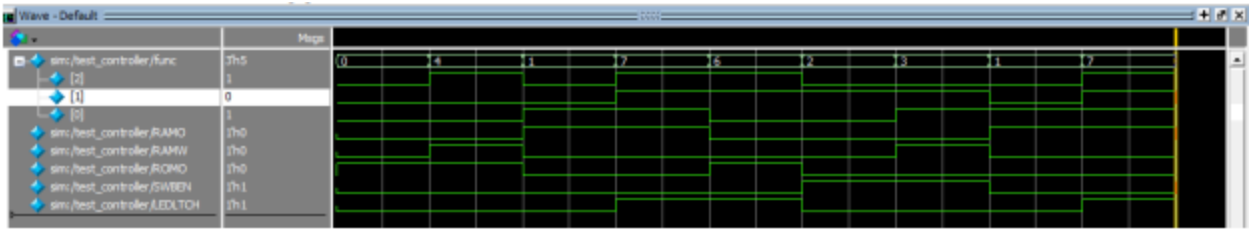
For playbus0 with the initial controller I found that functions 2 and 5 operated correctly.

Implementing one new function after each test I found that for the most part my code worked. However, for functions using the RAMW operation I had to modify the code to include the command:
next_state = xfer;
Without this it meant that the RAM was not correctly clocking the data in.



It is a good idea to simulate code beforehand as if it does not work you can easily look at the internal operation, something that isn't true for downloading it to the FPGA. It also reduces the compilation time which allows you to fix issues faster.
For each of the functions, the observed behaviour on the FPGA were consistent with the ModelSim simulations.

Development of a full-function controller - 08/12/20



For the EPROM and RAM functions I can read and write from different 0 to 7. For the switches and LEDs there is only one register and so only one address to write to.

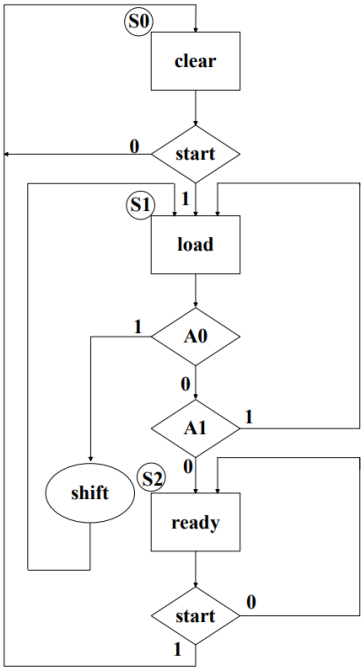
State Machines 2 - 19/01/21

As preparation for my exam I wrote a template for SystemVerilog state machine, this should cover most generic three state ASM charts.

```
//SystemVerilog State Machine template

module template (output logic output1, output2, output3
                 input logic clock, n_reset, input1, input2);

enum {state1, state2, state3} present_state, next_state;

always_ff @(posedge clock, negedge n_reset)
    begin: SEQ                              //Sequential label for modelsim
        if (!n_reset)                       //Reset
            present_state <= state1;
        else                                //Update state on clockedge
            present_state <= next_state;
    end

always_comb
    begin: COM                              //Combinational label for modelsim

        //Set default values of output
        output1 = '0
        output2 = '0
        output3 = '0
```

```
        unique case (present_state)         //Unique case label gives compiler
error if a state is missing
            state1 : begin
                output1 = '1;               //Assert unconditional output
                if (input1)                 //Conditional to progress to next
state
                    next_state = state2;
                else                        //Else stay in ready state
                    next_state = state1;
            end
            state2 : begin
                output2 = '1;               //Assert unconditional output
                if (input2)                 //Conditional to progress to next
state
                    next_state = state3;
                else
                    begin
                        output1 = '1;       //Assert conditional output
                        next_state = state1;//Else go to ready state
                    end
            end
            state3 : begin
                output3 = '1;               //Assert unconditional output
                next_state = state1;        //Progress to next state
            end
        endcase
    end
endmodule
```

Exam question 10




```
module Figure_3 (output logic clear, load, shift, ready
                 input logic clock, n_reset, start, [1:0] A);

enum {S0, S1, S2} present_state, next_state;

always_ff @(posedge clock, negedge n_reset)
    begin: SEQ                              //Sequential label for modelsim
        if (!n_reset)                       //Reset
            present_state <= S0;
        else                                //Update state on clockedge
            present_state <= next_state;
    end

always_comb
    begin: COM                              //Combinational label for modelsim

        //Set default values of output
        clear = '0
        load = '0
        shift = '0
```

```
        ready = '0
        unique case (present_state)         //Unique case label gives compiler
error if a state is missing
            S0 : begin
                clear = '1;                 //Assert unconditional output
                if (start)                  //Conditional to progress to load
state
                    next_state = S1;
                else                        //Else stay in clear state
                    next_state = S0;
            end
            S1 : begin
                load = '1;                  //Assert load output
                if (A[0])                   //Conditional to assert shift and
stay in load state
                    begin
                        shift = '1;
                        next_state = S1;
                    end
                else if (A[1])              //Conditional to stay in load state
                        next_state = S1;
                else
                        next_state = S2;    //Else go to ready state
            end
            S2 : begin
                ready = '1;                 //Assert ready output
                if (start)                  //Conditional to return to clear
state
                    next_state = S0;
                else                        //Else stay in ready state
                    next_state = S2;
            end
        endcase
    end
endmodule
```

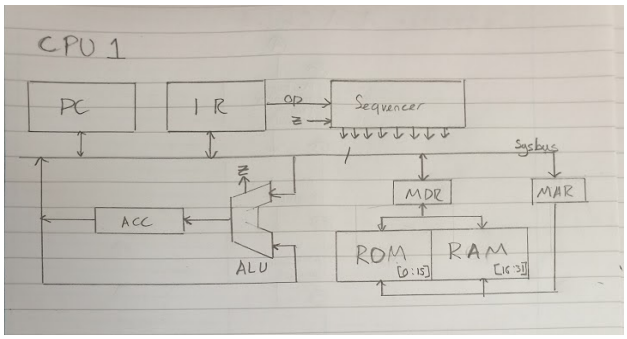
Testbench

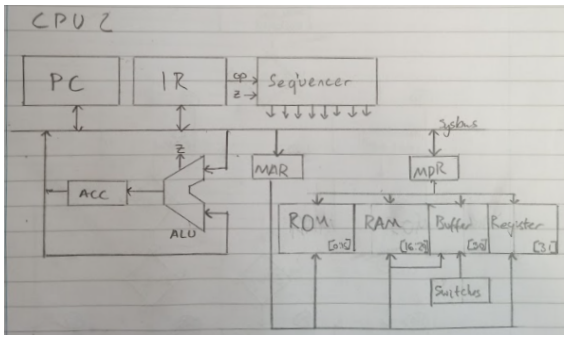
```
module test_Figure_3;   // No i/o

logic clear, load, shift, ready;
logic clock, n_reset, start, [1:0] A;

Figure_3 d0 (.*);       // only works if internal
                        // signals have the same names

initial
    begin
    //loop 1
    #10ns start = 0;
    #10ns start = 1;
    #10ns A = 0;
    #10ns start = 1;

    //loop 2
    #10ns start = 0;
    #10ns start = 1;
    #10ns A = 1;
    #10ns A = 2;
    #10ns A = 1;
    #10ns start = 0;
    #10ns start = 1;
    end

endmodule
```

D2: Digital Systems and Microprocessor Design Exercise
https://stackoverflow.com/questions/16424726/what-is-the-difference-between-verilog-and

Understanding the Microprocessor - 23/01/21

Architecture:



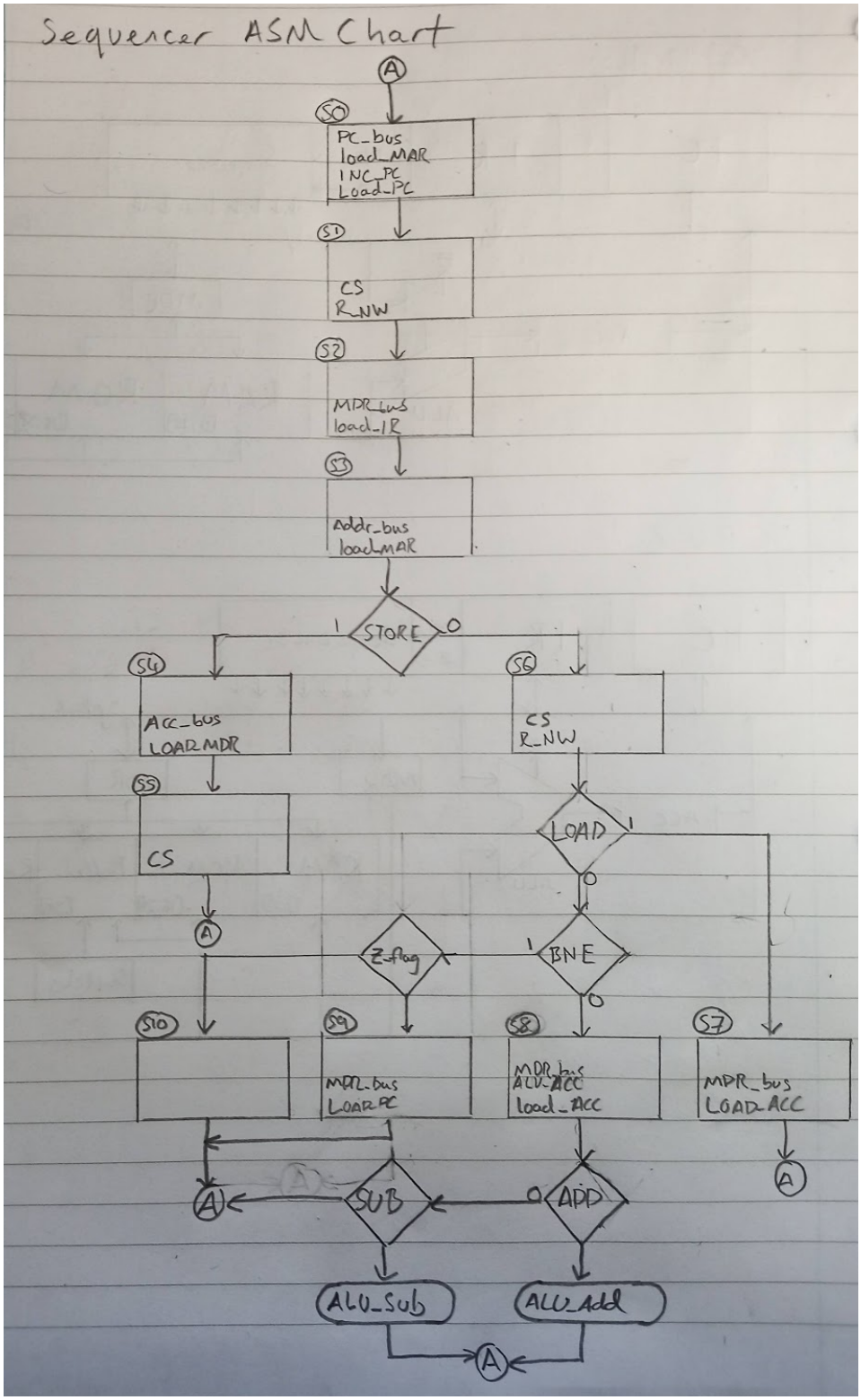


PC:
The program counter is a sequential logic module that records the position of the program through the ROM and outputs the position to the sysbus when the next instruction needs to be loaded.

IR:
The instruction register is a sequential logic module that reads the opcode from the sysbus and passes it to the sequencer.

Sequencer:
The sequencer is a state machine that controls the order of operation for the other modules. This is important to stop any bus contention occurring.

ACC_bus	Drive bus with contents of ACC (enable three-state output)
load_ACC	Load ACC from bus
PC_bus	Drive bus with contents of PC
load_IR	Load IR from bus
load_MAR	Load MAR from bus
MDR_bus	Drive bus with contents of MDR
load_MDR	Load MDR from bus
ALU_ACC	Load ACC with result from ALU
INC_PC	Increment PC and save the result in PC
Addr_bus	Drive bus with operand part of instruction held in IR
CS	Chip select. Use contents of MAR to set up memory address
R_NW	Read, not write. When false, contents of MDR are stored in memory
ALU_add	Perform an add operation in the ALU
ALU_sub	Perform a subtract operation in the ALU


Sequencer ASM Chart
A
S0
PC_bus
load_MAR
INC_PC
Load_PC
S1
CS
R_NW
S2
MDR_bus
load_IR
S3
Addr_bus
load_MAR
STORE
1
0
S4
ACC_bus
LOAD_MDR
S5
CS
R_NW
S6
CS
LOAD
1
0
A
Z_flag
BNE
1
0
S10
S9
MDR_bus
LOAD_PC
S8
MDR_bus
ALU_ACC
load_ACC
S7
MDR_bus
LOAD_ACC
A
A
SUB
0
ADD
ALU_Sub
ALU_Add
A

Basic Operation - 23/01/21

1) The file "rom.sv" contains a program. What does this program do? Where does it store data?

The program in the ROM is stored in the memory addresses 0 to 6. This program uses memory address 16 to store its working, this is the first address in RAM.

2) What happens at address 0 and why? Where does the program loop back to?

At address 0, in the ROM, the program stores the current value in the accumulator (inside the ALU) to address 16, in the RAM.

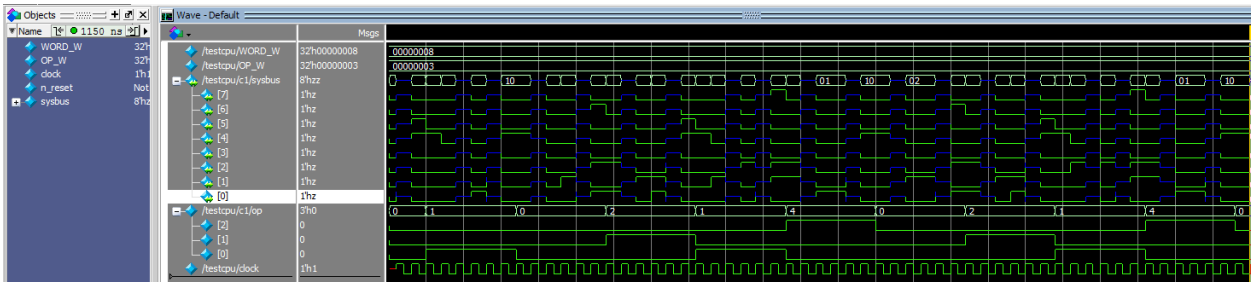
This program will loop back to address one because the BNE opcode sets the mdr to address one and the sequencer to load the program counter from the mdr.

3) The memory address registers in the ram and rom have 5 bits. How are the ram and rom organised to ensure that they do not overlap? How are the addresses decoded?

ROM covers the memory addresses 0 to 15.
RAM covers the memory addresses 16 to 31 (or 16 to 29 for CPU 2).

The three most significant bits of the sysbus is the opcode and so is ignored when looking for the address. The five least significant bits of the sysbus is the mar, which is a five bit address stored in binary.The ROM can only access the sysbus when the value of the most significant bit of the mar is 0.The RAM can only access the sysbus when the value of the most significant bit of the mar is 1. This stops bus contention.

4) How is the high impedance, 'Z', state of sysbus shown in the simulation?



In ModelSim the high impedance state is displayed as a blue line between the two logic levels.

5) SystemVerilog and ModelSim can represent ‘Z’ and ‘X’ states, but are these states “real”? In other words, is it possible to design logic that would detect ‘Z’ or ‘X’ states and display them?

‘Z’ and ‘X’ states are not “real” states as they cannot be verified with only voltage readings, which is all a digital system is capable of doing. However, it is possible to create logic that would identify ‘Z’ or ‘X’ states within itself as it would know at any point in its function when nothing is writing to the bus or if there is bus contention.

Memory Mapped Input and Output - 23/01/21

1) Sketch a diagram of the memory, going from address 0 to address 31, and showing which components are at each address.

Address	Function
[0:15]	ROM
[16:29]	RAM
[30]	Buffer
[31]	Register

2) Add the appropriate lines to cpu2 to instantiate one buffer and one register.

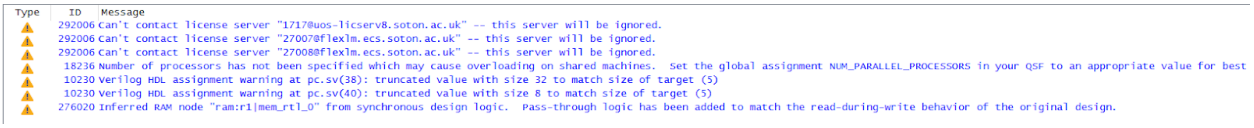
```
//instantiate 1 register and 1 buffer here

buffer #(.WORD_W(WORD_W), .OP_W(OP_W)) b1  (.*);
sequencer #(.WORD_W(WORD_W), .OP_W(OP_W)) s1  (.*);
ir #(.WORD_W(WORD_W), .OP_W(OP_W)) i1  (.*);
pc #(.WORD_W(WORD_W), .OP_W(OP_W)) p1 (.*);
alu #(.WORD_W(WORD_W), .OP_W(OP_W)) a1 (.*);
ram #(.WORD_W(WORD_W), .OP_W(OP_W)) r1 (.*);
rom #(.WORD_W(WORD_W), .OP_W(OP_W)) r2 (.*);
register #(.WORD_W(WORD_W), .OP_W(OP_W)) r3 (.*);
```

3) If the CPU attempts to read from address 30, both the buffer and the RAM will be enabled. Why is this going to be a problem?

This is a problem as both RAM and the buffer will assert their values onto the sysbus as they are both being read. The two values on the same bus will interfere with each other giving a signal that is likely neither of them. This is bus contention.

4) Modify this line so that mdr is not written to the bus when mar contains addresses 30 or 31. The RAM should continue to work correctly for other addresses.

```
//The following line in the ram module copies the contents of mdr to sysbus:
assign sysbus = (MDR_bus & mar[WORD_W-OP_W-1]
                  & ~(mar==30 | mar==31)) ? mdr : 'z;
```



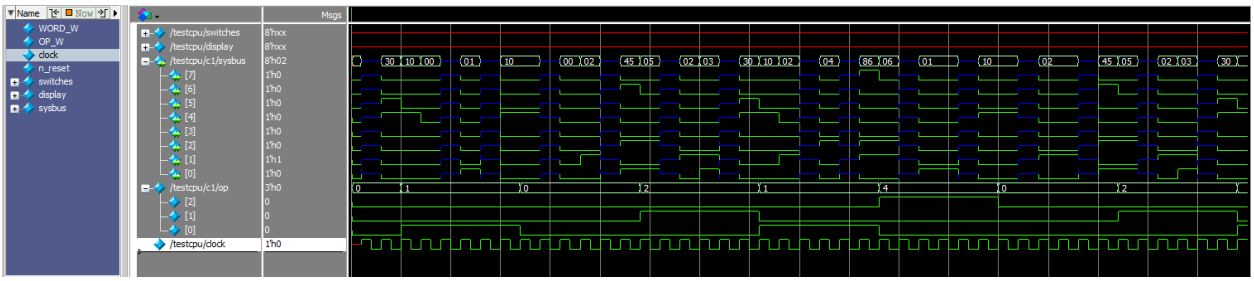
Mark Zwolinski
Sun 24/01/2021 16:11
To: butterworth j.d. (jdb1g20)

```
Logfile for user jdb1g20 attempt 5; ram.sv
Starting ModelSim
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 16:11:40 on Jan 24,2021
vlog -work jdb1g20work -sv ram.sv
-- Compiling module ram

Top level modules:
       ram
End time: 16:11:40 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
# vsim -L jdb1g20work -c -novopt -do "../do_vsim" ../work.testram
# Start time: 16:11:41 on Jan 24,2021
# Loading sv_std.std
# Loading ../work.testram
# Loading jdb1g20work.ram
# do ../do_vsim
# ---------------------------
# Modelsim Simulation
# ---------------------------
#
# RAM address        16 OK
# RAM address        17 OK
# RAM address        18 OK
# RAM address        19 OK
# RAM address        20 OK
# RAM address        21 OK
# RAM address        22 OK
# RAM address        23 OK
# RAM address        24 OK
# RAM address        25 OK
# RAM address        26 OK
# RAM address        27 OK
# RAM address        28 OK
# RAM address        29 OK
# RAM address        30 OK
# RAM address        31 OK
#
# ---------------------------
# End time: 16:11:41 on Jan 24,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

5) Modify “testcpu” so that it uses cpu2 instead of cpu1. You will also need to include switches and display as variables in the testbench.

```
module testcpu;

parameter int WORD_W = 8, OP_W = 3;

logic  clock, n_reset;
logic [WORD_W-1:0] switches;
logic [WORD_W-1:0] display;
wire [WORD_W-1:0] sysbus;

cpu2 #(.WORD_W(WORD_W), .OP_W(OP_W)) c1 (.*);
```

6) What do you see in the Waveform window for display and switches?

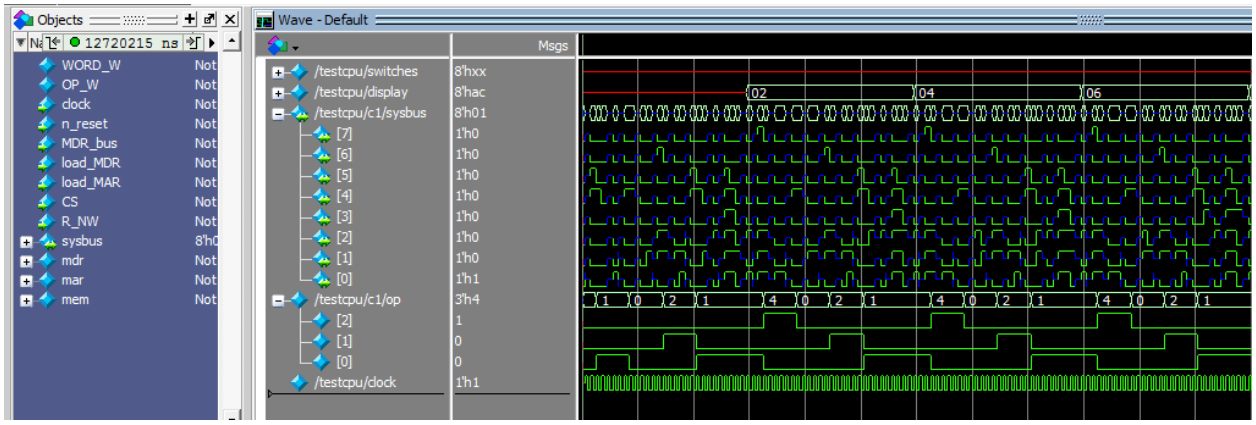


Both the display register and the switches are of unknown value as they are not defined at any point in the testbench or the program.

7) Modify the program in the rom module such that the result in the accumulator is STOREd to address 31 (display) as well as address 16. Note that you will have to move the contents of the ROM at addresses 4 to 6 up by one place and modify the ADD and BNE instructions. What do you now see in the simulation?

```
    0: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    1: mdr = {`LOAD, 5'd16};  //Load the contents of address into the
accumulator
    2: mdr = {`ADD, 5'd6};    //Add the contents of address to the accumulator
    3: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    4: mdr = {`STORE, 5'd31}; //Store the contents of accumulator
    5: mdr = {`BNE, 5'd7};    //Branch if result of last arithmetic operation
is not zero
    6: mdr = 2;               //contents used by another instruction
    7: mdr = 1;               //contents used by another instruction
```

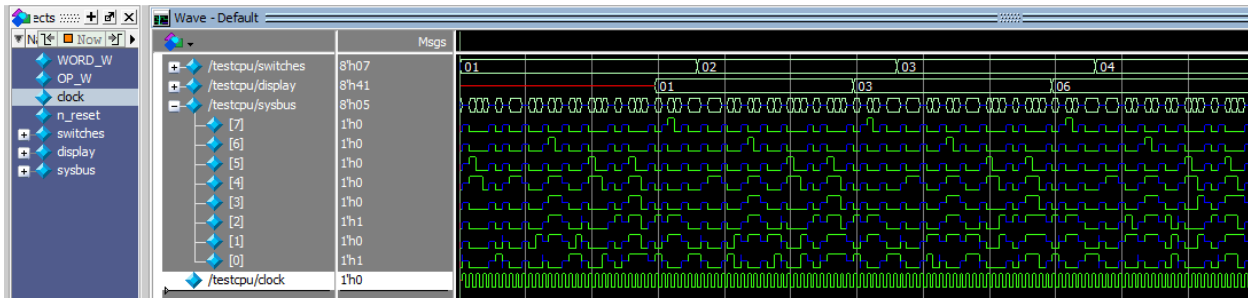
My modified program would output the value to the display register. This clearly shows the program counts up in twos.



8) Make a second change to the program in the rom module such that the data for the ADD operation comes from address 30 – the switches. Modify “testcpu” to apply different values to the switches during the simulation. What do you now see in the simulation?

```
    0: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    1: mdr = {`LOAD, 5'd16};  //Load the contents of address into the
accumulator
    2: mdr = {`ADD, 5'd30};    //Add the contents of address to the accumulator
    3: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    4: mdr = {`STORE, 5'd31}; //Store the contents of accumulator
    5: mdr = {`BNE, 5'd6};    //Branch if result of last arithmetic operation
is not zero
    6: mdr = 1;               //contents used by another instruction
```

My new program would take the values from the switches and add it to the current value, outputting it to the display register.



Extending the Microprocessor - 24/01/21

Creating a new version of the CPU, cpu3, I added the bitwise xor and bitwise not (complement) functions to act on the accumulator. This meant defining the opcodes, adding new enable lines between the ALU and sequencer, adding the functionality to the ALU and Sequencer and writing a new program in ROM to use these commands.

ALU:

```
    if (load_ACC)
      if (ALU_ACC)
        begin
        if (ALU_add)
          acc <= acc + sysbus;
        else if (ALU_sub)
          acc <= acc - sysbus;
        else if (ALU_xor)
          acc <= acc ^ sysbus;
        else if (ALU_comp)
          acc <= ~acc;
        end
      else
        acc <= sysbus;
```

Sequencer:

```
 s8: begin
       MDR_bus = 1'b1;
       ALU_ACC = 1'b1;
       load_ACC = 1'b1;
       if (op == `ADD)
         ALU_add = 1'b1;
       else if (op == `SUB)
         ALU_sub = 1'b1;
       else if (op == `XOR)
         ALU_xor = 1'b1;
       else if (op == `COMP)
         ALU_comp = 1'b1;
     Next_State = s0;
       end
```

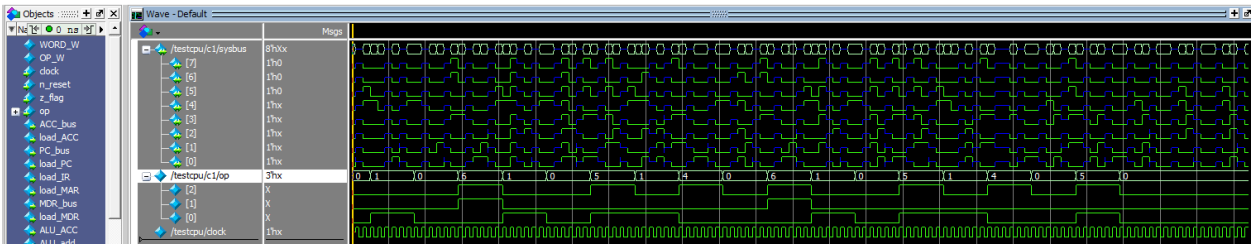
CPU:

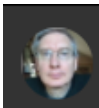
```
module cpu1 #(parameter WORD_W = 8, OP_W = 3)
                  (input logic clock, n_reset,
                   inout wire [WORD_W-1:0] sysbus);

logic ACC_bus, load_ACC, PC_bus, load_PC, load_IR, load_MAR,
MDR_bus, load_MDR, ALU_ACC, ALU_add, ALU_sub, ALU_xor, ALU_comp,
INC_PC, Addr_bus, CS, R_NW, z_flag;
```

ROM:

```
  case (mar)
    0:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    1:  mdr = {`LOAD,   5'd16}; //Load the contents of address into the
accumulator
    2:  mdr = {`COMP,   5'd8};  //Complement the contents of the accumulator
    3:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    4:  mdr = {`LOAD,   5'd16}; //Load the contents of address into the
accumulator
    5:  mdr = {`XOR,    5'd9};  //Xor the contents of address with the
accumulator
    6:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    7:  mdr = {`BNE,    5'd10}; //Branch if result of last arithmetic operation
is not zero
    8:  mdr = 1;                //contents used by another instruction
    9:  mdr = 170;              //contents used by another instruction
    10: mdr = 1;                //contents used by another instruction
    default: mdr = 0;           //rest of ROM is 0
  endcase
```

I then verified that the program was working correctly by simulating it in modelsim and going through each operation. I found that by the end of the first cycle the value in ram was 1'h55, which was to be expected when 1'hFF and 1'hAA are XORed together.





Mark Zwolinski
Sun 24/01/2021 15:02
To: butterworth j.d. (jdb1g20)

```
Logfile for user jdb1g20 attempt 1; alu.sv, sequencer.sv
Starting ModelSim
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 15:02:00 on Jan 24,2021
vlog -work jdb1g20work -sv alu.sv
-- Compiling module alu

Top level modules:
        alu
End time: 15:02:00 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 15:02:00 on Jan 24,2021
vlog -work jdb1g20work -sv sequencer.sv
-- Compiling module sequencer

Top level modules:
        sequencer
End time: 15:02:00 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
# vsim -L jdb1g20work -c -novopt -do "../do_vsim" ../work.testALU_seq
# Start time: 15:02:01 on Jan 24,2021
# Loading sv_std.std
# Loading ../work.testALU_seq
# Loading jdb1g20work.sequencer
# Loading jdb1g20work.alu
# do ../do_vsim
# ---------------------------
# Modelsim Simulation
# ---------------------------
#
# XOR operation works correctly
#
# ---------------------------
# End time: 15:02:01 on Jan 24,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
================================================================
Starting Verilator
================================================================
Starting Quartus
```

Encryption and Decryption - 25/01/21

l	0	1	1	0	0
k	0	1	0	1	1
output	0	0	1	1	1
output	0	0	1	1	1
l	0	1	1	0	0
key	0	1	0	1	1

To decrypt a XOR encrypted message the decryption is to XOR the encrypted message with the key again.

As the key is unknown to us and the number of possible keys (32) is small it is easy to brute force the solution by testing every possible key.

To test the output when the string “oiytmmvk” is decrypted with every possible key I wrote a program in ROM that would load in each value from the switches into a position in the RAM. It would then XOR each of these with a key outputting it to the display. Finally it would increment the key by one and repeat. This program meant I also had to modify the testbench so that it would input the value wanted.

As the length of my program was now longer than even the 32 total addresses in the 8 bit system, I had to increase the system size to a ten bit system. This provided me with 128 total addresses for my program to be stored.

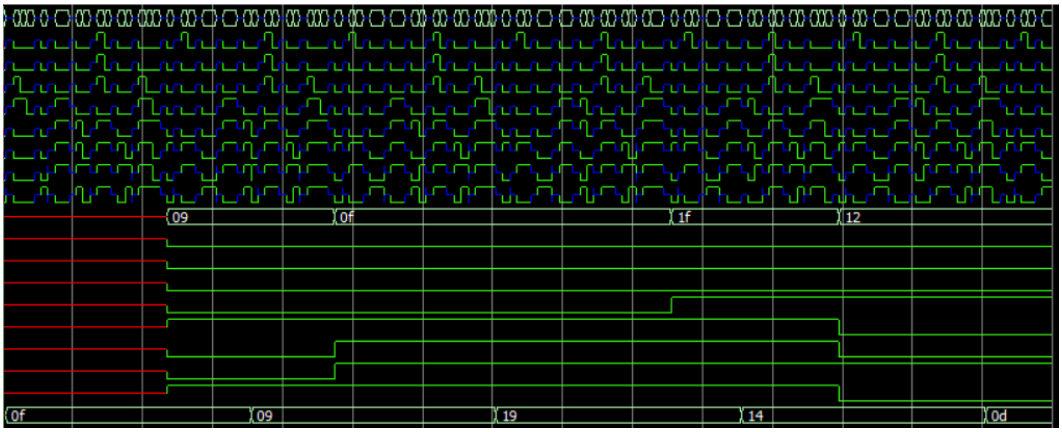
Updating to 10 bit from 8 bit meant that I had increase the word width so that the system could target each address. I did this by changing the parameter in each file. I also moved the addresses of the the tri-state buffer and the display registers, to positions 126 and 127, so that they weren’t taking ROM slots part way through the program.

I found that I did not need the subtract operation or the complement operation (that I implemented). However, as the total number of operations was still five, requiring 3 bits to encode them still, I did not see the value in removing them from my design.

```
  case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator
    16: mdr = {`BNE,    7'd52};

    22: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    23: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    24: mdr = {`STORE,  7'd127};  //Output to display
    25: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    26: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    29: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    30: mdr = {`STORE,  7'd127};  //Output to display
    31: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    32: mdr = {`XOR,    7'd80};   //XOR accumulator with key
```

```
    33: mdr = {`STORE,  7'd127};  //Output to display
    34: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    35: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    36: mdr = {`STORE,  7'd127};  //Output to display
    37: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    38: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    42: mdr = {`STORE,  7'd127};  //Output to display
    43: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    44: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    45: mdr = {`STORE,  7'd127};  //Output to display
    46: mdr = {`LOAD,   7'd80};   //Load key
    47: mdr = {`ADD,    7'd53};   //Add 1
    48: mdr = {`STORE,  7'd80};   //Store key
    49: mdr = {`XOR,    7'd54};
    50: mdr = {`BNE,    7'd55};   //Loop until all keys are tried

    52: mdr = 22;
    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 16;
        /*RAM:
64: encrypted character 1
65: encrypted character 2
66: encrypted character 3
67: encrypted character 4
68: encrypted character 5
69: encrypted character 6
70: encrypted character 7
71: encrypted character 8
80: key
*/
    default: mdr = 0;             //rest of ROM is 0
  endcase
  end
```

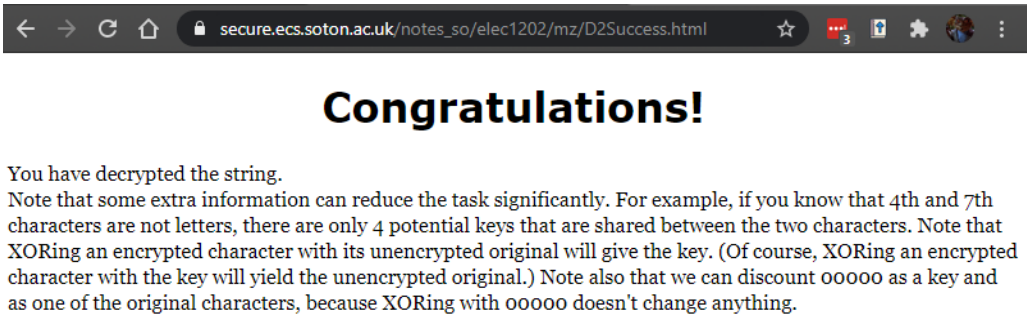
This is the start of the program in modelSIM as the value on the switches is loaded in.



Brute forcing through all 32 combinations gives the solutions:

0	OIYTMMVK	11	DBR FF]@	22	Y OB[[@]
1	NHXULLWJ	12	CEUXAAZG	23	X.NCZZA\
2	MK[VOOTI	13	BDTY@@[F	24	WQALUUNS
3	LJZWNNUH	14	AGWZCCXE	25	VP@MTTOR
4	KM]PIIRO	15	@FV[BBYD	26	USCNWWLQ
5	JL/QHHSN	16	YID]]F[	27	TRBOVVMP
6	IO RKKPM	17	.XHE//GZ	28	SUEHQQJW
7	HN.SJJQL	18	][KF DY	29	RTDIPPKV
8	GAQ/EE.C	19	/ZJG..EX	30	QWGJSSHU
9	F@P]DD B	20	[]M@YYB	31	PVFKRRIT
10	ECS.GG/A	21	Z/LAXXC.		

The tenth key is a link to the ecs intranet and provides the webpage below:



Encryption and Decryption with loops - 25/01/21

Before my invigilator made it clear that a brute force attack was sufficient. I began to try and make a program that could operate loops in such a way that it could change the target address.

It would do this by loading a “function” into memory it could then update the address by overwriting later lines with commands dependent on the outcome from the operation.

This method required the entire program to be written in ROM and the space for the largest function and the auto loader to be in the RAM. This meant that this design still required more address than the eight bits would allow.

I abandoned this method upon understanding that brute force was sufficient. By this point, I had written in pseudo code an auto-loader to load the program. A function to load the inputs and the brute force method with a loop to target different addresses.

```
//auto Loader
90: mdr = {`LOAD,   (`LOAD position_rom_start)};
91: mdr = {`ADD,    load_position};
92: mdr = {`STORE,  96}; //update next line
93: mdr = {`LOAD,   (`STORE position_ram_start)};
94: mdr = {`ADD,    position_load};
95: mdr = {`STORE,  97}; //update next line
96: mdr = {`LOAD,   program_in};
97: mdr = {`STORE,  program_out};
98: mdr = {`BNE,    90};

//input
100: mdr = {`LOAD,   switches};
101: mdr = {`STORE,  ACC_TEMP};
102: mdr = {`LOAD,   (`STORE position_character_start)};
103: mdr = {`ADD,    character_position};
104: mdr = {`STORE,   106};
105: mdr = {`LOAD,    ACC_TEMP};
106: mdr = {`STORE,  position};
107: mdr = {`BNE,   100};
```

```
//encryption-decryption
100: mdr = {`LOAD,   (`LOAD character_in_start)};
101: mdr = {`ADD,    position};
102: mdr = {`STORE,  106}; //update next line
103: mdr = {`LOAD,   (`STORE character_out_start)};
104: mdr = {`ADD,    position};
105: mdr = {`STORE,  108}; //update next line
106: mdr = {`LOAD,   character_in};
107: mdr = {`XOR,    key};
108: mdr = {`STORE,  character_out};
109: mdr = {`STORE,  Display};
110: mdr = {`LOAD,   position};
111: mdr = {`ADD,    1};
112: mdr = {`STORE,  position};
113: mdr = {`XOR,    length};
114: mdr = {`BNE,    100};
115: mdr = {`LOAD,   key};
116: mdr = {`ADD,    1};
117: mdr = {`STORE,  key};
118: mdr = {`XOR,    possible_keys};
119: mdr = {`BNE,    100};
```

XOR Then XNOR Encryption - 26/01/21

I changed the complement into a XNOR function.

```
always_ff @(posedge clock, negedge n_reset)
  begin
  if (!n_reset)
    acc <= 0;
  else
    if (load_ACC)
      if (ALU_ACC)
        begin
        if (ALU_add)
          acc <= acc + sysbus;
        else if (ALU_sub)
          acc <= acc - sysbus;
        else if (ALU_xor)
          acc <= acc ^ sysbus;
        else if (ALU_xnor)
          acc <= ~(acc ^ sysbus);
        end
      else
        acc <= sysbus;
  end
endmodule
```

The XNOR function will add further complexity to the encryption as it adds a second key on top of the existing function.

To decrypt this some one will have to know both keys and the order they were applied in. This increases the number of possible sets of keys to 2048. This is still manageable to brute force but will take orders of magnitude longer.

```
    case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator

    16: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    17: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    18: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    19: mdr = {`STORE,  7'd127};  //Output to display
    20: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    21: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    22: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    23: mdr = {`STORE,  7'd127};  //Output to display
    24: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    25: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    26: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
```

```
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    29: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    30: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    31: mdr = {`STORE,  7'd127};  //Output to display
    32: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    33: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    34: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    35: mdr = {`STORE,  7'd127};  //Output to display
    36: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    37: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    38: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    42: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    43: mdr = {`STORE,  7'd127};  //Output to display
    44: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    45: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    46: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    47: mdr = {`STORE,  7'd127};  //Output to display

    51: mdr = 21; //XOR Key
    52: mdr = 10; //XNOR key
    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 16;
    default: mdr = 0;             //rest of ROM is 0
  endcase
  end
```

XOR With Multiple Keys Encryption - 26/01/21

Finally I implemented a new program that would encrypt or decrypt the message using a key of multiple bytes. This program would read in both the string and a key, of length 3.

To brute force a key length of three it would have to try 32768 possible combinations of keys. However, if the length of the key is unknown it becomes exponentially harder to brute force the attack.

This method is scalable with different key lengths but can still be decrypted as long as the length of the key is less than the length of the encrypted string.

```
  case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator
    16: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
```

```
    17: mdr = {`STORE,  7'd72};   //Store the contents of accumulator
    18: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    19: mdr = {`STORE,  7'd73};   //Store the contents of accumulator
    20: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    21: mdr = {`STORE,  7'd74};   //Store the contents of accumulator

    22: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    23: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    24: mdr = {`STORE,  7'd127};  //Output to display
    25: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    26: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    29: mdr = {`XOR,    7'd74};   //XOR accumulator with key
    30: mdr = {`STORE,  7'd127};  //Output to display
    31: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    32: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    33: mdr = {`STORE,  7'd127};  //Output to display
    34: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    35: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    36: mdr = {`STORE,  7'd127};  //Output to display
    37: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    38: mdr = {`XOR,    7'd74};   //XOR accumulator with key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    42: mdr = {`STORE,  7'd127};  //Output to display
    43: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    44: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    45: mdr = {`STORE,  7'd127};  //Output to display
    46: mdr = {`BNE,    7'd55};   //Loop until all keys are tried

    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 22;
```

```
    /*RAM:
64: encrypted character 1
65: encrypted character 2
66: encrypted character 3
67: encrypted character 4
68: encrypted character 5
69: encrypted character 6
70: encrypted character 7
71: encrypted character 8
72: key 1
73: key 2
74: key 3
*/
    default: mdr = 0;             //rest of ROM is 0
  endcase
  end
```