Class:	CPE300L			Semester:	Fall 2021
Points		Document author:	Jerrod Batu		
		Author's email:	batuj1@unlv.nevada.edu		
		Document topic:	Postlab 3		
Instructor's comments:					

1. Introduction / Theory of Operation

Throughout this lab, I will learn about Quartus, VCS, and continue my knowledge of Verilog HDL. Specifically, I will have a better understanding of VCS for testbenches and combinational circuit designs. This lab will include a 7-segment in Verilog, magnitude comparators, ripple adder, and simple ALUS.

1. **Megafunction** in Quartus are ready-made, pre-tested practical blocks of code that expand existing design procedures. They decrease design assignments, extremely shorten tasks, and allow designers to optimize their time and energy on improving their system-level items and existing intellectual properties. Altera provides a library of megafunctions necessary to design more efficient logic synthesis and device applications.
2. The MegaWizard Plug-In Manager, provides several **applications** of custom megafunction variations to include in a design file. The main goal of these applications is to lessen the amount of instructions when instantiating specific functions. Some of these applications include: altfp_abs, altfp_add_sub, lpm_divide, lpm_and, lpm_or, etc.
   altfp_abs: floating-point absolute value megafunction
   altfp_add_sub: floating-point adder/subtractor megafunction
   lpm_divide: parameterized divider megafunction
   lpm_and: parameterized AND gate megafunction
   lpm_or: parameterized OR gate megafunction

2. Prelab

https://docs.google.com/document/d/10xN_bbupBwU2bnhr079n2DhvzONp04eo/edit?usp=sharing&ouid=102808507017671072128&rtpof=true&sd=true

This is the link to my prelab 3.

3. Results of Experiments

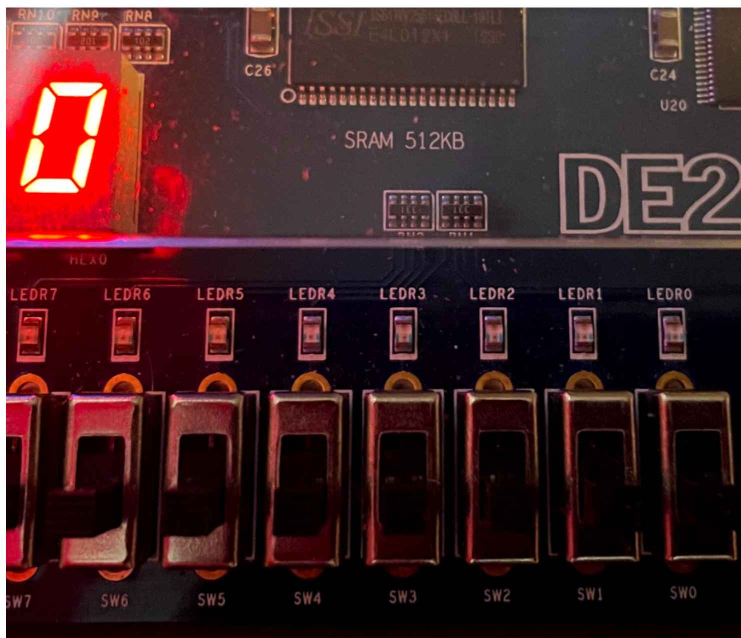
<u>Experiment 1</u>

a.

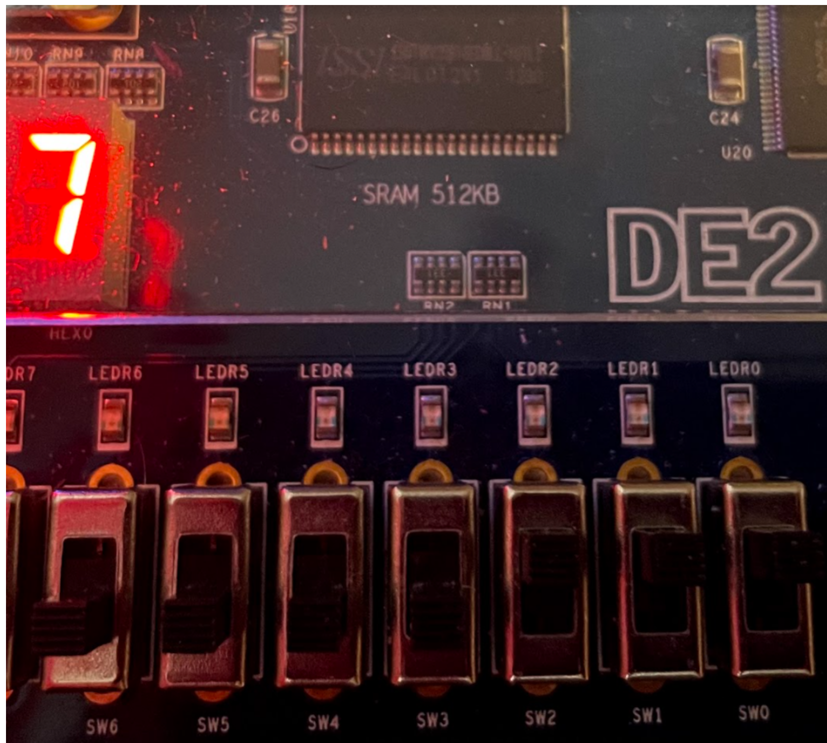
```
module seg7 (data,segments);
    input [3:0] data;
    output reg [6:0] segments;
    always @ (data) begin
        case (data)
            0 : segments = 7'b0000001;
            1 : segments = 7'b1001111;
            2 : segments = 7'b0010010;
            3 : segments = 7'b0000110;
            4 : segments = 7'b1001100;
            5 : segments = 7'b0100100;
            6 : segments = 7'b0100000;
            7 : segments = 7'b0001111;
            8 : segments = 7'b0000000;
            9 : segments = 7'b0000100;
            10: segments = 7'b0001000;
            11: segments = 7'b1100000;
            12: segments = 7'b0110001;
            13: segments = 7'b1000010;
            14: segments = 7'b0110000;
            15: segments = 7'b0111000;
            default: segments = 7'b1111111;
        endcase
    end
endmodule
```

This is my Verilog code for a 7-segment display.

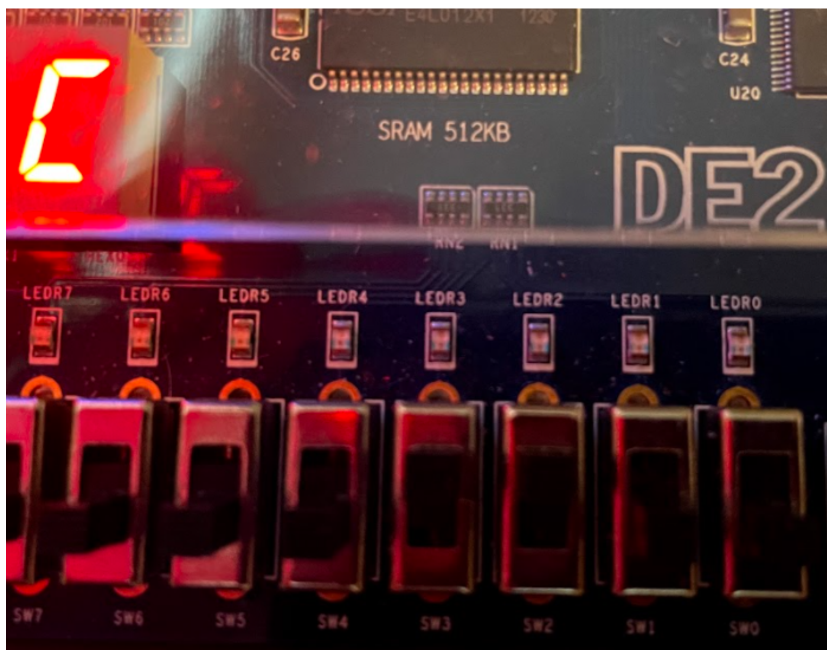
b.



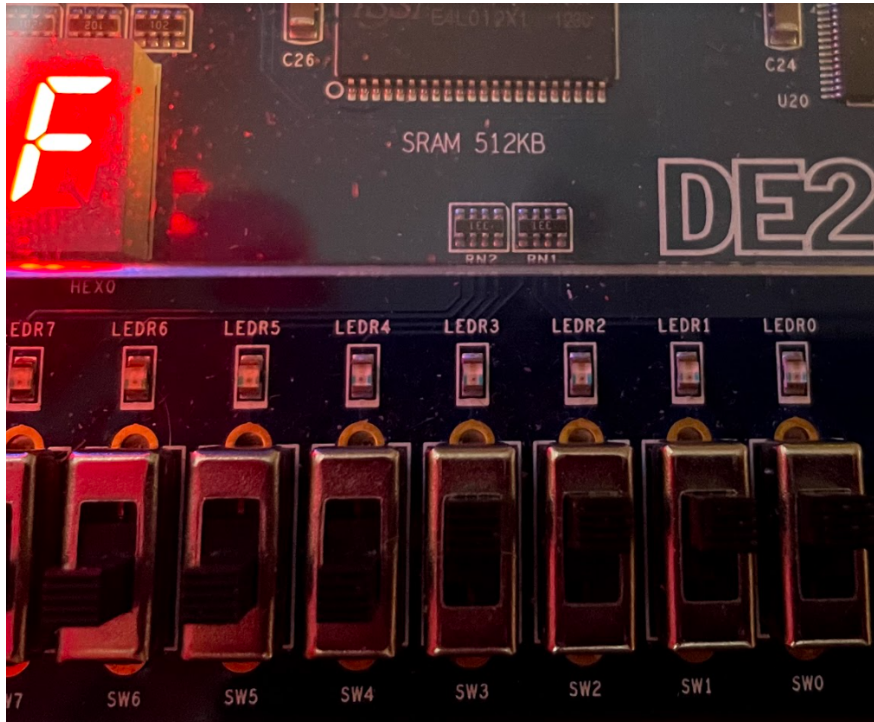
This is my DE2 board showing the 7-segment display of “0” when the data input is 0000.



This is my DE2 board showing the 7-segment display of "7" when the data input is 0111.



This is my DE2 board showing the 7-segment display of "C" when the data input is 1100.



This is my DE2 board showing the 7-segment display of “F” when the data input is 1111.

Experiment 2

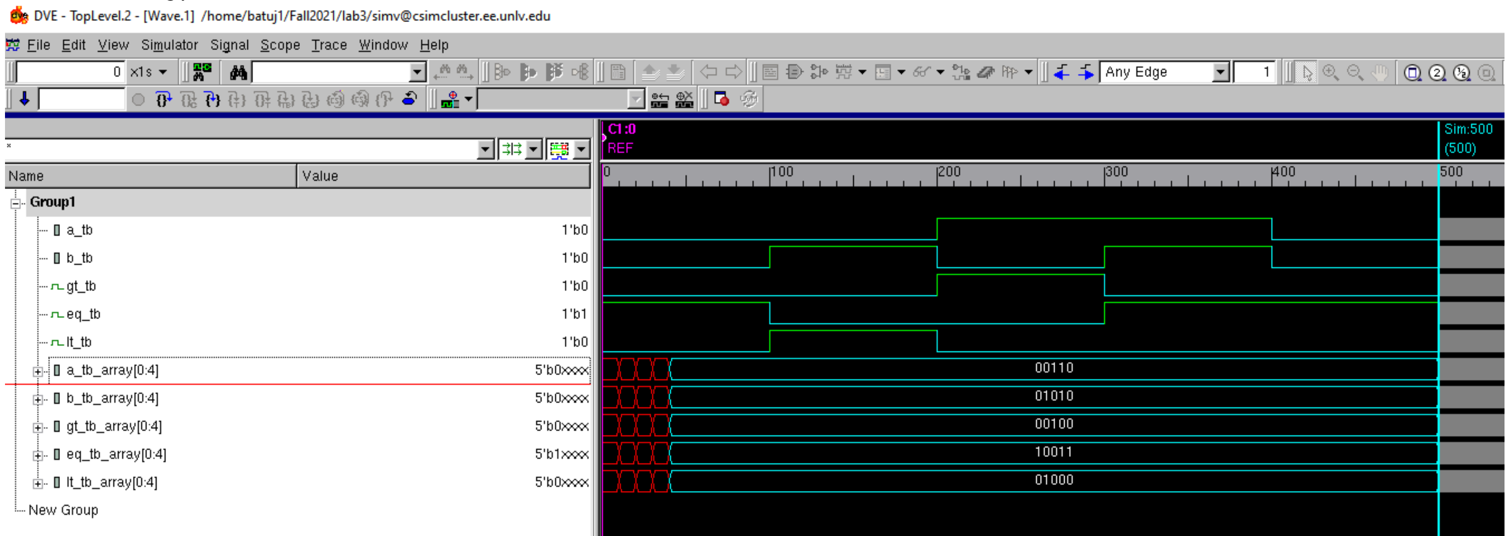
a.

```
module magCompare (a,b,gt,eq,lt);
    input a,b;
    output reg gt,eq,lt;

    always @ (a,b) begin
        gt = (a > b) ? 1 : 0; // greater than
        eq = (a == b) ? 1 : 0; // equal to
        lt = (a < b) ? 1 : 0; // less than
    end
endmodule

//magCompareTB testbench
module magCompareTB;
    parameter N = 4;
    reg a_tb;
    reg b_tb;
    wire gt_tb;
    wire eq_tb;
    wire lt_tb;
    reg [0:N] a_tb_array;
    reg [0:N] b_tb_array;
    reg [0:N] gt_tb_array;
    reg [0:N] eq_tb_array;
    reg [0:N] lt_tb_array;

    magCompare U0 (
        .a (a_tb),
        .b (b_tb),
        .gt (gt_tb),
        .eq (eq_tb),
        .lt (lt_tb)
    );

    initial begin
        $display("\t\ttime\ta_tb,\tb_tb,\tgt_tb,\teq_tb,\tlt_tb");
        $monitor("%d,\t%b,\t%b,\t%b,\t%b,\t%b",$time,a_tb,b_tb,gt_tb,eq_tb,lt_tb);
    end

    initial begin
        //initialization of input and output arrays
        a_tb_array [0] = 0;
        b_tb_array [0] = 0;
        gt_tb_array [0] = 0;
        eq_tb_array [0] = 1;
        lt_tb_array [0] = 0;
        #10
        a_tb_array [1] = 0;
        b_tb_array [1] = 1;
        gt_tb_array [1] = 0;
        eq_tb_array [1] = 0;
        lt_tb_array [1] = 1;
        #10
        a_tb_array [2] = 1;
        b_tb_array [2] = 0;
        gt_tb_array [2] = 1;
        eq_tb_array [2] = 0;
        lt_tb_array [2] = 0;
```

```
58          #10
59          a_tb_array [3] = 1;
60          b_tb_array [3] = 1;
61          gt_tb_array [3] = 0;
62          eq_tb_array [3] = 1;
63          lt_tb_array [3] = 0;
64          #10
65          a_tb_array [4] = 0;
66          b_tb_array [4] = 0;
67          gt_tb_array [4] = 0;
68          eq_tb_array [4] = 1;
69          lt_tb_array [4] = 0;
70      end
71
72      integer i;
73      always begin
74        for (i = 0; i <= N ; i = i + 1) begin
75          a_tb <= a_tb_array [i];
76          b_tb <= b_tb_array [i];
77          #100;
78      end
79          $display("\t\t\Test Finished");
80          $stop;
81      end
82  endmodule
83
```

This is my Verilog code for the magnitude comparator testbench from prelab 3.

b.

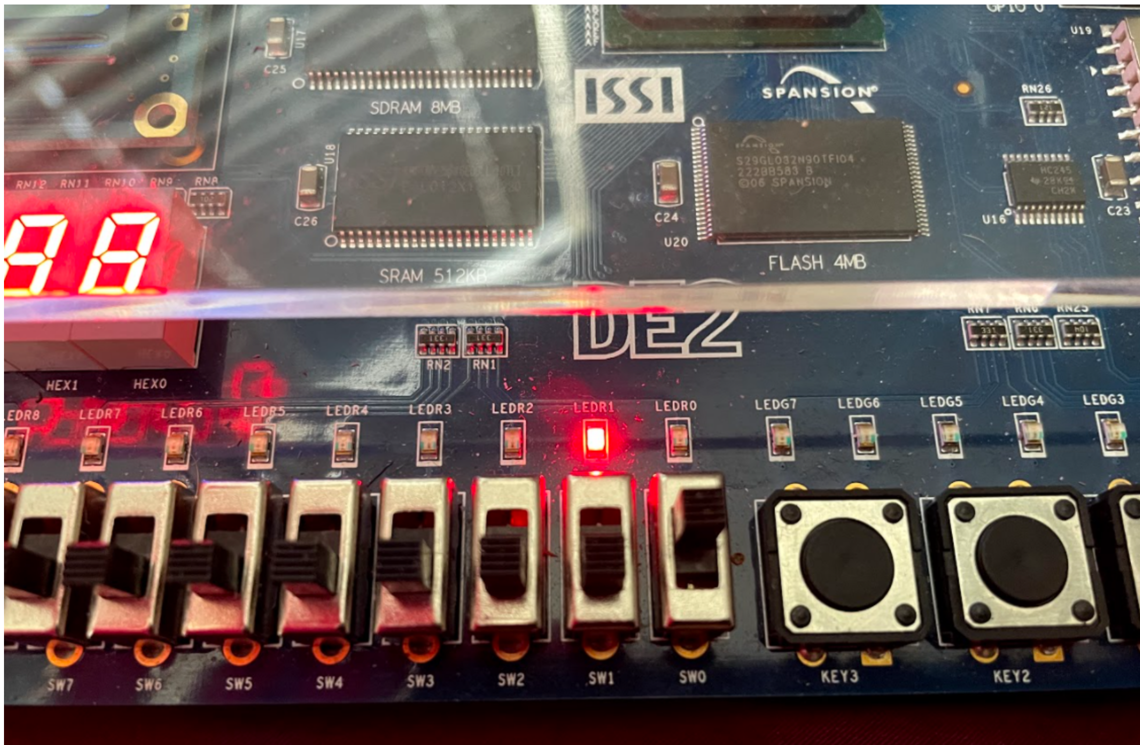


These are my VCS waveforms for the magnitude comparator testbench. The red waveforms represent the beginning delays at the start of the code.

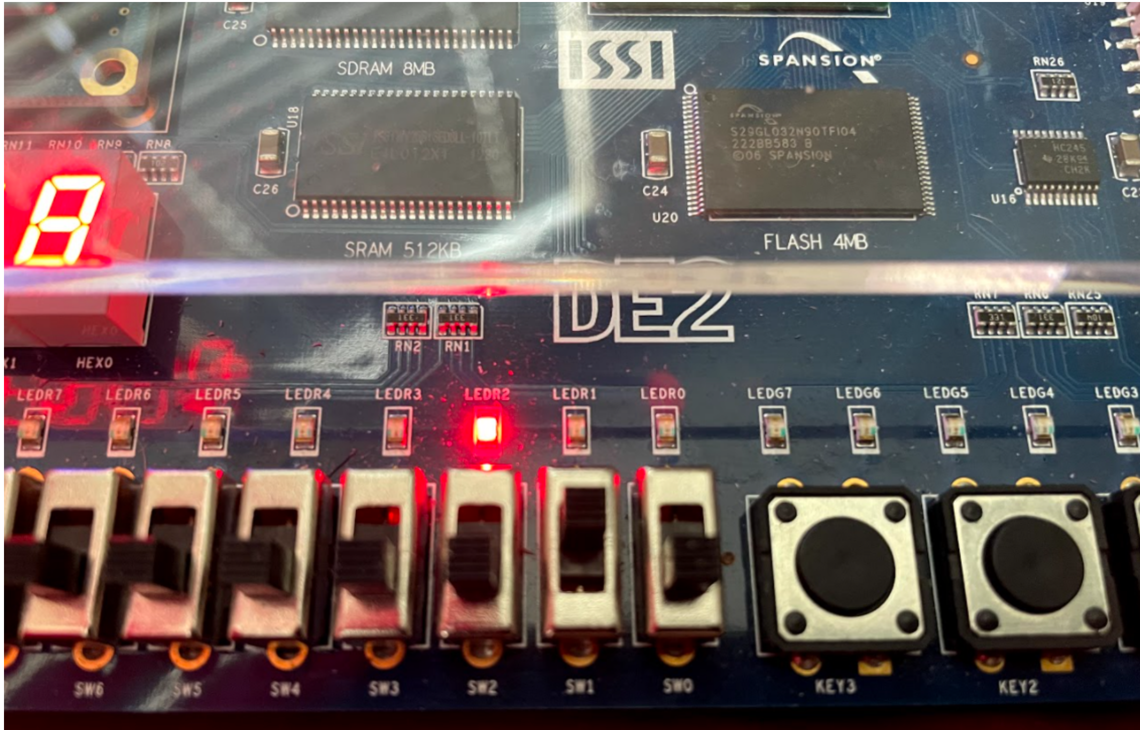
```
Chronologic VCS simulator copyright 1991-2017
Contains Synopsys proprietary information.
Compiler version N-2017.12-SP2-14; Runtime version N-2017.12-SP2-14;  Sep 14 13:49 2021
VCD+ Writer N-2017.12-SP2-14 Copyright (c) 1991-2017 by Synopsys Inc.
The file '/home/batuj1/Fall2021/lab3/inter.vpd' was opened successfully.
                  time    a_tb,   b_tb,   gt_tb,  eq_tb,  lt_tb
                     0,   0,      0,      0,      1,      0
                   100,   0,      1,      0,      0,      1
                   200,   1,      0,      1,      0,      0
                   300,   1,      1,      0,      1,      0
                   400,   0,      0,      0,      1,      0
                  Test Finished
```

This is the console output from VCS including the time, inputs, and results of the magnitude comparator testbench.

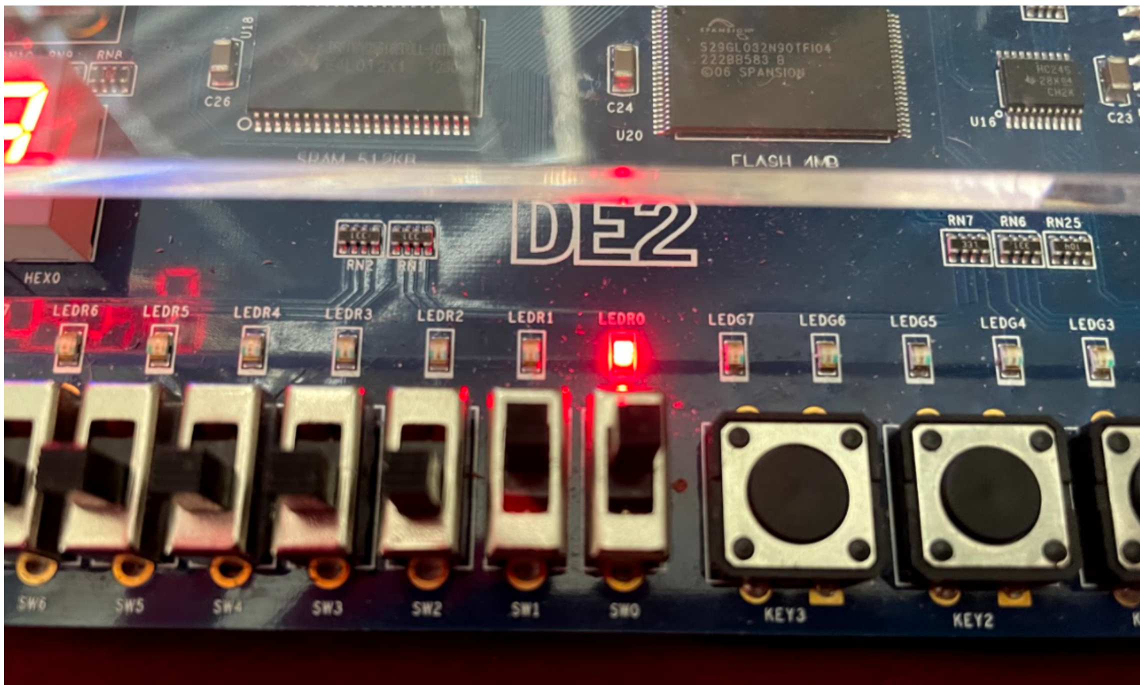
c.



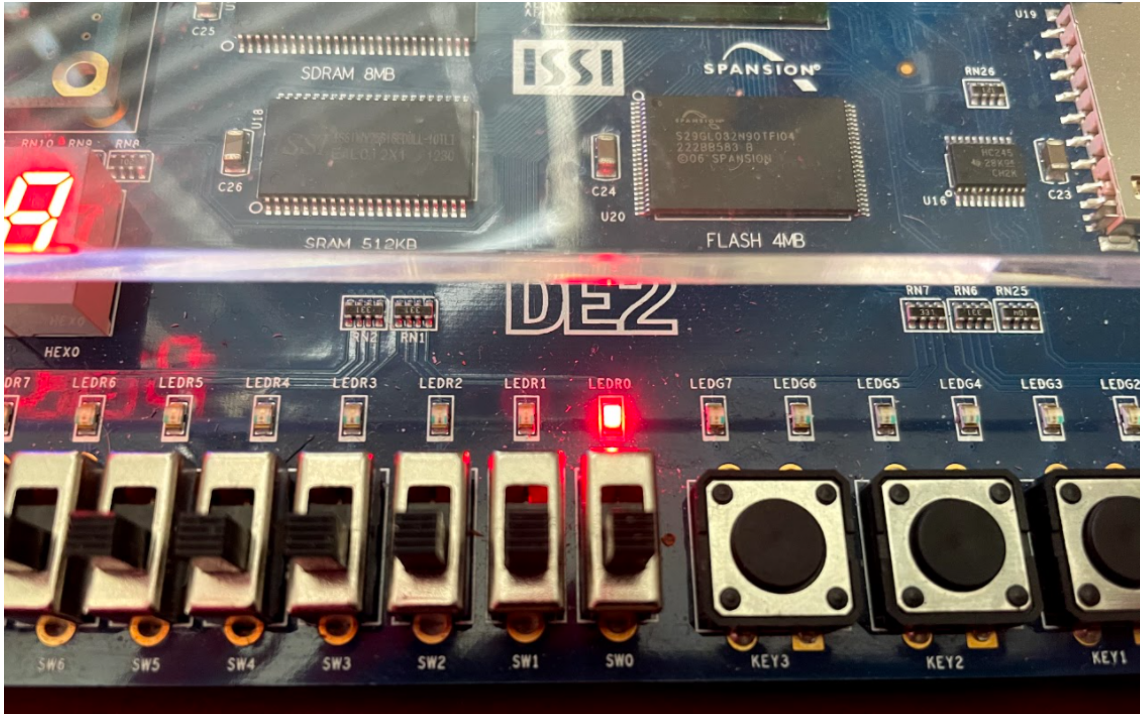
This is the DE2 board implementation of the magnitude comparator when a > b as a = 1 and b = 0.



This is the DE2 board implementation of the magnitude comparator when a < b as a = 0 and b = 1.



This is the DE2 board implementation of the magnitude comparator when a = b as a = 1 and b = 1.



This is the DE2 board implementation of the magnitude comparator when a = b as a = 0 and b = 0.

Experiment 3

a.

```
//4-bit Ripple Carry Adder
module rippleAdd (A, B, Cinra, Coutra, Sumra, Segments);
    input [3:0] A, B;
    input Cinra;
    output [3:0] Sumra;
    output Coutra;
    output reg [6:0] Segments;
    wire [3:1] carry;

    //instantiate four copies of the FullAdder
    FullAdd FA0 (A[0],B[0],Cinra,Sumra[0],carry[1]);
    FullAdd FA1 (A[1],B[1],carry[1],Sumra[1],carry[2]);
    FullAdd FA2 (A[2],B[2],carry[2],Sumra[2],carry[3]);
    FullAdd FA3 (A[3],B[3],carry[3],Sumra[3],Coutra);

    //7segment display
    always @ (Sumra) begin
        case (Sumra)
            0 : Segments = 7'b0000001;
            1 : Segments = 7'b1001111;
            2 : Segments = 7'b0010010;
            3 : Segments = 7'b0000110;
            4 : Segments = 7'b1001100;
            5 : Segments = 7'b0100100;
            6 : Segments = 7'b0100000;
            7 : Segments = 7'b0001111;
            8 : Segments = 7'b0000000;
            9 : Segments = 7'b0000100;
            10: Segments = 7'b0001000;
            11: Segments = 7'b1100000;
            12: Segments = 7'b0110001;
            13: Segments = 7'b1000010;
            14: Segments = 7'b0110000;
            15: Segments = 7'b0111000;
            default: Segments = 7'b1111111;
        endcase
    end
endmodule


//1-bit Full Adder
module FullAdd (X,Y,Cinfa,Sumfa,Coutfa);
    input  X, Y, Cinfa;
    output Sumfa, Coutfa;

    assign Sumfa  = X ^ Y ^ Cinfa;
    assign Coutfa = (X && Y) || ((X^Y) && Cinfa);
endmodule

```

This is my Verilog code for implementing a Full Adder module into a 4-bit Ripple Carry Adder including the 7-segment output display.

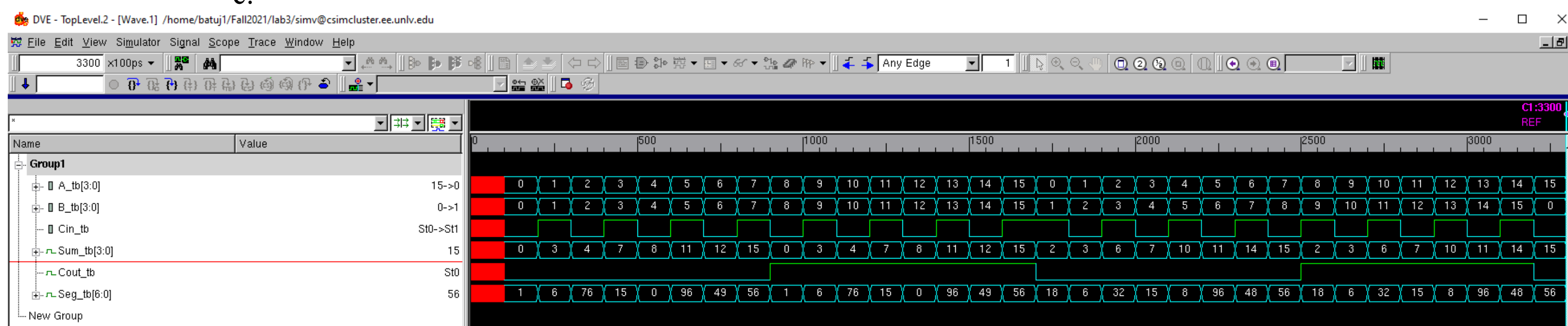
b.

```
`timescale 1 ns / 100 ps

//4-bit Ripple Carry Adder
module rippleAdd (A, B, Cinra, Coutra, Sumra, Segments);
    input [3:0] A, B;
    input Cinra;
    output [3:0] Sumra;
    output Coutra;
    output reg [6:0] Segments;
    wire [3:1] carry;

    //instantiate four copies of the FullAdder
    FullAdd FA0 (A[0],B[0],Cinra,Sumra[0],carry[1]);
    FullAdd FA1 (A[1],B[1],carry[1],Sumra[1],carry[2]);
    FullAdd FA2 (A[2],B[2],carry[2],Sumra[2],carry[3]);
    FullAdd FA3 (A[3],B[3],carry[3],Sumra[3],Coutra);

    //7segment display
    always @ (Sumra) begin
        case (Sumra)
            0 : Segments = 7'b0000001;
            1 : Segments = 7'b1001111;
            2 : Segments = 7'b0010010;
            3 : Segments = 7'b0000110;
            4 : Segments = 7'b1001100;
            5 : Segments = 7'b0100100;
            6 : Segments = 7'b0100000;
            7 : Segments = 7'b0001111;
            8 : Segments = 7'b0000000;
            9 : Segments = 7'b0000100;
            10: Segments = 7'b0001000;
            11: Segments = 7'b1100000;
            12: Segments = 7'b0110001;
            13: Segments = 7'b1000010;
            14: Segments = 7'b0110000;
            15: Segments = 7'b0111000;
            default: Segments = 7'b1111111;
        endcase
    end
endmodule


//1-bit Full Adder
module FullAdd (X,Y,Cinfa,Sumfa,Coutfa);
    input  X, Y, Cinfa;
    output Sumfa, Coutfa;

    assign Sumfa  = X ^ Y ^ Cinfa;
    assign Coutfa = (X && Y) || ((X^Y) && Cinfa);
endmodule


//4-bit Ripple Carry Adder TestBench
module rippleAddTB;
    parameter N = 32; //16 tests
    reg [3:0] A_tb;
    reg [3:0] B_tb;
```

```
58      reg        Cin_tb;
59      wire [3:0] Sum_tb;
60      wire       Cout_tb;
61      wire [6:0] Seg_tb;
62
63      rippleAdd U0 (
64          .A (A_tb),
65          .B (B_tb),
66          .Cinra (Cin_tb),
67          .Sumra (Sum_tb),
68          .Coutra (Cout_tb),
69          .Segments (Seg_tb)
70      );
71
72      initial begin
73          $display("\t\ttime\tA_tb,\tB_tb,\tCin_tb,\tSum_tb,\tCout_tb,\tSeg_tb");
74          $monitor("%d,\t%b,\t%b,\t%b,\t%b,\t%b,\t%b",$time,A_tb,B_tb,Cin_tb,Sum_tb,Cout_tb,Seg_tb);
75      end
76
77      integer i;
78      always begin
79          for (i = 0; i <= N; i = i + 1) begin
80              #10;
81            if (i >= 16) begin
82                  A_tb = i;
83                  B_tb = i + 1;
84                  Cin_tb = i - 1;
85            end
86            else begin
87              A_tb = i;
88              B_tb = i;
89              Cin_tb = i;
90            end
91
92          end
93              $display("\t\t\tTest Finished");
94              $stop;
95      end
96  endmodule
97
```

This is my Verilog code for the 4-bit Ripple Carry Adder testbench including the 7-segment output display.

c.

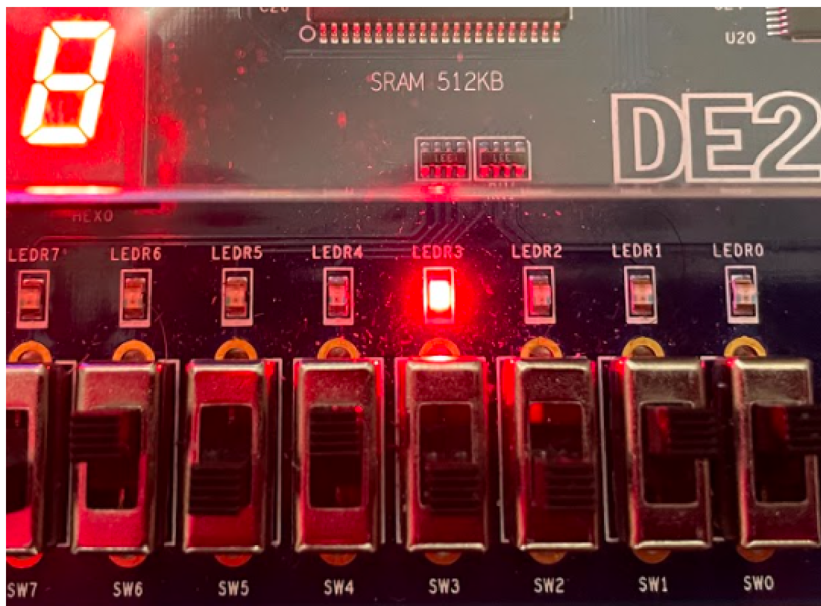


These are my waveforms from VCS that display the inputs and outputs of the 4-bit Ripple Carry Adder testbench including the 7-segment output display. The blocks of red lines indicate delay.

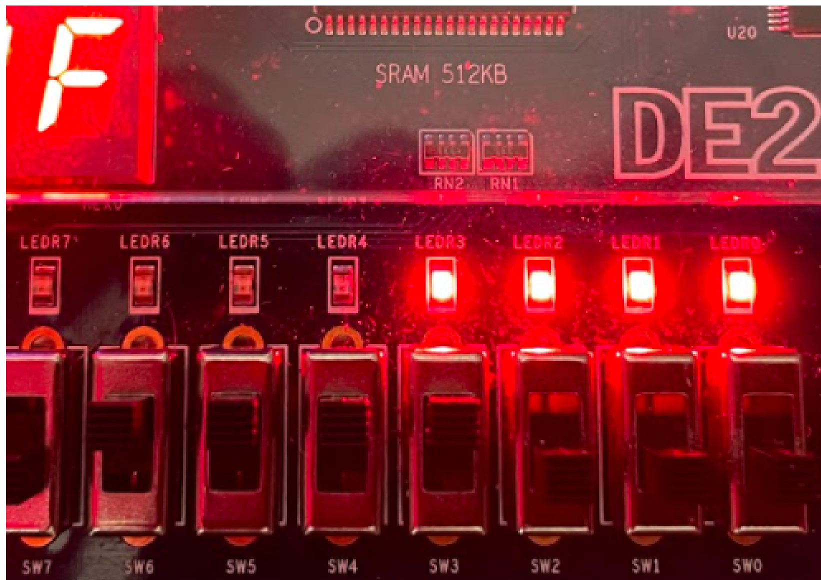
```
Type:        Severity:        Code: All

Contains Synopsys proprietary information.
Compiler version N-2017.12-SP2-14; Runtime version N-2017.12-SP2-14;  Sep 18 11:24 2021
VCD+ Writer N-2017.12-SP2-14 Copyright (c) 1991-2017 by Synopsys Inc.
The file '/home/batuj1/Fall2021/lab3/inter.vpd' was opened successfully.
                time    A_tb,   B_tb,   Cin_tb, Sum_tb, Cout_tb,        Seg_tb
                   0,   xxxx,   xxxx,   x,      xxxx,   x,      xxxxxxx
                  10,   0000,   0000,   0,      0000,   0,      0000001
                  20,   0001,   0001,   1,      0011,   0,      0000110
                  30,   0010,   0010,   0,      0100,   0,      1001100
                  40,   0011,   0011,   1,      0111,   0,      0001111
                  50,   0100,   0100,   0,      1000,   0,      0000000
                  60,   0101,   0101,   1,      1011,   0,      1100000
                  70,   0110,   0110,   0,      1100,   0,      0110001
                  80,   0111,   0111,   1,      1111,   0,      0111000
                  90,   1000,   1000,   0,      0000,   1,      0000001
                 100,   1001,   1001,   1,      0011,   1,      0000110
                 110,   1010,   1010,   0,      0100,   1,      1001100
                 120,   1011,   1011,   1,      0111,   1,      0001111
                 130,   1100,   1100,   0,      1000,   1,      0000000
                 140,   1101,   1101,   1,      1011,   1,      1100000
                 150,   1110,   1110,   0,      1100,   1,      0110001
                 160,   1111,   1111,   1,      1111,   1,      0111000
                 170,   0000,   0001,   1,      0010,   0,      0010010
                 180,   0001,   0010,   0,      0011,   0,      0000110
                 190,   0010,   0011,   1,      0110,   0,      0100000
                 200,   0011,   0100,   0,      0111,   0,      0001111
                 210,   0100,   0101,   1,      1010,   0,      0001000
                 220,   0101,   0110,   0,      1011,   0,      1100000
                 230,   0110,   0111,   1,      1110,   0,      0110000
                 240,   0111,   1000,   0,      1111,   0,      0111000
                 250,   1000,   1001,   1,      0010,   1,      0010010
                 260,   1001,   1010,   0,      0011,   1,      0000110
                 270,   1010,   1011,   1,      0110,   1,      0100000
                 280,   1011,   1100,   0,      0111,   1,      0001111
                 290,   1100,   1101,   1,      1010,   1,      0001000
                 300,   1101,   1110,   0,      1011,   1,      1100000
                 310,   1110,   1111,   1,      1110,   1,      0110000
                 320,   1111,   0000,   0,      1111,   0,      0111000
                Test Finished
```

This is my console output from VCS that displays the time for the test to finish, the inputs, and the outputs for the 4-bit Ripple Carry Adder testbench including the 7-segment output display.

d.



This is my DE2 board with the implementation of the 4-bit RCA displaying the output of 3 + 5 (0011 + 0101 = 1000 and no Cout). The 7-segment display shows the addition to be “8”, and the LEDs will follow the output of 1000.



This is my DE2 board with the implementation of the 4-bit RCA displaying the output of 8 + 7 (1000 + 0111 = 1111 and no Cout). The 7-segment display shows the addition to be “F”, and the LEDs will follow the output of 1111.

This is my DE2 board with the implementation of the 4-bit RCA displaying the output of 15 + 1 (1111 + 0001 = 0000 and 1 Cout). The 7-segment display shows the addition to be "0" because the addition goes out of the bit range, and the LEDs will follow the output of 0000 with LEDR5 as the Cout.

This is my DE2 board with the implementation of the 4-bit RCA displaying the output of 11 + 11 (1011 + 1011 = 0110 and 1 Cout). The 7-segment display shows the addition to be "6" because the addition goes out of the bit range, and the LEDs will follow the output of 0110 with LEDR5 as the Cout.

Experiment 4

a.

```
//simpleALU
module simpleALU (A, B, Cin, Cntrl, O, Cout,segments);
   input [3:0] A,B;
    input [2:0] Cntrl;
    input Cin;
    output reg [3:0] O;
    output reg Cout;
    output reg [6:0] segments;

    reg [3:0] Carry, Borrow;
    always @ (*)
    begin
        case(Cntrl)
            3'b000: begin                   //INC
                        O = A + 1;
                        if (A == 4'b1111)
                            begin
                                Cout = 1;
                            end
                        else
                            begin
                                Cout = 0;
                            end
                    end
            3'b001: begin                   //DEC
                        O = A - 1;
                        Cout = 0;
                    end
            3'b010: begin                   //ROR
                        O = {A[0],A[3],A[2],A[1]};
                        Cout = 0;
                    end
            3'b011: begin                   //SHR
                        O = A >> 1;
                        Cout = 0;
                    end
            3'b100: begin                   //AND
                        O = A & B;
                        Cout = 0;
                    end
            3'b101: begin                   //OR
                        O = A | B;
                        Cout = 0;
                    end
            3'b110: begin                   //ADD
                        O = A + B + Cin;
                        Carry = (A && B) || ((A && Cin) || (B && Cin));
                        Cout = Carry;
                   end
            3'b111: begin                   //SUB
                        O = A - B - Cin;
                        Borrow = (!A && (B ^ Cin)) || (B && Cin);
                        Cout = Borrow;
                    end
        endcase
    end

```

```
//7-Segment Display
always @ (O) begin
    case (O)
        0 : segments = 7'b0000001;
        1 : segments = 7'b1001111;
        2 : segments = 7'b0010010;
        3 : segments = 7'b0000110;
        4 : segments = 7'b1001100;
        5 : segments = 7'b0100100;
        6 : segments = 7'b0100000;
        7 : segments = 7'b0001111;
        8 : segments = 7'b0000000;
        9 : segments = 7'b0000100;
        10: segments = 7'b0001000;
        11: segments = 7'b1100000;
        12: segments = 7'b0110001;
        13: segments = 7'b1000010;
        14: segments = 7'b0110000;
        15: segments = 7'b0111000;
        default: segments = 7'b1111111;
    endcase
end
odule
```

This is my Verilog code for implementing a 4-bit ALU module including the 7-segment output display.

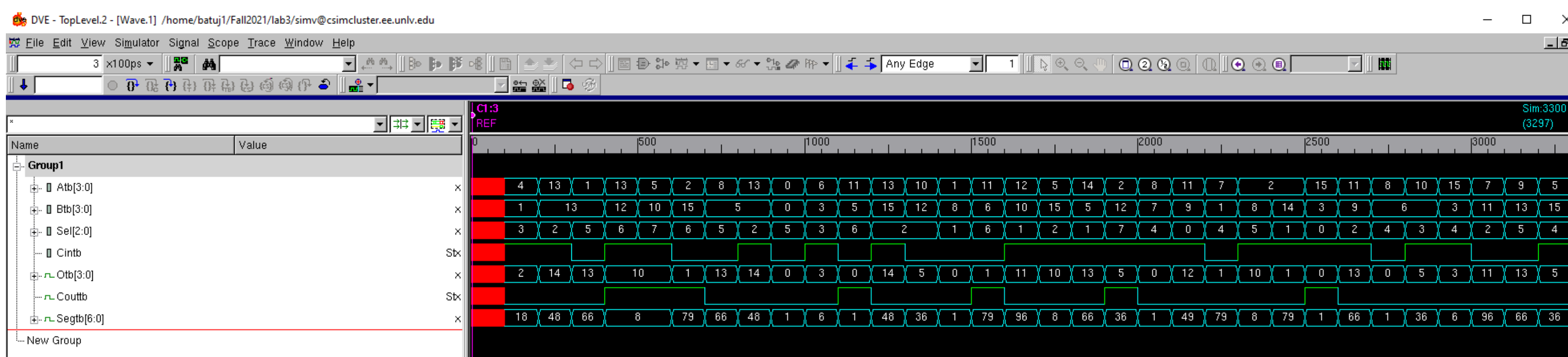
b.

```
`timescale 1 ns / 100 ps

//simpleALU
module simpleALU (A, B, Cin, Cntrl, O, Cout,segments);
   input [3:0] A,B;
    input [2:0] Cntrl;
    input Cin;
    output reg [3:0] O;
    output reg Cout;
    output reg [6:0] segments;

    reg [3:0] Carry, Borrow;
    always @ (*)
    begin
        case(Cntrl)
            3'b000: begin                       //INC
                        O = A + 1;
                        if (A == 4'b1111)
                            begin
                                Cout = 1;
                            end
                        else
                            begin
                                Cout = 0;
                            end
                      end
            3'b001: begin                       //DEC
                        O = A - 1;
                        Cout = 0;
                      end
            3'b010: begin                       //ROR
                        O = {A[0],A[3],A[2],A[1]};
                        Cout = 0;
                      end
            3'b011: begin                       //SHR
                        O = A >> 1;
                        Cout = 0;
                      end
            3'b100: begin                       //AND
                        O = A & B;
                        Cout = 0;
                      end
            3'b101: begin                       //OR
                        O = A | B;
                        Cout = 0;
                      end
            3'b110: begin                       //ADD
                        O = A + B + Cin;
                        Carry = (A && B) || ((A && Cin) || (B && Cin));
                        Cout = Carry;
                     end
            3'b111: begin                       //SUB
                        O = A - B - Cin;
                        Borrow = (!A && (B ^ Cin)) || (B && Cin);
                        Cout = Borrow;
                      end
        endcase
```

```
    end

    //7-Segment Display
    always @ (O) begin
        case (O)
            0 : segments = 7'b0000001;
            1 : segments = 7'b1001111;
            2 : segments = 7'b0010010;
            3 : segments = 7'b0000110;
            4 : segments = 7'b1001100;
            5 : segments = 7'b0100100;
            6 : segments = 7'b0100000;
            7 : segments = 7'b0001111;
            8 : segments = 7'b0000000;
            9 : segments = 7'b0000100;
            10: segments = 7'b0001000;
            11: segments = 7'b1100000;
            12: segments = 7'b0110001;
            13: segments = 7'b1000010;
            14: segments = 7'b0110000;
            15: segments = 7'b0111000;
            default: segments = 7'b1111111;
        endcase
    end
endmodule

//simple ALU testbench
module simpleALU_tb;
    parameter N = 32; //32 tests
    reg [3:0] Atb,Btb;
    reg [2:0] Sel;
    reg Cintb;
    wire [3:0] Otb;
    wire Couttb;
    wire [6:0] Segtb;

    simpleALU U0 (
        .A (Atb),
        .B (Btb),
        .Cin (Cintb),
        .Cntrl (Sel),
        .O (Otb),
        .Cout (Couttb),
        .segments (Segtb)
    );

    initial begin
        $display("\t\ttime\tAtb,\tBtb,\tCintb,\tSel,\tOtb,\tCouttb,\tSegtb");
        $monitor("%d,\t%b,\t%b,\t%b,\t%b,\t%b,\t%b,\t%b",$time,Atb,Btb,Cintb,Sel,Otb,Couttb,Segtb);
    end

    integer i;
    always begin
        for (i = 0; i <= N; i = i + 1) begin
            #10;
            Atb = $random;
            Btb = $random;
            Cintb = $random;
            Sel = $random;
        end
            $display ("\t\t\Test Finished");
            $stop;
    end
endmodule
```

This is my Verilog code for the 4-bit ALU module testbench including the 7-segment output display.

c.



These are my waveforms from VCS that display the inputs and outputs of the 4-bit ALU testbench including the 7-segment output display. The blocks of red lines indicate delay.

```
Type:        Severity:        Code: All
Chronologic VCS simulator copyright 1991-2017
Contains Synopsys proprietary information.
Compiler version N-2017.12-SP2-14; Runtime version N-2017.12-SP2-14;  Sep 18 11:53 2021
VCD+ Writer N-2017.12-SP2-14 Copyright (c) 1991-2017 by Synopsys Inc.
The file '/home/batuj1/Fall2021/lab3/inter.vpd' was opened successfully.
                   time     Atb,     Btb,     Cintb,   Sel,     Otb,     Couttb,  Segtb
                     0,     xxxx,    xxxx,    x,       xxx,     xxxx,    x,       xxxxxxx
                    10,     0100,    0001,    1,       011,     0010,    0,       0010010
                    20,     1101,    1101,    1,       010,     1110,    0,       0110000
                    30,     0001,    1101,    0,       101,     1101,    0,       1000010
                    40,     1101,    1100,    1,       110,     1010,    1,       0001000
                    50,     0101,    1010,    1,       111,     1010,    1,       0001000
                    60,     0010,    1111,    0,       110,     0001,    1,       1001111
                    70,     1000,    0101,    0,       101,     1101,    0,       1000010
                    80,     1101,    0101,    1,       010,     1110,    0,       0110000
                    90,     0000,    0000,    0,       101,     0000,    0,       0000001
                   100,     0110,    0011,    1,       011,     0011,    0,       0000110
                   110,     1011,    0101,    0,       110,     0000,    1,       0000001
                   120,     1101,    1111,    1,       010,     1110,    0,       0110000
                   130,     1010,    1100,    0,       010,     0101,    0,       0100100
                   140,     0001,    1000,    0,       001,     0000,    0,       0000001
                   150,     1011,    0110,    0,       110,     0001,    1,       1001111
                   160,     1100,    1010,    1,       001,     1011,    0,       1100000
                   170,     0101,    1111,    1,       010,     1010,    0,       0001000
                   180,     1110,    0101,    1,       001,     1101,    0,       1000010
                   190,     0010,    1100,    1,       111,     0101,    1,       0100100
                   200,     1000,    0111,    1,       100,     0000,    0,       0000001
                   210,     1011,    1001,    1,       000,     1100,    0,       0110001
                   220,     0111,    0001,    0,       100,     0001,    0,       1001111
                   230,     0010,    1000,    1,       101,     1010,    0,       0001000
                   240,     0010,    1110,    1,       001,     0001,    0,       1001111
                   250,     1111,    0011,    1,       000,     0000,    1,       0000001
                   260,     1011,    1001,    1,       010,     1101,    0,       1000010
                   270,     1000,    0110,    0,       100,     0000,    0,       0000001
                   280,     1010,    0110,    1,       011,     0101,    0,       0100100
                   290,     1111,    0011,    1,       100,     0011,    0,       0000110
                   300,     0111,    1011,    0,       010,     1011,    0,       1100000
                   310,     1001,    1101,    0,       101,     1101,    0,       1000010
                   320,     0101,    1111,    1,       100,     0101,    0,       0100100
                   Test Finished
```

This is my console output from VCS that displays the time for the test to finish, the inputs, and the outputs for the 4-bit ALU testbench including the 7-segment output display.

d. https://drive.google.com/file/d/1ukPazOBCfBhTPD_-QSV45y_pNUsIS62B/view?usp=sharing
This is a link to my video displaying the opcode from the table for the 4-bit ALU. The video will also be included within the submission zip file.

4. Answers to questions

Question 1: What is an unintentional latch in a Verilog Design? Is it a good or bad design practice?

An unintentional latch in a Verilog Design is when the user inserts a latch in place of where combinational logic can be found. An example of this is when the user takes out the default case within a case statement. Instead of a multiplexer being formed, a latch would take its place because the net is not assigned to any known values. These unintentional latches may also occur from any missing signals within the sensitivity list. Unintentional latches are also called unintended latches or inferred latches. These unintentional latches are considered as bad design practice as the user is unintentionally creating these latches. By accidentally making these latches, the original design is altered; therefore, the user would need to go back into the Verilog code and rewrite it to where the combinational logic would form multiplexers.

Question 2: Why do we need a testbench? Is the waveform simulation not good enough?

We need a testbench to verify the functionality of a design and to report the inputs and outputs in a readable format within the console. The waveform simulation is only good enough in specific circumstances. By using a testbench, the user can check a large number of signals rather than by manually inputting them into the waveform simulation. In addition, the testbench, itself, can generate a periodic clock signal, acquire signal waveforms, and create a simulation report. Forcing inputs within a waveform simulation may be time consuming when there are multiple signals; hence, the testbench would make verification of functionality much quicker.

Question 3: Explain the differences between $monitor and $display.

$monitor and $display have several differences. $monitor is a computing program for editing and viewing, whereas $display is a screen that only shows graphics or text. $display and $write both display arguments in the order that they are laid out within the argument list. The key difference between the two is that $display is used when values are to be printed to the console, and $monitor is to be called only one time to print the value of a variable whenever it is to be altered. $monitor is utilized to monitor signals when values change throughout the compilation of the Verilog code, and $display is utilized every time to print values or display the immediate values of signals.

5. Conclusions & Summary

This lab was pretty straightforward as the lab instructions were somewhat clear. The only parts I did have confusion was for experiment 2 where the instructions said to create a Magnitude Comparator from prelab 2; however, this was a typo instead of prelab 2 it should've been prelab 3. The ripple adder was a great review because I remember doing a similar lab in CPE 200 L.

Experiment 1 seemed to have been carried over from CPE 200 L as the 7-segment display module is somewhat identical to the one in the previous class. Experiment 4 would probably be the most challenging as the 4-bit ALU implementation involved if statements within cases to include the cout within the outputs. I enjoyed reading upon testbenches as they were my struggle in the CPE 200 L class, and I am more comfortable with writing testbenches now than I was before. In addition, the hardest part of this lab was filming the video for the 4-bit ALU operations because the video took up a lot of data. Overall, I enjoyed this lab, and my knowledge of testbenches has increased as well as the implementation of our Verilog code into the DE2 board.