CSC258H1: Pre-lab Exercise for Lab 3

Wang, Weiqing

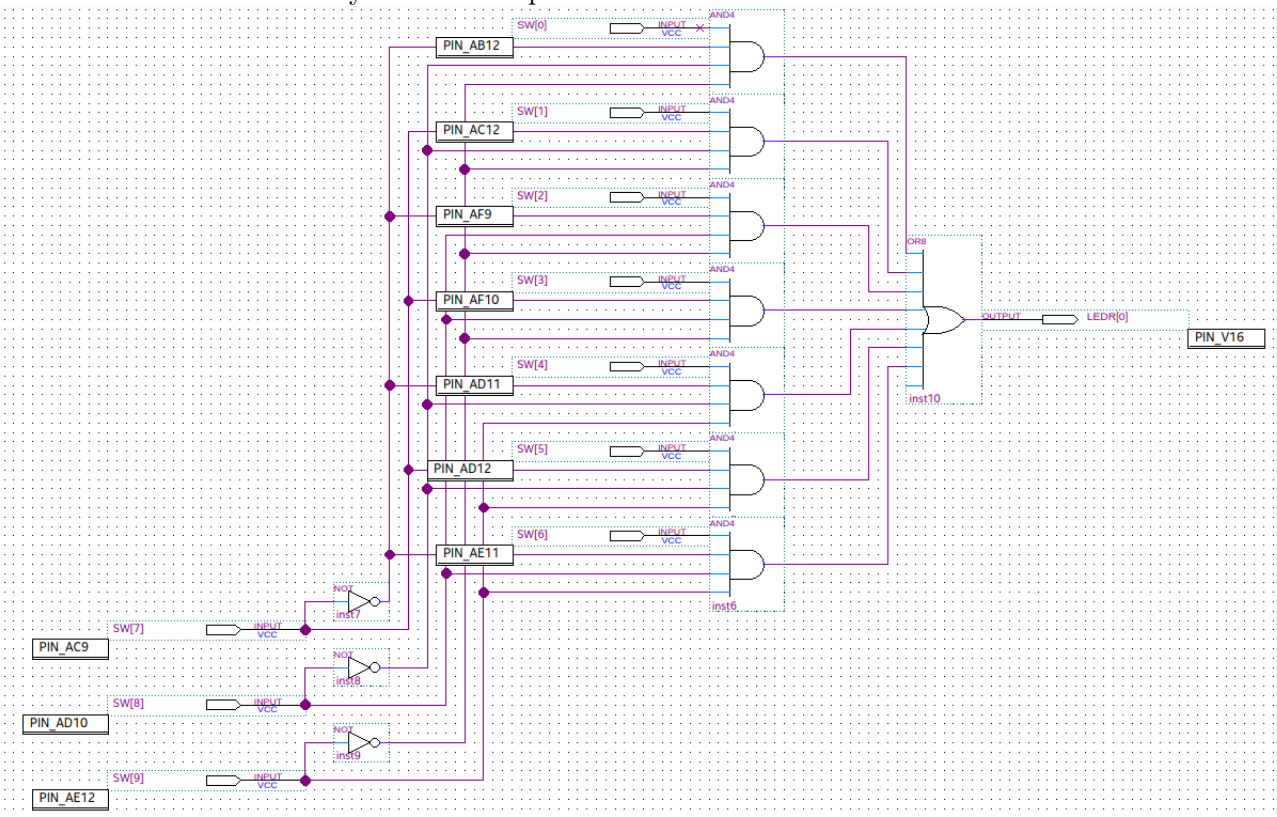
Due: January 28, 2018 before 12 a.m.

1 Part I

1. Draw a schematic showing your code structure with all wires, inputs and outputs labelled. Be prepared to explain it to the TA as a part of your preparation.

Solution.

Here is the schematic for my 7-to-1 Multiplexer:



2. Write Verilog code for a 7-to-1 multiplexer, based on the template you provided above. Use switches SW_{9-7} on the DE1-SoC board as the `MuxSelect` input and switches SW_{6-0} as the `Input` data inputs. Connect the output to $LEDR_0$.

Solution.

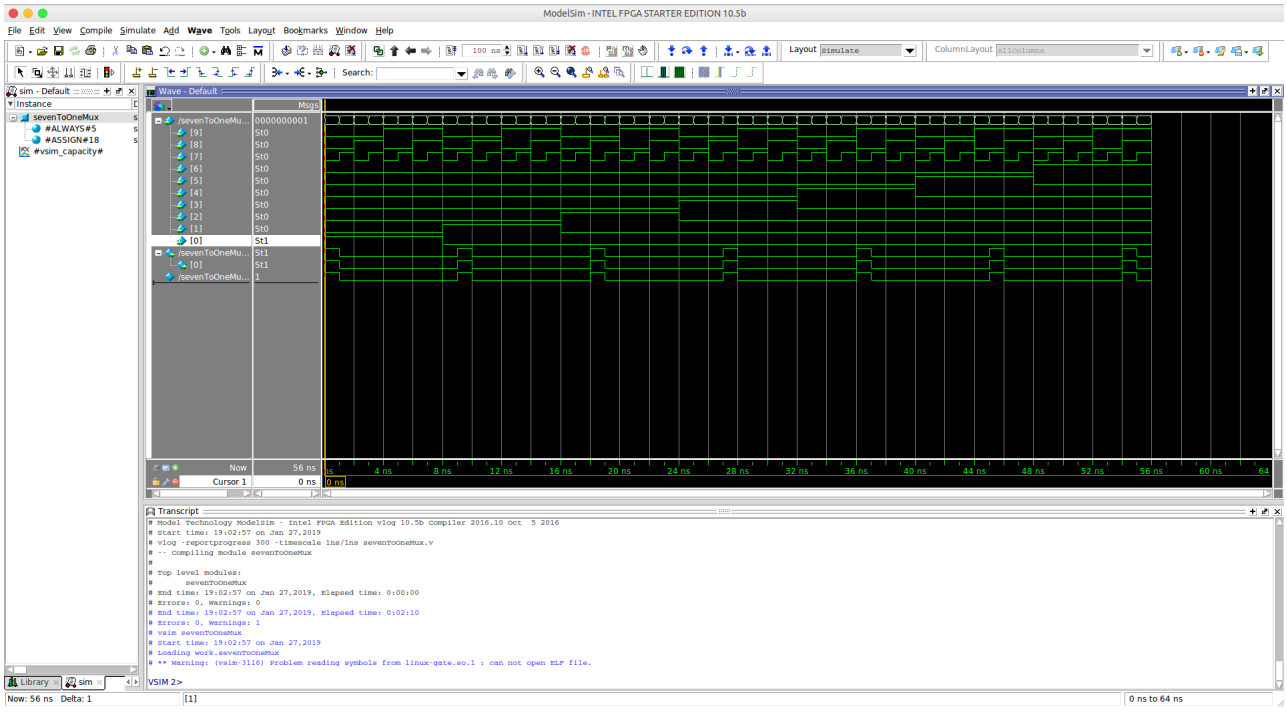
The following is the Verilog code I wrote for the 7-to-one multiplexer.

```
module sevenToOneMux(SW, LEDR);
    input [9:0] SW;
    output [0:0] LEDR;
    reg Out;
    always @(*)
    begin
        case(SW[9:7])
            3'b000: Out = SW[0];
```

```
9               3'b001: Out = SW[1];
10              3'b010: Out = SW[2];
11              3'b011: Out = SW[3];
12              3'b100: Out = SW[4];
13              3'b101: Out = SW[5];
14              3'b110: Out = SW[6];
15              default: Out = 1'b0;
16          endcase
17      end
18      assign LEDR[0] = Out;
19  endmodule
```

3. Simulate your circuit with ModelSim for different values of `MuxSelect` and `Input`. You must include a screen shot of simulations output as part of your pre-lab.

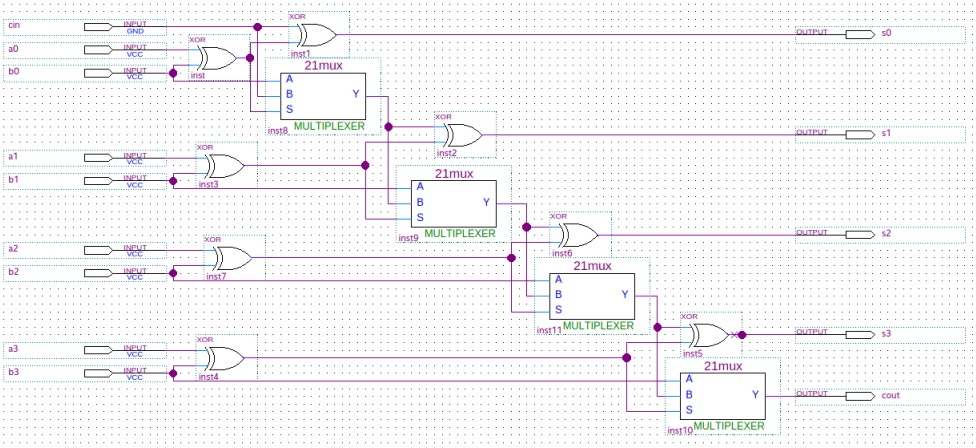
Solution.
Here is the screenshot for my simulation on ModelSim:



2 Part II

1. Draw a schematic showing your code structure with all wires, inputs and outputs labelled. Your schematic should resemble Figure 1(d), though it should also contain module and signal labels, and shows external connections to the switches and LEDs. Be prepared to explain it to the TA as part of your preparation.

Solution. The following is the schematic for my Four-bit ripple-carry adder circuit:



2. Write a Verilog module for the full adder sub-circuit and write another Verilog module that instantiates four instances of this full adder. Name the input ports A, B and cin, and the output ports S and $cout$. Note: You should **NOT** use the arithmetic addition operator + in your Verilog implementation of the full-adder. Doing so will earn you 0 marks for this part

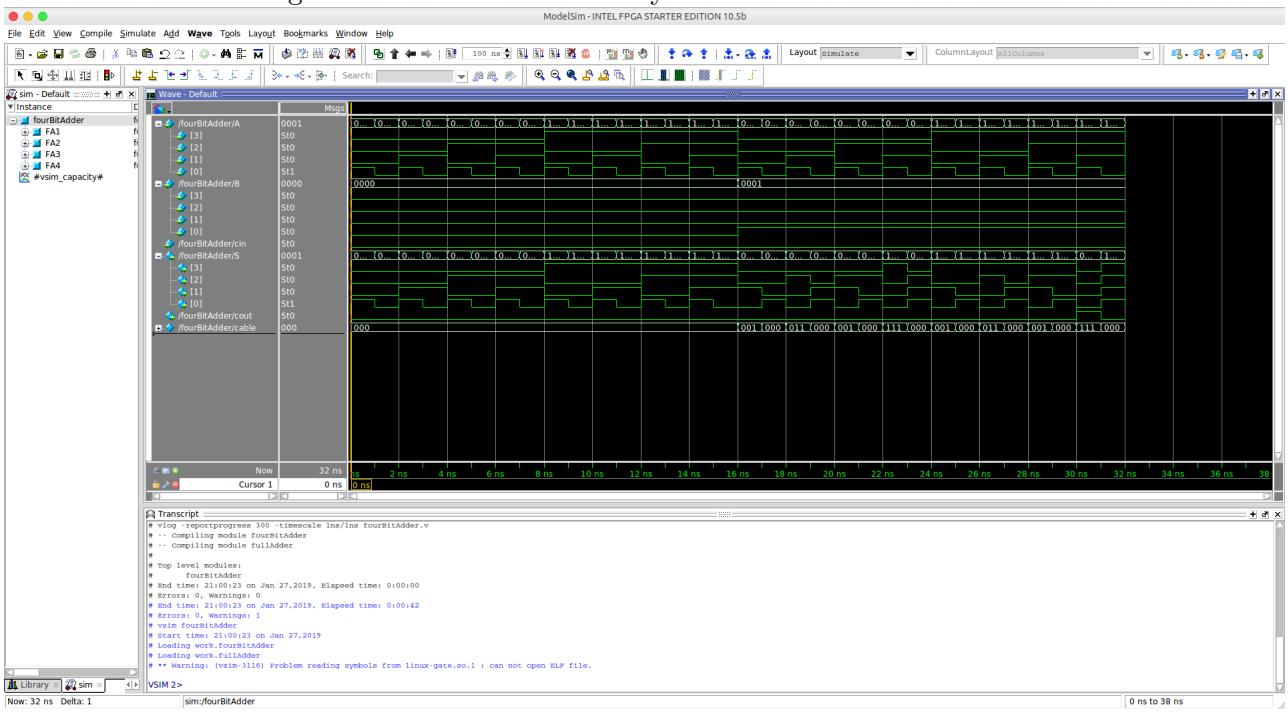
Solution. The following is the Verilog code I wrote for my circuit:

```
module fourBitAdder(A, B, cin, S, cout);
    input [3:0] A;
    input [3:0] B;
    input cin;
    output [3:0] S;
    output cout;
    wire [2:0] cable;
    fullAdder FA1(
        .a(A[0]),
        .b(B[0]),
        .cin(cin),
        .s(S[0]),
        .cout(cable[0]));
    fullAdder FA2(
        .a(A[1]),
        .b(B[1]),
        .cin(cable[0]),
        .s(S[1]),
        .cout(cable[1]));
    fullAdder FA3(
        .a(A[2]),
```

```
22          .b(B[2]),
23          .cin(cable[1]),
24          .s(S[2]),
25          .cout(cable[2]));
26      fullAdder FA4(
27          .a(A[3]),
28          .b(B[3]),
29          .cin(cable[2]),
30          .s(S[3]),
31          .cout(cout));
32  endmodule
33
34  module fullAdder(a, b, cin, s, cout);
35      input a, b, cin;
36      output s, cout;
37
38      assign s = a^b^cin;
39      assign cout = (a & b) | (cin & (a^b));
40  endmodule
```

3. Simulate your 4-bit ripple-carry adder with ModelSim for intelligently chosen values of A and B and *cin*. You should include a screenshot of it in the pre-lab. Note that as circuits get more complicated, you will not be able to simulate or test all possible cases. This means that you can test only a subset. Here *intelligently chosen* means to find particular *corner cases* that exercise key aspects of the circuit. An example would be a pattern that shows that the carry signals are working. Be prepared to explain why your test cases are good enough.

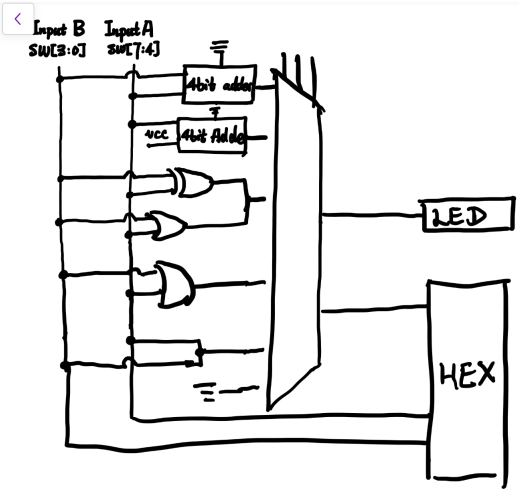
Solution. The following is the the screenshot for my simulation:



3 Part III

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Your schematic should contain a block diagram of your design showing any design hierarchy. You should show the multiplexer that is implied by the case statement for your ALU as well as all inputs to this multiplexer. Also show all connections to switches and LEDs. Be prepared to explain it to the TA.

Solution. The following is the schematic for my ALU:



2. Write a Verilog module for the ALU including all inputs and outputs.

Solution. Here is my Verilog module:

```
module ALU(SW, KEY, LEDR, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);
    input [7:0] SW;
    input [2:0] KEY;
    output [6:0] HEX0;
    output [6:0] HEX1;
    output [6:0] HEX2;
    output [6:0] HEX3;
    output [6:0] HEX4;
    output [6:0] HEX5;
    output [7:0] LEDR;

    wire [4:0] addAToB;
    fourBitAdder FA1(.SW({2'b00, SW[7:0]}), .LEDR(addAToB[4:0]));

    wire [4:0] addOneToA;
    fourBitAdder FA2(.SW({2'b00, SW[7:4], 4'b0001}), .LEDR(addOneToA[4:0]));

    reg [7:0] ALUout;
    always @(*)
    begin
        case (KEY[2:0])
            3'b000: ALUout[7:0] = {3'b000, addOneToA[4:0]};
            3'b001: ALUout[7:0] = {3'b000, addAToB[4:0]};
```

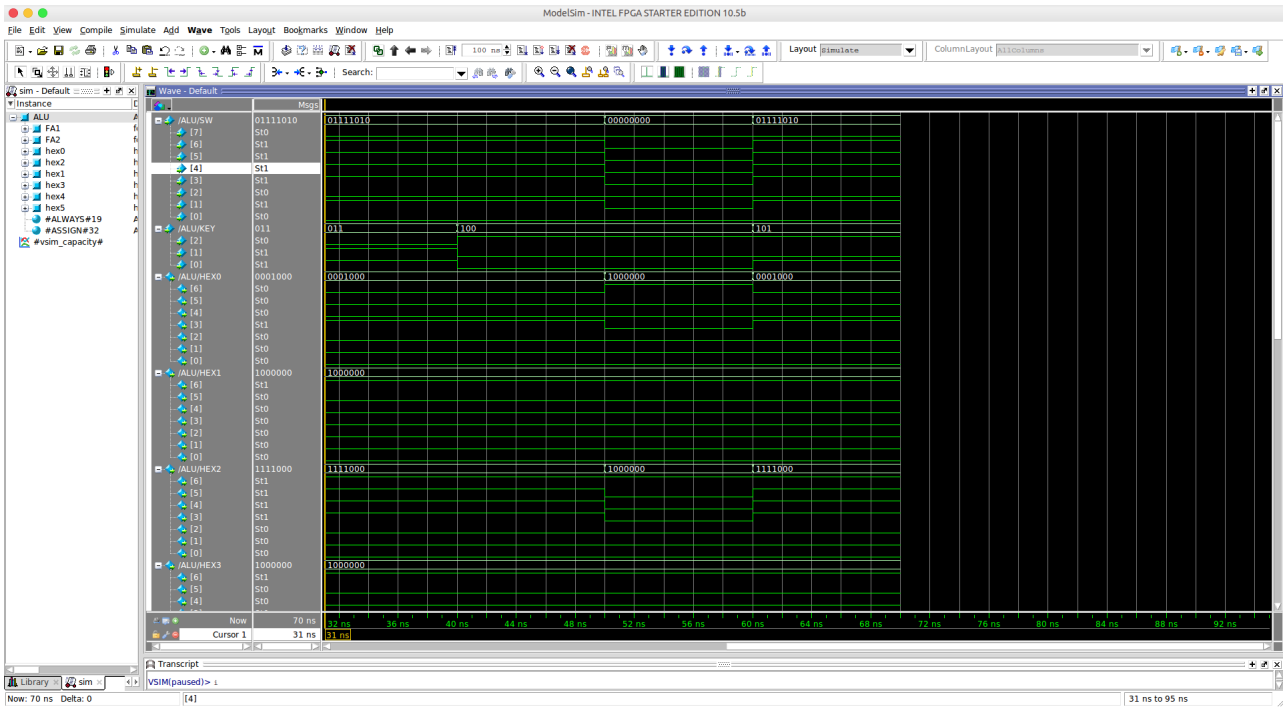
```
            3'b010: ALUout[7:0] = {3'b000, SW[7:4] + SW[3:0]};
            3'b011: ALUout[7:0] = {SW[7:4] | SW[3:0], SW[7:4] ^ SW[3:0]};
            3'b100: ALUout[7:0] = {7'b0000000, (|SW[7:0])};
            3'b101: ALUout[7:0] = SW[7:0];
            default: ALUout[7:0] = 8'b0000_0000;
        endcase
    end

    assign LEDR[7:0] = ALUout[7:0];
    hexDecoder hex0(.SW(SW[3:0]), .HEX(HEX0[6:0]));     //B
    hexDecoder hex2(.SW(SW[7:4]), .HEX(HEX2[6:0]));     //A
    hexDecoder hex1(.SW(4'b0000), .HEX(HEX1[6:0]));     //0000
    hexDecoder hex3(.SW(4'b0000), .HEX(HEX3[6:0]));     //0000
    hexDecoder hex4(.SW(ALUout[3:0]), .HEX(HEX4[6:0]));     //ALUout[3:0]
    hexDecoder hex5(.SW(ALUout[7:4]), .HEX(HEX5[6:0]));     //ALUout[7:4]
endmodule

module hexDecoder (SW, HEX);
    input [3:0] SW;
    output [6:0] HEX;

    assign HEX[0] = ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
            SW[3] & ~SW[2] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[1] = SW[2] & SW[1] & ~SW[0] |
            SW[3] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[2] = SW[3] & SW[2] & ~SW[0] |
            SW[3] & SW[2] & SW[1] |
            ~SW[3] & ~SW[2] & SW[1] & ~SW[0];

    assign HEX[3] = SW[2] & SW[1] & SW[0] |
            ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
            SW[3] & ~SW[2] & SW[1] & ~SW[0];

    assign HEX[4] = ~SW[3] & SW[0] |
            ~SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & ~SW[1] |
            ~SW[3] & SW[2] & ~SW[1] & SW[0] |
            ~SW[3] & SW[2] & SW[1] & SW[0] |
            SW[3] & ~SW[2] & ~SW[1] & SW[0];

    assign HEX[5] = ~SW[3] & ~SW[2] & SW[0] |
            ~SW[3] & ~SW[2] & SW[1] |
```

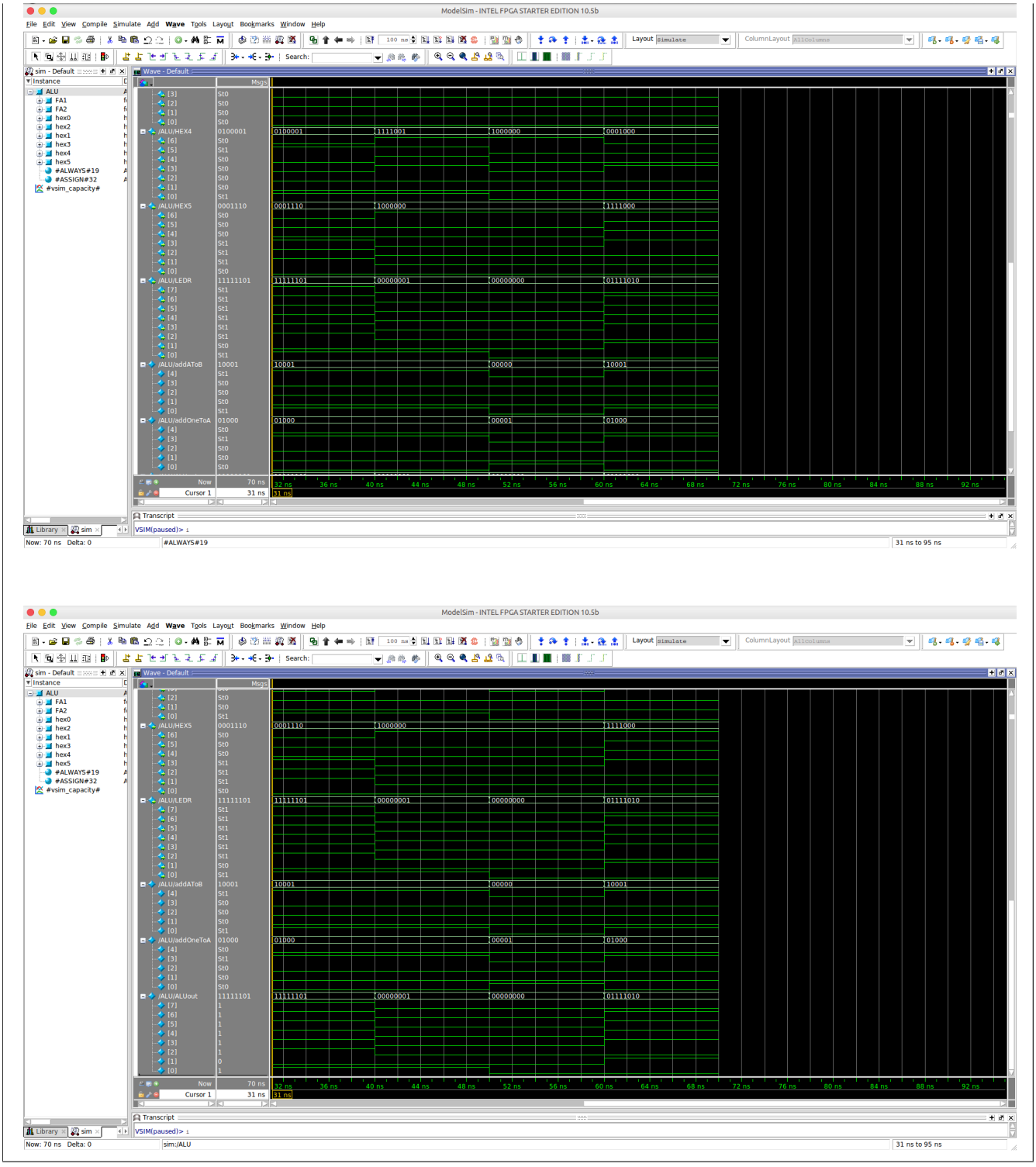
```
            ~SW[3] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & SW[0];

    assign HEX[6] = ~SW[3] & ~SW[2] & ~SW[1] |
            ~SW[3] & SW[2] & SW[1] & SW[0] |
            SW[3] & SW[2] & ~SW[1] & ~SW[0];
endmodule

module fourBitAdder(SW, LEDR);
    input [9:9] SW;
    output [4:0] LEDR;
    wire [2:0] cable;

    fullAdder FA1(
        .a(SW[4]),
        .b(SW[0]),
        .cin(SW[9]),
        .s(LEDR[0]),
        .cout(cable[0])
    );

    fullAdder FA2(
        .a(SW[5]),
        .b(SW[1]),
        .cin(cable[0]),
        .s(LEDR[1]),
        .cout(cable[1])
    );

    fullAdder FA3(
        .a(SW[6]),
        .b(SW[2]),
        .cin(cable[1]),
        .s(LEDR[2]),
        .cout(cable[2])
    );

    fullAdder FA4(
        .a(SW[7]),
        .b(SW[3]),
        .cin(cable[2]),
        .s(LEDR[3]),
        .cout(LEDR[4])
    );
endmodule

module fullAdder(a, b, cin, s, cout);
    input a;
    input b;
```

```
122     input cin;
123     output s;
124     output cout;
125
126     assign s = a^b^cin;
127     assign cout = (a & b) | (cin & (a^b));
128
129 endmodule
```

3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must include screenshots of output waveforms as part of your pre-lab.

Solution. The following are the screenshots for simulations:




ModelSim - INTEL FPGA STARTER EDITION 10.5b
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
sim - Default
Instance
ALU
FA1
FA2
hex0
hex2
hex1
hex3
hex4
hex5
#ALWAYS#19
#ASSIGN#32
#vsim_capacity#
Wave - Default
Msgs
/ALU/HEX4 0100001 0100001 1111001 1000000 0001000
/ALU/HEX5 0001110 0001110 1000000 1111000
/ALU/LEDR 11111101 11111101 00000001 00000000 01111010
/ALU/addAToB 10001 10001 00000 10001
/ALU/addOneToA 01000 01000 00001 01000
Now 70 ns
Cursor 1 31 ns
32 ns 36 ns 40 ns 44 ns 48 ns 52 ns 56 ns 60 ns 64 ns 68 ns 72 ns 76 ns 80 ns 84 ns 88 ns 92 ns
Transcript
VSIM(paused)>
Now: 70 ns Delta: 0
#ALWAYS#19
31 ns to 95 ns
ModelSim - INTEL FPGA STARTER EDITION 10.5b
/ALU/ALUout 11111101 11111101 00000001 00000000 01111010
sim:/ALU