

CSC258 PRELAB3

Tingfeng Xia (1003780884)

October 1, 2019

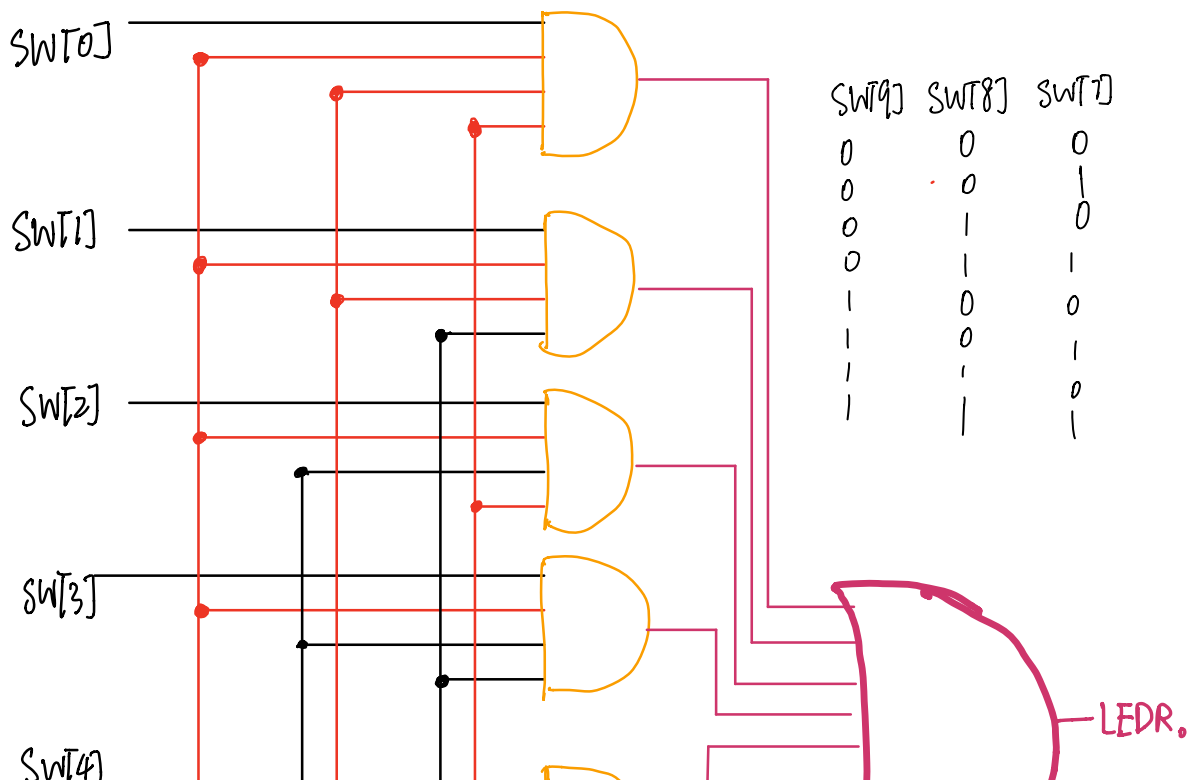
Using SW_{6-0} as the data inputs and SW_{9-7} as the select signals, display on $LEDR_0$ the output of a 7-to-1 multiplexer using the case statement style as shown above.

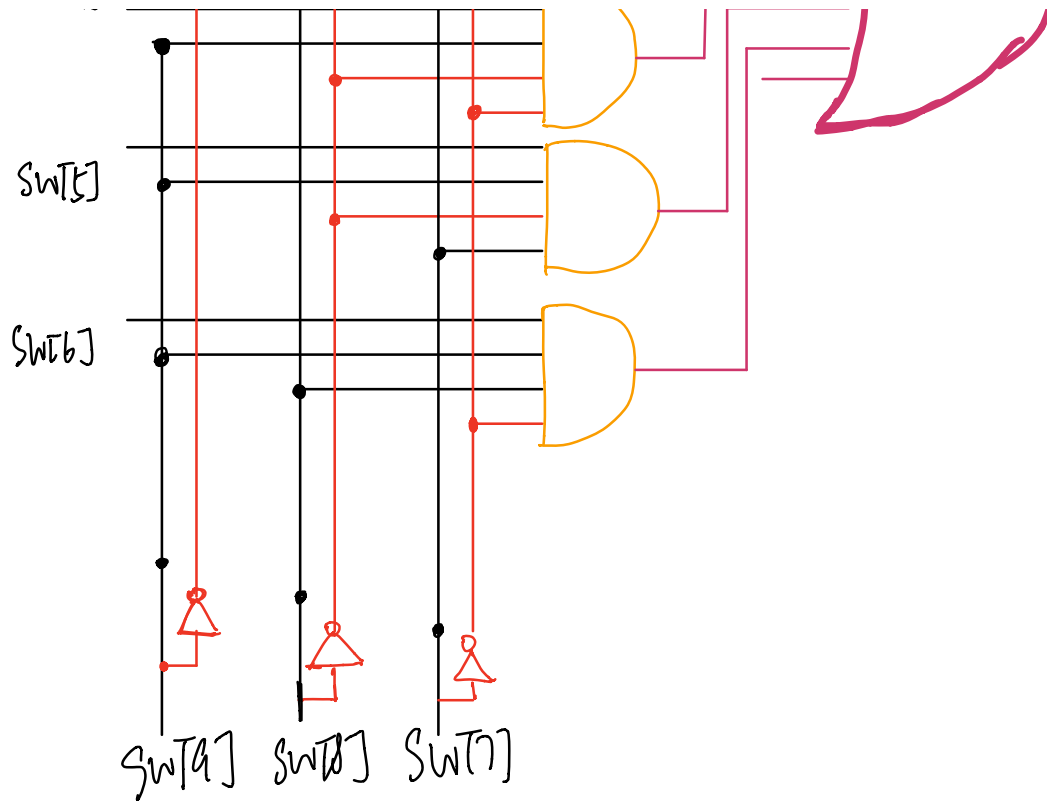
PART I

1. Draw a schematic showing your code structure with all wires, inputs and outputs labeled. Be prepared to explain it to the TA as part of your preparation. (PRELAB)
2. Write Verilog code for a 7-to-1 multiplexer, based on the template provided above. Use switches SW_{9-7} on the DE1-SoC board as the MuxSelect inputs and switches SW_{6-0} as the Input data inputs. Connect the output to $LEDR_0$. (PRELAB)
3. Simulate your circuit with ModelSim for different values of MuxSelect and Input. You must include a screenshot of the simulation output as part of your prelab. (PRELAB)
4. Create a new Quartus Prime project for your circuit. Make sure it is stored in your W:\ drive.
5. Compile the project.
6. In Quartus Prime, select Tools > Netlist Viewers > RTL Viewer and observe the circuit that got produced from your Verilog code. Show this to the TA when you demonstrate the functionality of your circuit to the TA. Note that Quartus uses slightly different symbol for the multiplexer than what you have seen in class: select inputs are denoted by name instead of being drawn from the top of the symbol. (IN-LAB)
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs. When you are sure that it works correctly, demonstrate the circuit behaviour to your TA. (IN-LAB)

1 Schematic of 7to1 multiplexer: inputs: SW_{6-0} select: SW_{9-7}
out: $LEDR_0$

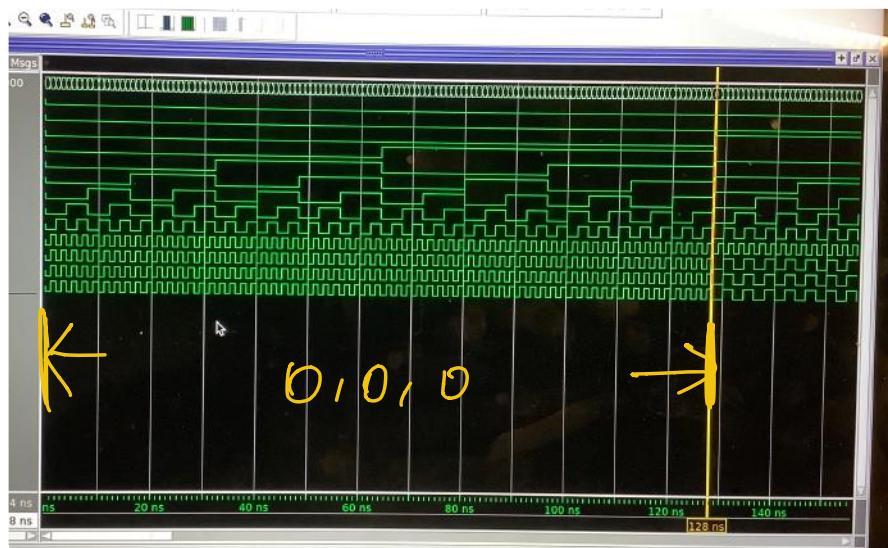
*idea: $2^3 = 8$ cases to consider, for each case, use and gates to specify the requirements; OR everything together to get final output.





2. Please see the appendix for max71.v

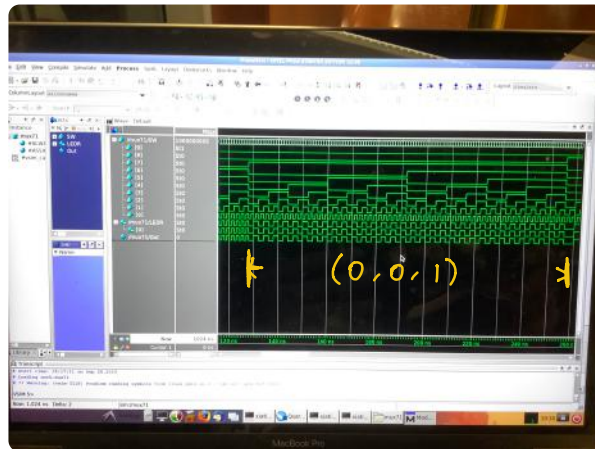
3. When control switches = (0, 0, 0)



output follows SW[0]

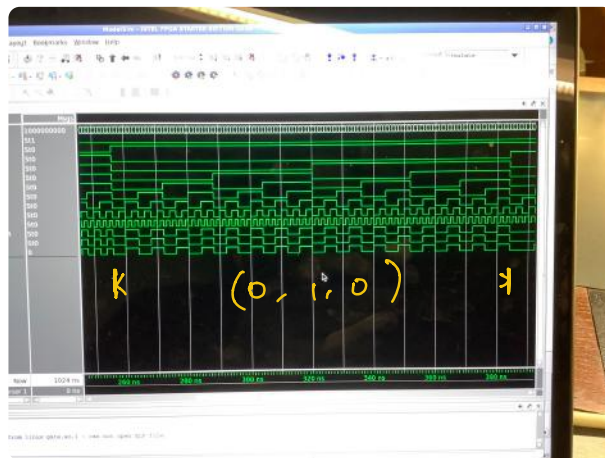
000

Case 3 switches are
 $(0, 0, 1)$, output LEDR₀
 follows SW12



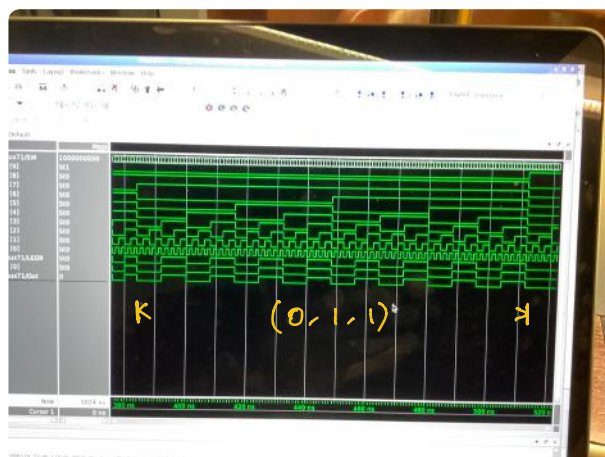
001

Case 3 switches are
 $(0, 1, 0)$, output LEDR₀
 follows SW13



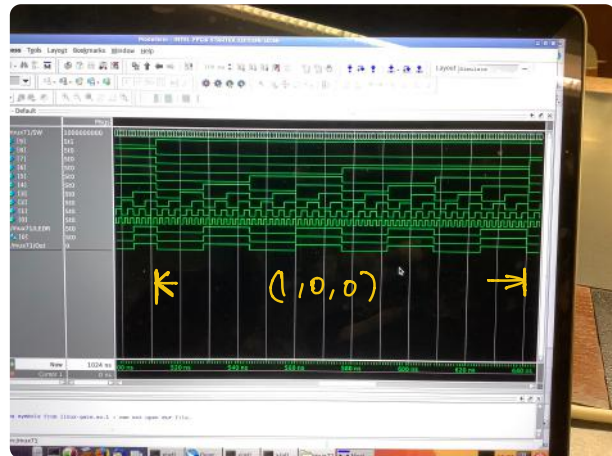
010

Case 3 switches are $(0, 1, 1)$
 output LEDR₀ follows SW13



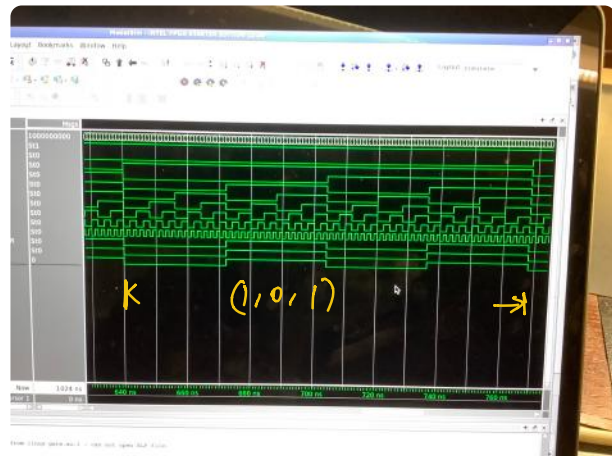
011

Case 3 switches are
 $(1,0,0)$. output LEDR₀
 follows SWT4]



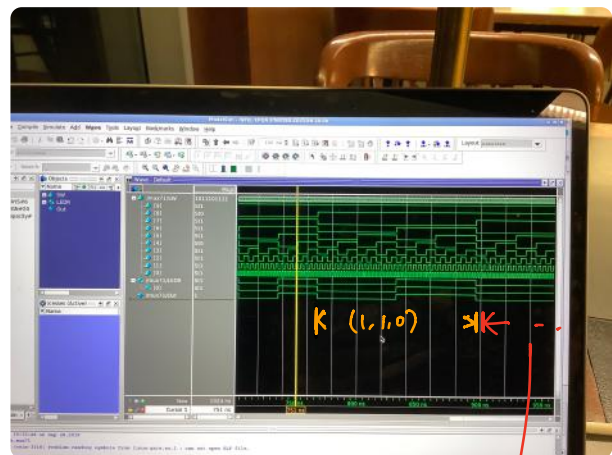
100

Case 3 switches are
 $(1,0,1)$. output LEDR₀
 follows SWT5]



101

Case 3 switches are $(1,1,0)$
 output LEDR₀ follows SWT6]

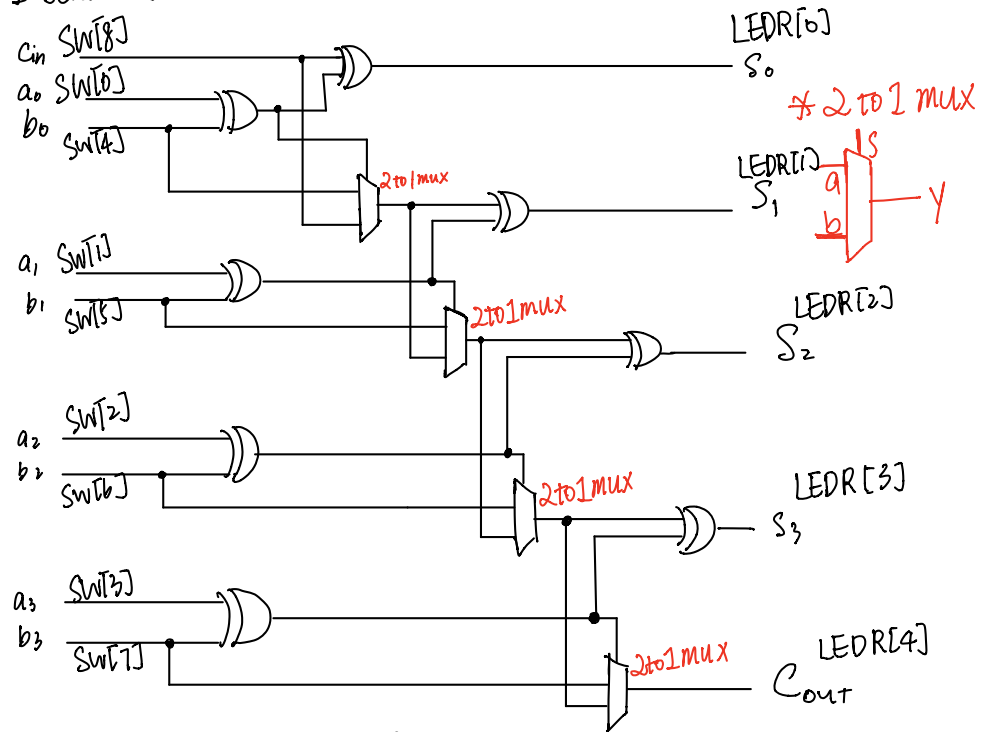


110

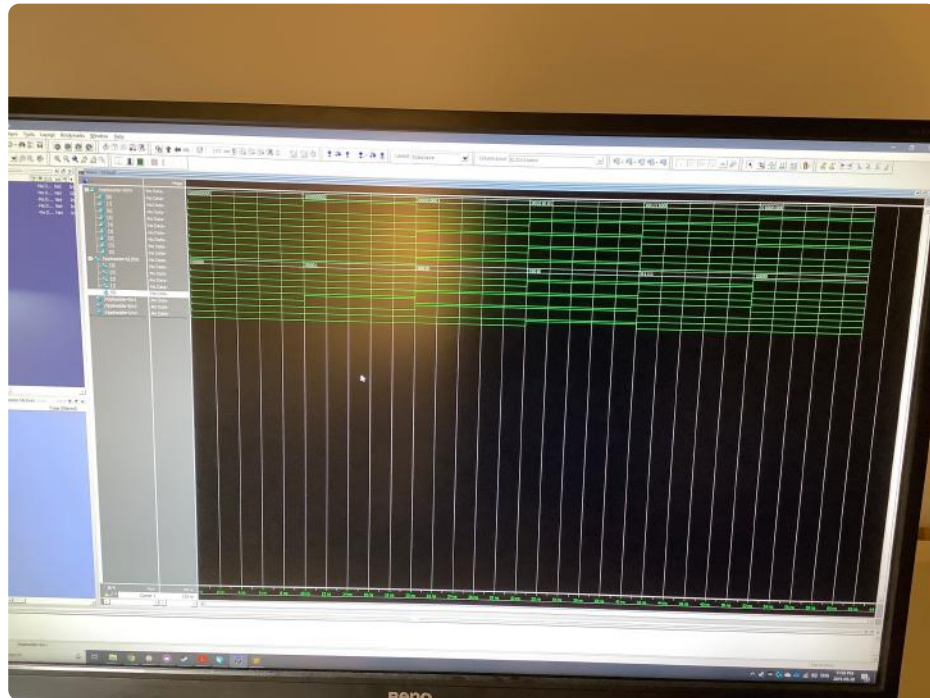
default

PART 2

1. Schematic

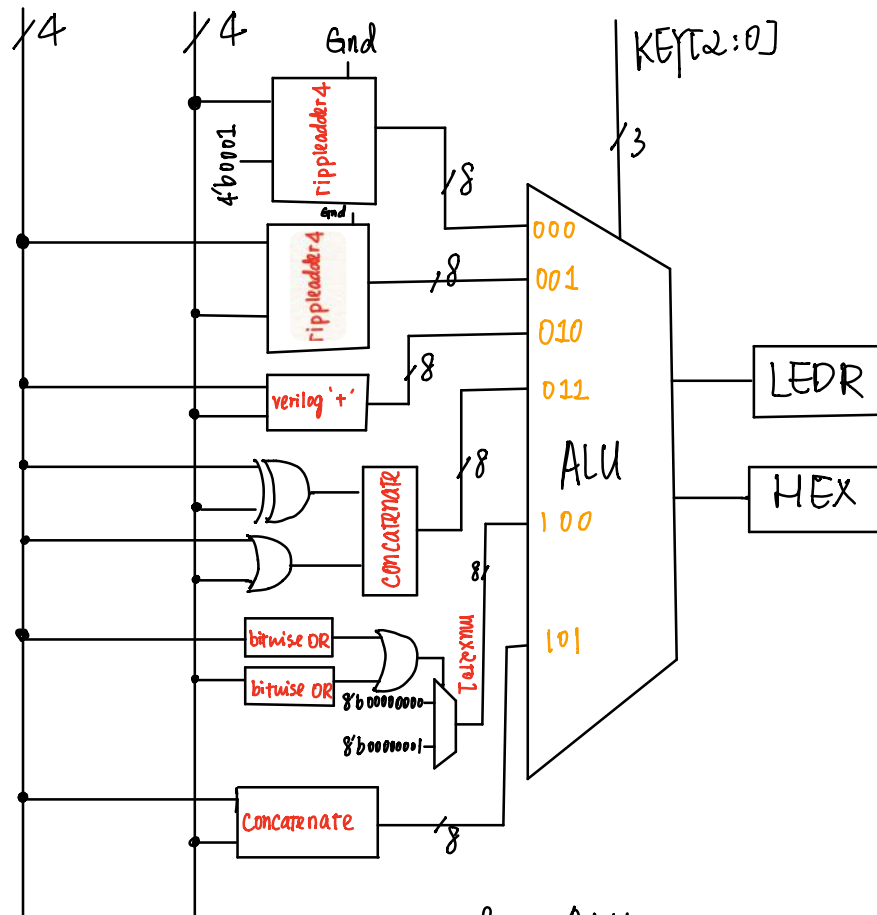


2. please check appendix for rippleadder4.v
3. please check appendix for rippleadder4.clo for test cases that I used. Here's a picture of the result.



PART 3 1. Schematic

Input B Input A
SW[3:0] SW[7:4]



2. Please Check appendix for ALU.v

3. please check appendix for ALU.do ; here are the screen shots of the simulation outcome .

.1 Appendix rippleadder.v

```

module rippleadder4(SW, LEDR);
    // SW[3:0] number 1
    // SW[7:4] number 2
    // SW[8:8] carry initial
    input [8:0] SW;

    output [4:0] LEDR; // 4 bit result, one bit carry
    // connecting the four full adders
    wire w1;
    wire w2;
    wire w3;

    fulladder f1(
        .cin(SW[8]),
        .a(SW[4]),
        .b(SW[0]),
        .cout(w1),
        .s(LEDR[0])
    );

    fulladder f2(
        .cin(w1),
        .a(SW[5]),
        .b(SW[1]),
        .cout(w2),
        .s(LEDR[1])
    );

    fulladder f3(
        .cin(w2),
        .a(SW[6]),
        .b(SW[2]),
        .cout(w3),
        .s(LEDR[2])
    );

    fulladder f4(
        .cin(w3),
        .a(SW[7]),
        .b(SW[3]),
        .cout(LEDR[4]),
        .s(LEDR[3])
    );

endmodule

// full adder
module fulladder(cin, a, b, s, cout);
    //     input a;
    //     input b;
    //     input cin;
    //     output s;
    //     output cout;
    //

```

```

//      assign s = a^b^cin;
//      assign cout = (a & b) | (cin & (a^b));
input cin;
input a;
input b;
output cout;
output s;

wire w1;

mux2to1 mux(
    .x(b),
    .y(cin),
    .s(w1),
    .m(cout)
);

XOR x1(
    .a(a),
    .b(b),
    .f(w1)
);

XOR x2(
    .a(cin),
    .b(w1),
    .f(s)
);
endmodule

// define a XOR module
module XOR(a, b, f);
    input a;
    input b;
    output f;
    assign f = a ^ b;
endmodule

// mux2to1 from lab2
module mux2to1(x, y, s, m);
    input x; //selected when s is 0
    input y; //selected when s is 1
    input s; //select signal
    output m; //output

    assign m = s & y | ~s & x;
endmodule

```

.2 Appendix rippleadder.do

```

# Set the working dir, where all compiled Verilog goes.
vlib work

# Compile all Verilog modules in mux.v to working dir;
# could also have multiple Verilog files.
# The timescale argument defines default time unit

```

```

# (used when no unit is specified), while the second number
# defines precision (all times are rounded to this value)
vlog -timescale 1ns/1ns rippleadder4.v

# Load simulation using mux as the top level simulation module.
vsim rippleadder4

# Log all signals and add some signals to waveform window.
log {/*}
# add wave {/*} would add all items in top level simulation module.
add wave {/*}

# 0000 + 0000 = 0000 simple case
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0
force {SW[8]} 0
run 10ns

# 0000 + 0000 with initial carry
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0
force {SW[8]} 1
run 10ns

# 0001 + 0001 = 0010 (with no ini carry)
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0
force {SW[8]} 0
run 10ns

# 0101 + 0101 = 1010 no overflow
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 1
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 0
force {SW[6]} 1

```

```

force {SW[7]} 0
force {SW[8]} 0
run 10ns

# 1000 + 0111 = 1111 no overflow
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 1
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 0
force {SW[8]} 0
run 10ns

# 1000 + 1000 = (1)0000 overflow
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 1
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 1
force {SW[8]} 0
run 10ns

```

.3 Appendix mux7.v

```

module mux71(SW, LEDR);
    input [9:0] SW;
    output [0:0] LEDR;

    reg Out;
    always @(*)
    begin
        case (SW[9:7])
            3'b000: Out = SW[0];
            3'b001: Out = SW[1];
            3'b010: Out = SW[2];
            3'b011: Out = SW[3];
            3'b100: Out = SW[4];
            3'b101: Out = SW[5];
            3'b110: Out = SW[6];
            default: Out = 1'b0; // fall back
        endcase
    end

    assign LEDR[0] = Out; // Assign the outputs

endmodule;

```

.4 Appendix mux7.do

```
# Set the working dir, where all compiled Verilog goes.
vlib work

# Compile all Verilog modules in mux.v to working dir;
# could also have multiple Verilog files.
# The timescale argument defines default time unit
# (used when no unit is specified), while the second number
# defines precision (all times are rounded to this value)
vlog -timescale 1ns/1ns mux71.v

# Load simulation using mux as the top level simulation module.
vsim mux71

# Log all signals and add some signals to waveform window.
log {/*}
# add wave {/*} would add all items in top level simulation module.
add wave {/*}

force {SW[9]} 0 0, 1 512
force {SW[8]} 0 0, 1 256 -r 512
force {SW[7]} 0 0, 1 128 -r 256
force {SW[6]} 0 0, 1 64 -r 128
force {SW[5]} 0 0, 1 32 -r 64
force {SW[4]} 0 0, 1 16 -r 32
force {SW[3]} 0 0, 1 8 -r 16
force {SW[2]} 0 0, 1 4 -r 8
force {SW[1]} 0 0, 1 2 -r 4
force {SW[0]} 0 0, 1 1 -r 2
run 1024ns
```

.5 Appendix ALU.v

```
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;
    // two wires for arithmetic operations
    wire [4:0] addOneToA;
    wire [4:0] addAToB;

    // set hex1 and hex3 to zero
    assign HEX1[6:0] = 7'b0000001;
    assign HEX3[6:0] = 7'b0000001;

    // two 4 bit ripple adders
    rippleadder4 ra1(
        .SW({1'b0, SW[7:4]}, 4'b0001),
        .LEDR(addOneToA[4:0])
    );
endmodule
```

```

);
rippleadder4 ra2(
    .SW({1'b0, SW[7:4], SW[3:0]}),
    .LEDR(addAToB[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])}; // bitwise or the 8 inputs
        3'b101: ALUout[7:0] = SW[7:0];
        default: ALUout[7:0] = 8'b0000_0000;
    endcase
end

assign LEDR[7:0] = ALUout[7:0];

// HEX0 and HEX2 shows B and A respectively
hexdecoder hex0(
    .SW(SW[3:0]),
    .HEX(HEX0[6:0])
);
hexdecoder hex2(
    .SW(SW[7:4]),
    .HEX(HEX2[6:0])
);

hexdecoder hex4(
    .SW(ALUout[3:0]),
    .HEX(HEX4[6:0])
); //First four bits
hexdecoder hex5(
    .SW(ALUout[7:4]),
    .HEX(HEX5[6:0])
); //Second four bits
endmodule

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

    hex0 u0(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[0])
    );

    hex1 u1(
        .x(SW[3]),

```



```

        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[1])
    );

    hex2 u2(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[2])
    );

    hex3 u3(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[3])
    );

    hex4 u4(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[4])
    );

    hex5 u5(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[5])
    );

    hex6 u6(
        .x(SW[3]),
        .y(SW[2]),
        .z(SW[1]),
        .w(SW[0]),
        .m(HEX[6])
    );

endmodule

module hex0(x, y, z, w, m);
    input x;
    input y;
    input z;
    input w;
    output m;

    assign m = (~x & ~y & ~z & w) | (~x & y & ~z & ~w) | (x & y & ~z & w) | (x & ~y & z & w);

```

```
endmodule
```

```
module hex1(x, y, z, w, m);
```

```
    input x;
```

```
    input y;
```

```
    input z;
```

```
    input w;
```

```
    output m;
```

```
    assign m = (~x & y & ~z & w) | (x & z & w) | (y & z & ~w) | (x & y & ~w);
```

```
endmodule
```

```
module hex2(x, y, z, w, m);
```

```
    input x;
```

```
    input y;
```

```
    input z;
```

```
    input w;
```

```
    output m;
```

```
    assign m = (x & y & ~w) | (x & y & z) | (~x & ~y & z & ~w);
```

```
endmodule
```

```
module hex3(x, y, z, w, m);
```

```
    input x;
```

```
    input y;
```

```
    input z;
```

```
    input w;
```

```
    output m;
```

```
    assign m = (~x & y & ~z & ~w) | (~x & ~y & ~z & w) | (y & z & w) | (x & ~y & z & ~w);
```

```
endmodule
```

```
module hex4(x, y, z, w, m);
```

```
    input x;
```

```
    input y;
```

```
    input z;
```

```
    input w;
```

```
    output m;
```

```
    assign m = (~x & w) | (~y & ~z & w) | (~x & y & ~z);
```

```
endmodule
```

```
module hex5(x, y, z, w, m);
```

```
    input x;
```

```
    input y;
```

```
    input z;
```

```
    input w;
```

```
    output m;
```

```
    assign m = (~x & ~y & w) | (~x & ~y & z) | (~x & z & w) | (x & y & ~z & w);
```

```

endmodule

module hex6(x, y, z, w, m);
    input x;
    input y;
    input z;
    input w;
    output m;

    assign m = (~x & ~y & ~z) | (~x & y & z & w) | (x & y & ~z & ~w);

endmodule

module rippleadder4(SW, LEDR);
    // SW[3:0] number 1
    // SW[7:4] number 2
    // SW[8:8] carry initial
    input [8:0] SW;

    output [4:0] LEDR; // 4 bit result, one bit carry
    // connecting the four full adders
    wire w1;
    wire w2;
    wire w3;

    fulladder f1(
        .cin(SW[8]),
        .a(SW[4]),
        .b(SW[0]),
        .cout(w1),
        .s(LEDR[0])
    );

    fulladder f2(
        .cin(w1),
        .a(SW[5]),
        .b(SW[1]),
        .cout(w2),
        .s(LEDR[1])
    );

    fulladder f3(
        .cin(w2),
        .a(SW[6]),
        .b(SW[2]),
        .cout(w3),
        .s(LEDR[2])
    );

    fulladder f4(
        .cin(w3),
        .a(SW[7]),
        .b(SW[3]),
        .cout(LEDR[4]),
        .s(LEDR[3])
    );

```

```

    );

endmodule

// full adder
module fulladder(cin, a, b, s, cout);
    //      input a;
    //      input b;
    //      input cin;
    //      output s;
    //      output cout;
    //
    //      assign s = a^b^cin;
    //      assign cout = (a & b) | (cin & (a^b));
    input cin;
    input a;
    input b;
    output cout;
    output s;

    wire w1;

    mux2to1 mux(
        .x(b),
        .y(cin),
        .s(w1),
        .m(cout)
    );

    XOR x1(
        .a(a),
        .b(b),
        .f(w1)
    );

    XOR x2(
        .a(cin),
        .b(w1),
        .f(s)
    );
endmodule

// define a XOR module
module XOR(a, b, f);
    input a;
    input b;
    output f;
    assign f = a ^ b;
endmodule

// mux2to1 from lab2
module mux2to1(x, y, s, m);
    input x; //selected when s is 0
    input y; //selected when s is 1
    input s; //select signal
    output m; //output

```

```

    assign m = s & y | ~s & x;
endmodule

```

.6 Appendix ALU.do

```

# Set the working dir, where all compiled Verilog goes.
vlib work

# Compile all Verilog modules in mux.v to working dir;
# could also have multiple Verilog files.
# The timescale argument defines default time unit
# (used when no unit is specified), while the second number
# defines precision (all times are rounded to this value)
vlog -timescale 1ns/1ns ALU.v

# Load simulation using mux as the top level simulation module.
vsim ALU

# Log all signals and add some signals to waveform window.
log {/*}
# add wave {/*} would add all items in top level simulation module.
add wave {/*}

# 000, make the output equal to plus one, 0111 + 1 = 1000
force {KEY[2]} 0
force {KEY[1]} 0
force {KEY[0]} 0
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 0
run 10ns

# 000, make the output equal to plus one, 1111 + 1 = 10000. overflow by one bit
force {KEY[2]} 0
force {KEY[1]} 0
force {KEY[0]} 0
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

# 001, A+B. 0000+1111 = 1111
force {KEY[2]} 0
force {KEY[1]} 0
force {KEY[0]} 1

```

```

force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

```

```

# 001, A+B. 0101+0001 = 0110
force {KEY[2]} 0
force {KEY[1]} 0
force {KEY[0]} 1
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 0
force {SW[6]} 1
force {SW[7]} 0
run 10ns

```

```

# 010 verilog + operator 0000+1111 = 1111, same as case 1 in 001
force {KEY[2]} 0
force {KEY[1]} 0
force {KEY[0]} 1
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

```

```

# 010 verilog + operator, 0101 + 0001 = 0110, same as case 2 in 001
force {KEY[2]} 0
force {KEY[1]} 1
force {KEY[0]} 0
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 0
force {SW[6]} 1
force {SW[7]} 0
run 10ns

```

```

# 011, XOR in the lower bits and OR in higher bits, 1000 and 0111, all one
force {KEY[2]} 0
force {KEY[1]} 1
force {KEY[0]} 1

```



```

force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

```

```

# 011, XOR in the lower bits and OR in higher bits, 0101 and 1111, OR = 1111 XOR=1010
force {KEY[2]} 0
force {KEY[1]} 1
force {KEY[0]} 1
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 1
force {SW[3]} 0
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

```

```

# 100 case where all zero, 0000_0000
force {KEY[2]} 1
force {KEY[1]} 0
force {KEY[0]} 0
force {SW[0]} 0
force {SW[1]} 0
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0
run 10ns

```

```

# 100 case where all one, 0000_0001
force {KEY[2]} 1
force {KEY[1]} 0
force {KEY[0]} 0
force {SW[0]} 1
force {SW[1]} 1
force {SW[2]} 1
force {SW[3]} 1
force {SW[4]} 1
force {SW[5]} 1
force {SW[6]} 1
force {SW[7]} 1
run 10ns

```

```

# 100 case where some one, 0000_0001
force {KEY[2]} 1
force {KEY[1]} 0
force {KEY[0]} 0

```

```
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 1
force {SW[3]} 1
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 1
force {SW[7]} 0
run 10ns
```

```
# 101 case appear input, 01001101
force {KEY[2]} 1
force {KEY[1]} 0
force {KEY[0]} 1
force {SW[0]} 1
force {SW[1]} 0
force {SW[2]} 1
force {SW[3]} 1
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 1
force {SW[7]} 0
run 10ns
```

```
# 101 case appear input, 1100_1110
force {KEY[2]} 1
force {KEY[1]} 0
force {KEY[0]} 1
force {SW[0]} 0
force {SW[1]} 1
force {SW[2]} 1
force {SW[3]} 1
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 1
force {SW[7]} 1
run 10ns
```