San Francisco Bay University

EE461 Verilog-HDL
Homework #4

Due day: 11/17/2024

Instruction:

1. **Push answer sheets/source code to Github**
2. **Please follow the code style rule like programs on handout.**
3. **Overdue homework submission could not be accepted.**
4. **Takes academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)**

1. Design a detector in RTL level to detect if all bits are *0*s or all *1*s for an 8-bits input. If all bits are *0*s, one of two outputs, "*zeroflag*" is *1*. If all are *1*s, the other of two outputs, "*oneflag*" is *1*. After that, write the gate level module to compare two designs with the testbench.

RTL level

```
module detector_8bit(input [7:0] data_in, output reg zeroflag, oneflag);
   always @(*) begin
      zeroflag = (data_in == 8'b00000000);  // Check if all bits are 0
      oneflag  = (data_in == 8'b11111111);  // Check if all bits are 1
   end
endmodule
```

Gate Level

```
module detector_gate(input [7:0] data_in, output zeroflag, oneflag);
   wire [7:0] not_data;  // Inverted input

   // Invert the input for zeroflag detection
   not u1 (not_data[0], data_in[0]);
   not u2 (not_data[1], data_in[1]);
   not u3 (not_data[2], data_in[2]);
   not u4 (not_data[3], data_in[3]);
   not u5 (not_data[4], data_in[4]);
   not u6 (not_data[5], data_in[5]);
   not u7 (not_data[6], data_in[6]);
   not u8 (not_data[7], data_in[7]);

   // AND all bits for zeroflag
   and g1 (zeroflag, not_data[0], not_data[1], not_data[2], not_data[3],
               not_data[4], not_data[5], not_data[6], not_data[7]);
```

```
    // AND all bits for oneflag
    and g2 (oneflag, data_in[0], data_in[1], data_in[2], data_in[3],
                data_in[4], data_in[5], data_in[6], data_in[7]);
endmodule
```

Testbench

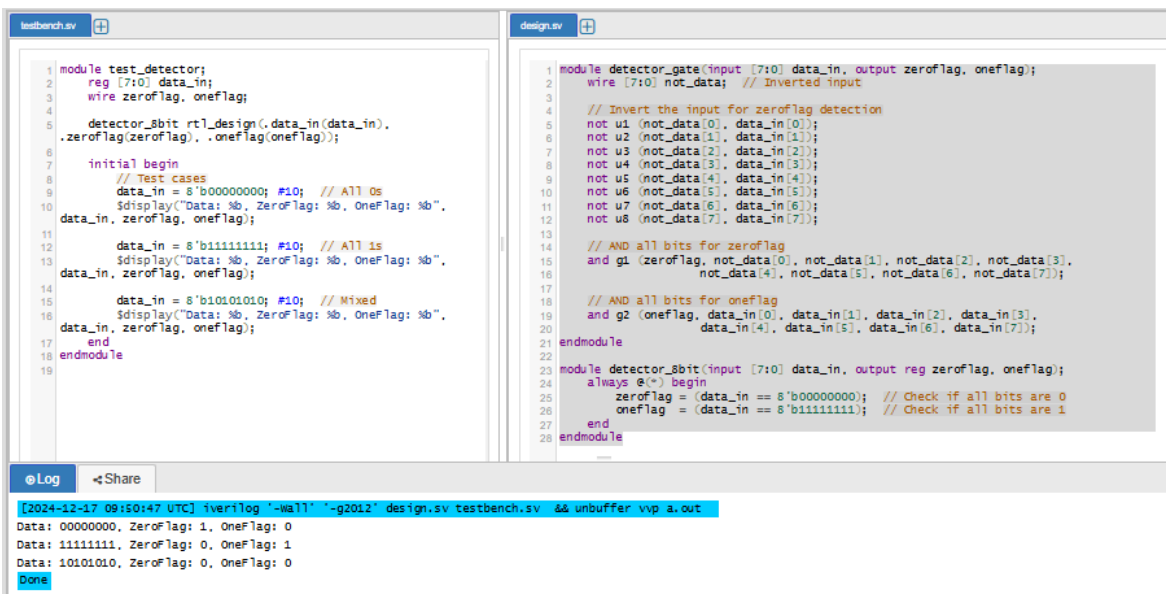
```
module test_detector;
    reg [7:0] data_in;
    wire zeroflag, oneflag;

    detector_8bit rtl_design(.data_in(data_in), .zeroflag(zeroflag), .oneflag(oneflag));

    initial begin
        // Test cases
        data_in = 8'b00000000; #10;  // All 0s
        $display("Data: %b, ZeroFlag: %b, OneFlag: %b", data_in, zeroflag, oneflag);

        data_in = 8'b11111111; #10;  // All 1s
        $display("Data: %b, ZeroFlag: %b, OneFlag: %b", data_in, zeroflag, oneflag);

        data_in = 8'b10101010; #10;  // Mixed
        $display("Data: %b, ZeroFlag: %b, OneFlag: %b", data_in, zeroflag, oneflag);
    end
endmodule
```

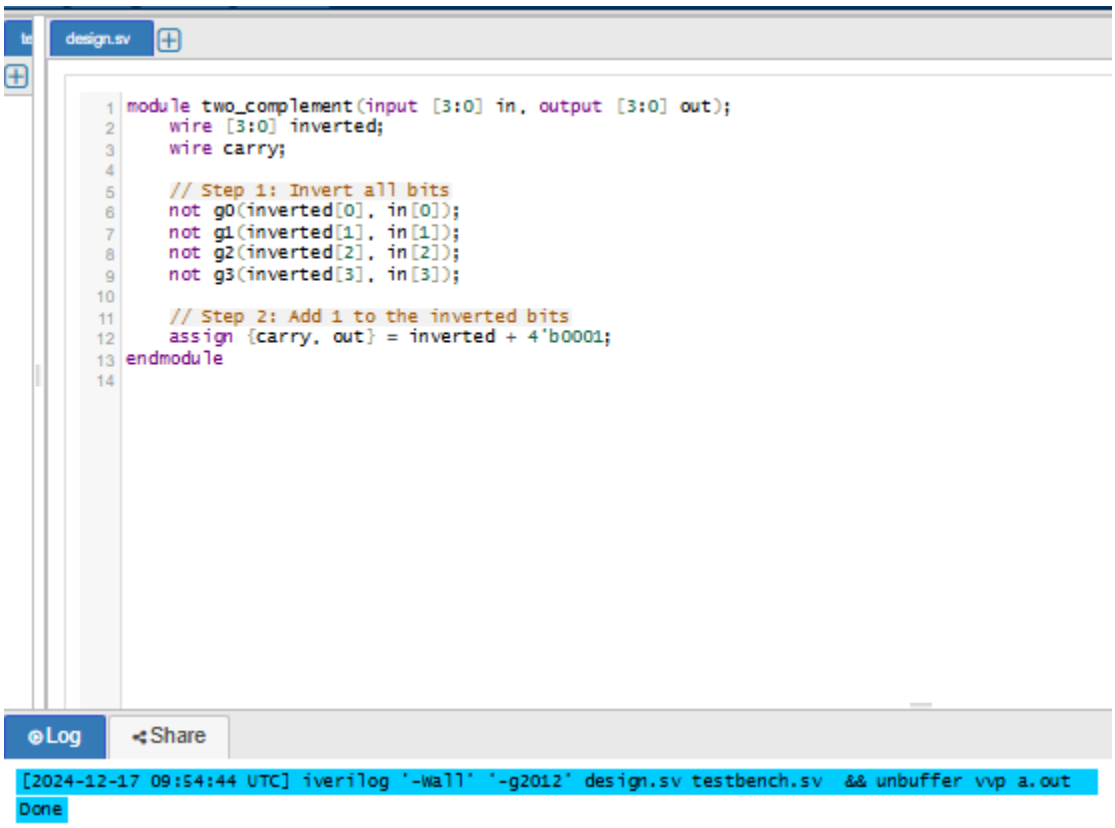


2. Design 4-bits 2's complement number converter in the gate level module

```
module two_complement(input [3:0] in, output [3:0] out);
    wire [3:0] inverted;
    wire carry;

    // Step 1: Invert all bits
    not g0(inverted[0], in[0]);
    not g1(inverted[1], in[1]);
    not g2(inverted[2], in[2]);
    not g3(inverted[3], in[3]);

    // Step 2: Add 1 to the inverted bits
    assign {carry, out} = inverted + 4'b0001;
endmodule
```



3. Design 4-1 mux in continuous assign. Write a testbench to assign *x* & *z* to select-bits to observe what you are going to get.

MODULE

```
module mux4to1(input [3:0] data_in, input [1:0] sel, output out);
   assign out = (sel == 2'b00) ? data_in[0] :
                (sel == 2'b01) ? data_in[1] :
                (sel == 2'b10) ? data_in[2] :
                                 data_in[3];
endmodule
```

TESTBENCH

```
module test_mux4to1;
   reg [3:0] data_in;
   reg [1:0] sel;
   wire out;

   mux4to1 uut(.data_in(data_in), .sel(sel), .out(out));

   initial begin
      data_in = 4'b1010;
      sel = 2'b00; #10;
      $display("Sel: %b, Output: %b", sel, out);

      sel = 2'b01; #10;
      $display("Sel: %b, Output: %b", sel, out);

      sel = 2'b10; #10;
      $display("Sel: %b, Output: %b", sel, out);

      sel = 2'b11; #10;
      $display("Sel: %b, Output: %b", sel, out);
   end
endmodule
```

testbench.sv

```
module test_mux4to1;
    reg [3:0] data_in;
    reg [1:0] sel;
    wire out;

    mux4to1 uut(.data_in(data_in),
.sel(sel), .out(out));

    initial begin
        data_in = 4'b1010;
        sel = 2'b00; #10;
        $display("Sel: %b, Output: %b",
sel, out);

        sel = 2'b01; #10;
        $display("Sel: %b, Output: %b",
sel, out);

        sel = 2'b10; #10;
        $display("Sel: %b, Output: %b",
sel, out);

        sel = 2'b11; #10;
        $display("Sel: %b, Output: %b",
sel, out);
    end
endmodule
```

design.sv

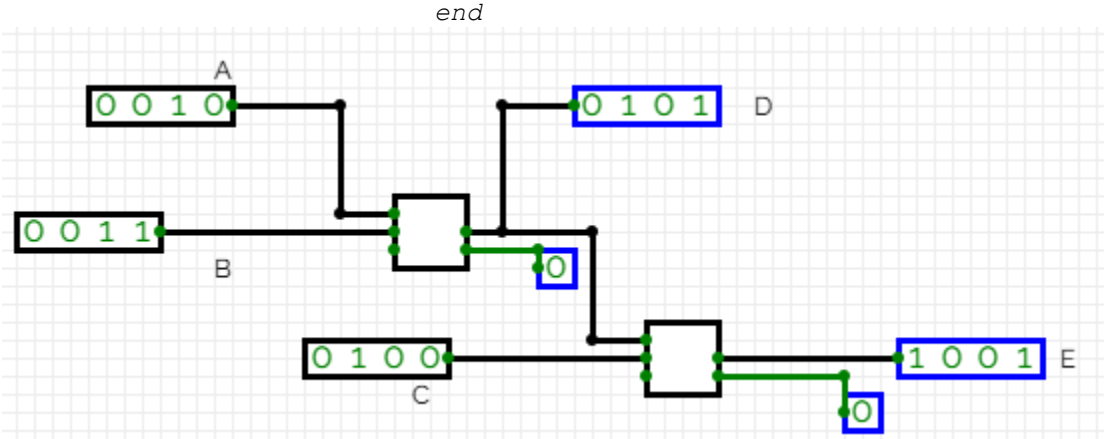
```
module mux4to1(input [3:0] data_in, input [1:0] sel, output out);
    assign out = (sel == 2'b00) ? data_in[0] :
                 (sel == 2'b01) ? data_in[1] :
                 (sel == 2'b10) ? data_in[2] :
                                  data_in[3];
endmodule
```

Log | Share

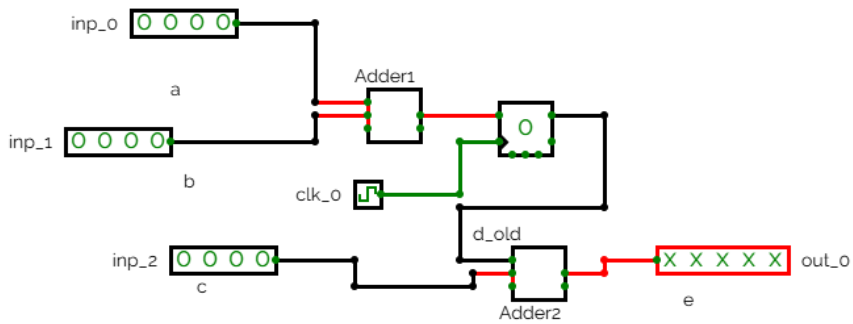
```
[2024-12-17 10:38:29 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Sel: 00, Output: 0
Sel: 01, Output: 1
Sel: 10, Output: 0
Sel: 11, Output: 1
Done
```

4. Draw the circuit schematics for the following two always block and compare what the difference is.

```
a. always @(a, b, c) begin
       d = a + b;
       e = d + c;
   end
```




```
b. always @(a, b, c) begin
       e = d + c;
       d = a + b;
   end
```



5. When designing a 5-to-1 mux by "case" structure, inferred latch will be generated if design isn't in proper way after logic synthesis. Please fix it and compare the new design hardware schematic with original one.

```
module mux(a,b,c,d,e,sel,out);
        input a,b,c,d,e;
        input[2:0] sel;
        output out;
        reg     out;

       always @(a, b, c, d, e, sel) begin
            case(sel)
                3'b000: out=a;
                3'b001: out=b;
                3'b010: out=c;
                3'b011: out=d;
                3'b100: out=e;
             endcase
       end
endmodule
```

To avoid the latch, a default case can be included. Since a latch is inferred when a variable (out) is not assigned under all conditions, adding a default makes sure that out always gets a value.

FIXED CODE

```
module mux(a,b,c,d,e,sel,out);
input a,b,c,d,e;
   input[2:0] sel;
   output out;
                    reg   out;

always @(a, b, c, d, e, sel) begin
   case(sel)
      3'b000: out = a;
      3'b001: out = b;
      3'b010: out = c;
      3'b011: out = d;
      3'b100: out = e;
```

```
        default: out = 1'b0;  // Default case to prevent latch
    endcase
end

  endmodule
```