



# 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 0s or all 1s for an 8-bits input. If all bits are 0s, one of two outputs, "zeroflag" is 1. If all are 1s, 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

```

The screenshot shows a Verilog IDE with two files: `testbench.sv` and `design.sv`. The `testbench.sv` file contains a testbench module that instantiates the `detector_8bit` module and applies three test cases: all 0s, all 1s, and a mixed pattern (10101010). The `design.sv` file contains the implementation of the `detector_8bit` module, which uses inverters to create `not_data` signals, then uses `and` gates to calculate the `zeroflag` (all bits are 0) and `oneflag` (all bits are 1). The log window at the bottom shows the simulation results for the three test cases.

```

testbench.sv
1 module test_detector;
2   reg [7:0] data_in;
3   wire zeroflag, oneflag;
4
5   detector_8bit rtl_design(.data_in(data_in),
6     .zeroflag(zeroflag), .oneflag(oneflag));
7
8   initial begin
9     // Test cases
10    data_in = 8'b00000000; #10; // All 0s
11    $display("Data: %b, ZeroFlag: %b, OneFlag: %b",
12      data_in, zeroflag, oneflag);
13
14    data_in = 8'b11111111; #10; // All 1s
15    $display("Data: %b, ZeroFlag: %b, OneFlag: %b",
16      data_in, zeroflag, oneflag);
17
18    data_in = 8'b10101010; #10; // Mixed
19    $display("Data: %b, ZeroFlag: %b, OneFlag: %b",
20      data_in, zeroflag, oneflag);
21  end
22 endmodule
23
design.sv
1 module detector_8bit(input [7:0] data_in, output zeroflag, oneflag);
2   wire [7:0] not_data; // Inverted input
3
4   // Invert the input for zeroflag detection
5   not u1 (not_data[0], data_in[0]);
6   not u2 (not_data[1], data_in[1]);
7   not u3 (not_data[2], data_in[2]);
8   not u4 (not_data[3], data_in[3]);
9   not u5 (not_data[4], data_in[4]);
10  not u6 (not_data[5], data_in[5]);
11  not u7 (not_data[6], data_in[6]);
12  not u8 (not_data[7], data_in[7]);
13
14  // AND all bits for zeroflag
15  and g1 (zeroflag, not_data[0], not_data[1], not_data[2], not_data[3],
16    not_data[4], not_data[5], not_data[6], not_data[7]);
17
18  // AND all bits for oneflag
19  and g2 (oneflag, data_in[0], data_in[1], data_in[2], data_in[3],
20    data_in[4], data_in[5], data_in[6], data_in[7]);
21 endmodule
22
23 module detector_8bit(input [7:0] data_in, output reg zeroflag, oneflag);
24   always @(*) begin
25     zeroflag = (data_in == 8'b00000000); // Check if all bits are 0
26     oneflag = (data_in == 8'b11111111); // Check if all bits are 1
27   end
28 endmodule

```

Log

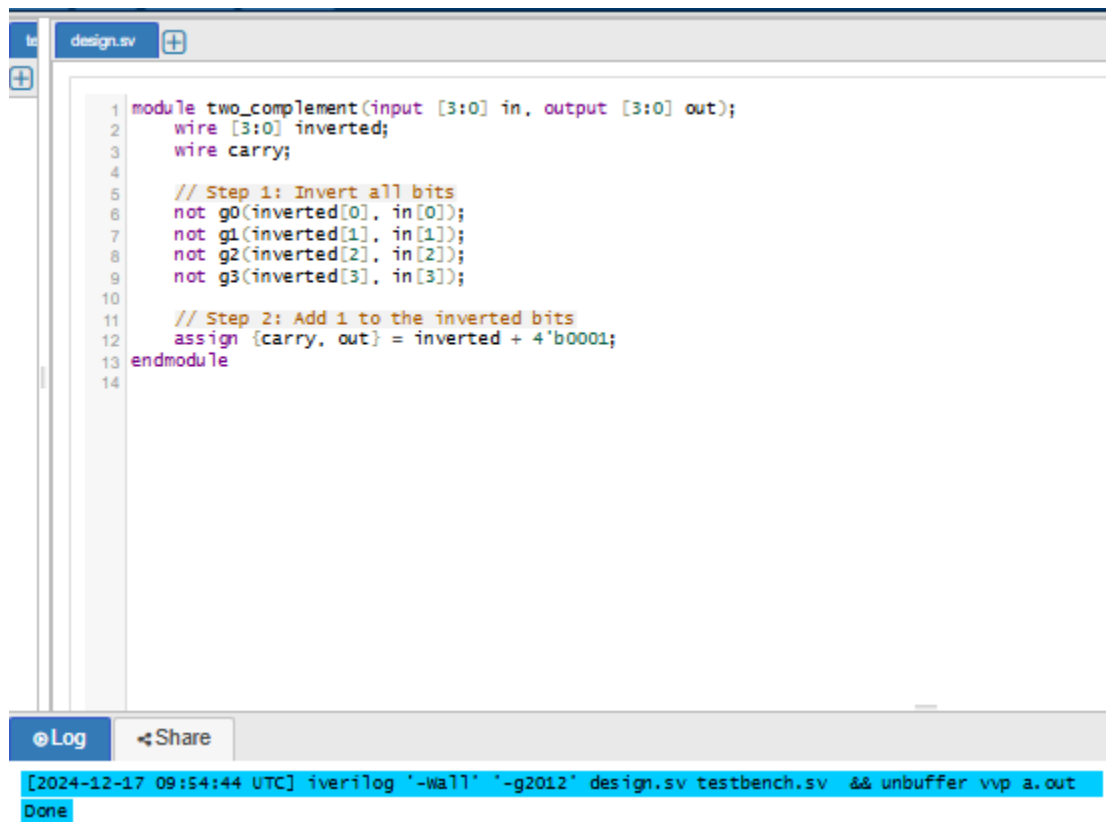
```

[2024-12-17 09:50:47 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
Data: 00000000, ZeroFlag: 1, OneFlag: 0
Data: 11111111, ZeroFlag: 0, OneFlag: 1
Data: 10101010, ZeroFlag: 0, OneFlag: 0
Done

```

## 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
```



```
1 module two_complement(input [3:0] in, output [3:0] out);  
2     wire [3:0] inverted;  
3     wire carry;  
4  
5     // Step 1: Invert all bits  
6     not g0(inverted[0], in[0]);  
7     not g1(inverted[1], in[1]);  
8     not g2(inverted[2], in[2]);  
9     not g3(inverted[3], in[3]);  
10  
11     // Step 2: Add 1 to the inverted bits  
12     assign {carry, out} = inverted + 4'b0001;  
13 endmodule  
14
```

Log Share

[2024-12-17 09:54:44 UTC] iverilog '-wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out  
Done

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
```

```

1 module test_mux4to1;
2   reg [3:0] data_in;
3   reg [1:0] sel;
4   wire out;
5
6   mux4to1 uut(.data_in(data_in),
7               .sel(sel), .out(out));
8
9   initial begin
10    data_in = 4'b1010;
11    sel = 2'b00; #10;
12    $display("Sel: %b, Output: %b",
13            sel, out);
14    sel = 2'b01; #10;
15    $display("Sel: %b, Output: %b",
16            sel, out);
17    sel = 2'b10; #10;
18    $display("Sel: %b, Output: %b",
19            sel, out);
20    sel = 2'b11; #10;
21    $display("Sel: %b, Output: %b",
22            sel, out);
23  end
24 endmodule

```

```

1 module mux4to1(input [3:0] data_in, input [1:0] sel, output out);
2   assign out = (sel == 2'b00) ? data_in[0] :
3               (sel == 2'b01) ? data_in[1] :
4               (sel == 2'b10) ? data_in[2] :
5               data_in[3];
6 endmodule
7

```

Log   Share

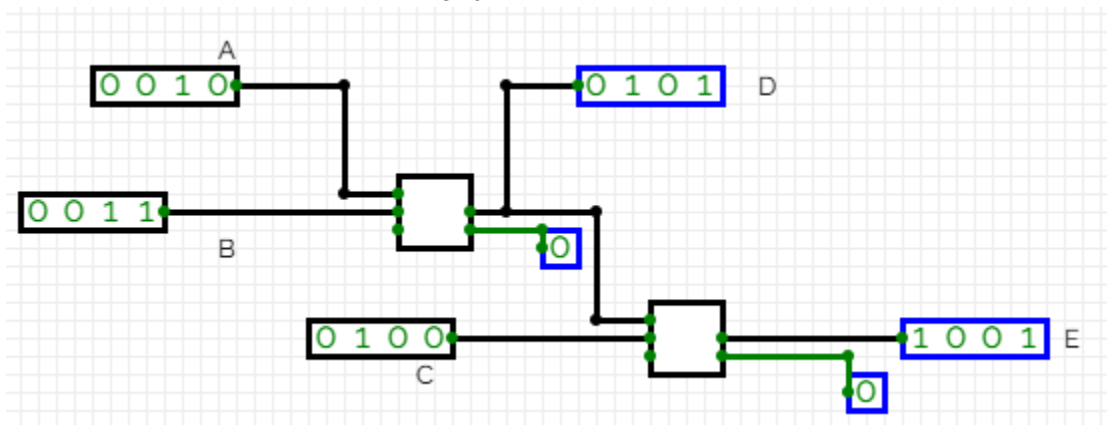
[2024-12-17 10:38:29 UTC] iverilog -Wall -g2012 design.sv testbench.sv && unbuffer vvp a.out

Sel: 00, Output: 0  
Sel: 01, Output: 1  
Sel: 10, Output: 0  
Sel: 11, Output: 1

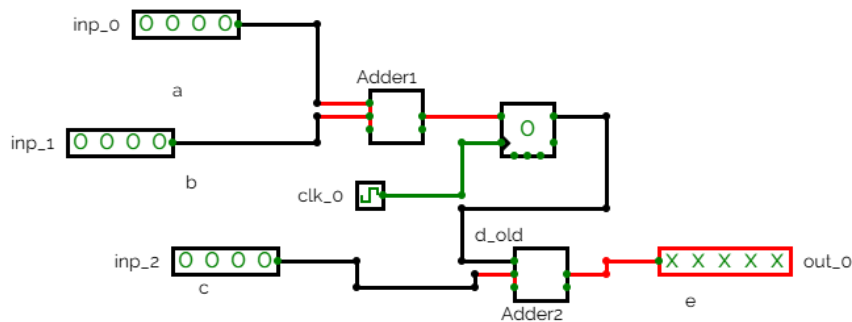
Done

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
```