Nang Thiri Wutyi
20113
10/9/2024

Verilog Lab #3 Combinational Logic

1. **XOR3 Circuit**

```
`timescale 1ns/100ps
module xor3 (input a_i, b_i, c_i, output y_o);
  wire aBar_w, bBar_w, cBar_w;
  wire im1_w, im2_w, im3_w, im4_w;

  not (aBar_w, a_i),
      (bBar_w, b_i),
      (cBar_w, c_i);

  nand (im1_w, aBar_w, bBar_w, c_i),
       (im2_w, aBar_w, b_i, cBar_w),
       (im3_w, aBar_w, bBar_w, cBar_w),
       (im4_w, a_i, b_i, c_i);

  nand (y_o, im1_w, im2_w, im3_w, im4_w);
endmodule
```

Majority Circuit

```
`timescale 1ns/100ps
module maj3(input a_i, b_i, c_i, output y_o);
  wire im1_w, im2_w, im3_w;

  and (im1_w, a_i, b_i),
      (im2_w, b_i, c_i),
      (im3_w, c_i, a_i);

  or (y_o, im1_w, im2_w, im3_w);
endmodule
```

Combining XOR3 and Majority into a 1-bit Full Adder

```
`include "maj3.v"
`include "xor3.v"
`timescale 1ns/100ps
module AddOneBitP (input a_i, b_i, ci_i, output sum_o, co_o);
  xor3 Uxor3 (.a_i(a_i), .b_i(b_i), .c_i(ci_i), .y_o(sum_o));
  maj3 Umaj3 (.a_i(a_i), .b_i(b_i), .c_i(ci_i), .y_o(co_o));
endmodule
```

2. **1-bit Full Adder using assign**

```
`timescale 1ns/100ps
module add_1bit(input a, b, ci, output s, co);
  assign {co, s} = {(a & b) | (b & ci) | (a & ci), a ^ b ^ ci};
endmodule
```

2-to-4 Decoder using Conditional Assignment

```
`timescale 1ns/100ps

module dcd2_4(input a, b, output d0, d1, d2, d3);

  assign {d3, d2, d1, d0} = ( {a, b} == 2'b00 ) ? 4'b0001 :

                            ( {a, b} == 2'b01 ) ? 4'b0010 :

                            ( {a, b} == 2'b10 ) ? 4'b0100 :

                            ( {a, b} == 2'b11 ) ? 4'b1000 : 4'b0000;

endmodule
```

Majority Logic using always Block

```
`timescale 1ns/100ps

module maj3(input a, b, c, output reg y);

  always @(a, b, c) begin

    y = (a & b) | (b & c) | (a & c);

  end

endmodule
```

3. **Majority Logic using always Block**

```
`timescale 1ns/100ps

module maj3(input a, b, c, output reg y);
```

```
  always @(a, b, c) begin
    y = (a & b) | (b & c) | (a & c);
  end
endmodule
```

4. **Comparison of Mux Implementations**

4.1 MUX using if-else (IfMux8)

```
module IfMux8 (output reg y, input [7:0] i, input [2:0] sel);
  always @(i, sel) begin
    if (sel == 3'd0) y = i[0];
    else if (sel == 3'd1) y = i[1];
    else if (sel == 3'd2) y = i[2];
    else if (sel == 3'd3) y = i[3];
    else if (sel == 3'd4) y = i[4];
    else if (sel == 3'd5) y = i[5];
    else if (sel == 3'd6) y = i[6];
    else if (sel == 3'd7) y = i[7];
  end
endmodule
```

MUX using case (CaseMux8)

```
module CaseMux8 (output reg y, input [7:0] i, input [2:0] sel);
  always @(i, sel) begin
    case (sel)
      3'd0: y = i[0];
    3'd1: y = i[1];

    3'd2: y = i[2];

    3'd3: y = i[3];

    3'd4: y = i[4];

    3'd5: y = i[5];

    3'd6: y = i[6];

    3'd7: y = i[7];

   endcase

 end

endmodule
```

5. **Race Condition with Multiple Initial Blocks**

```
module race_condition();

 reg b;

 initial begin

   b = 0;

 end

 initial begin

   b = 1;

 end

endmodule
```

6. **Exercises**
   a.
   1) **Fix: Missing sensitivity list**

      ```
      // design.sv
      ```

```
`timescale 1ns/100ps

module my_design(input a, input b, input c, output reg y);

  // Using @(*) ensures all relevant inputs are in the sensitivity list.
  always @(*) begin
    y = (a & b) | (b & c) | (a & c); // Example logic (Majority gate)
  end

endmodule
```

2) **Fix: Conflict between always blocks setting the same variable C**

```
always@(B or E)
C = |B | ^E;
```

3) **Fix: Clock enable issue**

```
always@(posedge clock) begin
 if(A)
   Q <= D;
end

always@(Q or E) begin
 case (Q)
   1'b0: F = E;
   default: F = 1;
```

```
  endcase

end
```

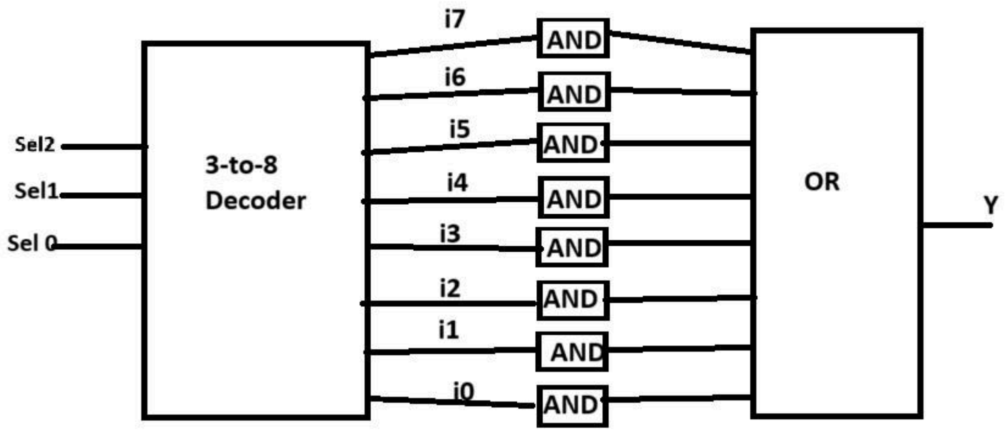
4) Fix: Wire conflict between two modules

```
module top;

  wire B;

  bar u1 (A, B);

  bar u2 (C, B);

endmodule


module bar (input D, output wire E);

  assign E = ~D;

endmodule
```

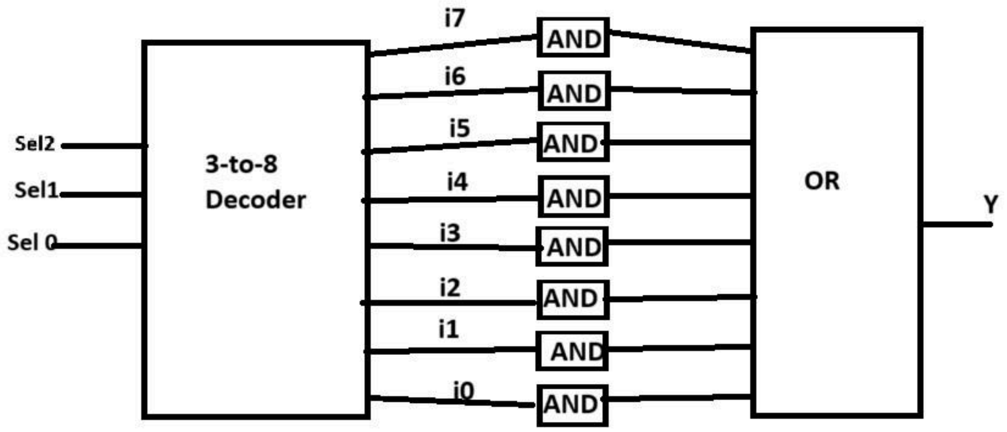
5) Fix: Use always block and logic gates together

```
module foo(input A,B; output reg E);

  wire C,D;

  always@(posedge clock) E = B & D;

  assign C = A ^ D;

  assign D = C | B;

endmodule
```

b. According to the IfMux modules



According to CaseMux8 module



Both approaches use a decoder, AND gates, and an OR gate in the same way, so the block diagrams are identical in terms of components and connections.