# DAY-5
# #100DAYSOFRTL

## Problem statement :-

1. Build an AND gate using both an assign statement and a combinational always block. (Since assign statements and combinational always blocks function identically, there is no way to enforce that you're using both methods)

**Write your solution here**

[Load a previous submission] Load

```
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    output wire out_assign,
    output reg out_alwaysblock
);
    assign out_assign=a&b;
    always @(*)
            out_alwaysblock=a&b;



endmodule

```

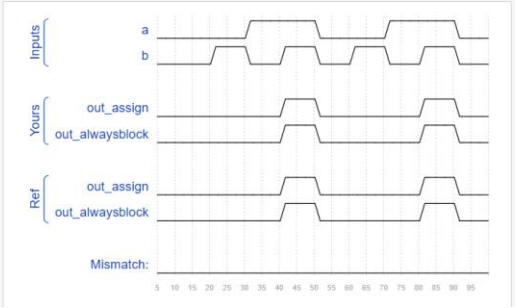
**Status: Success!**

You have solved 26 problems. See my progress...

**Timing diagrams for selected test cases**

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).

**AND gate**



**2.** Build an XOR gate three ways, using an assign statement, a combinational always block, and a clocked always block. Note that the clocked always block produces a different circuit from the other two: There is a flip-flop so the output is delayed.

## Write your solution here

[Load a previous submission] Load

```
// synthesis verilog_input_version verilog_2001
module top_module(
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff   );

assign out_assign =a^b;

   always@(*)
      out_always_comb=a^b;

    always@(posedge clk)
     out_always_ff =  a^b;



endmodule
```

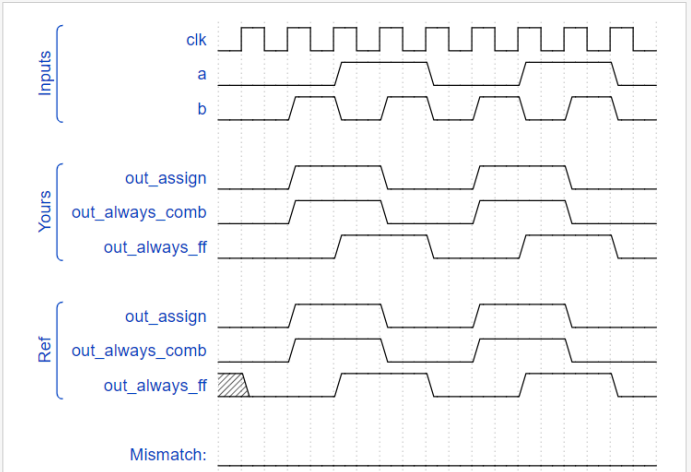
## Status: Success!

You have solved 27 problems. See my progress...

### Timing diagrams for selected test cases

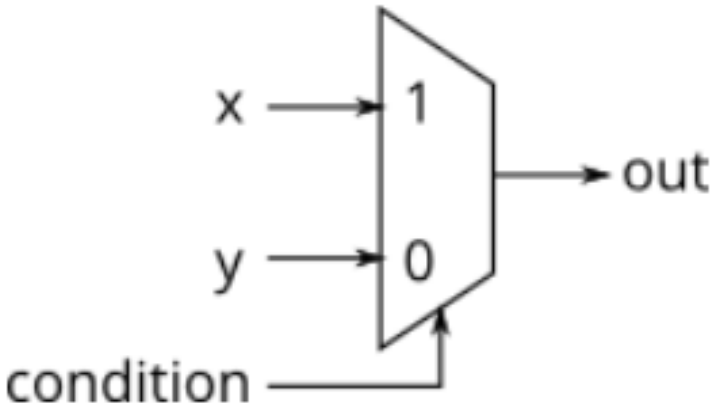
These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).

**XOR gate**



**3.** An `if` statement usually creates a 2-to-1 multiplexer, selecting one input if the condition is true, and the other input if the condition is false.

```
    always @(*) begin
    if (condition) begin
        out = x;
    end
    else begin
        out = y;
    end
end
```



## Write your solution here

[Load a previous submission] Load

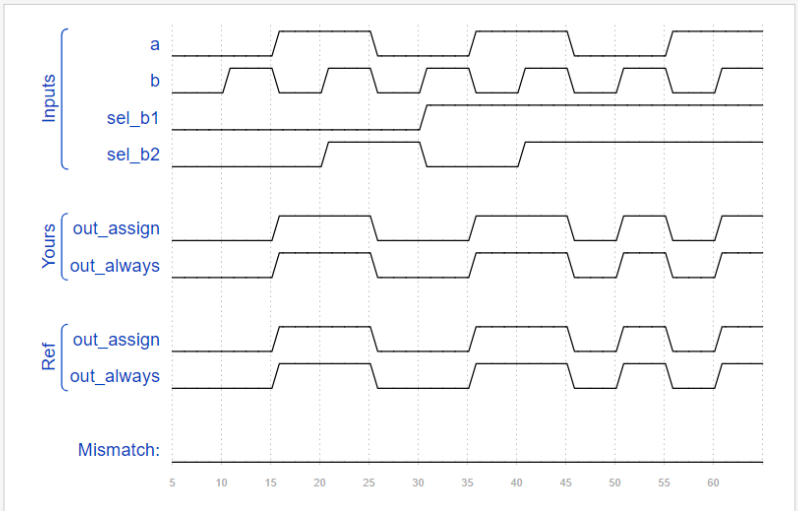
```verilog
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    input sel_b1,
    input sel_b2,
    output wire out_assign,
    output reg out_always   );


    assign out_assign=(sel_b1 && sel_b2 )? b:a;

    always @(*)
        begin

            if (sel_b1 && sel_b2 )
                out_always=b;

            else
                out_always=a;

        end


endmodule

```

# Status: Success!

You have solved 28 problems. See my progress...

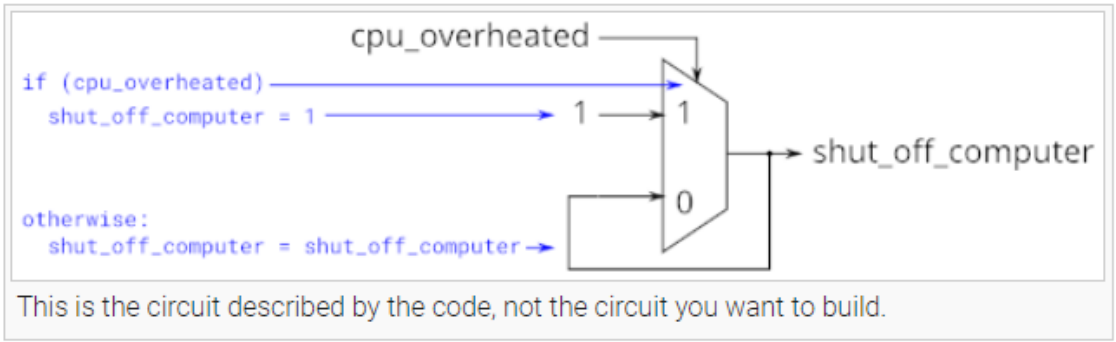
## Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).



**4.** The following code contains incorrect behaviour that creates a latch. Fix the bugs so that you will shut off the computer only if it's really overheated, and stop driving if you've arrived at your destination or you need to refuel.

```
always @(*) begin
    if (cpu_overheated)
        shut_off_computer = 1;
end

always @(*) begin
    if (~arrived)
        keep_driving = ~gas_tank_empty;
end
```




This is the circuit described by the code, not the circuit you want to build.

## Write your solution here

[Load a previous submission] Load

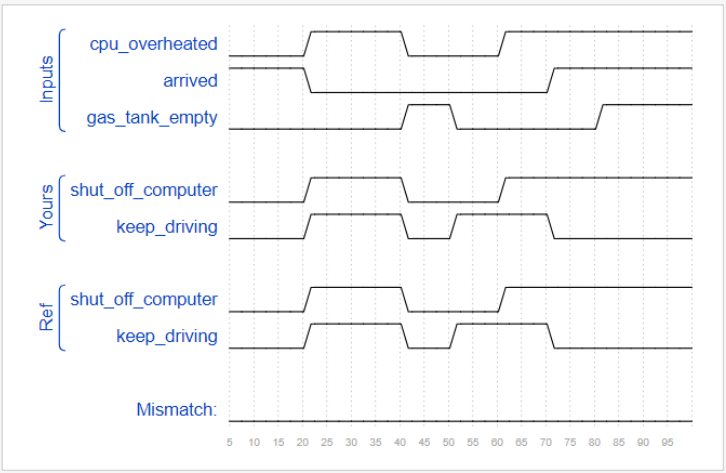
```
// synthesis verilog_input_version verilog_2001
module top_module (
    input      cpu_overheated,
    output reg shut_off_computer,
    input      arrived,
    input      gas_tank_empty,
    output reg keep_driving  ); //

    always @(*) begin
        if (cpu_overheated)
           shut_off_computer = 1;
        else
             shut_off_computer = 0;

    end

    always @(*) begin
        if (~arrived && ~gas_tank_empty)
           keep_driving = ~gas_tank_empty;
        else
             keep_driving=0;

    end

endmodule

```

## Status: Success!

You have solved 29 problems. See my progress...

### Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).



**5.** Case statements are more convenient than if statements if there are a large number of cases. So, in this exercise, create a 6-to-1 multiplexer. When `sel` is between 0 and 5, choose the corresponding data input. Otherwise, output 0. The data inputs and outputs are all 4 bits wide.

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input [2:0] sel,
    input [3:0] data0,
    input [3:0] data1,
    input [3:0] data2,
    input [3:0] data3,
    input [3:0] data4,
    input [3:0] data5,
    output reg [3:0] out   );//

    always@(*) begin  // This is a combinational circuit
        case(sel)

            3'b000:out=data0;
            3'b001:out=data1;
            3'b010:out=data2;
            3'b011:out=data3;
            3'b100:out=data4;
            3'b101:out=data5;
            default:out=3'b000;

 endcase

  end

endmodule

```

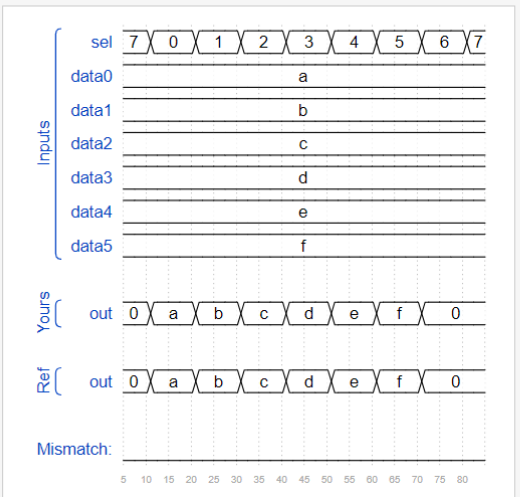
## Status: Success!

You have solved 30 problems. See my progress...

### Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).

**Sel chooses one of the data inputs**



**6.** Build a 4-bit priority encoder. For this problem, if none of the input bits are high (i.e., input is zero), output zero. Note that a 4-bit number has 16 possible combinations.

## Write your solution here

[Load a previous submission] Load

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input [3:0] in,
    output reg [1:0] pos  );

    always@(*)
        case (in)
            4'd0:pos=2'd0;
            4'd1:pos=2'd0;
            4'd2:pos=2'd1;
            4'd3:pos=2'd0;
            4'd4:pos=2'd2;
            4'd5:pos=2'd0;
            4'd6:pos=2'd1;
            4'd7:pos=2'd0;
            4'd8:pos=2'd3;
            4'd9:pos=2'd0;
            4'd10:pos=2'd1;
            4'd11:pos=2'd0;
            4'd12:pos=2'd2;
            4'd13:pos=2'd0;
            4'd14:pos=2'd1;
             4'd15:pos=2'd0;
        endcase

endmodule
```

Submit | Submit (new window)

Upload a source file...

## Status: Success!

You have solved 31 problems. See my progress...

### Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).

**Priority encoder**