ECE 2300 Spring 2025

# Verilog Tutorial Session

Nitish (graduated)
Phil (graduated)
Nikita (graduated)
Andrew (atb78)

# Verilog

- Verilog is a **Hardware Description Language (HDL)**. It is used to describe the structure and behavior of the hardware. The final product of a Verilog “program” is the generated hardware circuit

- Verilog is **NOT** a sequential programming language.
- And is NOT even a programming language

# Recap: Continuous Assignment Statements (assign)

- Describes only combinational logic

- These statements are re-evaluated **anytime** any of the inputs on the right hand side changes.

```
assign <out> = <in0> <operator> <in1> ... ;
```

# Q1: Find the bug in this 2 input AND gate!

```
module assign_example(A,B,Y);
    input A;
    input B;
    output Y;

    Y = A & B;
endmodule
```

# Q1: Fixed version.

```
module assign_example(A,B,Y);
    input A;
    input B;
    output Y;

    assign Y = A & B;
endmodule
```

# Some rules (assign)

- Assign statements can only be used to set signals declared as wires.
  - Wire
  - Input
  - Output

- Assign statements cannot be used inside *always* blocks.
  - More on this later.

# always Statements

- These statements are executed when a signal in the sensitivity list changes.
  - otherwise, signals keep their old value.
- Can be used to describe sequential logic.

```
module flipflop(D,CLK,Q);
    input D;
    input CLK;
    output reg Q;

    always @(posedge CLK) begin
        Q <= D;
    end
endmodule
```

# Some rules  (always blocks)

- The sensitivity list for an always block specifies the signals that cause the block to update.
  - in the sensitivity list updates a block when any signal in the circuit changes.
- Every signal is either a **wire** or a **reg**.
- You can only assign to a reg inside always blocks.
- You can read from a wire or a reg inside always block.

# Wire vs. Reg Recap

- Every signal is either a wire or a reg

- Wires are used for connecting things; think of them as of physical wires
  - they can only be read or assigned
  - no value can be stored in them
  - you can only assign to a wire outside always blocks using **assign** statement
  - module I/O are AUTOMATICALLY considered as wires

- Regs represent data storage elements, **not necessarily registers**
  - can be synthesized to FFs, latches, combinational circuit
  - might even not be synthesizable!
  - you can only assign to a reg inside always blocks using = or <=

# Back to always Statements

- Can also be used to describe combinational logic.

```
module always_ex(A,B,C,Y);
    input A;
    input B;
    input C;
    output reg Y;

    reg interm;

    always @(*) begin
        interm = A & B;
        Y      = ~interm | C;
    end
endmodule
```

=

```
module assign_ex(A,B,C,Y);
    input A;
    input B;
    input C;
    output Y;

    wire interm;


    assign interm = A & B;
    assign Y      = ~interm | C;

endmodule
```

# Q2: Find the bug in this module

```
module  if_test_module (A,Y);
    input [1:0] A;
    output reg  Y;

    always@(*) begin
        if (A == 2'b00) begin
            Y = 1;
        end
        else if (A == 2'b01) begin
            Y = 0;
        end
        else if (A == 2'b10) begin
            Y = 0;
        end
    end
endmodule
```

# Blocking vs. Non-Blocking Assignments Recap

- Assignments to regs inside always blocks can be blocking and non-blocking

- Blocking Statements (=) are used to express combinational logic, i.e. it infers dependencies between variables in a combinational circuit
  - you can model their execution as a “standard software program”
  - this whole “program” is executed at once following the event in the sensitivity list

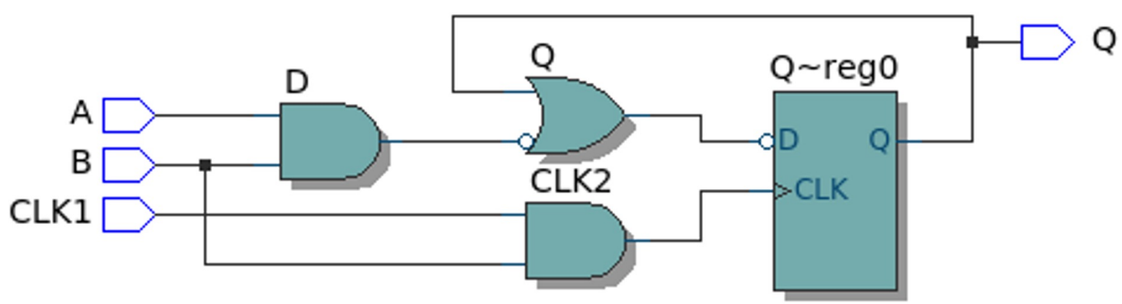
- Non-Blocking Statements (<=) represent sequential logic
  - each blocking statement is executed independently and concurrently with others

# Viewing Your Design

Sometimes, it's very helpful to see what your design is synthesised to:

*Quartus > Tools -> Netlist_Viewers -> RTL Viewer*

```
module tffp (CLK1, A, B, Q);
  input CLK1, A, B;
  output reg Q;

  wire CLK2;
  assign CLK2 = B & CLK1;

  reg D;

  always @ (posedge CLK2)
  begin
    D = ~(A & B);
    Q = ~(D | Q);
  end

endmodule
```

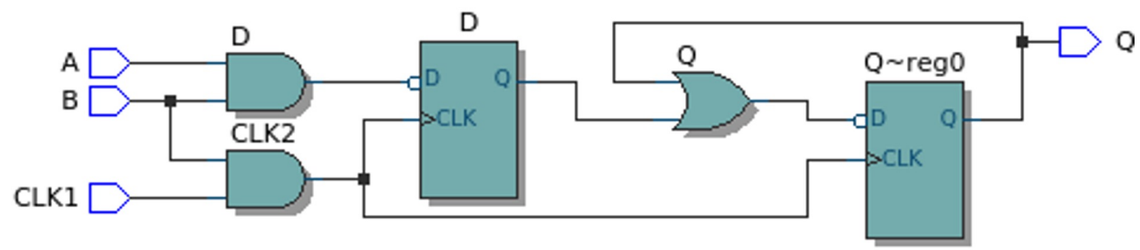


# Viewing Your Design

Sometimes, it's very helpful to see what your design is synthesised to:

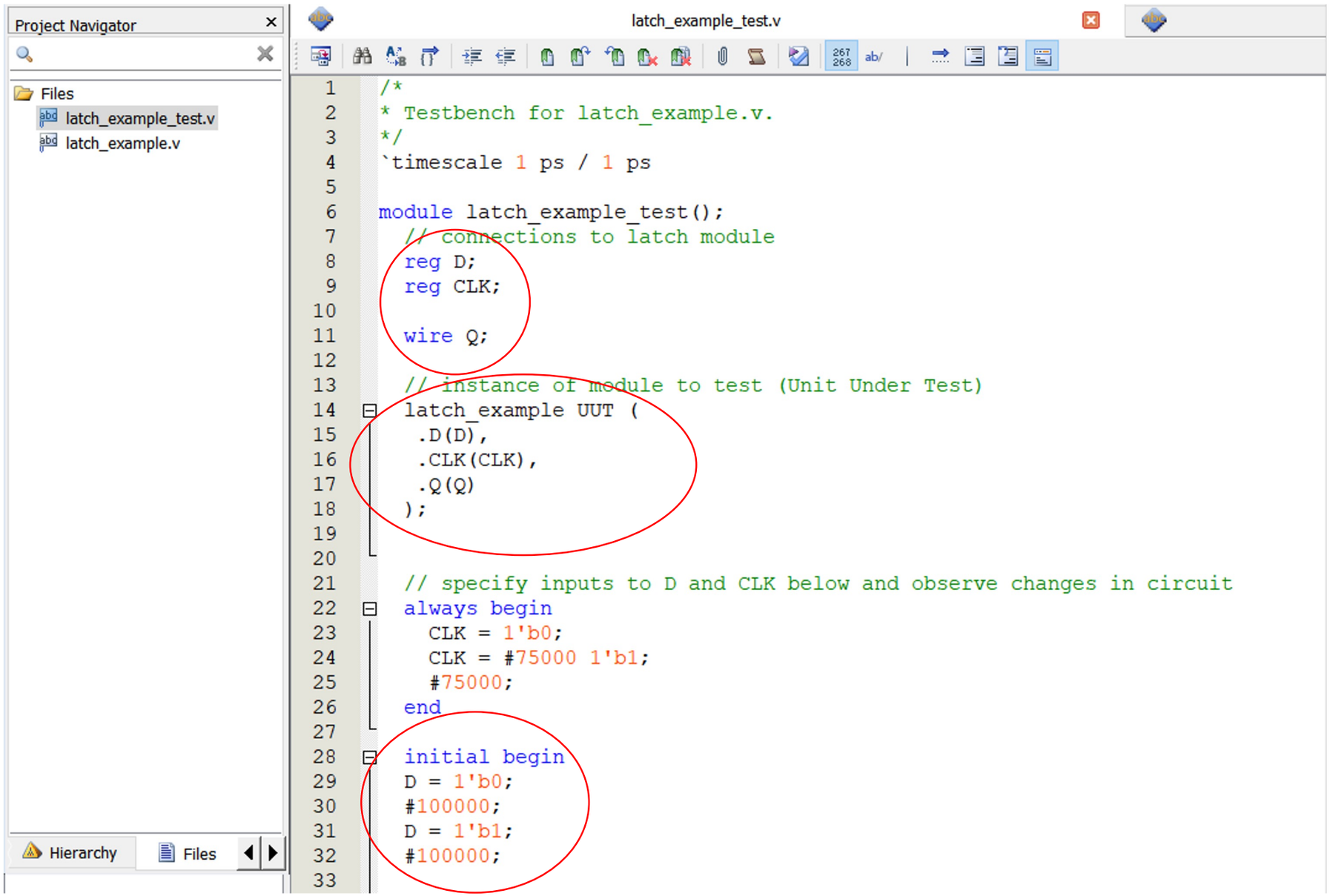
*Quartus > Tools -> Netlist_Viewers -> RTL Viewer*

```
module tffp (CLK1, A, B, Q);
  input CLK1, A, B;
  output reg Q;

  wire CLK2;
  assign CLK2 = B & CLK1;

  reg D;

  always @ (posedge CLK2)
  begin
    D <= ~(A & B);
    Q <= ~(D | Q);
  end

endmodule
```



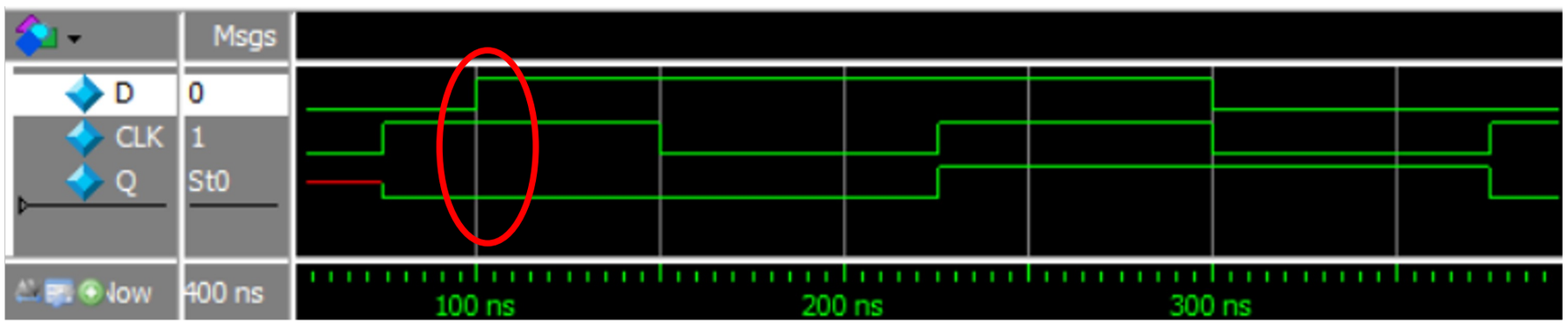
# Testbench

- A Verilog module that is used to test another module, called the design under test (DUT).
- Contains statements to apply inputs to the DUT.
- The input and desired output patterns are called test vectors.


Project Navigator
Files
latch_example_test.v
latch_example.v
Hierarchy
Files
latch_example_test.v
1 /*
2 * Testbench for latch_example.v.
3 */
4 `timescale 1 ps / 1 ps
5
6 module latch_example_test();
7 // connections to latch module
8 reg D;
9 reg CLK;
10
11 wire Q;
12
13 // instance of module to test (Unit Under Test)
14 latch_example UUT (
15 .D(D),
16 .CLK(CLK),
17 .Q(Q)
18 );
19
20
21 // specify inputs to D and CLK below and observe changes in circuit
22 always begin
23 CLK = 1'b0;
24 CLK = #75000 1'b1;
25 #75000;
26 end
27
28 initial begin
29 D = 1'b0;
30 #100000;
31 D = 1'b1;
32 #100000;
33

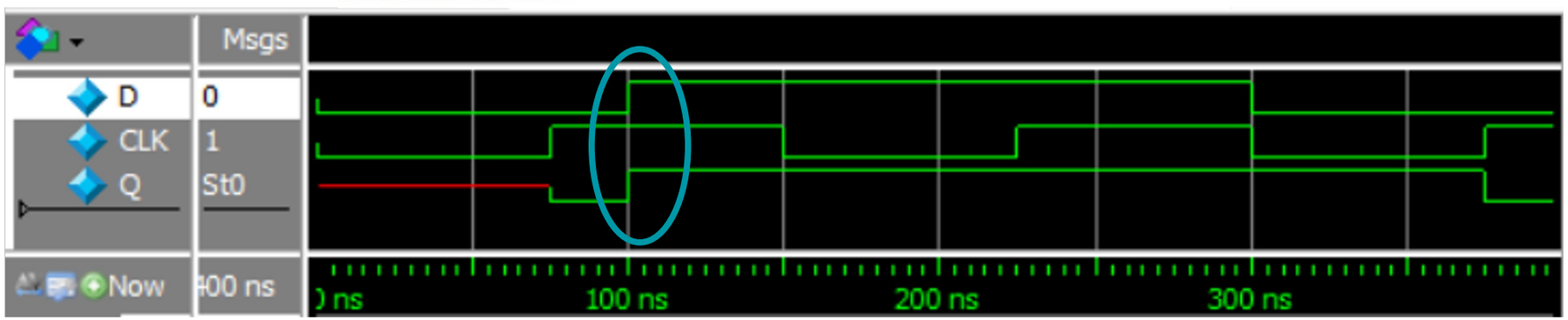
# Debugging using a waveform

```
module latch_example(D, CLK, Q);
    input D;
    input CLK;
    output reg Q;

    always @(CLK) begin
        if (CLK) begin
            Q <= D;
        end
    end
endmodule
```



# Latch fixed!

```
module latch_example(D, CLK, Q);
    input D;
    input CLK;
    output reg Q;

    always @(CLK,D) begin
        if (CLK) begin
            Q <= D;
        end
    end
endmodule
```



# Common Quartus Issues

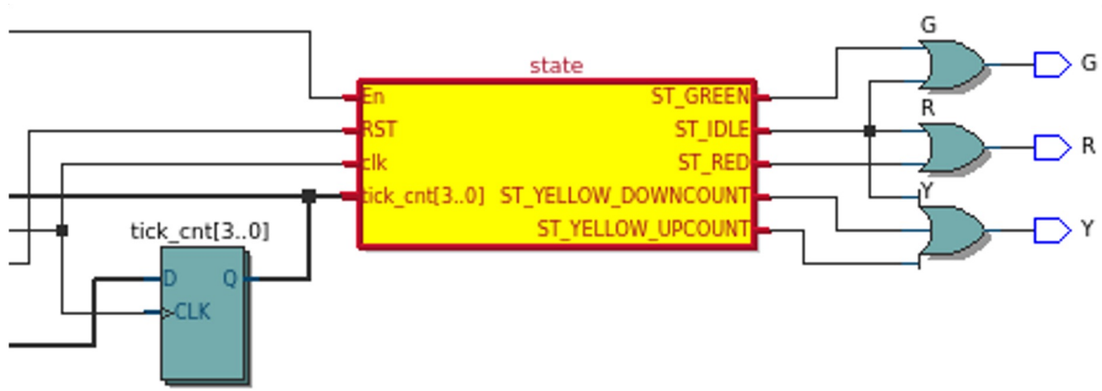
- “Cannot open project”: Likely an issue with the downloaded zip. Delete the zip and the folder the contents were unzipped to. Redownload the zip and try again
- On ecelinux “Simulation failed to launch”: Using v18 instead of v23, use module load quartus/quartus-prime-lite-v23
- “No Simulation Test Bench Specified”: Assignment -> Settings and select the EDA Tool Settings -> Simulation tab. Under NativeLink settings select compile test bench and select the testbench there if available. If not click testbenches and add a new testbench, selecting the testbench Verilog file.

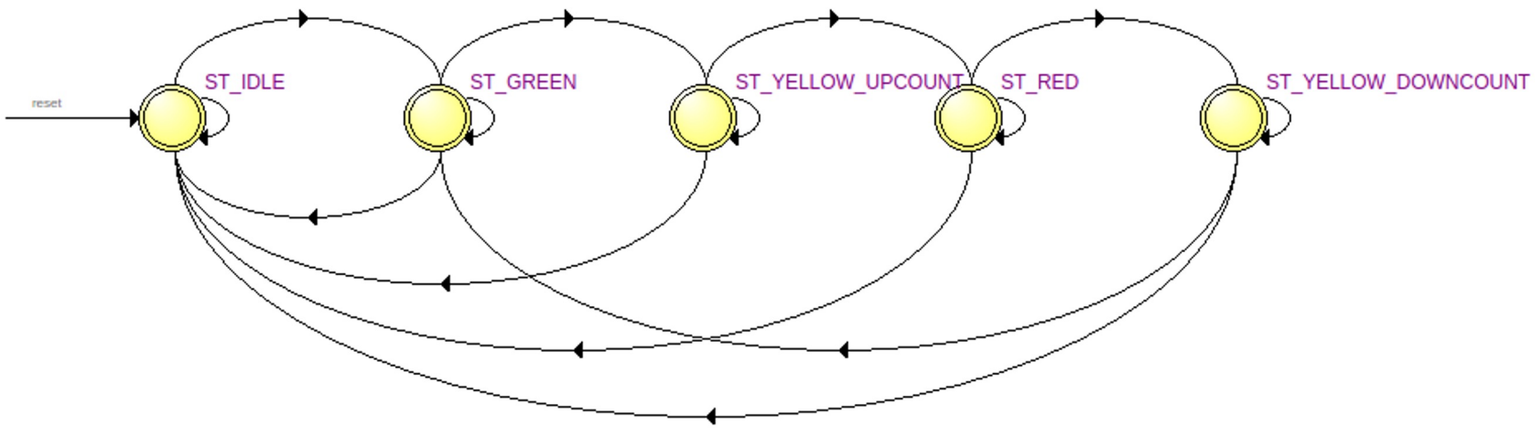
# Case Study: Traffic Light Controller.

1. Implementation of a simple traffic light controller
2. Correct behavior:
    1. After RESET, all Red (R), Yellow (Y), and Green (G) lights are ON
    2. When EABLE (En) comes (`1), the light is switched to G
    3. After some time T1, G is switching to Y
    4. After some time **T2 < T1**, Y is switching to R
    5. <same for R -> Y -> G)
    6. If En goes down (`0), the traffic light keeps the current light until En is back to `1

# Case Study: Traffic Light Controller.

You can view the FSM diagram from the RTL viewer if you click on the yellow block

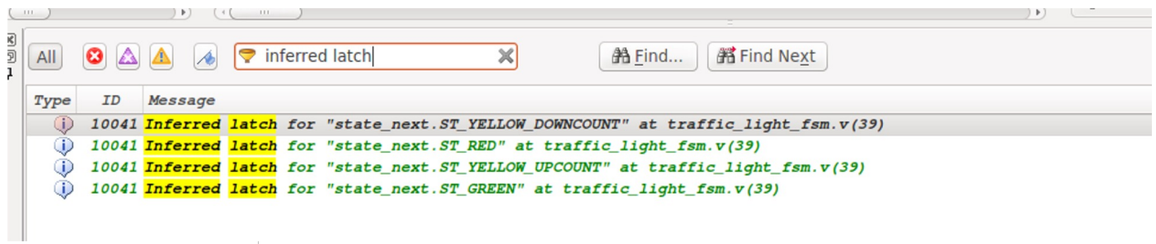


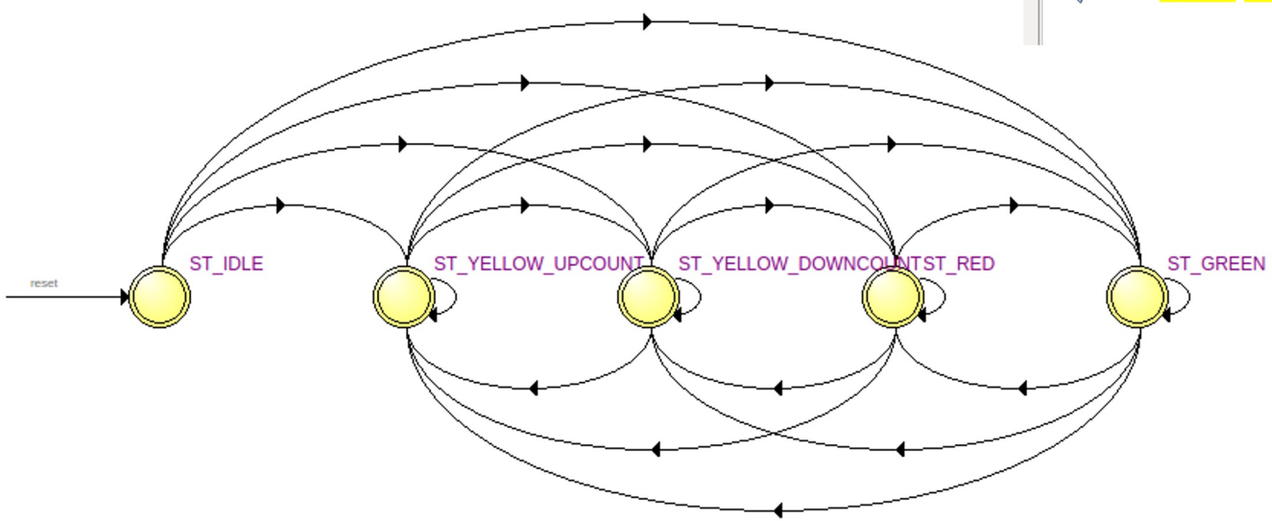



# Case Study: Traffic Light Controller.

In this case study, the FSM is incorrect:

1) it has too many transitions due to **inferred latches**
2) its logic is broken



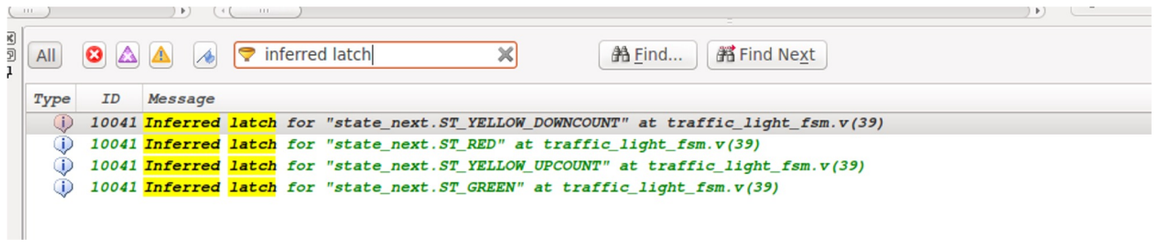



# Case Study: Traffic Light Controller.

1. Download traffic_light_fsm example from CMS (*VerilogTutorial*).
2. The code contains three bugs:
    1. inferred latch
    2. bug in the FSM logic (tests fail)
    3. another bug that you will have to figure out yourself (tests fail)
3. Work in groups of 1-3 to try to figure out and fix the bugs.
4. We will discuss results and debug strategy afterwards.

# Systematic Debugging, fixing inferred latches

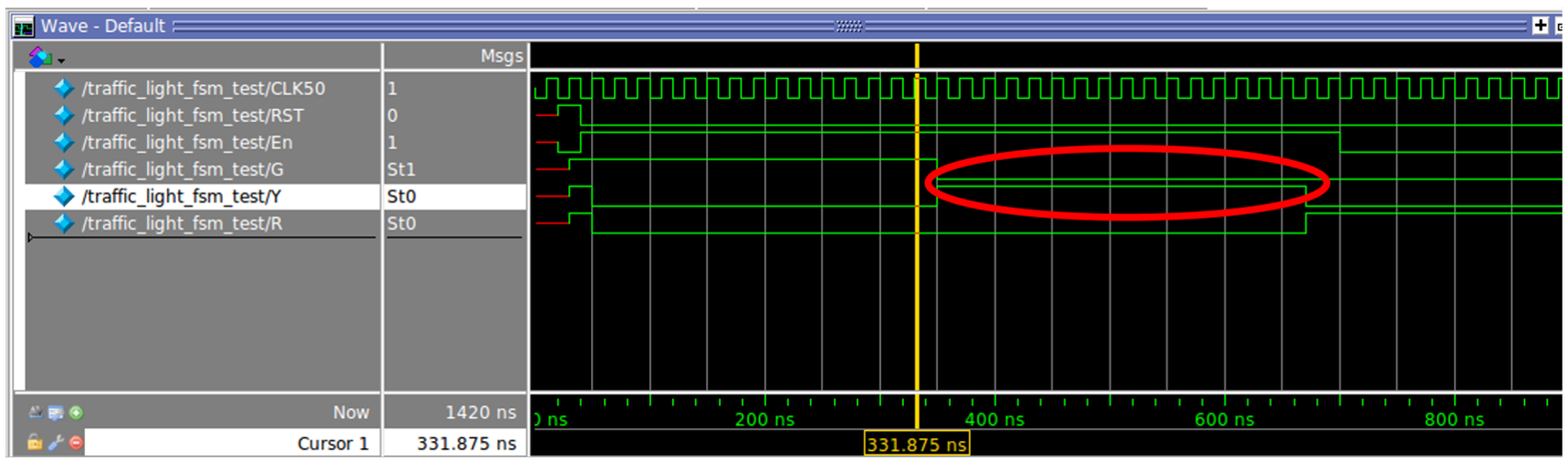
1. Based on the compilation report, the inferred latches are somewhere in the FSM state_next logic




1. After careful inspection of the state_next part of the FSM, one can easily find that the '*state_next*' signal has no default value assigned
2. We fix it by adding the line '*state_next = state;*' before the case statement (line 39)

# Systematic Debugging, finding bug #1

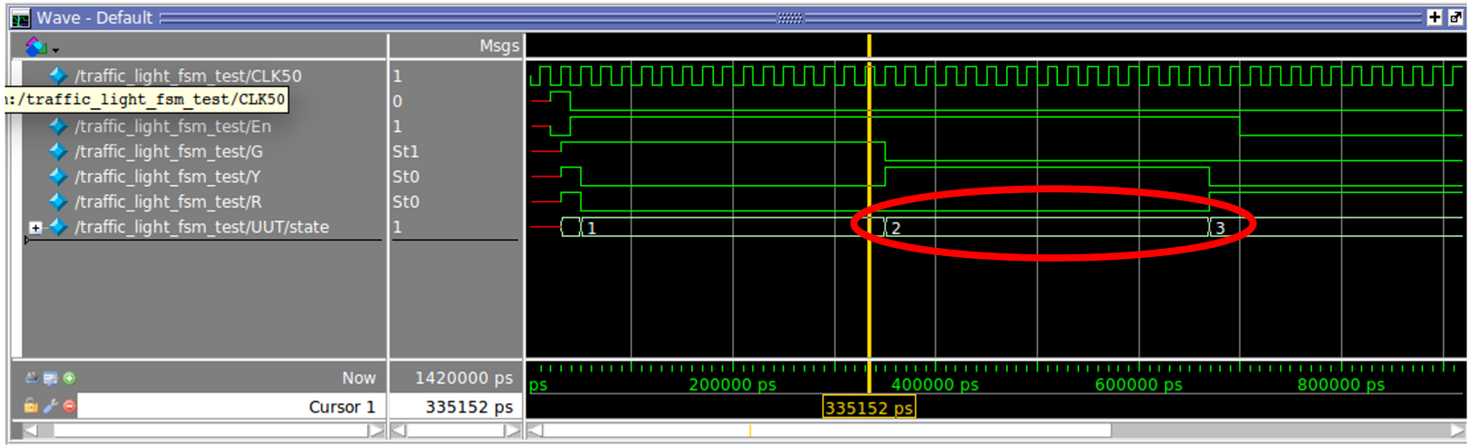
1. Based on the simulation output, things become wrong when the Yellow light is turning to Red
2. We open the waveforms and find this place




1. For some reason, the Yellow light stays for too long (T2 = T1)

# Systematic Debugging, finding bug #1

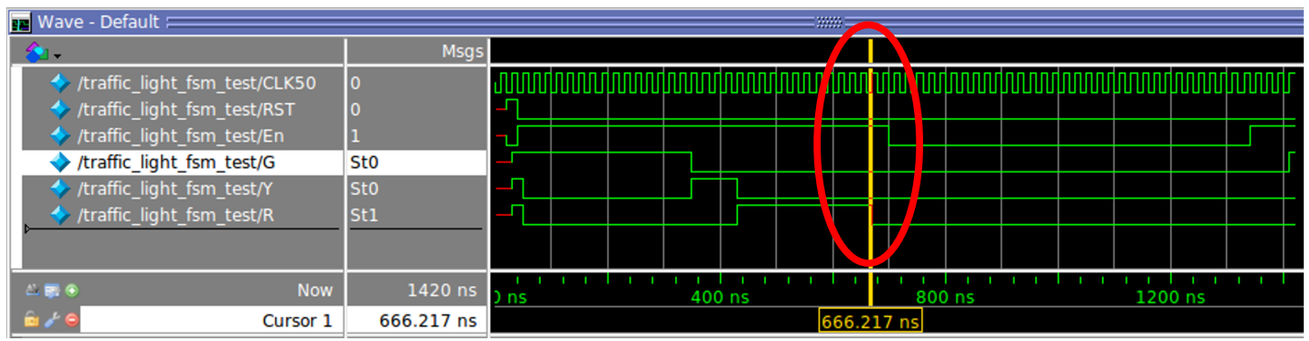
4. We add one more signal to the waveforms: the state of the FSM




4. We notice that the state stucks in ‘2’ (*ST_YELLOW_UPCOUNT*) for too long
5. We go to the corresponding line in the FSM and observe that the condition for the switch to *ST_RED* is based on ‘*tick*’ rather than ‘*short_tick*’
6. The bug is caught!

# Systematic Debugging, finding bug #2

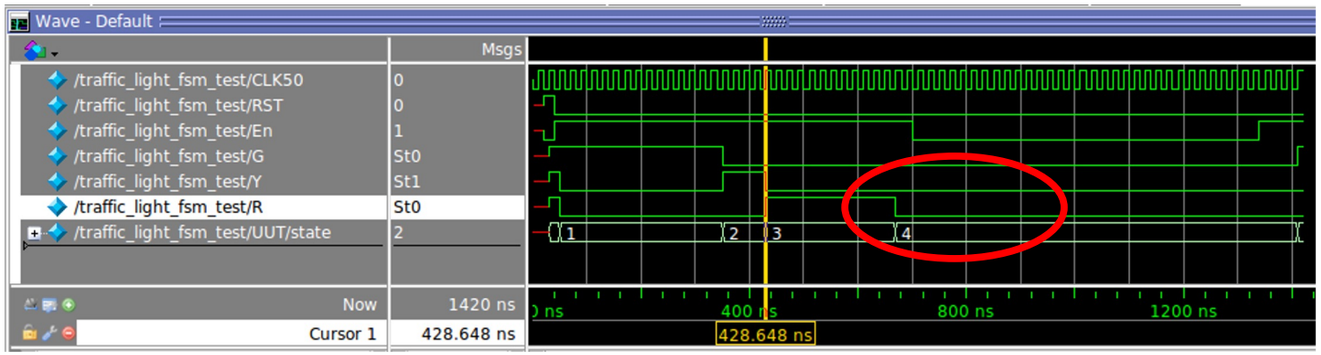
1. Based on the simulation output, we see that the test fails when counting backwards, after the Red light
2. We open the waveforms and locate the time when it happens




1. So Yellow is not going UP after Red

# Systematic Debugging, finding bug #2

4. As before, we add the state signal to the waveforms to check it at this point




4. We notice that the state is switching to 5 (*ST_YELLOW_DOWNCOUNT*), but the value of the Yellow signal is not changing
5. Perhaps, the bug is in the output logic
6. We go to the output logic and see that the Yellow signal is missing the OR condition when *state == ST_YELLOW_DOWNCOUNT*
7. The bug is fixed!

# Common Verilog Compiler Messages and Pitfalls

- **Assigning to wire in always block:** “Error (10137): Verilog HDL Procedural Assignment error at universalShift.v(22): object "myErrorWire" on left-hand side of assignment must have a variable data type”
- **Assigning to reg outside always block:** One of several errors: “Error (10170): Verilog HDL syntax error at universalShift.v(17) near text "="; expecting ".", or "("” or “Error (10219): Verilog HDL Continuous Assignment error at universalShift.v(17): object "myErrorReg" on left-hand side of assignment must have a net type”
- **Bit truncation warning “Warning (10230):** Verilog HDL assignment warning at universalShift.v(17): truncated value with size 10 to match size of target (8)”

# Common Verilog Compiler Messages and Pitfalls

- **Inferred latches “Warning (10240):** Verilog HDL Always Construct warning at universalShift.v(17): inferring latch(es) for variable "accidentalLatch", which holds its previous value in one or more paths through the always construct
- **Trying to assign to a wire in 2 places:** “Error (10028): Can't resolve multiple constant drivers for net "DoubleDrivenWire" at universalShift.v(17)”
- **Driving Input Ports “Error (10231):** Verilog HDL error at universalShift.v(15): value cannot be assigned to input "drivenInput"” followed by “Error (10031): Net "drivenInput" at universalShift.v(15) is already driven by input port "drivenInput", and cannot be driven by another signal”

# Common Runtime issues and bugs

- **Forgetting to connect inputs to units:** Here the simulator might tell that the value is “X” or “XX.XX” and the corresponding waveform will be red. Fix it by carefully inspecting all signals this output depends on.
- **Unexpected additional delay cycle (the waveforms are shifted by one cycle):** This often happens when the logic that was supposed to be combinational is written as sequential which introduces an additional FF and the delay.
- **Mismatch in signal names:** The simulator might report that it can not find some signals or design units. Please, make sure the names of signals and DUT’s exactly match the names in the testbench
- **If a testbench fails, you can always open it and see what exactly is not working**