VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CO3098 - LSI DESIGN LAB

LAB 1

BOUND FLASHER

Instructor: **PROF. NGUYEN THIEN AN**

Student: Nguyen Khanh Nam - 2153599

HO CHI MINH CITY, FEBRUARY 2024

Contents

1 Problem Implementation

1.1 Problem-Solving Strategies

The problem can be effectively solved by dividing it into states because there are multiple operations. Beside, in order to make the lamps turn on or off gradually, a delay mechanism needs to be applied. In this problem, I will use an integer **counter** variable that will count up to a specific number **TIMER**. Once the **counter** attains the value of **TIMER**, the operation to either turn on or turn off the lamps will be executed.

1.2 Finite State Machine

An effective solution can be devised by employing a Finite State Machine (FSM) approach. By breaking down the task into eight distinct states, we can systematically address each operation outlined in the question. Each state in the FSM corresponds to a specific phase of the task and is designed to execute the required operations in a structured manner. This modular and organized approach not only enhances the clarity of the solution but also facilitates efficient management of the entire process.

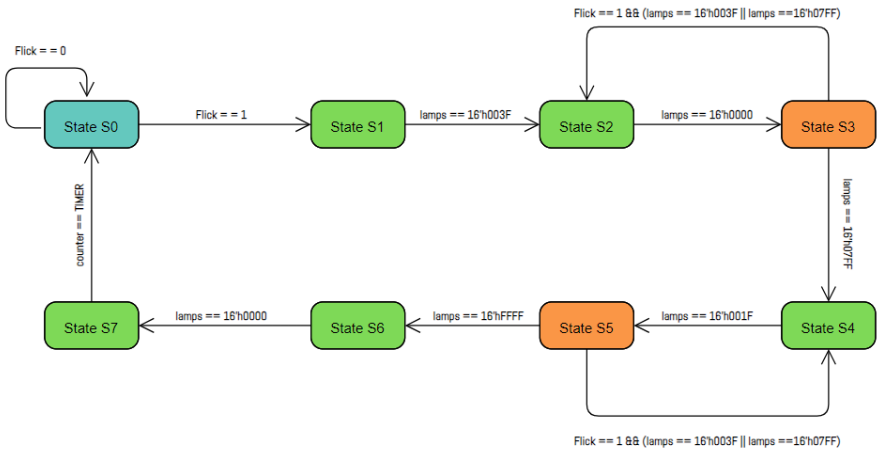


Figure 1: Finite State Machine for the problem.

1.3 Code Implementation

First of all, I will define inputs, output, states and some internal signals.

```
`timescale 1ns / 1ps

module boundFlasher(
    input clk, flick,
    output [15:0] lamps
);

    // Define states
    parameter S0 = 0;
    parameter S1 = 1;
```

```
    parameter S2 = 2;
    parameter S3 = 3;
    parameter S4 = 4;
    parameter S5 = 5;
    parameter S6 = 6;
    parameter S7 = 7;

    // Internal signals
    parameter TIMER = 200;
    integer counter = 0;
    reg [3:0] state = S0;
    reg [15:0] temp;
```

Break down the variables:

- **clk**: clock input signal.
- **flick**: flick input signal.
- **lamps**: 16 bits output signal.
- **S0 - S7**: states.
- **TIMER**: the number that the counter needs to be counted up to.
- **counter**: an integer variable used for delay purpose.
- **state**: current state.
- **temp**: a temporary variable used to keep track of the status of the lamps.

Next step is to implement the finite state machine based on the above section.

```
    // State machine
    always @(posedge clk) begin
        case(state)
            S0:
                begin
                    temp <= 16'h0000;
                    if(flick == 0) state <= S0;
                    else state <= S1;
                end
            S1:
                begin
                    counter <= counter + 1;
                    if(counter == TIMER) begin
                        temp <= (temp << 1) + 1;
                        counter <= 0;
                    end
                    if(temp == 16'h003F) begin
                        state <= S2;
                        counter <= 0;
                    end
```

```
            end
        S2:
        begin
            counter <= counter + 1;
            if (counter == TIMER) begin
                temp <= temp >> 1;
                counter <= 0;
            end
            if (temp == 16'h0000) begin
                state <= S3;
                counter <= 0;
            end
        end
        S3:
        begin
            counter <= counter + 1;
            if (counter == TIMER) begin
                temp <= (temp << 1) + 1;
                counter <= 0;
            end
            // Case flick
            if(flick == 1 && temp <= 16'h07FF) begin
                if(temp == 16'h003F || temp == 16'h07FF) begin
                    counter <= 0;
                    state <= S2;
                end
            end
            // Case no flick
            else begin
                if(temp == 16'h07FF) begin
                    counter <= 0;
                    state <= S4;
                end
            end
        end
        S4:
        begin
            counter <= counter + 1;
            if (counter == TIMER) begin
                temp <= temp >> 1;
                counter <= 0;
            end
            if (temp == 16'h001F) begin
                counter <= 0;
                state <= S5;
            end
        end
        S5:
        begin
```

```
                counter <= counter + 1;
                if(counter == TIMER) begin
                    temp <= (temp << 1) + 1;
                    counter <= 0;
                end
                // Case flick
                if(flick == 1 && temp <= 16'h07FF) begin
                    if(temp == 16'h003F || temp == 16'h07FF) begin
                        counter <= 0;
                        state <= S4;
                    end
                end
                // Case no flick
                else begin
                    if(temp == 16'hFFFF) begin
                        counter <= 0;
                        state <= S6;
                    end
                end
            end
            S6:
            begin
                counter <= counter + 1;
                if(counter == TIMER) begin
                    temp <= temp >> 1;
                    counter <= 0;
                end
                if (temp == 16'h0000) begin
                    counter <= 0;
                    state <= S7;
                end
            end
            S7:
            begin
                temp <= 16'hFFFF;
                counter <= counter + 1;
                if(counter == TIMER) begin
                    state <= S0;
                    counter <= 0;
                end
            end
            endcase
        end
    // Assign output
    assign lamps = temp;

endmodule
```

The descriptions for the states have been demonstrated detailed in the designspec file.

1.4 Simulation

1.4.1 Normal Test

For testing purpose, I will create a testbench file with the below information:

```
module boundFlasher_tb;
    reg clk;
    reg flick;
    wire [15:0] lamps;

    boundFlasher UUT (
        .clk(clk),
        .flick(flick),
        .lamps(lamps)
    );
    // Create clock
    always #1 clk = !clk;
    initial begin
        // Reset the thing
        clk = 0;
        flick = 0;
        #4;

        // Normal test
        flick = 1;
        #4;
        flick = 0;
        #40000;
        $finish;
    end
    initial begin
        $recordfile ("waves");
        $recordvars ("depth=0", boundFlasher_tb);
    end
endmodule
```

The output of the module will be as below.

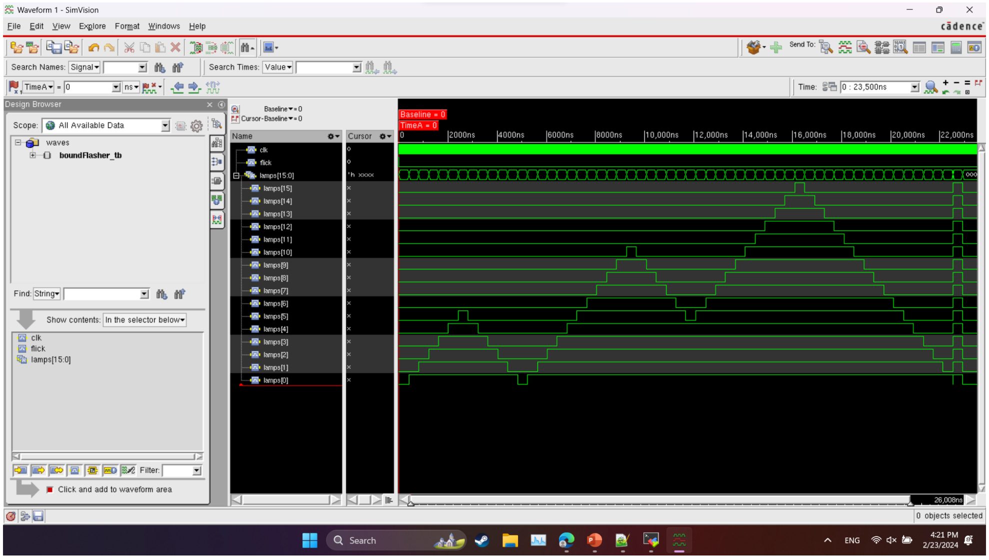
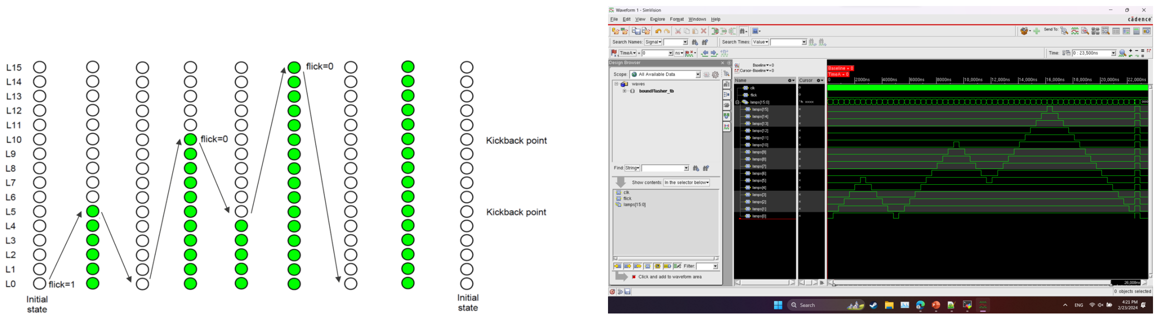


Figure 2: Normal Test without kickback.

It can be seen that, in normal condition, the module work well as expected. The waveform has the same shape as in the theory.



(a) Expected Output (b) Actual Output

Figure 3: Normal Test without additional condition.

1.4.2 Additional Condition Testing

Similar methods are employed in this section. Below is the content of the testbench file.

```
module boundFlasher_tb;
    reg clk;
    reg flick;
    wire [15:0] lamps;

    boundFlasher UUT (
        .clk(clk),
        .flick(flick),
        .lamps(lamps)
    );
    // Create clock
    always #1 clk = !clk;
    initial begin
        // Reset the thing
        clk = 0;
        flick = 0;
        #4;

        // Normal test
        flick = 1;
        #4;
        flick = 0;

        // Slide flick waveform test
        @(UUT.state == 3) begin
            #3500;
            flick = 1;
        end
        @(UUT.state == 2) flick = 0;
        #40000;
        $finish;
    end
    initial begin
        $recordfile ("waves");
        $recordvars ("depth=0", boundFlasher_tb);
    end
endmodule
```

And here is the output of the program.

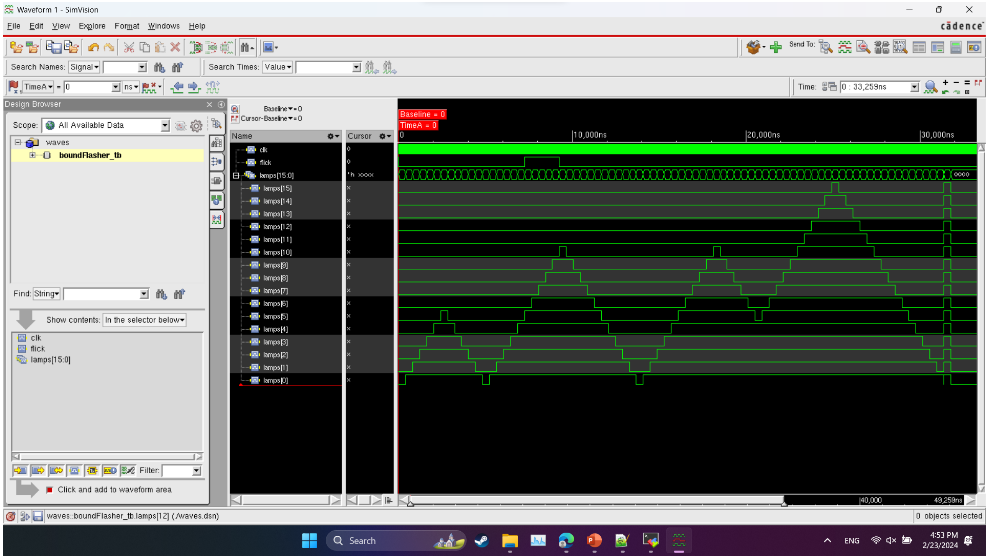
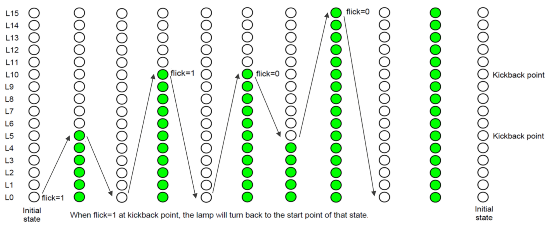
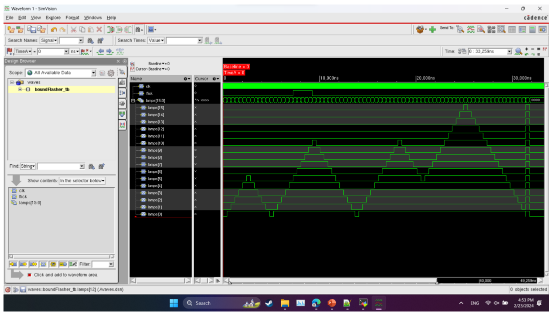


Figure 4: Test with Additional Condition.

Compare between the expected output in the slide with the recent output we will have:



(a) Expected Output



(b) Actual Output

Figure 5: Normal Test without additional condition.

1.4.3 Extra Case

In this section, I will create my own test case to see if the system works well as I expected or not. The content below belongs to the testbench file of this test case.

```
module boundFlasher_tb;
    reg clk;
    reg flick;
    wire [15:0] lamps;

    boundFlasher UUT (
        .clk(clk),
        .flick(flick),
        .lamps(lamps)
    );
    // Create clock
    always #1 clk = !clk;
    initial begin
        // Reset the thing
        clk = 0;
        flick = 0;
        #4;

        // Normal test
        flick = 1;
        #4;
        flick = 0;

        // Myself flick waveform test
        @(UUT.state == 5) flick = 1;
        @(UUT.state == 4) flick = 0;
        @(UUT.state == 5) begin
            #2000;
            flick = 1;
        end
        @(UUT.state == 4) flick = 0;
        #40000;
        $finish;
    end
    initial begin
        $recordfile ("waves");
        $recordvars ("depth=0", boundFlasher_tb);
    end
endmodule
```

Here is the output picture.

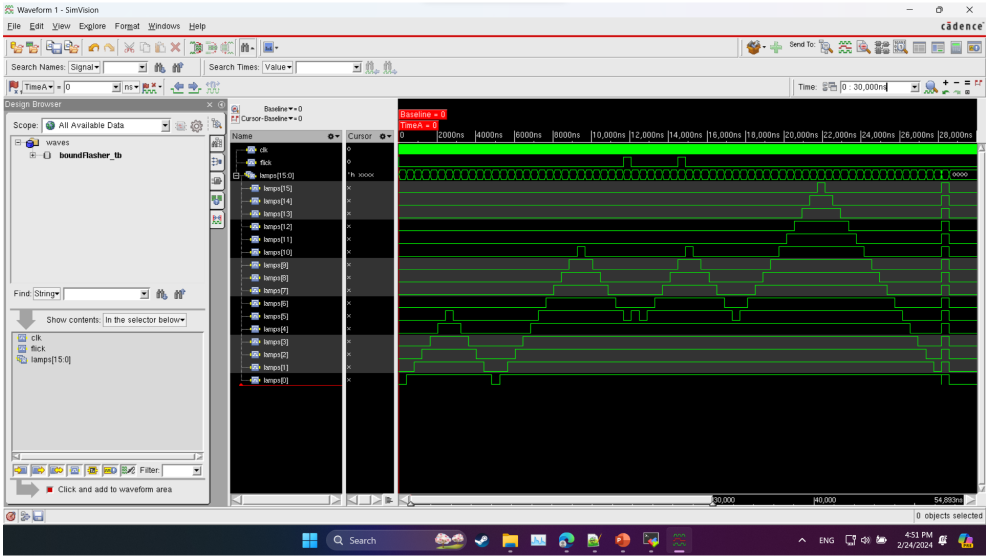


Figure 6: Extra case waveform.

What actually happened is that in state S5 at the beginning, the Flick signal was triggered so that when lamps[5] turned on, it would become the kickback point so that it would go back to state S4. Next, when it came back to state S5, after some delay time, the Flick signal was triggered again at the time lamps[9] turned on. So that when the lamps[10] turned on, it will then become the kickback point and go back to state S4. After all, the system will work normally.

The source code of the problem will be available at: GitHub