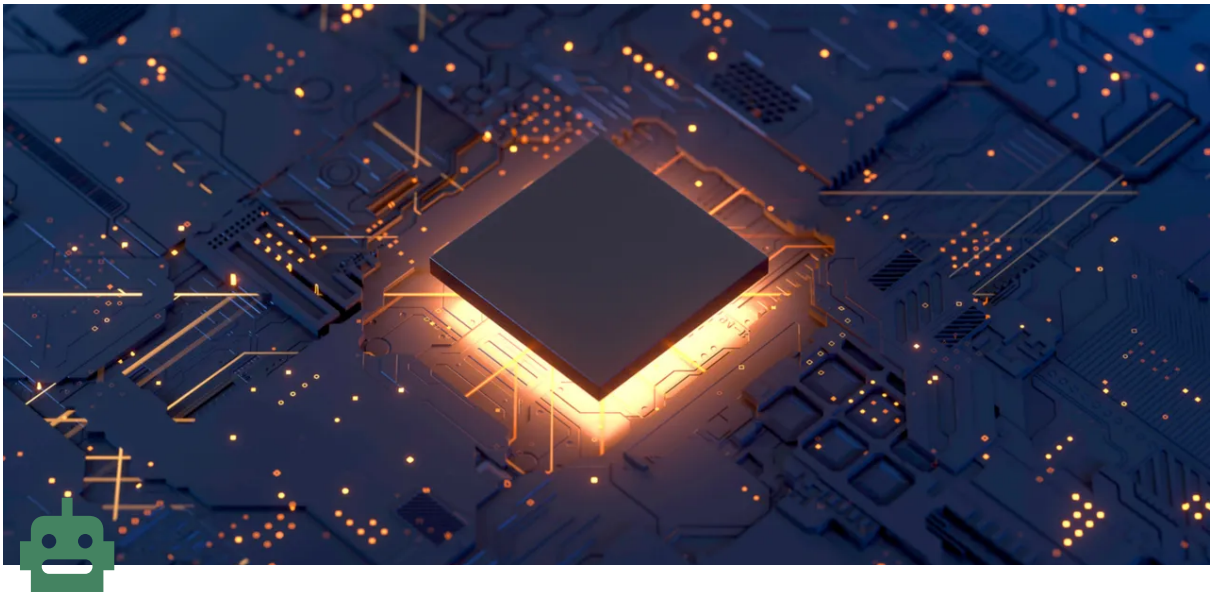


IAC Lab 3

Task 0

Started off the lab by downloading a verification package, `GTest`, with the following command.

```
sudo apt install libgtest-dev
```

I then navigated to the main testbench folder, which contains a `main.cpp` file and a `doit.sh` file for compiling the code. The main.cpp file contains simple setup and teardown functions, and also several testcases.

While the first testcase `AddTest` is guaranteed to pass (as the sum of 2 and 4 is 6), I created a second

```
TEST_F(TestAdd, AddTest)
{
    EXPECT_EQ(add(2, 4), 6);
}

TEST_F(TestAdd, AddTest2)
{
    EXPECT_EQ(add(3, 7), 9);
}
```

testcase `AddTest2` which fails no matter what (as the sum of 3 and 7 is not 9). Running the `doit.sh` script to compile and run the `main.cpp` file, we obtain the following in the output.

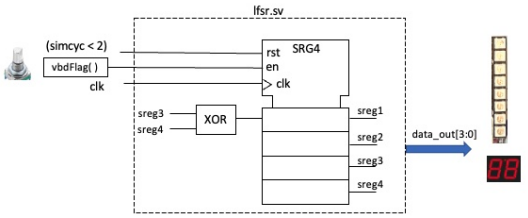
The `AddTest2` unit test fails in this instance, with red output as shown.

```
main.cpp:27: Failure
Expected equality of these values:
  add(3, 7)
    Which is: 10
  9
[  FAILED  ] TestAdd.AddTest2 (0 ms)
[----------] 2 tests from TestAdd (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] TestAdd.AddTest2
```

Task 1

I proceeded to create the component lfsr.sv, which is a Linear Feedback Shift Register (LFSR). The LFSR module produces output bits with a linear function of the previous state and contains the following inputs and output signals -

- a reset signal `rst`, asserted for the first 2 simulation cycles `(simcyc < 2)`
- an enable signal `en`, asserted when `vbdFlag()` is high
- a clock signal `clk`
- an output signal 4 bits wide, `data_out[3:0]`, which comprises each bit in the shift register.



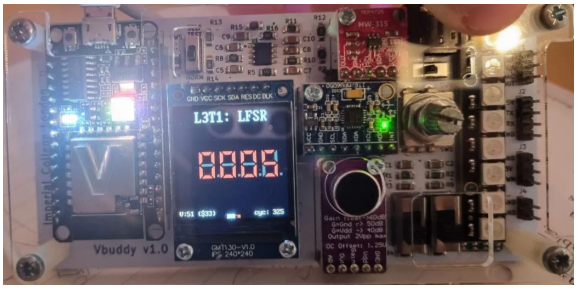

```
module lfsr (
    input logic clk,
    input logic en,
    input logic rst,
    output logic [4:1] data_out
);

    logic [4:1] sreg;

always_ff @ (posedge clk, posedge rst)
    if (rst) sreg <= 4'b1;
    else if (en) sreg <= {sreg[3:1], sreg[4]^sreg[3]};

assign data_out = sreg;
endmodule
```

The reset signal is asynchronous, meaning that we have to consider it when triggering the flip-flop, as given by the line `always_ff @ (posedge clk, posedge rst)`.

The 3rd and 4th bit of the LFSR undergoes an exclusive-OR (XOR) operation each clock cycle to give the 1st bit of the LFSR for the new clock cycle, with `sreg <= {sreg[3:1], sreg[4]^sreg[3]}`.

This gives the primitive polynomial `1 + X^3 + X^4`.

We visualise our results by printing the value of `data_out` on the 7-segment display on VBuddy. As expected, we obtain a pseudo-random binary sequence (PRBS) ranging between 0 and 15. When the rotary encoder is pressed, the number stops as the enable signal `en` is now de-asserted.



We then proceed to run the verification tests, comprising the verification testbench file `verify.cpp` and script `verify.sh` . In the verification testbench file, a unit test checks if the output of the initial and first state matches are expected, which is 1 and 2 respectively. The entire sequence is then tested, as it repeats every 15 cycles, a feature of a PRBS. As shown in the output, the module works as expected.

```
TEST_F(TestDut, SequenceTest)
{
    std::vector<int> expected = {
        0b0001, 0b0010, 0b0100, 0b1001,
        0b0011, 0b0110, 0b1101, 0b1010,
        0b0101, 0b1011, 0b0111, 0b1111,
        0b1110, 0b1100, 0b1000, 0b0001};

    for (int exp : expected)
    {
        EXPECT_EQ(top->data_out, exp);
        runSimulation();
    }
}
```

```
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from TestDut
[ RUN      ] TestDut.InitialStateTest
[       OK ] TestDut.InitialStateTest (0 ms)
[ RUN      ] TestDut.SequenceTestMini
[       OK ] TestDut.SequenceTestMini (0 ms)
[ RUN      ] TestDut.SequenceTest
[       OK ] TestDut.SequenceTest (0 ms)
[----------] 3 tests from TestDut (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
```

For the challenge, we are tasked to modify `lfsr.sv` into `lfsr_7.sv`, a 7-bit PRBS generator, with the 7th order primitive polynomial `1 + X^3 + X^7`, and test with the `verify_7.sh` script.

We do the simple modification by extending the output signal `data_out` to 7 bits and then applying a XOR operation to the 3rd and 7th bit during each clock cycle to yield the new 1st bit. Running the verification scripts `verify_7.cpp` and `verify_7.sh` we see that our module passes the testcases.

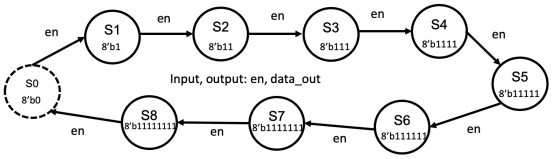
```
module lfsr (
    input logic clk,
    input logic en,
    input logic rst,
    output logic [7:1] data_out
);

    logic [7:1] sreg;

always_ff @ (posedge clk, posedge rst)
    if (rst) sreg <= 7'b1;
    else if (en) sreg <= {sreg[6:1], sreg[3]^sreg[7]};

assign data_out = sreg;
endmodule
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestDut
[ RUN      ] TestDut.InitialStateTest
[       OK ] TestDut.InitialStateTest (0 ms)
[ RUN      ] TestDut.SequenceTest
[       OK ] TestDut.SequenceTest (0 ms)
[----------] 2 tests from TestDut (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

Task 2

In this task, we design a FSM that simulates the starting lights of Formula 1 racing - a series of red lights turn ON one by one, until all lights are ON. Then all of them turn OFF simultaneously after a random delay.

We seek to implement this in System Verilog.



- We start off by implementing the necessary input signals - clock (`clk`), enable (`en`), reset (`rst`), and the 8-bit output signal, `data_out [7:0]`
- We then declare a type `my_state`, which can take on a series (an enumeration) of values from `S0` to `S8`. There will be 2 instances of the `my_state` variable in the FSM, `current_state` and `next_state`
- At each clock cycle, we check if the reset signal is asserted. If it is asserted, we reset the current state to S0, else we set the

current state to the next possible state

- We then implement the next state logic, assigning the next state according to the state model, if the enable signal `en` is asserted.
- Lastly, we assign the output based on the current state, from `8'b0` (an 8-bit array composed only of zeroes) to `8'b11111111` (an 8-bit array composed only of ones). This will help the programming of the light strip on VBuddy later.

```
module lfsr (
    input logic clk,
    input logic en,
    input logic rst,
    output logic [7:0] data_out
);

    typedef enum {S0, S1, S2, S3, S4, S5, S6, S7, S8} my_state;
    my_state current_state, next_state;

always_ff @ (posedge clk)
    if (rst) current_state <= S0;
    else current_state <= next_state;

always_comb
    case (current_state)
    S0: if (en) next_state = S1;
        else next_state = current_state;
    S1: if (en) next_state = S2;
        else next_state = current_state;
    S2: if (en) next_state = S3;
        else next_state = current_state;
    S3: if (en) next_state = S4;
        else next_state = current_state;
    S4: if (en) next_state = S5;
        else next_state = current_state;
    S5: if (en) next_state = S6;
        else next_state = current_state;
    S6: if (en) next_state = S7;
        else next_state = current_state;
    S7: if (en) next_state = S8;
        else next_state = current_state;
    S8: if (en) next_state = S0;
        else next_state = current_state;
    default: next_state = S0;
    endcase
```

```
always_comb
    case (current_state)
        S0: data_out = 8'b0;
        S1: data_out = 8'b1;
        S2: data_out = 8'b11;
        S3: data_out = 8'b111;
        S4: data_out = 8'b1111;
        S5: data_out = 8'b11111;
        S6: data_out = 8'b111111;
        S7: data_out = 8'b1111111;
        S8: data_out = 8'b11111111;
        default: data_out = 8'b0;
    endcase

endmodule
```

We then proceed to verify the functionality of the module `f1_fsm.sv` using the verification testbench file `verify.cpp` and `verify.sh`.

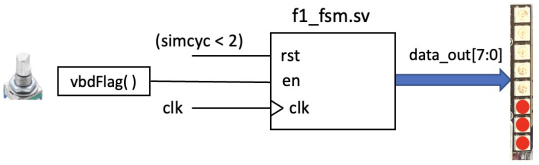
The unit tests check if the initial value of `data_out` is 0, and whether the number of `1` bits in `data_out` increases according to the model as the simulation progresses.

The module works as anticipated.

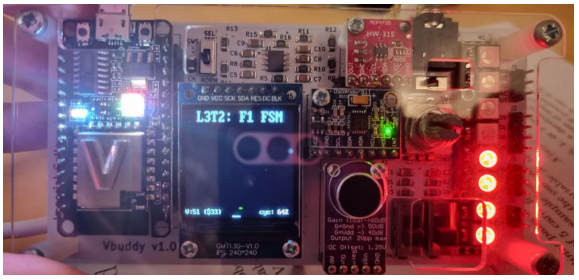
```
TEST_F(TestDut, InitialStateTest)
{
    top->rst = 1;
    runSimulation();
    EXPECT_EQ(top->data_out, 0x00);
}

TEST_F(TestDut, FSMTest)
{
    top->rst = 1;
    runSimulation();
    EXPECT_EQ(top->data_out, 0b0000);

    top->rst = 0;

    std::vector<int> expected = {
        0b0000'0000, 0b0000'0001, 0b0000'0011, 0b0000'0111,
        0b0000'1111, 0b0001'1111, 0b0011'1111, 0b0111'1111,
        0b1111'1111, 0b0000'0000};

    for (int exp : expected)
    {
        EXPECT_EQ(top->data_out, exp);
        runSimulation();
    }
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestDut
[ RUN      ] TestDut.InitialStateTest
[       OK ] TestDut.InitialStateTest (0 ms)
[ RUN      ] TestDut.FSMTest
[       OK ] TestDut.FSMTest (0 ms)
[----------] 2 tests from TestDut (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

We then proceeded to drive the pixel bar and cycle through the F1 light sequence using the rotary switch with the `vbdFlag()` function in mode 1 (one-shot mode) to drive the `en` signal as shown.



After uploading the code to VBuddy, we observe how the individual lights in the light bar activate sequentially as the rotary encoder is pressed several times. When all lights are lit, the next press resets all lights and the entire light bar turns off.



Task 3

We proceeded to explore the `clktick.sv` System Verilog module, which provides a clock impulse every N+1 clock cycles.

The clktick.sv module works by storing an logic variable `count` which tracks the number of cycles elapsed since the last clock impulse. The `count` variable is a 16-bit number.

Initially, `count` is set to N, the number of clock cycles between each impulse. On each positive edge of the clock, `count` decrements by 1. When `count` eventually equals zero, the output logic signal `tick` is asserted and the count variable is reset back to N.

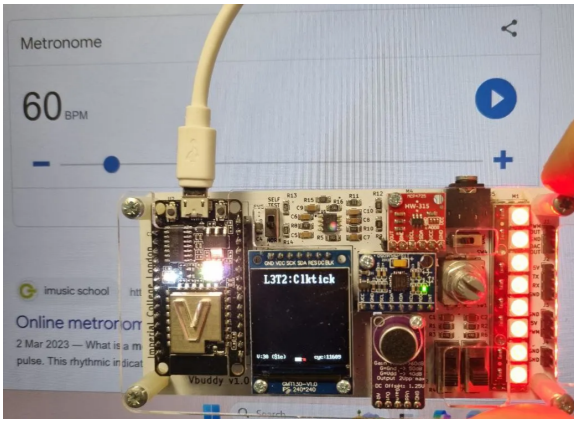
In the testbench file `clktick_tb.cpp`, we set the value of N to the rotary encoder value via `vbdValue()`.

Every time the output signal `tick` is asserted, `vbdBar()` will flash on and off in an alternating manner. This is done by applying a XOR operation to the prior `lights` variable with 0xFF, causing the value of lights to alternate between 0x00 and 0xFF each tick.

We seek to calibrate the circuit (under simulation) to find what value of N gives us a tick period of 1 second. After compiling and testing the `clktick.sv` module, we use a metronome to generate a beat at 60 bpm. We then adjust the rotary

```
module clktick #(
    parameter WIDTH = 16
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // enable signal
  input  logic [WIDTH-1:0] N,            // clock divided by N+1
  output logic                     tick      // tick output
);

logic [WIDTH-1:0] count;

always_ff @ (posedge clk)
    if (rst) begin
        tick <= 1'b0;
        count <= N;
        end
    else if (en) begin
        if (count == 0) begin
            tick <= 1'b1;
            count <= N;
            end
        else begin
            tick <= 1'b0;
            count <= count - 1'b1;
            end
        end
endmodule
```

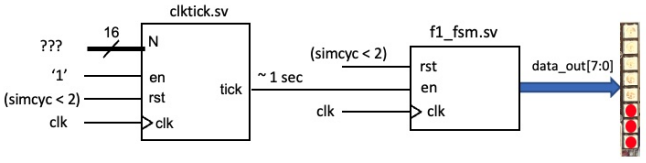
```
// Display toggle neopixel
if (top->tick)
{
    vbdBar(lights);
    lights = lights ^ 0xFF;
}
```



switch to change the flash rate of the pixels to match the metronome.
The `vbdValue()` shown on bottom left of the TFT display would be the value for N which gives a tick period of 1 second.

The reason for this calibration is that the Verilator simulation is NOT in real time. Every computer will work at different rates and therefore takes different amount of time to simulate one cycle of the clock signal `clk`. For my computer, N is around 30 for a tick period of 1 sec (one tick pulse every second).

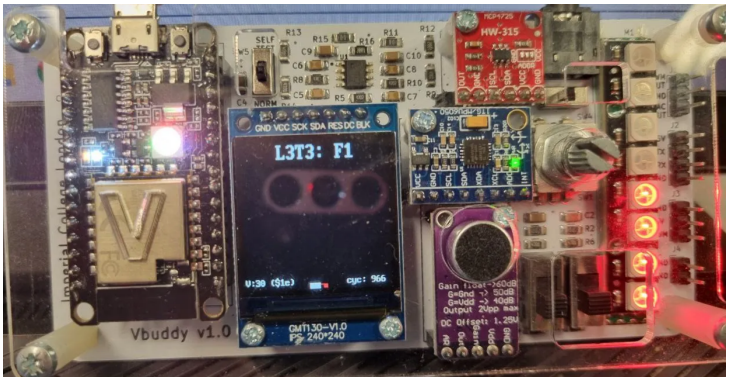
As a challenge, I sought to combine `clkctick.sv` with `f1_fsm.sv` so that the F1 light sequence is cycled through automatically with 1 second delay per state transition.



We create a simple top-level module top.sv which contains both `clktick.sv` and `f1_fsm.sv` System Verilog modules. Using a logic connector count, we feed the tick output from the `clktick` module into the enable input `en` of the `f1_fsm` module. Whenever the F1 FSM is enabled, the number of positive bits in the `data_out` output increments by 1. This is then connected to the LED light bar on VBuddy via the `vbdBar()` function in the testbench file `challenge_tb.cpp`.

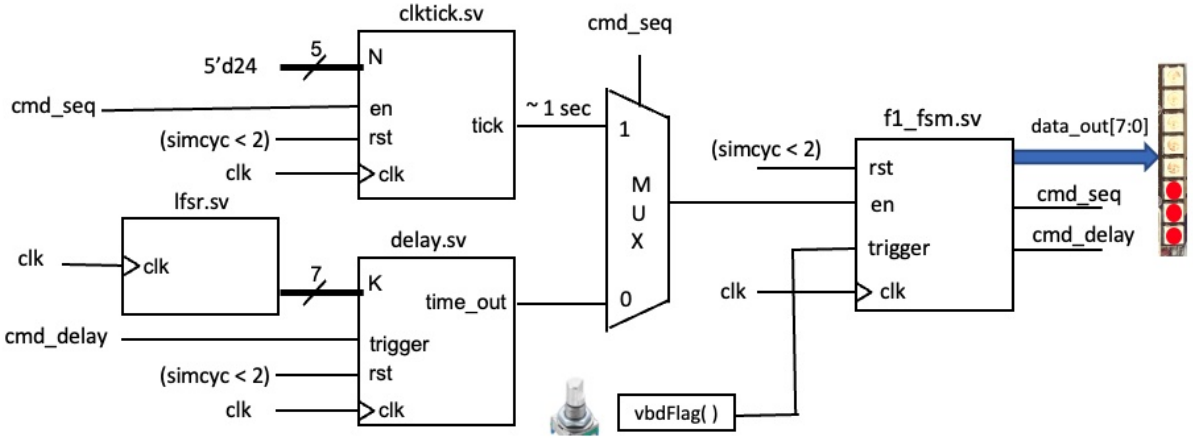
Altogether, after compiling and testing the module, we get a light bar which works similar to that of a F1 light bar.

```
module top #(
    parameter WIDTH = 16
)(
    input logic clk,
    input logic en,
    input logic rst,
    input logic [WIDTH-1:0] N,
    output logic [7:0] data_out
);

logic [WIDTH-1:0] count;

clktick myClock (
    .clk (clk),
    .rst (rst),
    .en (en),
    .N (N),
    .tick (count)
);

f1_fsm myF1 (
    .clk (clk),
    .en (count),
    .rst (rst),
    .data_out (data_out)
);

endmodule
```



Task 4

Lastly, we seek to implement a full version of the F1 starting light. The diagram below shows a full version of the F1 light design that combines all the components created so far.



Initially, `cmd_seq` will be 1 and `cmd_delay` will be 0.

1. The clock module clktick.sv will provide 8 clock pulses at 1s intervals. The clock pulses will pass through the multiplexer and into the F1 FSM as enable signals, causing the number of positive bits in the output signal `data_out` to increase in sequence and the lights to light up one by one.
2. When `data_out[7:0] = 8'b11111111`, `cmd_seq = 0` and `cmd_delay = 1`. This will trigger the multiplexer to receive inputs from the `delay.sv` module instead of `clktick.sv`.
3. The Linear Feedback Shift Register (LFSR) module, `lfsr.sv`, generates a 7-bit random number which is the number of clock cycles delay.sv will wait

for before asserting the `time_out` signal.

4. When the `time_out` signal is asserted, the multiplexer passes it to the enable input of f1_fsm.sv, causing the `data_out` to go to zero and all the lights to turn off.

On the testbench, we can then use two new VBuddy functions to measure the reaction time -

1. Once all the lights are off after a random delay, the testbench calls the `vbdInitWatch()` function to start the VBuddy stop watch.
2. The user reacts to the lights going OFF and presses the switch as quickly as possible. `VBuddy` automatically records the elapsed time since the stop watch started.
3. The testbench calls `vbdElapsed()` function to read the reaction time in milliseconds.
4. The testbench reports by sending it to VBuddy as a message on the TFT screen. This can be displayed in decimal using a binary to BCD converter, or in hex.

To implement all these, we create an overall top-level module top.sv, which contains the corresponding input and output signals.

- `clk` is the input clock signal
- `rst` is the input reset signal
- `cmd_seq` is the input signal which activates `clktick.sv` which will send pulses in 1 second intervals in real-life, and instructs the multiplexer to select between `clktick.sv` and `delay.sv` for the listening of impulses
- `cmd_delay` is the input signal which activates `delay.sv`, which takes the randomly generated number by the LFSR and applies a respective random delay to the output pulse
- `N` is the 5-bit input signal which we have previously established to be 0 - it is the approximate number of clock cycles elapsed for 1 second to occur and is used to mark the intervals between individual lights activating
- `data_out` is the 8-bit input signal which represents the individual lights to be lit up on the pixel bar on VBuddy (after a bitmask of `0xFF` is applied)
- `cmd_seq_output` is the output signal which signals that the lights are still being lighted up in sequence - it is asserted when `data_out` is between `8'b0` to `8'b1111111` and de-asserted when `data_out` changes value to `8'b11111111`

```
module top #(
    parameter WIDTH = 5
)(
    input logic clk,
    input logic rst,
    input logic cmd_seq,
    input logic cmd_delay,
    input logic trigger,
    input logic [WIDTH-1:0] N,
    output logic [7:0] data_out,
    output logic cmd_seq_output,
    output logic cmd_delay_output
);

logic [7:1] lfsrToDelay;

logic tick;
logic timeout;
logic fsmEnable;
assign fsmEnable = cmd_seq ? tick : timeout;

lfsr_7 myLFSR (
    .clk (clk),
    .rst (rst),
    .data_out (lfsrToDelay)
);

delay myDelay (
    .clk (clk),
    .rst (rst),
    .trigger (cmd_delay),
    .n (lfsrToDelay),
    .time_out (timeout)
);

clktick myClock (
    .clk (clk),
    .rst (rst),
    .en (cmd_seq),
    .N (N),
    .tick (tick)
);

f1_fsm myF1 (
    .clk (clk),
    .en (fsmEnable),
    .rst (rst),
    .trigger (trigger),
    .data_out (data_out),
    .cmd_seq (cmd_seq_output),
    .cmd_delay (cmd_delay_output)
);

endmodule
```

- `cmd_delay_output` is the output signal which signals that all lights are lighted up and there is now a delay between all lights turning on and off - it is asserted only when `data_out` is `8'b11111111`

```
case (current_state)
    S0: begin
            data_out = 8'b0;
            cmd_seq = 1'b1;
            cmd_delay = 1'b0;
        end
    S1: begin
            data_out = 8'b1;
            cmd_seq = 1'b1;
            cmd_delay = 1'b0;
        end
    S2: begin
            data_out = 8'b11;
            cmd_seq = 1'b1;
            cmd_delay = 1'b0;
        end
```

We represent the multiplexer using a ternary condition, passing on the value from `clktick.sv` if `cmd_seq` is asserted, and `delay.sv` otherwise -

```
assign fsmEnable = cmd_seq ? t
ick : timeout
```

Most of the System Verilog files are untouched, however for the F1 FSM file `f1_fsm.sv` additional assignment instructions are added at each case to cater for `cmd_seq` and `cmd_delay`.

We make some more modifications to the testbench file. We need to force `cmd_delay` to only last for 1 clock cycle as we feed the output signal `cmd_delay_output` in the current clock cycle as the input `cmd_delay` for the next clock cycle. We do this by implementing a Boolean `isDelayPassed`.

```
// Force delay to only last 1 cycle
bool isDelayPassed = false;

// Force delay pulse to only last 1 cycle
if (top->cmd_delay_output and isDelayPassed == false){
    top->cmd_delay = 1;
    isDelayPassed = true;
} else if (top->cmd_delay_output and isDelayPassed == true) {
    top->cmd_delay = 0;
} else {
    isDelayPassed = false;
}
```

When `cmd_delay_output` is initially asserted, we pass it to `cmd_delay`, however we mark the Boolean as `True`. In subsequent clock cycles where `cmd_delay_output` is asserted, this will be ignored and `cmd_delay` remains at 0. Only when `cmd_delay_output` is de-asserted will the Boolean be marked back as `False` again.

Some modifications are also needed for the testbench file in order to

```
// Measure reaction time when all lights are null
bool in_sequence, is_timing;
```

obtain the reaction time. We declare 2 Booleans, `in_sequence` and `is_timing`.

If `top->data_out` is positive, which means that at least 1 light is lit up, we can safely assume that the light sequence is being carried out, and we mark `in_sequence` as True.

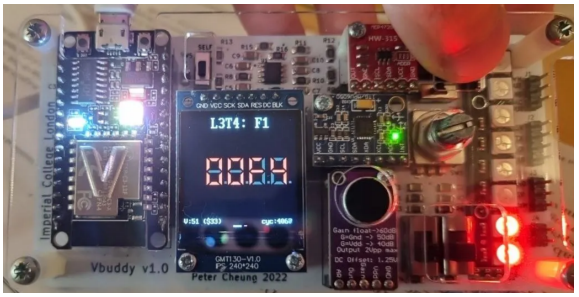
Once `top->data_out` becomes zero, which means that all lights are off, the light sequence is no longer active and we are now seeking to determine the reaction time of the user. We do so by starting the internal stopwatch with `vbdInitWatch()` and setting `in_sequence` to False and `is_timing` to True.

In the next clock cycle, if we are timing (`is_timing` is True), we enter the else-if condition. If the rotary encoder is pressed, `flag_val` is asserted and we obtain the runtime via `vbdElapsed()`. We then display each individual bit of the runtime in hexadecimal using a combination of right shifts and bitmasks. We clean up by setting `is_timing` and `in_sequence` to False.

After testing and compiling the code, we are able to obtain a fully functioning F1 countdown timer. With fast reaction time, I obtained a reaction time of `0xF4` milliseconds, which translates to 244ms.

Another attempt with delayed reaction recorded a longer reaction

```
// Get reaction time
if (top->data_out){
    in_sequence = true;
}
int flag_val = vbdFlag();

if(!is_timing){
    top->trigger = flag_val;
} else if (flag_val) {
    is_timing = false;
    in_sequence = false;
    int runtime = vbdElapsed();
    vbdHex(4, (runtime >> 12) & 0xF);
    vbdHex(3, (runtime >> 8) & 0xF);
    vbdHex(2, (runtime >> 4) & 0xF);
    vbdHex(1, (runtime) & 0xF);
}
if (top->data_out == 0){
    if (in_sequence) {
        is_timing = true;
        in_sequence = false;
        vbdInitWatch();
    }
}
```



time of about `0x161` milliseconds, which translates to 353ms.