EIE2 Instruction Set Architecture & Compiler (IAC)

Lab 3 - Finite State Machines (FSM)

Peter Cheung, @saturn691, V1.4 - 30 Oct 2025

Objectives

By the end of this experiment, you should be able to:

- be aware of industry standard testing techniques
- design and test a PRBS generator using a linear feedback shift register (LFSR)
- display 8-bit value on neopixel bar on Vbuddy
- specify a FSM in SystemVerilog
- design a FSM to cycle through the Formula 1 starting light sequence
- understand how the ***clktick.sv*** module works, and calibrate it for 1 sec tick period
- automatically cycle through F1 lights at 1 second interval
- optionally implement the full F1 starting light machine and test your reaction.

Clone this repo to your local disk.

Introduction

Hardware design is often a process that takes years from design to fabrication (except FPGAs), so any errors would be costly in time and money.

Therefore a lot of time in industry is trying to minimise errors in hardware, by catching them before they are sent out to fabrication. Whilst there are plenty of techniques, this is beyond the scope of the course. We will introduce a basic approach to testing, also known as **verification**. There will be no need to write any tests for this lab.

The assessed coursework will be verified using a framework, called GTest. GTest is an industry standard, with programs such as LLVM tested using GTest (source). Make sure, you have it installed, by running the following command:

```
        # Ubuntu
        sudo apt install libgtest-dev
        # If it doesn't work right away, see this blog post:
        # https://stackoverflow.com/questions/13513905/how-to-set-up-
googletest-as-a-shared-library-on-linux

        # MacOs
        brew install googletest
```

Then navigate to task0 and open the `main.cpp` file. Add a test case in this block. Take your time to understand the gist of this code.

```
TEST_F(TestAdd, AddTest2)
{
    // Create a test case here. Maybe fail this to see what happens?
}
```

Then run the `doit.sh` file, you should get something like (assuming you created a passing testcase):

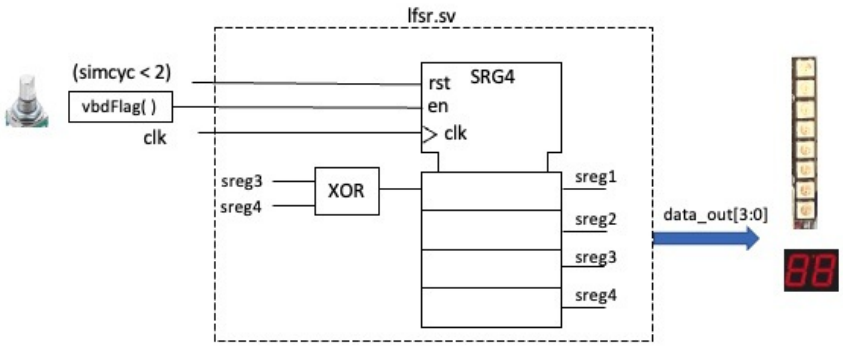
```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestAdd
[ RUN      ] TestAdd.AddTest
[       OK ] TestAdd.AddTest (0 ms)
[ RUN      ] TestAdd.AddTest2
[       OK ] TestAdd.AddTest2 (0 ms)
[----------] 2 tests from TestAdd (0 ms total)

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

Step 1 - Create the component lfsr.sv

Open the *Lab3-FSM* folder in VS code. In folder ***task1***, create the component **lfsr.sv** guided by Lecture 4 slide 17. This is your top-level circuit for this task, as described:

- All four bits of the shift register output are brought out as data_out[3:0].
- `en` is the enable signal.
- Reset is asynchronous (hint: add it to the sensitivity list) and brings the state back to 1 (not 0).



Step 2 - Verify the LFSR

Use the attached testbench script (`verify.sh`) to check your answer.

TEST YOURSELF CHALLENGE

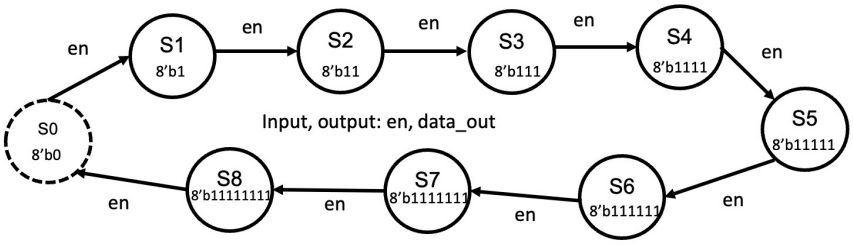
Based on the ***primitive polynomial*** table in Lecture 4 slide 16, modify ***lfsr_7.sv*** into a 7-bit (instead of 4-bit) PRBS generator. Test your design, using the `verify_7.sh` script. The 7th order primitive polynomial is:

$$1 + X^3 + X^7$$

Step 1 - Create the component f1_fsm.sv

Formula 1 (F1) racing has starting light consists of a series of red lights that turn ON one by one, until all lights are ON. Then all of them turn OFF simultaneously after a random delay.

The goal of this task is to design a FSM that cycles through the sequence according to the following FSM:



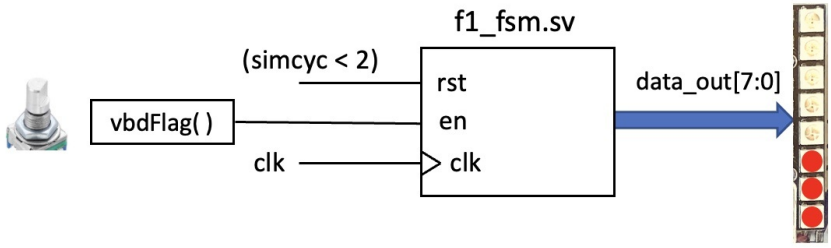
Based on the notes from Lecture 5, implement this state machine in SystemVerilog.

Step 2 - Verify the FSM

Use the attached testbench script (`verify.sh`) to check your answer.

Step 3 - Connect the FSM to Vbuddy

Drive the neopixel bar and cycle through the F1 light sequence. You should use the switch on the rotary switch with the ***vbdFlag()*** function (in mode 1) to drive the *en* signal as shown below:



Write the testbench ***f1_fsm_tb.cpp***, similar to how you wrote your other testbenches in Lab1/Lab2.

Compile and test your design. Each time you press the switch, you should step through the FSM and cycle through the F1 light sequence.

You should also display this result on the neopixel strip using the ***vbdBar( )*** function:

```
vbdBar(top->data_out & 0xFF);
```

Note that ***vbdBar()*** takes an unsigned 8-bit integer parameter between the value 0 and 255. Therefore you must mask *data_out* with 0xFF.

In Lecture 4 slides 9 & 10, you were introduced to the ***clktick.sv*** module. The interface signals for this module is:

```
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
);
```

In the *task3* folder of this repo, you are provided with the testbench ***clktick_tb.cpp*** and shell script ***clktick.sh*** to build and test the ***clktick*** module.

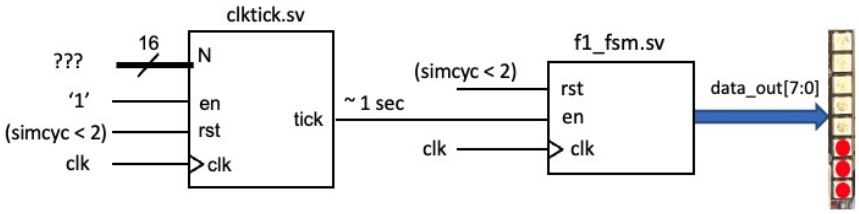
The testbench flashes the neopixel strip LEDs on and off at a rate determined by N. Our goal is to calibrate the circuit (under simulation) to find what value of N gives us a tick period of 1 sec.

Compile and test the ***clktick.sv*** module. Use the metronome app on Google (just search for metronome) to generate a beat at 60 bpm. Now adjust the rotary switch to change the flash rate of the neopixels to match the metronome. The ***vbdValue()*** shown on bottom left of the TFT display is the value for N which gives a tick period of 1 second! (Why?)

The reason that we need to do this calibration is that the Verilator simulation of your design is NOT in real time. Every computer will work at different rate and therefore takes different amount of time to simulate one cycle of the clock signal *clk*. For a 14" M1 Macbook Pro (my computer), N is around 24 for a tick period of 1 sec (i.e. one tick pulse every second).

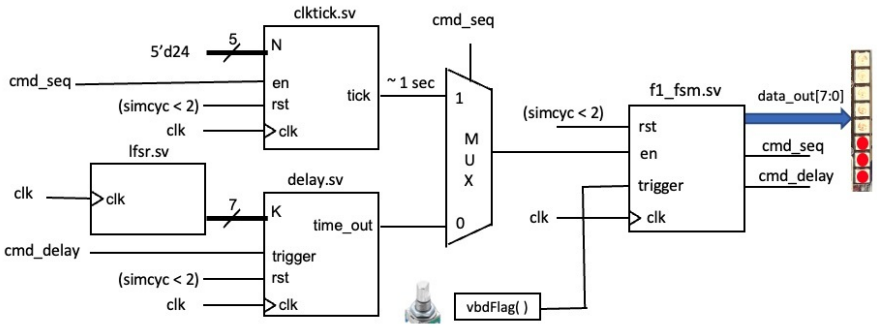
TEST YOURSELF CHALLENGE

Implement the following design by combining ***clkctick.sv*** with ***f1_fsm.sv*** so that the F1 light sequence is cycle through automatically with 1 second delay per state transition.



Complete this task only if you have time. It is challenging and fun, but also you may find this time consuming.

The follow diagram shows a full version of the F1 light design that combines all many components you have created so far.



The ***delay.sv*** module is provided. This module is from Lecture 5 slides 16 & 17. When trigger is asserted (goes from low to high), it starts counting K clock cycles. At which time, *time_out* goes high for one clock cycle. This works in a similar way to clktick.sv, except: 1. Instead of the *en* signal, we use a *trigger* signal, which is edge. 2. The FSM can only be triggered again after the _trigger_signal has returned to zero.

You also need to modify ***f1_fsm.sv*** to include a trigger input which kicks off the whole sequence. It also has two additional output signals:

1. *cmd_seq* which is high during the sequencing of *data_out[7:0]* from 8'b1 to 8'b11111111.
2. *cmd_delay* which triggers the start of the ***delay.sv*** component.

You may use the 7-bit LFSR from Task 1 to provide the random delay between all LED ON to all LED OFF.

Finally, in the testbench, you may use two new Vbuddy functions added in version 1.1 to measure the reaction time:

1. Once all the lights are OFF after a random delay, the testbench calls ***vbdInitWatch()*** function to start Vbuddy's stop watch.
2. 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. You may want to display this in binary, using a binary to BCD converter, or in hex.