Lab 1: Linear Feedback Shift Registers (LFSRs)

1 Introduction

1.1 Purpose

The purpose of this assignment is to refresh your skills on register transfer-level (RTL) design, waveform simulation, and stimulus-based behavioral verification. Basic RTL design is covered more extensively in other courses like ECE 6443 (VLSI System and Design) and ECE 6463 (Advanced Hardware Design). This lab aims to refresh your knowledge on these topics while also developing a component useful for your final chip implementation.

1.2 Background

A Linear Feedback Shift Register (LFSR) is a sequential digital circuit composed of flip-flops arranged as a shift-register. On each clock cycle, the register shifts its stored bits, and a new bit is fed back based on a linear combination (typically XOR) of selected bit positions, known as taps. The initial state of the register is called the **seed**, which determines the resulting sequence.

LFSRs produce pseudo-random bit sequences which occur *periodically*. This means that they output sequences in a fixed cycle. The sequence length that can be produced by an LFSR is based on its tap positions and can be as large as $2^n - 1$ for an n-bit LFSR.

LFSRs are widely used in hardware testing (Built-In Self-Test – BIST), error detection (Cyclic Redundancy Check – CRC), pseudo-random number generation, and even cryptography, due to their pseudo-random properties and extremely small logic size. In this class, it is highly likely that you will want to emulate receiving data from high-speed off-chip interfaces. LFSRs are easily capable of generating hundreds of gigabits of data per second, whereas the off-chip interfaces we use in this course will struggle to achieve high data rates (we will discuss off-chip interfaces later in the course).

2 PRBS7 Specification

The Pseudorandom Binary Sequence of order 7 (PRBS7) is a maximal-length sequence of $2^7 - 1 = 127$ bits, generated by a 7-bit LFSR defined by the polynomial:

$$x^7 + x^6 + 1$$

This polynomial indicates there are feedback taps at bits D6 and D5 (bit D6 being the most significant bit, and bit D0 the least significant bit).

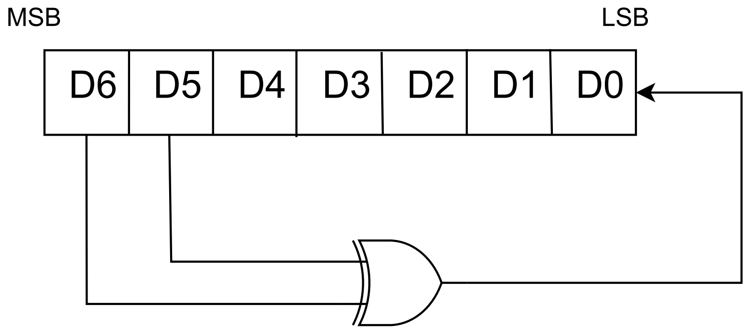

Figure 1: Logical diagram of an LFSR implementing the PRBS7 specification

Specifically, the LFSR operates as follows:

1. XOR bits D6 and D5.
2. Shift the register bits one position (left shift).
3. Insert the XOR result into the least significant bit position (D0).

Initialized with a **non-zero** seed, the PRBS7 generates a deterministic and reproducible 127-bit sequence, cycling through all possible **non-zero** 7-bit patterns before repeating. A seed of zero will produce a sequence of all zeroes.

2.1 Example Output

Assuming the initial seed is 7'b1100111, the LFSR will produce the following output values using the PRBS7 specification:

Cycle	LFSR Output
0	1100111
1	1001110
2	0011101
3	0111010
4	1110101
...	...
126	0110011
127	1100111
...	...

Table 1: Example output sequence from PRBS7 given a seed of 7'b1100111.

Note how the seed value (the value on cycle 0) is repeated on cycle 127, which occurs after all 127 non-zero values have been output in cycles 0-126.

3 Objective

- Implement a 7-bit LFSR SystemVerilog module which follows the PRBS7 specification.
- Use the following module interface:

```
module lfsr (
  input  logic       clk,
  input  logic       reset,    // active-high synchronous reset
  input  logic       load,     // load seed into LFSR
  input  logic       enable,   // enable LFSR shift
  input  logic [6:0] seed,     // 7-bit seed value
  output logic [6:0] lfsr_out  // current LFSR state
);
```

  - The LFSR behavior is controlled using the `reset`, `load`, and `enable` signals.
  - When `reset` is high, the register is reset to all 1s.
  - When `load` is high, `seed` is copied into the LFSR.
  - When `enable` is high, the LFSR shifts one step on each clock cycle, updating its state according to the PRBS7 specification.
  - `reset` takes precedence over `load`, and `load` takes precedence over `enable`.
  - If none of the control signals are asserted, the LFSR retains its current value.
  - All signals are synchronous to `clk`.

- Create one or more **blackbox** testbenches to test your design (do not access any internal signals of the design under test (DUT) in the testbench)
- Submit the following files to the autograder:
  - `lfsr.sv` (design)
  - `lfsr_tb_*.sv` (one or more testbenches)

  The `*` indicates a wildcard pattern match, meaning `*` can be any valid character in a unix filepath. Example filenames might include `lfsr_tb_1.sv`, `lfsr_tb_2.sv`, `lfsr_tb_reset.sv`, etc. For instructions on how to submit files to the autograder, please see the instructions on BrightSpace.

3.1 Testbench Output

Testbenches must report their status by printing the string `@@@PASS` if the design behavior is correct or `@@@FAIL` if the design behavior is incorrect. These strings must be printed **verbatim** (case-sensitive, and including the `@` symbol). Testbenches should print **one** of the two statuses and **only once**. If a testbench prints `@@@PASS` and/or `@@@FAIL` more than once, or both together, the testbench will be ignored and not included in the grading.

It is therefore recommended that you `$display("@@@PASS")` only at the end of your testbench after all tests have passed (right before `$finish()`) and call `$finish()` immediately after any `$display("@@@FAIL")` (you do not need to identify more than 1 bug in a design). You are free to display any other string for your own purposes, such as `"pass"`, `"TEST FAILED"`, etc. as they will be ignored in the autograder.

4 Grading

Both your design **and** your testbench(es) will be graded by the autograder. Your design will be checked for correctness by running it against several hidden instructor test cases. You will receive points for each test that you pass. Your testbench(es) will be checked for coverage and thoroughness by running it/them against several hidden instructor buggy designs. You will receive points for each buggy design you catch, meaning at least 1 testbench correctly reports the design as failing. Test benches will be ignored if they report a correct design as buggy. Your final score is determined by the total score of your **best** submission at the time of the deadline.

This assignment has a firm deadline (check BrightSpace). No late submissions will be accepted.

5 Tips

- This design portion of this assignment can be accomplished with very few lines of code. The majority of time will likely be spent on creating testbenches which can catch the buggy designs.
- Buggy designs are constructed by taking a specific item from the specification and then modifying it to be incorrect. Therefore, a thorough test strategy is to take every line of the specification and ensure that each item has a dedicated test for it within at least one of your testbenches.
- Tests on the autograder are ordered roughly by difficulty (i.e. Test 1 is easier than Test 2, which is easier than Test 3, etc.). If you are passing the more advanced tests but failing the easy tests, try to check for simpler problems with your design/testbenches.
- The earlier you start on the assignment, the more chances you will have to submit to the autograder with feedback.