# EE 209 Lab 6 – We Value Your Feedback

## 1 Introduction

In this lab you will implement an encryption/decryption engine using a linear feedback shift register (LFSR). Using these LFSRs we can create a system that will encrypt a stream of bytes for storage or transmission and then be decrypted upon receipt. This will be a simulation only lab where we view the results via simulation and ensure our design works rather than placing it on the actual FPGA board.

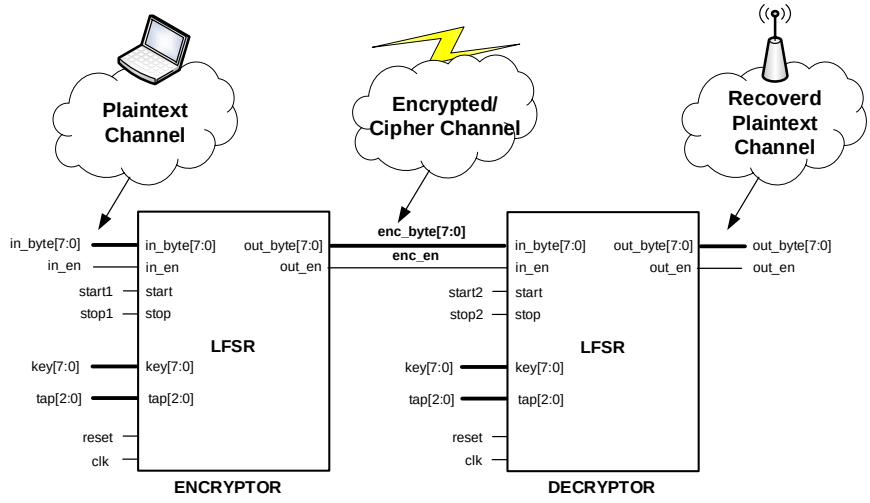
## 2 What you will learn

This lab is intended to help you integrate many of the concepts taught in the first portion of this course. You will design a state machine for control, a shift register for the datapath, and other "glue" logic to combine various components.

> **Important:** *Read through this entire document once or twice. Then go back and study the overall circuit diagram on page 3. Then perform the prelab. Only at that point should you start attempting to write your Verilog code.*

## 3 Background Information and Notes

**Encryption Systems:** To maintain confidentiality and access control, data generated by an entity ("plaintext" data) can be transformed to an encrypted form ("ciphertext" data) for storage or transmission. When data is retrieved, an inverse transform should be performed to recover the original ("plaintext") data. One approach to do this is a linear-feedback shift register (LFSR) where both transmitter and receiver both know a secret key value which allows them to communicate (i.e. perform an inverse transformation of encrypted data). The system relies on the fact that **((A XOR B) XOR B) = A**.

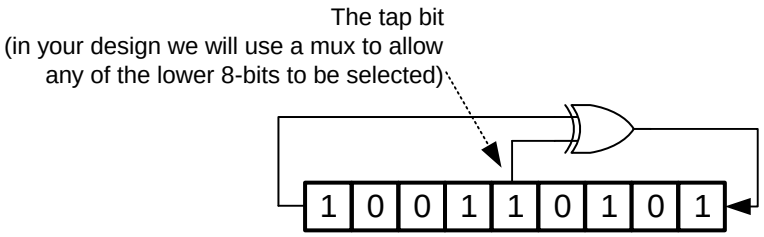



**View of the encryption/decryption system using LFSRs.**

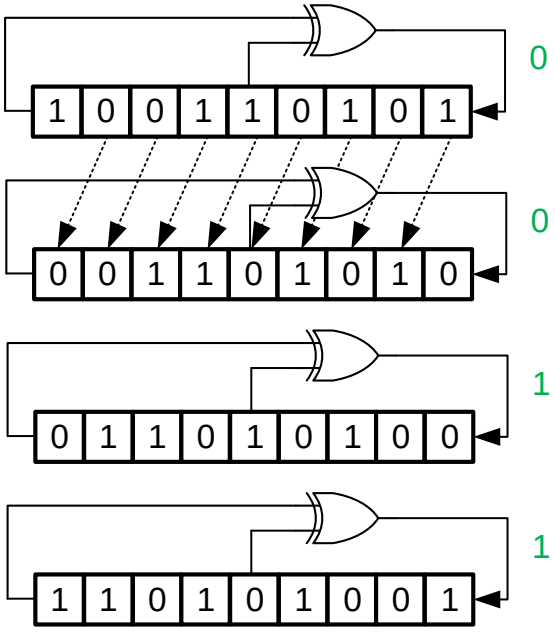
An LFSR relies on producing a pseudo-random sequence of values that is then XOR'ed with the original data stream to produce the cipher text. If the receiver can generate the same sequence of pseudo-random values then it can XOR the ciphertext with these values to recover the original.

**plaintext XOR LFSR-value => ciphertext**
**ciphertext XOR LFSR-value => plaintext**

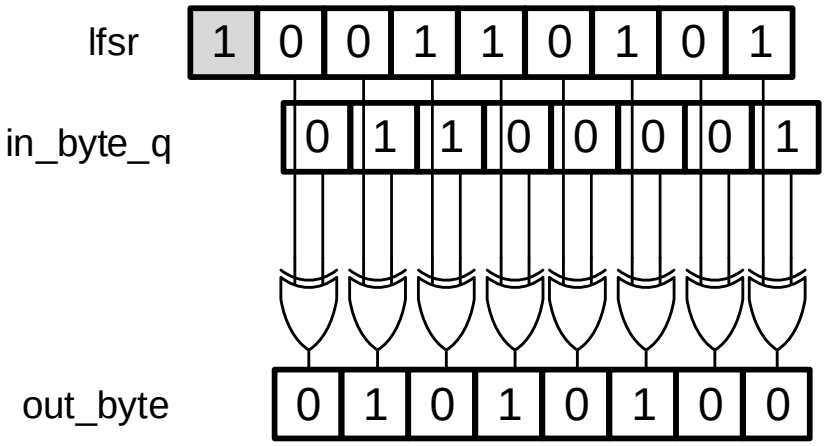
**LFSRs:** A linear feedback shift register is a shift register that is initialized with a seed (key) value. Then each clock cycle the bits are shifted one step in a given direction (say to the left). The left-most bit will be lost but before it is dropped, it will be XOR'd with a selected bit from somewhere else in the shift register. The selected bit is called the "tap". By XOR'ing the tap and the last bit, we produce the next bit that will be input into the right side of the register.



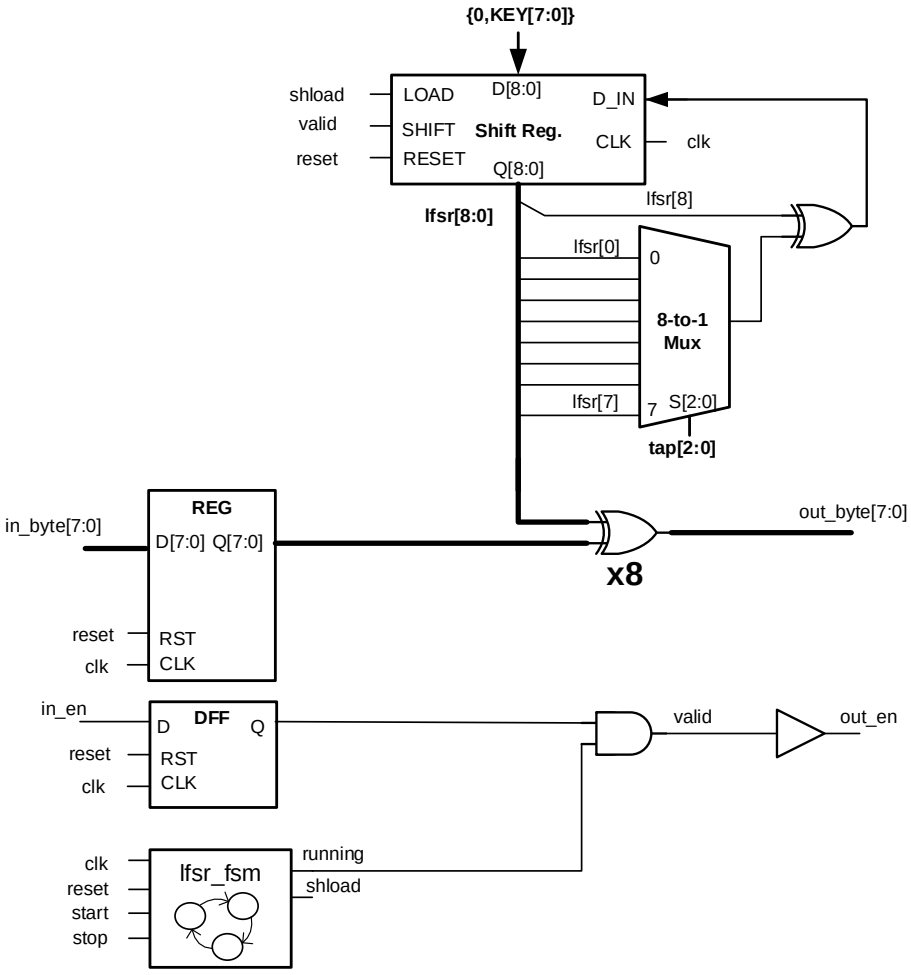

Run over successive clock cycles, this system creates a pseudo-random sequence of bits.



At each time step the LFSR value will be XOR'ed with the incoming byte of data (plaintext) to produce an output byte (cipertext).



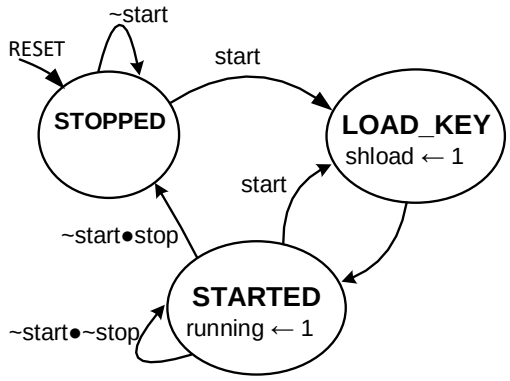

**The Design**: Study the diagram and then read about its operation below.



The LFSR design below uses a shift register that can either a.) load new data (i.e. the key), b.) shift to the left, or c.) hold its current value. The next input bit (D_IN of the shift register) bit is generated by XORing the LFSR's leftmost bit with a selectable tap bit (using an 8-to-1 mux).

The cipher text (**out_byte**) is produced by XORing the lower 8-bits of the LFSR with the 8-bits of the registered input byte. An enable (**in_en**) is used to indicate when an incoming byte actually represents valid data (as opposed to garbage data during an unused clock cycle). Both **in_data** and **in_en** are registered to ensure we have a full clock cycle to perform the desired operation and send it to whatever logic is connected to the outputs. Thus the internal logic of the LFSR looks at **in_byte_q** and **in_en_q** (the registered versions of the actual input data). We generate not only the output data (**out_byte**) but **out_en** to indicate when that **out_byte** represents actual encrypted data and should be consumed by whatever logic is connected to the outputs. Again there may be dead cycles where no new data is input. Thus, **in_en** indicates to our LFSR when new input data is valid and we generate **out_en** to tell the next logic device when our output data is valid.

A state machine will govern the operation of the system by controlling when it starts (i.e. loads a new key value) and when it stops (i.e. forces out_en to 0 and stops the LFSR from shifting despite valid input data being present). The LFSR should only shift when valid input data is present and the system is in the STARTED state. A state machine diagram is presented below.




## 4 Prelab

Given a key of 0x9e and using tap bit 1, manually (paper and pen) generate the first eight LFSR values (each value should be a 9-bit value shown in hex such as: 0x14f or 0x02d). Then assume an input data sequence of 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68. What would the 8 encrypted output values be? Place your answers in a text file named: **prelab.txt** and submit them with your files. To be clear we expects 2 sequences of 8 hex values. The first is the LFSR values (these should be 9-bit answers shown in hex). The second is the encrypted values (these should be 8-bit answers shown in hex).

# 5 Procedure

1. Download the project skeleton zip file from our website and extract it to a folder. Then load the project file (the file with the .xise extension) in Xilinx's Project Navigator
2. Open the **shlreg9.v** Verilog file. This is where the basic 9-bit shift register should be implemented. It should function according to the following function table:

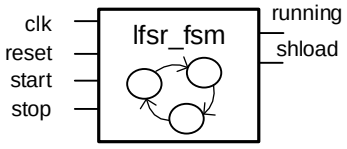
| CLK | RESET | LOAD | SHIFT | Q*[8:0] |
|---|---|---|---|---|
| 0,1 | X | X | X | Q[8:0] |
| PosEdge | 1 | X | X | 0 |
| PosEdge | 0 | 1 | 0 | D[8:0] |
| PosEdge | 0 | 0 | 1 | {Q[7:0],D_IN} |
| PosEdge | 0 | 0 | 0 | Q[8:0] |

We have provided the raw D-FFs for the 9-bits. Your job is to use the LOAD and SHIFT inputs along with D[8:0] and D_IN to design the logic that provides the inputs to these flip flops. Note that for each FF the next value is one of these 3 options: the old value of Q[i], the new D[i] input, or the shifted input (i.e. Q[i-1] or D_in for Q[0]). We have provided a 3-to-1 mux in case you find it useful.

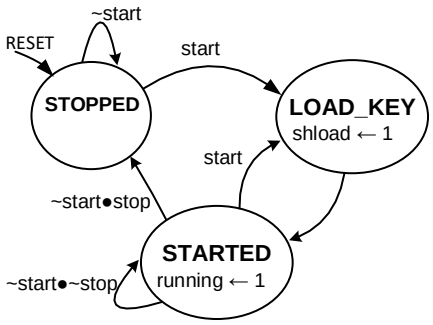
3. Back in the overall LFSR design (i.e **lfsr.v** file), create an instance of your shift register.
    a. Note: That the key input is only 8-bits while the shift register is 9-bits. Thus the D-input to the shift register can be formed by prepending a 0 to the key. This can be done in Verilog using the bit concatenation operator {}. By separating the signals by commas and placing them in {} Verilog will create a concatenated signal (i.e. {1'b0, key} will concatenate a 1-bit value of 0 and the entire 8-bit key signal to form a 9-bit result signal).
    b. Also implement the feedback to D_IN by adding an XOR gate and 8-to-1 mux (you should be able to find an 8-to-1 mux in one of your previous designs) selected by the **tap[2:0]** input.

4. Add the eight XOR gates to produce the ciphertext (**out_byte**).

5. Add a new source file (**lfsr_fsm.v**) to your design. (To do this select Project..New Source..Verilog Module and enter the appropriate filename, and follow the dialog boxes to add appropriate input/output shown below). In this file, design the state machine to implement the state diagram shown earlier in this document and reprinted below.



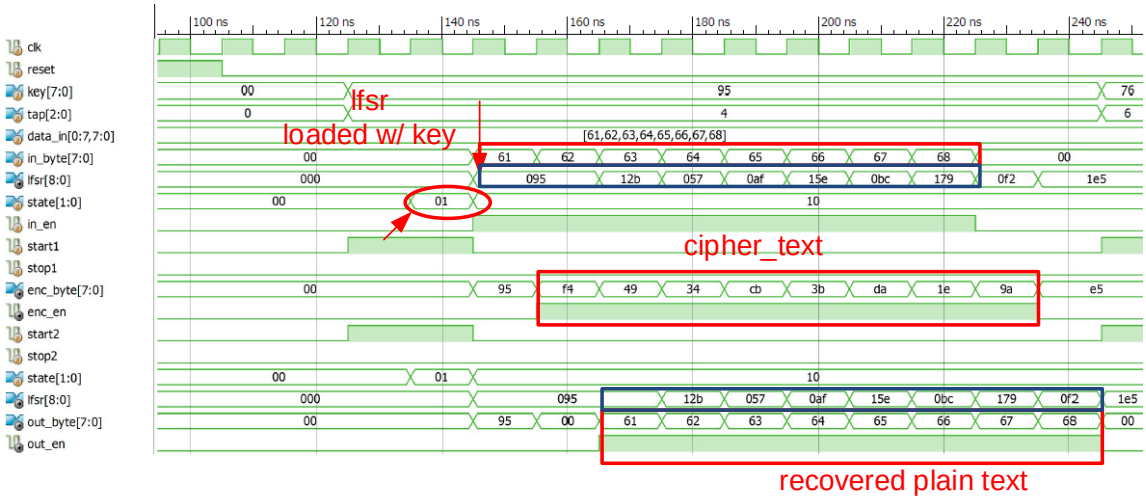

Block Diagram




State Diagram

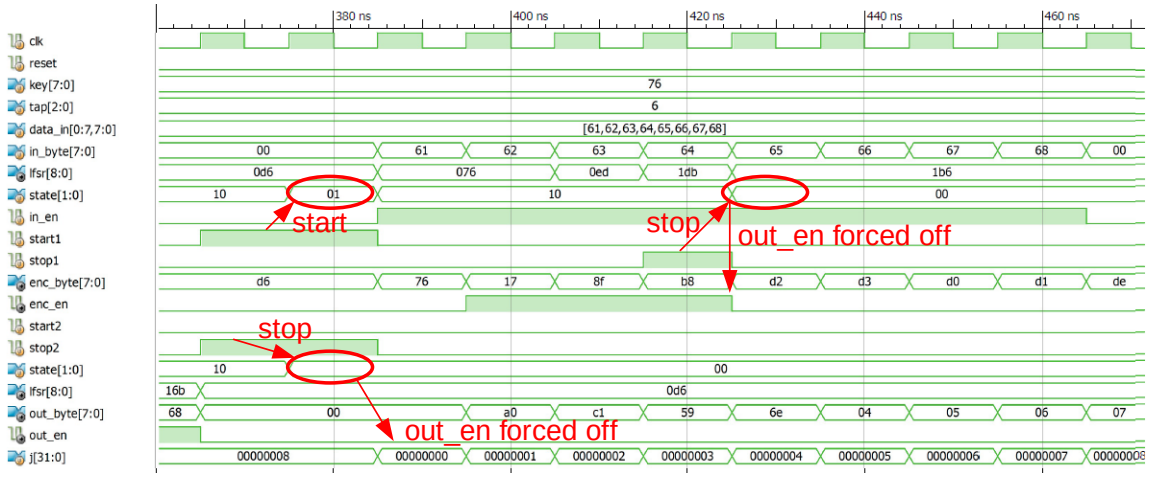
6. Back in the overall LFSR design (i.e **lfsr.v** file), create an instance of your state machine and connect the correct inputs and outputs. Use its outputs to produce the shift register's **load** and **shift** signals as well as **out_en**.

7. Simulate your design using the provided testbench. The testbench provides input stimulus for 3 runs of the LFSR. Correct results are shown below along with an explanation.




Here we start the operation when **start1** goes high. This causes the state machine to move to the LOAD_KEY state in the next cycle and actually load the LFSR with the key in the following cycle. Data is then input (shown when **in_en** = 1) and the byte sequence 0x61, 0x62, …, etc. is input. These bytes are XOR'd with the LFSR values (you should be able to double-check the LFSR values by manually shifting and XORing the tap with the MSB for each cycle). The sequence has a 1 clock delay at the output (i.e. **enc_byte**) since we register the **in_byte**. The **enc_byte** is then input to the decrypting LFSR which then recovers the original.


Above is another run of the LFSR with a different key and tap.




In this last run we start the encryptor but stop it (via **stop1)** half way through. This should cause the state to return to STOPPED and thus out_en should be forced off. The decryptor LFSR is stopped and never started so it should never have its output enabled.

8. In the testbench, add one more test case sequence using the key and tap from the prelab. (You may copy and paste the code from Test sequence 1 or 2 in the testbench, altering the key and tap to create this new test sequence.). Run the simulation and verify your prelab answers are correct. If not determine where you went wrong and go back and update your prelab answers or the design if that is the cause of the error.

9. Once your design is complete and working, show your simulation to your TA and get their sign-off on the rubric sheet. Then submit your files online.

**Interested students may read more about LFSRs for encryption or about more advanced encryption algorithms suitable for hardware implementation such as AES.**

# 6 EE 209 Lab 5 Grading Rubric

Student Name: _____

TA sign-off (correct simulation): _____

| Item | Outcome | Score | Max. |
|---|---|---|---|
| Design<br>• Correct LFSR prelab sequence | Yes / No | | 1 |
| • Correct Encrypted output prelab sequence | Yes / No | | 1 |
| • Correct slreg9.v implementation (can hold, load, and shift left) | Yes / No | | 3 |
| • Valid state machine design | Yes / No | | 2 |
| • Correct tap selection and D_IN implementation | Yes / No | | 1 |
| • 4th Test sequence added to testbench (tap = 1, key = 0x9e) | Yes / No | | 1 |
| • Correct simulation (TA Sign-off) | Yes / No | | 2 |
| SubTotal | | | 10 |
| Late Deductions (-1 pts. per day) | | | |
| Total | | | 10 |
| Open Ended Comments: | | | |