4DM4 Lab. #1
An Intro to LFSRs and Stream Ciphers

(version - Sunday, Sept. 25, 2022)

All Lab demos are in ITB Room 234, on posted DEMO DAYS

Demo-days:	Tues, Wed, Thurs - Sept. 27,28,29 - 2022
Lab reports Due by:	Sunday Oct 2, 2022 (11:59 pm)

Motivation:

According to the US "National Academy of Engineering" (NAE), "cyber security" has emerged as an outstanding challenge in the 21st century. Cyber security is driving the tremendous investments into advanced classical super-computing systems, and Quantum Computing systems. You will likely encounter cyber security issues at some point in your future careers. The labs in 4DM4 will introduce you to some basic and advanced concepts in cyber security. The labs are new in 2022, and prof. is developing these as the class progresses, so please be patient. Hopefully, the labs will be informative and fun.

In the class notes, we have learned about a very simple and secure encryption technique called "Shannon's One-Time-Pad" (OTP). The key idea of Shannon's OTP is that both the sender and receiver of an encrypted message each know the secret key. To encrypt a message with M bits, the key is ideally a perfectly random number with M bits. In general, it is virtually impossible to generate perfectly random numbers on a classical computer, using silicon circuits. A common way to generate "pretty good" random numbers is to use a LFSR ("Linear Feedback Shift Register"). In lab #1, we will experiment with a simple LFSR. This lab is not too difficult, but it requires some programming.

A first goal of lab #1 is for everyone to learn about LFSRs. A second goal is for everyone to sharpen their programming skills, to prepare for Lab #2, in which we will hopefully program a real state-of-the-art Internet encryption algorithm, called Chacha20. You will need to be comfortable programming, and reading and writing data-streams in bits, hexadecimal numbers, and decimal numbers, in order to complete lab #2.

Part A - LFSRs

1. Introduction to LFSRs:

LFSRs are frequently used in digital systems. (i) LFSRs can be used to compute 'Cyclic Redundancy Check' (CRC) checksums, on a transmitted bit-stream. The checksums typically can detect many bit errors that might occur. (ii) LFSRs can be used to compute pseudo-random number streams. Random number streams are often used as 'One-Time-Keys' for encrypting and decrypting messages. (iii) LFSRs can be used to compute simple "Digital Signatures", which indicate if the data in a bit-stream has been changed.

The 16-bit LFSR used for CRC checksum in the IEEE 802.3 Ethernet transmission standard CSMA/CD (Carrier Sense Multiple Access, with Collision Detection) is shown below.

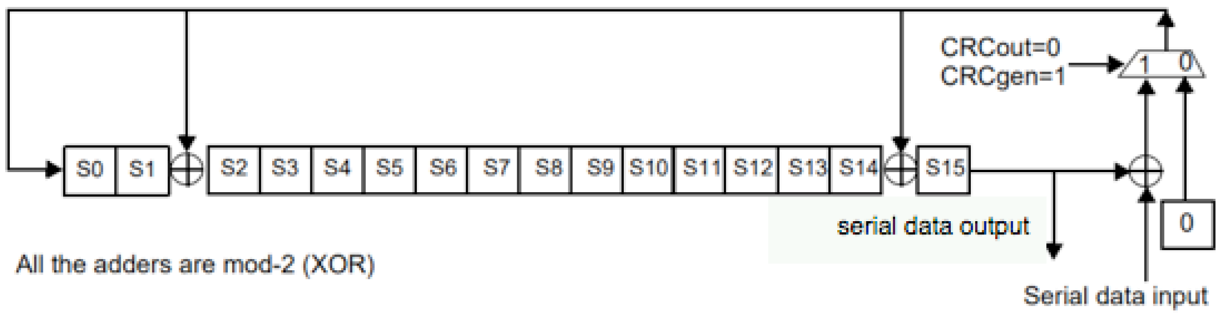


Fig. 1. 16-bit LFSR used in the IEEE 803.2 standard to compute CRC checksums.

In the IEEE figure above, the state of the LFSR is denoted as a 16-bit vector S0, S1, …., S15. This notation is unusual, compared to the LFSR literature. The LFSR literature usually labels the least-significant bit as bit 1, situated at the RIGHT side, as shown below. A larger LFSR with 22 bits of state is shown below in Fig. 2. It labels the bits in the state in the more popular order used in the literature. (Note that Matlab however labels its bits starting at 1 from the LEFT side, so you need to distinguish between LFSR bit numbers and Matlab bit numbers.)

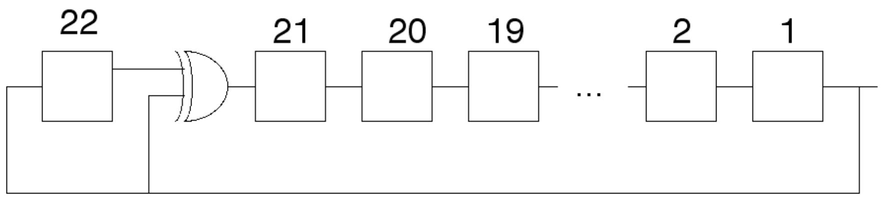


Fig. 2. 22-bit LFSR, using the more common notation for the bits of state.

The LFSR in FIG. 2 uses a feedback polynomial that is written as x^22 + x^21 + 1. This polynomial indicates that the feedback occurs in bits x^22 and x^21 (where bit 1 is the right-most bit.) A table of LFSR feedback polynomials is included at the end of the lab.

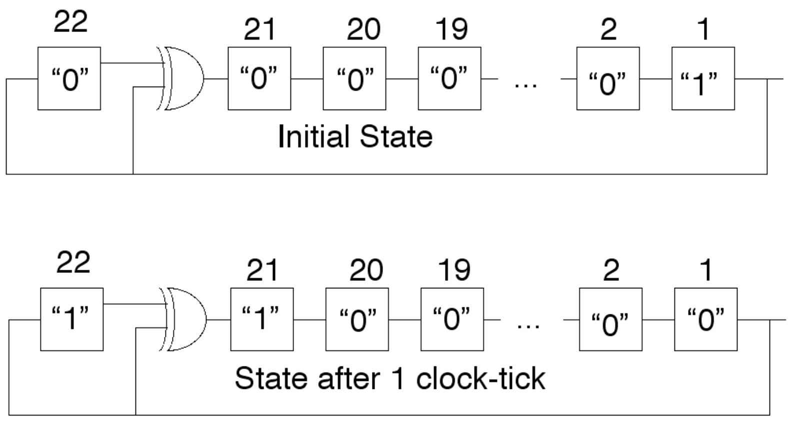


Fig. 3a. LFSR Initial state = (0,0,1) decimal. (Logical values shown as "0" or "1").
Fig. 3b. LFSR state after 1 clock-tick = (48,0,0) decimal.

The state of the LFSR in Fig. 2 can be represented many ways: as (a) BIT-vector (with 22 bits), (b) or as a HEX-vector (with 6 hexadecimal digits) or (c) as DECIMAL-vector (with 3 decimal numbers representing 3 bytes, respectively, where each byte has a value between 0 and 255). We will often need to change between representations (ie BITS-to-DECIMAL and the reverse), so in this lab we will get used to converting between these representations.

A CRC Checksum:

To perform a CRC checksum, the starting state of the LFSR in Fig. 1 is initialized to 2 decimal numbers (0,0). The data to transmit is shifted in 1 bit at a time, on the 'serial-data-input' wire. (The mux control-signal CRCgen = 1, to enable the serial-data-input to affect the serial-data-output.) Assuming that the first serial-data-input bit is "1", then after the 1st clock tick, the state will become (160,1) decimal, and the next 'serial-data-output' bit will become available at bit S15.

A Random Number Generator:

To generate random numbers, the starting state at time t=0 can be initialized to any valid 22-bit state. For the LFSR in FIG. 2 or 3, we can use the initial state = (0,0,1) decimal. (Note, for this LFSR the left-most decimal number only has 6 bits, so its range is between 0 and 63.) The 'serial-data-input' signal shown in Fig. 1 is kept at '0', so it is not drawn in Fig. 2 or 3. Before the 1st clock tick, the "1" in the right-most bit of state gets fed-back into the LFSR, as shown in Fig. 3a (and it also recorded into the DATA_OUT vector, described below.) After the first clock-tick, the state becomes (48,0,0) decimal. The output of the LFSR will repeat, once the 22-bit state returns to the 'initial-state'. (The initial-state can also be defined as any other state the LFSR visits.) If we define the initial-state at time t=1 to be (0,0,1) decimal, then the state after 1 clock tick (at time t=2) will be (48,0,0) decimal.

Exercise Part (A) - Generate Random Numbers.

(A1) Try to use the LFSR in Fig. 2, to generate many random bits. Write clear and well-documented code. You can record these bits in a long vector called DATA_OUT. Each clock tick adds 1 bit to the DATA_OUT vector. Set the initial-state to (0,0,1) decimal, and run the LFSR. Observe the DATA_OUT bit-stream, for many clock ticks. It should appear to be a random stream of bits.
(If your computer is slow, you can use a smaller LFSR, ie one with 20 bits, or 18 bits.)

(A2) Does the LFSR reach a 'steady-state', ie does the output observed on the DATA_OUT bit-stream repeat ? Specifically, can you find the same sequence of 22 bits in DATA_OUT, at two different locations within the bit-stream ? If so, that sequence of 22 bits has repeated. What is the period of the output stream (ie how many clock-ticks expire, before it repeats? Equivalently, how many random bits are generated in one period?)

(A3) Assuming your LFSR has a period, record the random data generated in one period to the vector DATA_OUT, so you can read the data later. You can write bits to a file, it will be hard to interpret a million bits or more. Therefore, convert the sequence of bits in DATA_OUT into a sequence of bytes, and write out a decimal number for each byte. The number of bytes is given by floor(period/8). You can call this file “my_random_numbers.m’. (It might take 1-2 minutes of computer time to generate these numbers and write them to a file.) Print out the first page of your file, to include in your lab report.

(You can write a stream of decimal numbers into a matlab file, with say 16 values per row. Open a file using “fid = fopen(‘my_random_numbers.m’, ‘w’);
Write 16 numbers to the file using “fprintf(fid,’%3g, ’, vector); and “fprintf(fid,’…\n’);”
where “vector’ has 16 decimal numbers.

Your .m file can look like this (so Matlab can execute the file, to load the data into memory).

```
RANDOM_DATA_OUT(1,1:num_bytes) = [… % this square bracket starts the vector
123, 241, 13,  ….. 178,…            % you can write 16 decimal numbers per line
…..                                 % use as many lines as you need
];                                  % this square bracket ends the vector
```

(A4) Define a “0-run” as a string of consecutive 0s, which are preceded by a non-0 character, and followed by a non-0 character. (A non-0 character is a 1, or a null character, or an end of file character). Define a “1-run” as a string of consecutive 1s, which are preceded by a non-1 character, and followed by a non-1 character. (A non-0 character is a 1, or a null character, or an end of file character).

Let X(k) be the number of observations of 0-runs of length k, in one period (for k=1,2,3,….).
Let Y(k) be the number of observations of 1-runs of length k, in one period (for k=1,2,3,….).

Consider only 0-runs first. The conditional-probability that a 0-run of length k occurs, given that we are only considering 0-runs, is given by X(k) / (sum of all observed X(k)). We know that the sum of all these conditional probabilities over all k must = 1, by definition.

Consider only 1-runs next. The conditional-probability that a 1-run of length k occurs, given that we are only considering 1-runs, is given by Y(k) / (sum of all observed Y(k)). We know that the sum of all these conditional probabilities over all k must = 1, by definition.

Consider the unconditional-probability that a 0-run or 1-run of length k occurs, given that we are considering both 0-runs and 1-runs. These probabilities are given by :

X(k)/(sum of all observed X(k) and sum of all observed Y(k))
Y(k)/(sum of all observed X(k) and sum of all observed Y(k))

Now consider a perfectly-random bit stream. Given us a formula for the conditional-probability that a 0-run of length k occurs, given that we are considering only 0-runs in a perfectly random bit-stream. Try to justify your formula.

Give us a formula for the conditional-probability that a 1-run of length k occurs, given that we are considering only 1-runs in a perfectly random bit-stream. Try to justify your formula.

(A5) Create a table for 0-runs, with 4 rows, and 24 columns. The first row shows the number k, for k = 1 to 24. The 2nd row shows the number of observations of 0-runs of length k, for k=1…24, in one period of your LFSR (assuming your LFSR has a period). The 3rd row shows the observed experimental conditional-probability that a 0-run of length k occurs, given that we are using only your observations of 0-runs. (If your LFSR does not have a period, run the LFSR for a million click ticks.) The 4th row shows a theoretical answer, the conditional-probability that a 0-run of length k occurs, in a perfectly-random bit stream. (Use your formula from part A4.) When you print these numbers, if the numbers become very small, use a floating point format, ie try fprintf('%10.8f, ' ….

Are there any discrepancies between the theoretical and experimental conditional probabilities ? Try to explain any discrepancy between the experimental and theoretical conditional probabilities, if any.

(A6) Create a table for 1-runs, with 4 rows, and 24 columns. Follow the same instructions as in A5.

Are there any discrepancies between the theoretical and experimental conditional probabilities ? Try to explain any discrepancy between the experimental and theoretical conditional probabilities, if any.

Exercise Part (B): A Simple Stream Cipher

(B1) Pick any image you like from the www, and save it to a file. You can crop the image, to make it smaller and easier to work with. Read the image into matlab, to create a bit-map image, ie try "A = imread(my_image.png)" or "A = imread(my_image.jpg)".

According to Matlab documentation, the bit-map image A is a 3D array with R rows, C columns, and a depth of 3 numbers. Each number is between 0 and 255, and represents a color intensity (for 3 colors R,G,B). You can view the image in matlab with "image(A)". Show the image in your lab report.

(My version of Matlab is inconsistent: A is sometimes read as a 2D matrix, and sometimes read as a 3D matrix. To make Matlab realize A is a 3D matrix, type and execute these lines:

```
        R_matrix = A(:,:,1);        G_matrix = A(:,:,2);          B_matrix = A(:,:,3);
```

Thereafter, if you call "[rows,cols,depth] = size(A)", and Matlab should view A as a 3D matrix.)

(B2) Read the stream of pseudo-random numbers from the LFSR, that you wrote to a file in part A3. Write the numbers into a matrix RAND_matrix, with the same size as the A matrix. (Recall that you wrote your random numbers to a matlab file in part A3. Call that script in your matlab code, to load those numbers into your memory.) If you do not have enough random numbers, then we really should use a larger LFSR. However, to simplify lab. #1 you can also re-use some of the same random numbers. (This of course invalidates Shannon's "One-Time-PAD" principle, and makes the cipher very weak.) You can create a 3D matrix of zeros called RAND_matrix with the command RAND_matrix = zeros(rows,cols,depth);

You can then write your LFSR numbers into the RAND_matrix.
Explain how you did part B2 in your lab report.

(B3) You need to XOR the 3D matrix A with a 3D matrix of random numbers RAND_matrix, to encrypt the matrix A. I did this as follows: Visit each element of A and the corresponding element of RAND_matrix. These elements are decimal integers between 0 and 255. You can convert each of them to a vector of 8 bits, XOR the two bit vectors, and convert the resulting bit-vector back unto a decimal number. Write that decimal number into the element A_encrypted. You can try other ways in Matlab, to accomplish this XOR functionality too. Explain how you accomplish part B3 in your lab report.

(B4) View the encrypted image, using the matlab image command. Convert the matrix A_encrypted to unsigned 8 bit integers. Use the following command:

```
image (uint8(A_encrypted))
```

It should appear to be "random noise". Show the image in your lab report.

(B5) Decrypt the ciphertext, by XOR-ing the encrypted image matrix with the random matrix used for encryption. (Do the reverse process from what was described in B3.)
View the decrypted image, using the following command:

```
image (uint8(A_decrypted))
```

You should see the original image. Explain how you did part B5 in your lab report.
Show the image in your lab report.

(B6) Ideally, your software will be general, ie you can view any image from the web, save it to a file, and crop it. You then run your software: view the image in Matlab, then encrypt it, view the encrypted image in Matlab, then decrypt it, and view the decrypted image in Matlab. You should only need to enter a file name, to read the original image file. The software should do the rest.

SUBMISSION:

Submit your report on Avenue-to-Learn, by the due date.

SUBMISSION FORMAT:
Create a lab report file called 4DM4-Lab-1-report-XX-YY, where XX and YY are the initials of your team members.

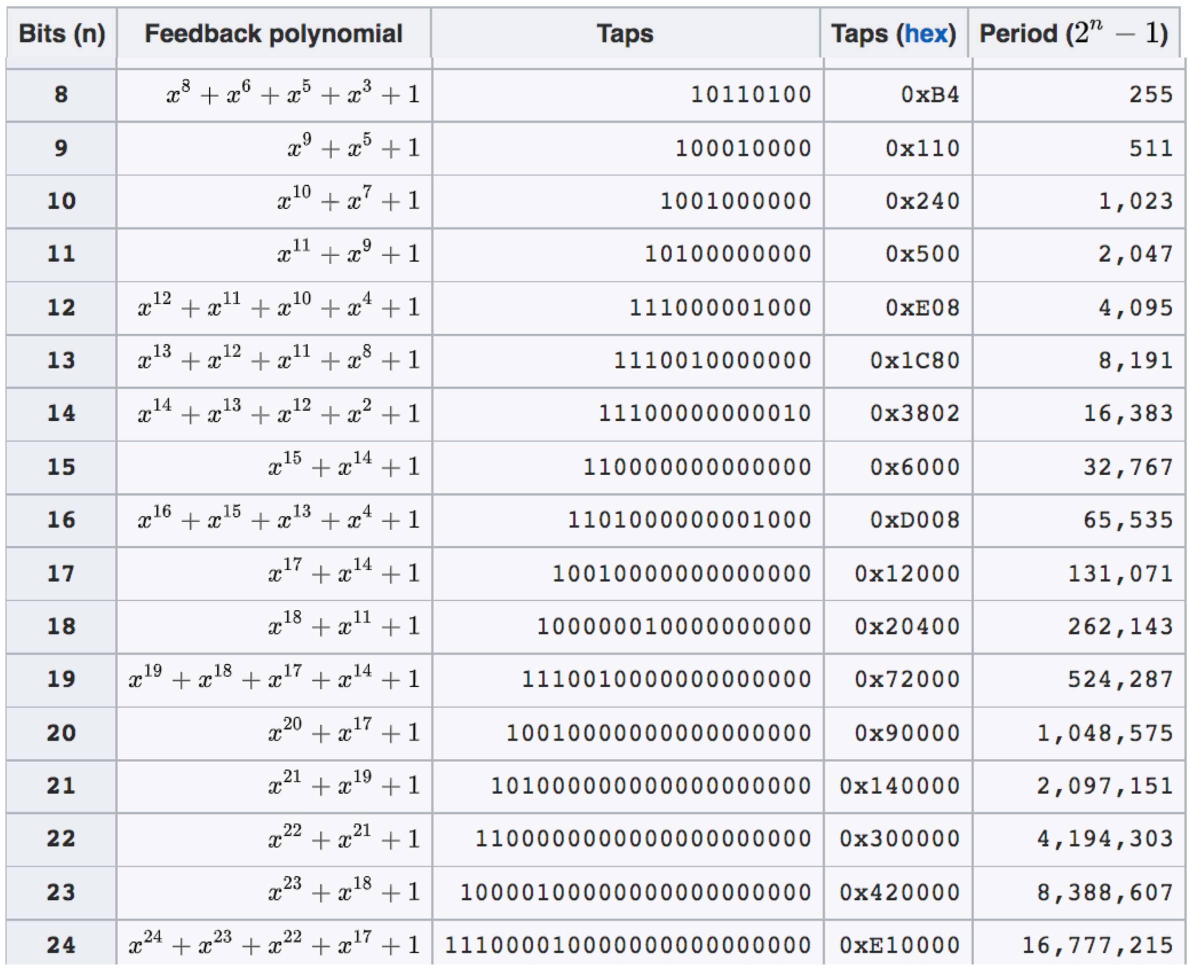
Include your brief report in one PDF file, called LAB-1-REPORT. Include your well-documented matlab code (or python or Java or C code), in a separate PDF documents. Print your well-documented matlab code (or python or Java) to a nicely formatted readable PDF file.
A non-expert reader should be able to determine all important steps in your code, from your comments. Comments are worth marks. Try to make your code efficient. Efficient code is worth marks.

You do not need to put a lot of effort into preparing this report, if you have demonstrated that everything works to the TA. Please summarize what you have done concisely, so we each have a record of what you have done. We use these reports to show the CEAB (Canadian Engineering Accreditation Board) what we teach.

If you are taking 6DM4, please do the lab alone. Please demonstrate the lab to the professor, during the tutorial on Thurs. Sept 29.

LAB-DEMOS: You will need to book a lab-demo time-slot with a TA. Each group will get a 10 minute time-slot, to demo their lab to a TA. *Please be ready to explain your software to a TA, to convince them that everything works, in 10 minutes.* If time permits, the TA may ask questions, may ask you to modify and run your software with different a slight change, etc. (Unfortunately, each group can only book 1 time-slot. If you miss your time-slot, there probably will not be another time for you to demo.)
We will post instructions on how to book a LAB-DEMO, closer to the demo dates.

Here is a table of some common LFSR Feedback Polynomials (from Wikipedia).



Bits (n)	Feedback polynomial	Taps	Taps (hex)	Period $(2^n - 1)$
8	$x^8 + x^6 + x^5 + x^3 + 1$	10110100	0xB4	255
9	$x^9 + x^5 + 1$	100010000	0x110	511
10	$x^{10} + x^7 + 1$	1001000000	0x240	1,023
11	$x^{11} + x^9 + 1$	10100000000	0x500	2,047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	111000001000	0xE08	4,095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	1110010000000	0x1C80	8,191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	11100000000010	0x3802	16,383
15	$x^{15} + x^{14} + 1$	110000000000000	0x6000	32,767
16	$x^{16} + x^{15} + x^{13} + x^4 + 1$	1101000000001000	0xD008	65,535
17	$x^{17} + x^{14} + 1$	10010000000000000	0x12000	131,071
18	$x^{18} + x^{11} + 1$	100000010000000000	0x20400	262,143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	1110010000000000000	0x72000	524,287
20	$x^{20} + x^{17} + 1$	10010000000000000000	0x90000	1,048,575
21	$x^{21} + x^{19} + 1$	101000000000000000000	0x140000	2,097,151
22	$x^{22} + x^{21} + 1$	1100000000000000000000	0x300000	4,194,303
23	$x^{23} + x^{18} + 1$	10000100000000000000000	0x420000	8,388,607
24	$x^{24} + x^{23} + x^{22} + x^{17} + 1$	111000010000000000000000	0xE10000	16,777,215