Philip Nevins
5/12/2022
ENGR 271

Lab 0xC: Linear-Feedback SRs

Introduction

In this lab we explore LFSRs, also known as Linear-Feedback Shift Registers. These are circuits that produce a pseudo-random output based on their design. The ones we look at in this lab have something called a max-length sequence, which means that before repeating, it will shift through $2^n - 1$ states, with the -1 part being the all zeros state. This can be mapped back into the LFSR with additional logic, which we also explore in this lab.

Part A: Fibonacci vs Galois LFSRs

1. In procedure #1, we explore how a 4-bit Fibonacci LFSR works. We are tasked with building one out of DFFs, according to the lab instructions and test it with a Logic Analyzer. In Figure 1, we can see the schematic we built. In Figure 2, we can see the Logic Analyzer output.

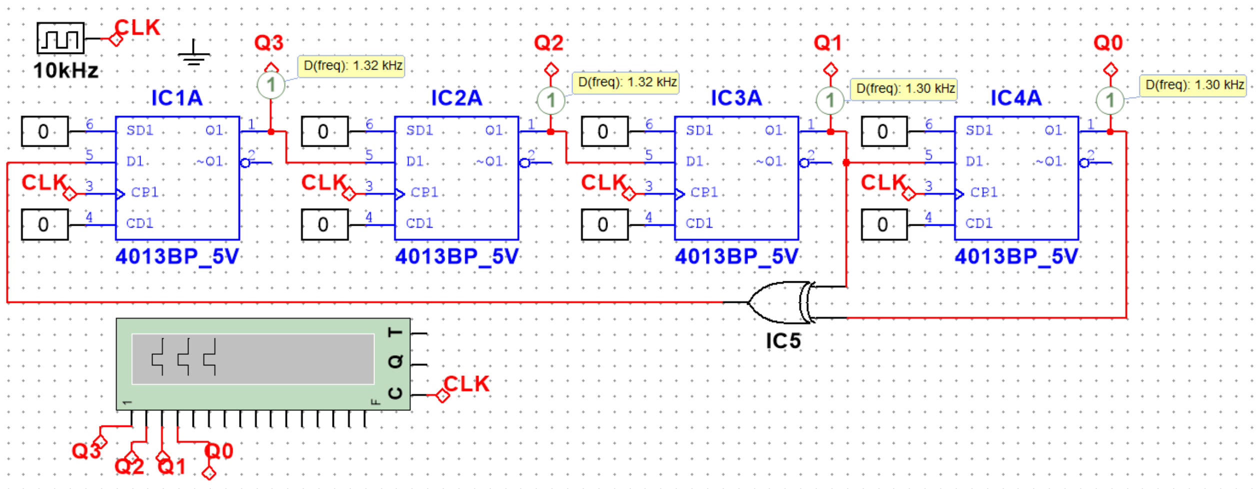


Figure 1: 4-bit Fibonacci LFSR with DFFs

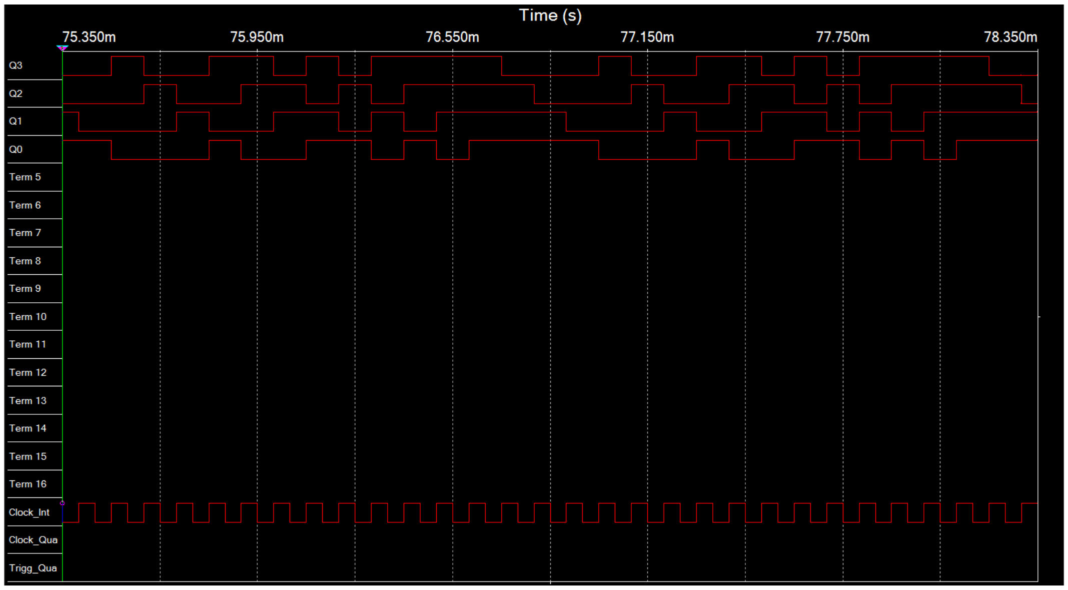


Figure 2: Logic Analyzer output for 4-bit Fib LFSR (3 clocks per time division)

Below in Table 1, we can see the outputs organized in binary based on Figure 2, with their decimal numbers attached. In this table, you can observe that we do not have the all zeros state, like we mentioned in the introduction. This is expected and we will be adding additional circuitry to map this into the sequence in procedure 2.

Q3	Q2	Q1	Q0	Decimal
0	0	0	1	1
1	1	0	0	12
0	1	1	0	6
0	0	1	1	3
1	1	0	1	13
1	0	1	0	10
0	1	0	1	5
1	1	1	0	14
0	1	1	1	7
1	1	1	1	15
1	0	1	1	11
1	0	0	1	9

1	0	0	0	8
0	1	0	0	4
0	0	1	0	2

Table 1: 4-bit Fibonacci Output Results

2. In procedure 2, we are tasked with adding additional circuitry to map the all zeros, invalid state, into the sequence. This can be seen in Figure 3. We know that a NOR gate produces a 1 when all inputs are 0, which causes the Q3 DFF output to be a 1. So when we see state 0000 come in, it forces it to 1000, therefore mapping it back into the sequence.

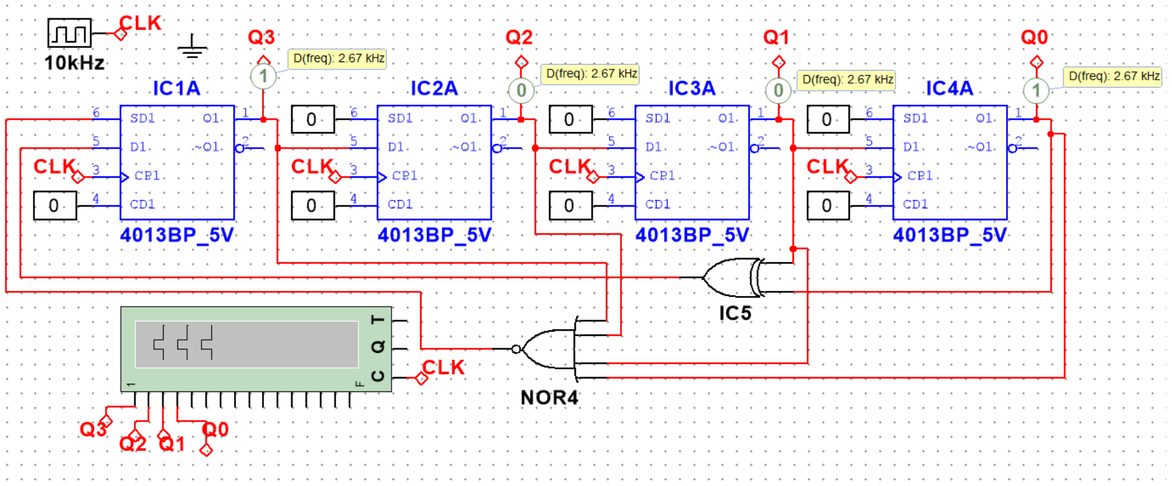


Figure 3: 4-bit Fib LFSR with DFFs with additional circuitry to map 0000 state in

3. In procedure 3, we explore how a 4-bit Galois LFSR works. We are tasked with building it out of DFFs, based on the lab schematics and testing it with a Logic Analyzer. In Figure 4, we can see the schematic we built. In Figure 5, we can see the output results from the Logic Analyzer. Table 2 shows our results compiled with the binary sequence and decimal number. This will make it easy to compare Table 1 & 2 in procedure 4.

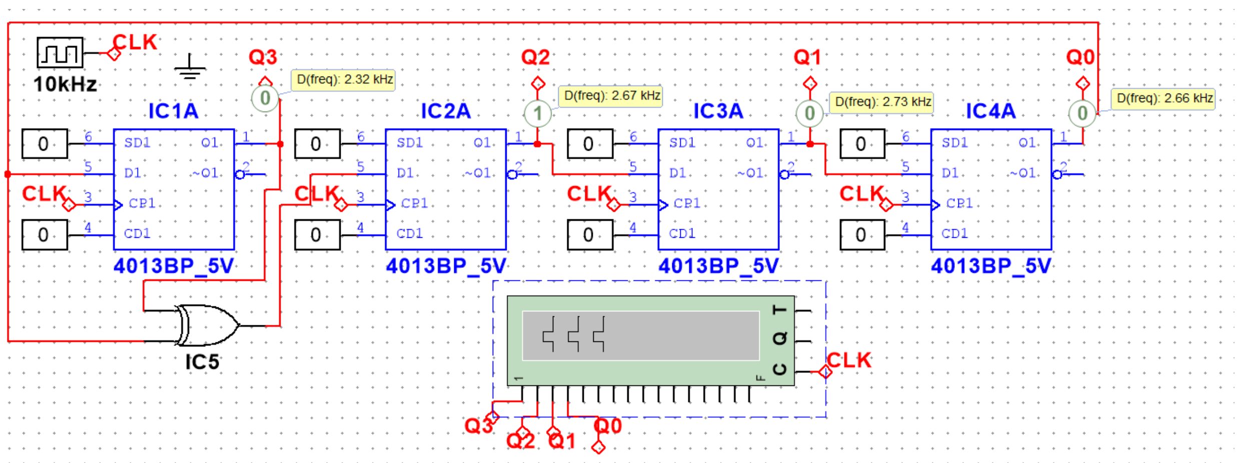


Figure 4: 4-bit Galois LFSR with DFFs

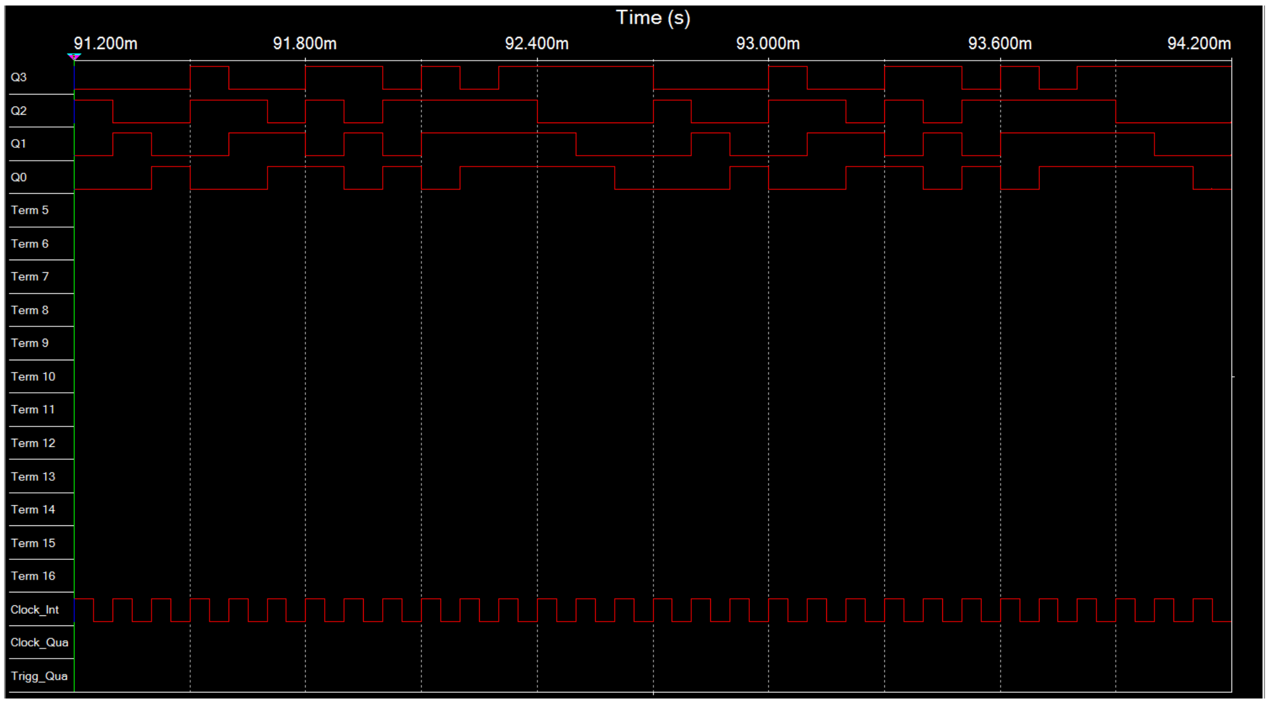


Figure 5: Logic Analyzer output for 4-bit Galois LFSR (3 clocks per time division)

Q3	Q2	Q1	Q0	Decimal
0	0	0	1	1
1	0	0	0	8
0	1	0	0	4
0	0	1	0	2
1	0	0	1	9
1	1	0	0	12
0	1	1	0	6
1	0	1	1	11
0	1	0	1	5
1	0	1	0	10
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15
0	1	1	1	7
0	0	1	1	3

Table 2: 4-bit Galois Output Results

4. In this procedure, we are tasked with comparing our results from the 4-bit Galois and Fibb LFSRs made with DFFs. There is a difference in the sequence, as we can see when comparing Table 1 & 2. There is no timing difference or difference in propagation delay, as we can see from the graphs in Figure 2 and Figure 5. Both graphs have the same number of states in the sequence and same clocks per time division. This is expected since they both are built with the same DFFs and are almost identical in design, except for the XOR gate inputs / output. Since they have the same number of XOR gates in the design, we would not expect any differences in the timing of the outputs of the LFSR. These XOR gates are referred to as taps.

5. In procedure 5, we are tasked with adding an additional bit to the Fibonacci LFSR making it a 5-bit and having it be self-correcting (no all zero state). We can map the all zeros state into our sequence the same way we did above, by using a NOR gate. When we add another bit, we must change the taps according to the table given in the lab instructions. This will change the taps

from 0,1 to 0,2, which means Q2 will be connected instead of Q1. The schematic for our 5-bit Fib LFSR can be seen in Figure 6. We are also tasked with testing it with a Logic Analyzer, which can be seen in Figure 7. We compile our results into a table, which can be seen in Table 3.

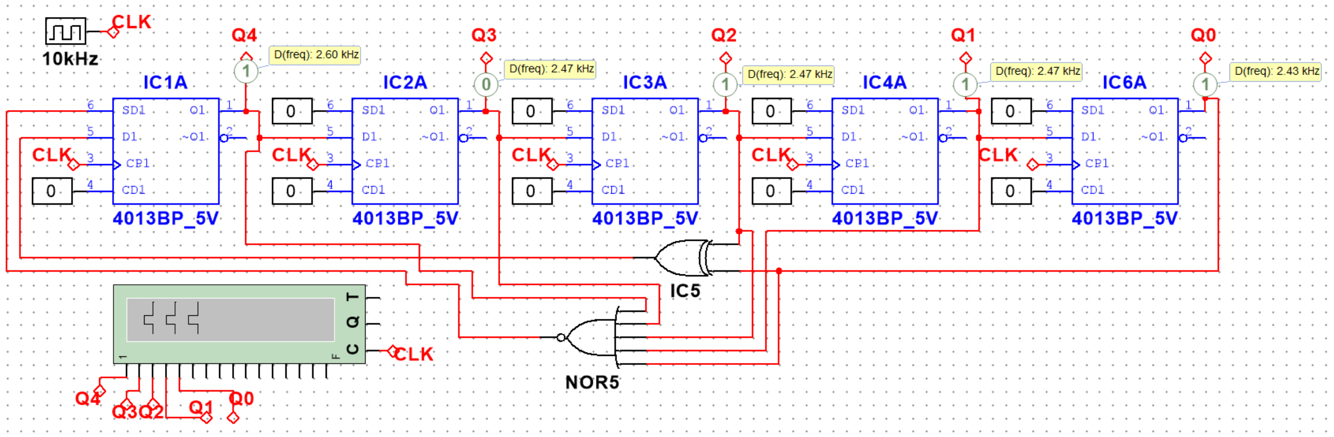


Figure 6: 5-bit Fib LFSR with additional circuitry to map 0000 state in

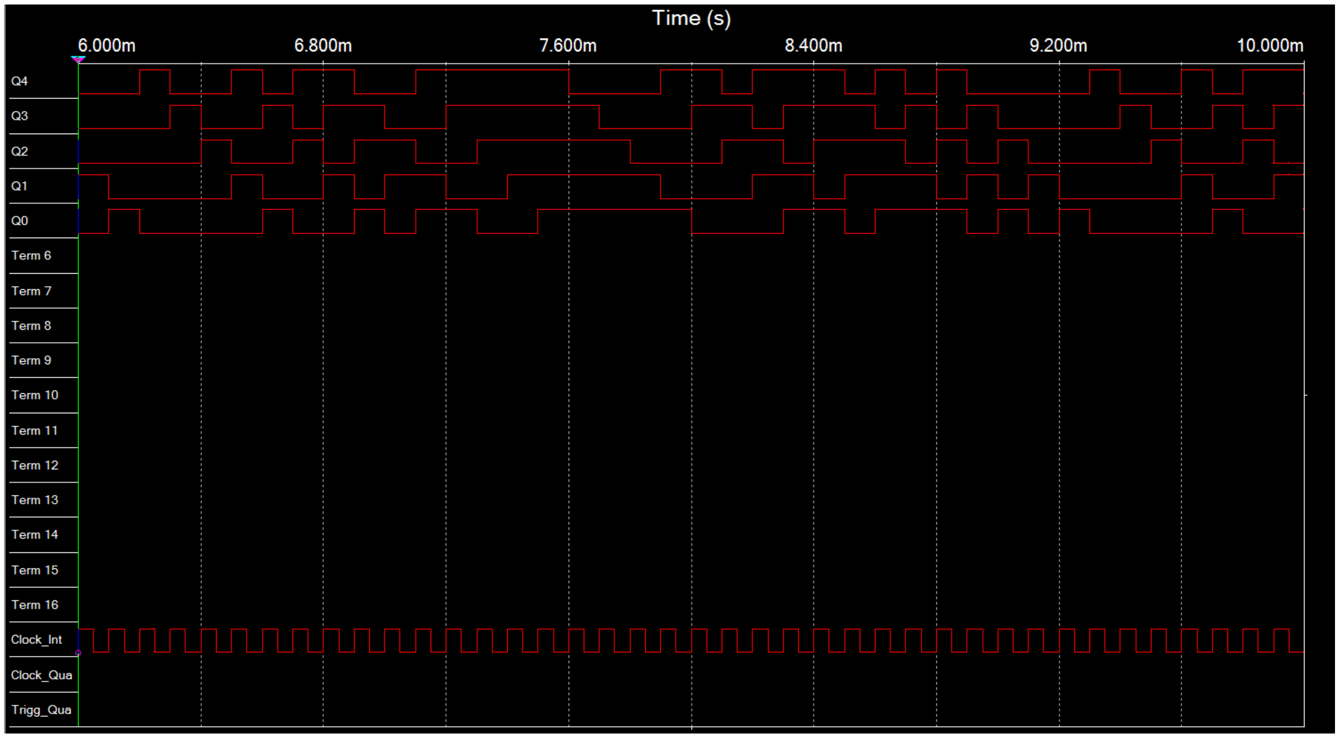


Figure 7: Logic Analyzer output for 5-bit Fib LFSR (4 clocks per time division)

Q4	Q3	Q2	Q1	Q0	Deci mal		Q4	Q3	Q2	Q1	Q0	Deci mal
0	0	0	0	1	1		0	1	1	1	1	15
1	0	0	0	0	16		0	0	1	1	1	7
0	1	0	0	0	8		0	0	0	1	1	3
0	0	1	0	0	4		1	0	0	0	1	17
1	0	0	1	0	18		1	1	0	0	0	24
0	1	0	0	1	9		0	1	1	0	0	12
1	0	1	0	0	20		1	0	1	1	0	22
1	1	0	1	0	26		1	1	0	1	1	27
0	1	1	0	1	13		1	1	1	0	1	29
0	0	1	1	0	6		0	1	1	1	0	14
1	0	0	1	1	19		1	0	1	1	1	23
1	1	0	0	1	25		0	1	0	1	1	11
1	1	1	0	0	28		1	0	1	0	1	21
1	1	1	1	0	30		0	1	0	1	0	10
1	1	1	1	1	31		0	0	1	0	1	5
							0	0	0	1	0	2

Table 3: 5-bit Fibonacci LFSR Output Results

Part B: Word Generator and Logic Analyzer

6. In this procedure, we are tasked with using a Word Generator and Logic Analyzer to output a 4-bit LFSR. In Figure 7, we can see the circuit we will use for both procedure 6 and 7. In Figure 8, we can see the file we loaded into the Word Generator. We used a python script to produce this text file, which was given to us in the lab instructions.

   In Figure 9, we can see the results from our Logic Analyzer. It should be noted that the clock cycles are 100 per time division. If we consider that on the Y-axis, 5 = Q3 and 1 = Q0, we have an identical sequence to the 4-bit Galois LFSR we built in Part A, procedure 3. This is expected, as these are not true random generators. They are just pseudo random generators.

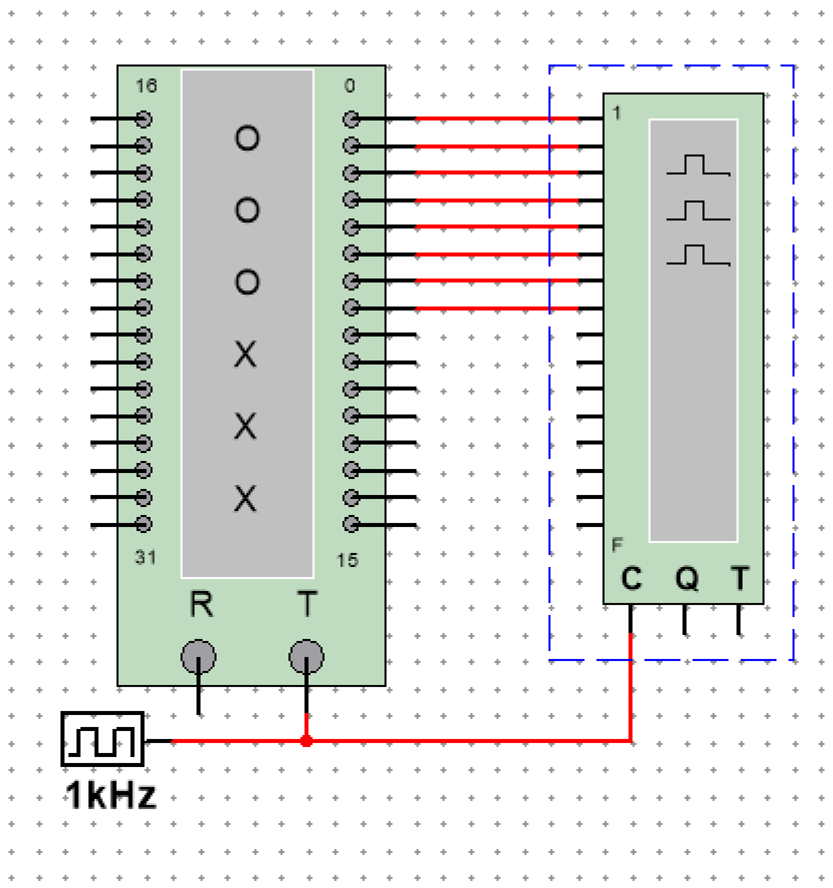
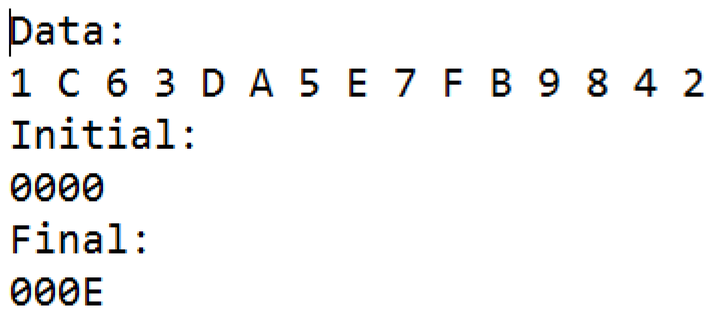


Figure 7: Word Generator and Logic Analyzer



```
Data:
1 C 6 3 D A 5 E 7 F B 9 8 4 2
Initial:
0000
Final:
000E
```

Figure 8: Word Generator Load File 4-bit Galois

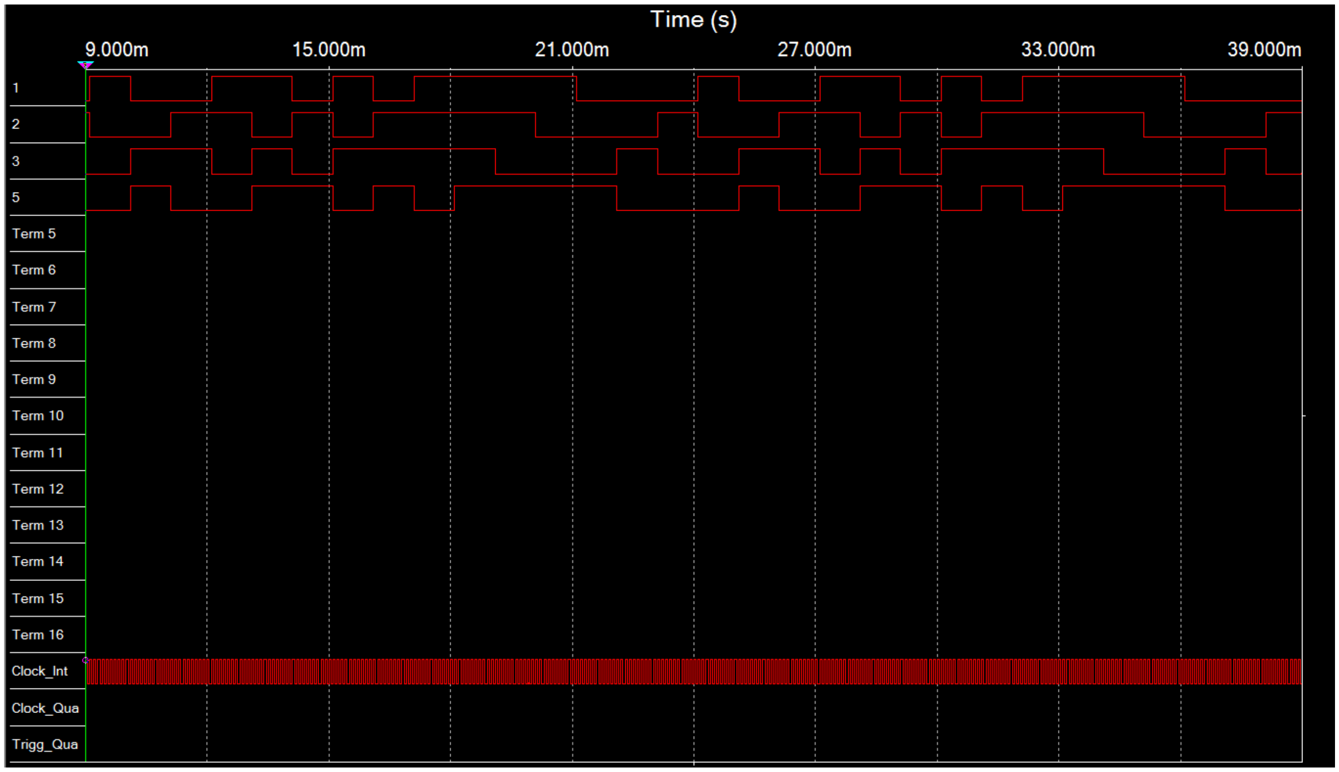
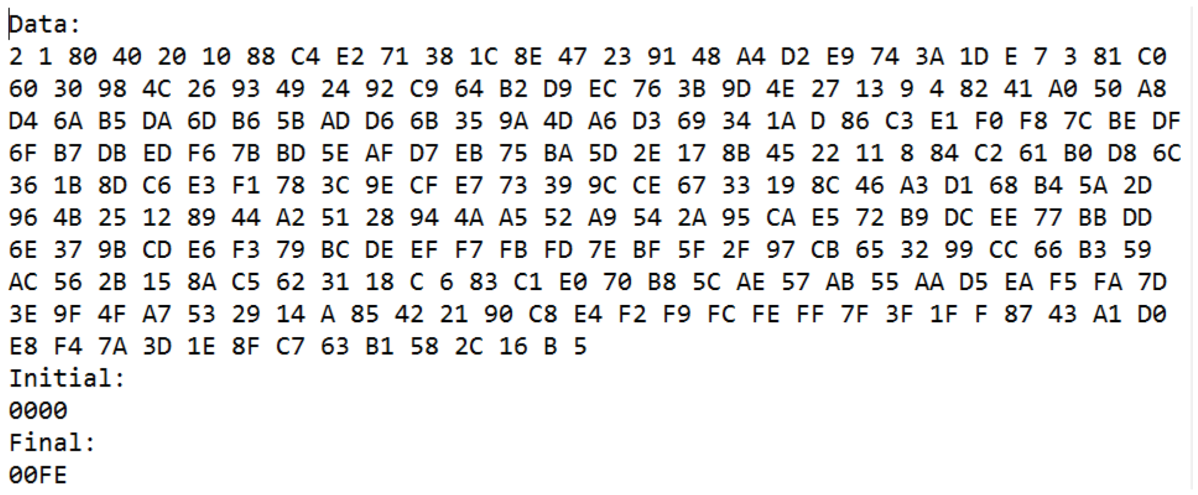


Figure 9: Logic Analyzer Results, 4-bit Galois LFSR via Word Generator

7. In procedure 7, we simulate a 8-bit Fibonacci LFSR using a Word Generator. In Figure 10, we can see the file we loaded into the Word Generator. We used a python script to produce this text file, which was given to us in the lab instructions.

   In Figure 11, we can see the Logic Analyzer results. To effectively show the majority of the results without the graph being too squished together to read, we have cut the Logic Analyzer results off at FF = 256 (FF can be seen in Figure 10), but we can confidently say that the sequence did continue as expected.



```
Data:
2 1 80 40 20 10 88 C4 E2 71 38 1C 8E 47 23 91 48 A4 D2 E9 74 3A 1D E 7 3 81 C0
60 30 98 4C 26 93 49 24 92 C9 64 B2 D9 EC 76 3B 9D 4E 27 13 9 4 82 41 A0 50 A8
D4 6A B5 DA 6D B6 5B AD D6 6B 35 9A 4D A6 D3 69 34 1A D 86 C3 E1 F0 F8 7C BE DF
6F B7 DB ED F6 7B BD 5E AF D7 EB 75 BA 5D 2E 17 8B 45 22 11 8 84 C2 61 B0 D8 6C
36 1B 8D C6 E3 F1 78 3C 9E CF E7 73 39 9C CE 67 33 19 8C 46 A3 D1 68 B4 5A 2D
96 4B 25 12 89 44 A2 51 28 94 4A A5 52 A9 54 2A 95 CA E5 72 B9 DC EE 77 BB DD
6E 37 9B CD E6 F3 79 BC DE EF F7 FB FD 7E BF 5F 2F 97 CB 65 32 99 CC 66 B3 59
AC 56 2B 15 8A C5 62 31 18 C 6 83 C1 E0 70 B8 5C AE 57 AB 55 AA D5 EA F5 FA 7D
3E 9F 4F A7 53 29 14 A 85 42 21 90 C8 E4 F2 F9 FC FE FF 7F 3F 1F F 87 43 A1 D0
E8 F4 7A 3D 1E 8F C7 63 B1 58 2C 16 B 5
Initial:
0000
Final:
00FE
```

Figure 10: Word Generator Load File 8-bit Fib

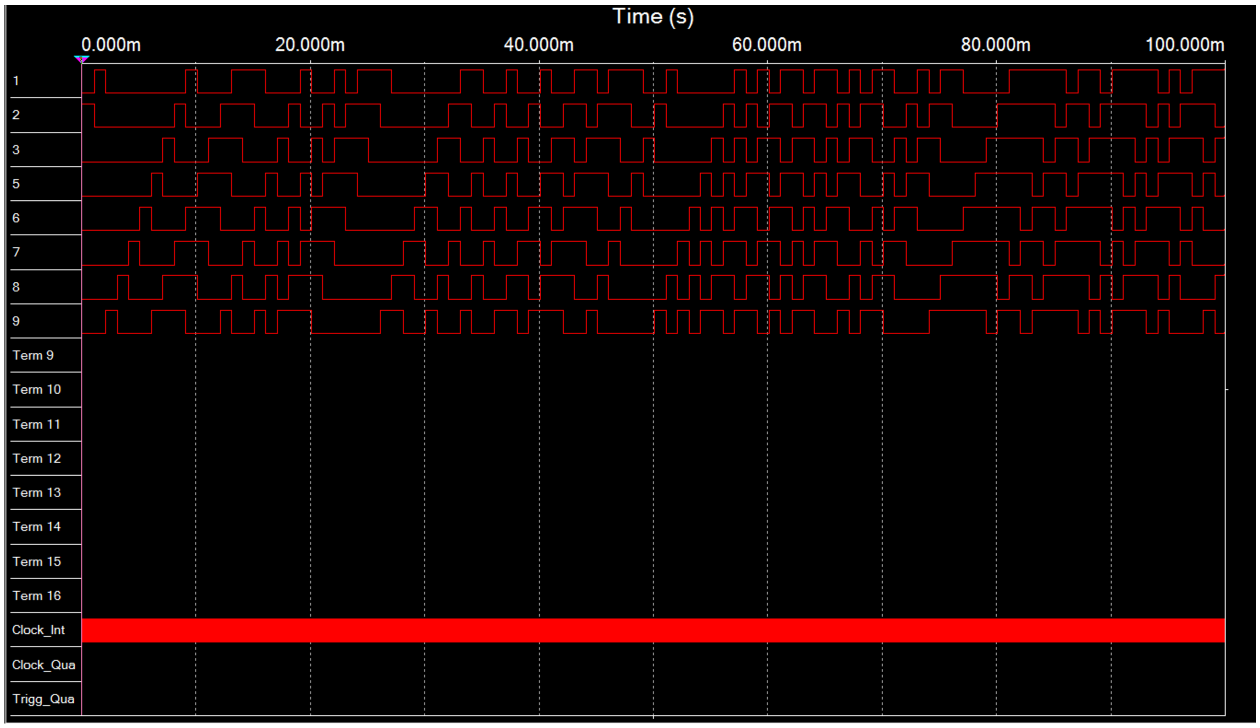


Figure 11: Logic Analyzer Results, 8-bit Fib LFSR via Word Generator

Part C: Digital Pseudo-Random Noise Generator

8. In this procedure, we are given a schematic in KiCAD (Figure 12, taken from the lab instructions) and tasked with porting it into Multisim, using the 74194 IC in Multisim and finding a replacement for the speaker that is used. We used an oscilloscope as our output, so we can assess the waveforms that result from the circuit. The new circuit can be seen in Figure 13. In Figure 14, we can see the resulting waveform from our new circuit.

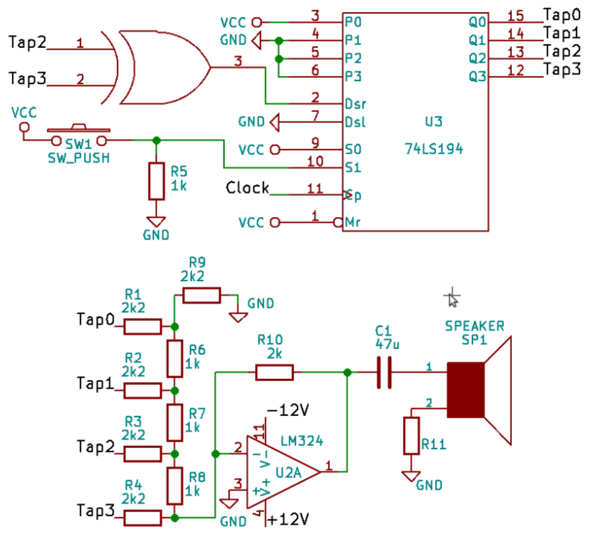


Figure 12: KiCAD schematic from lab instructions

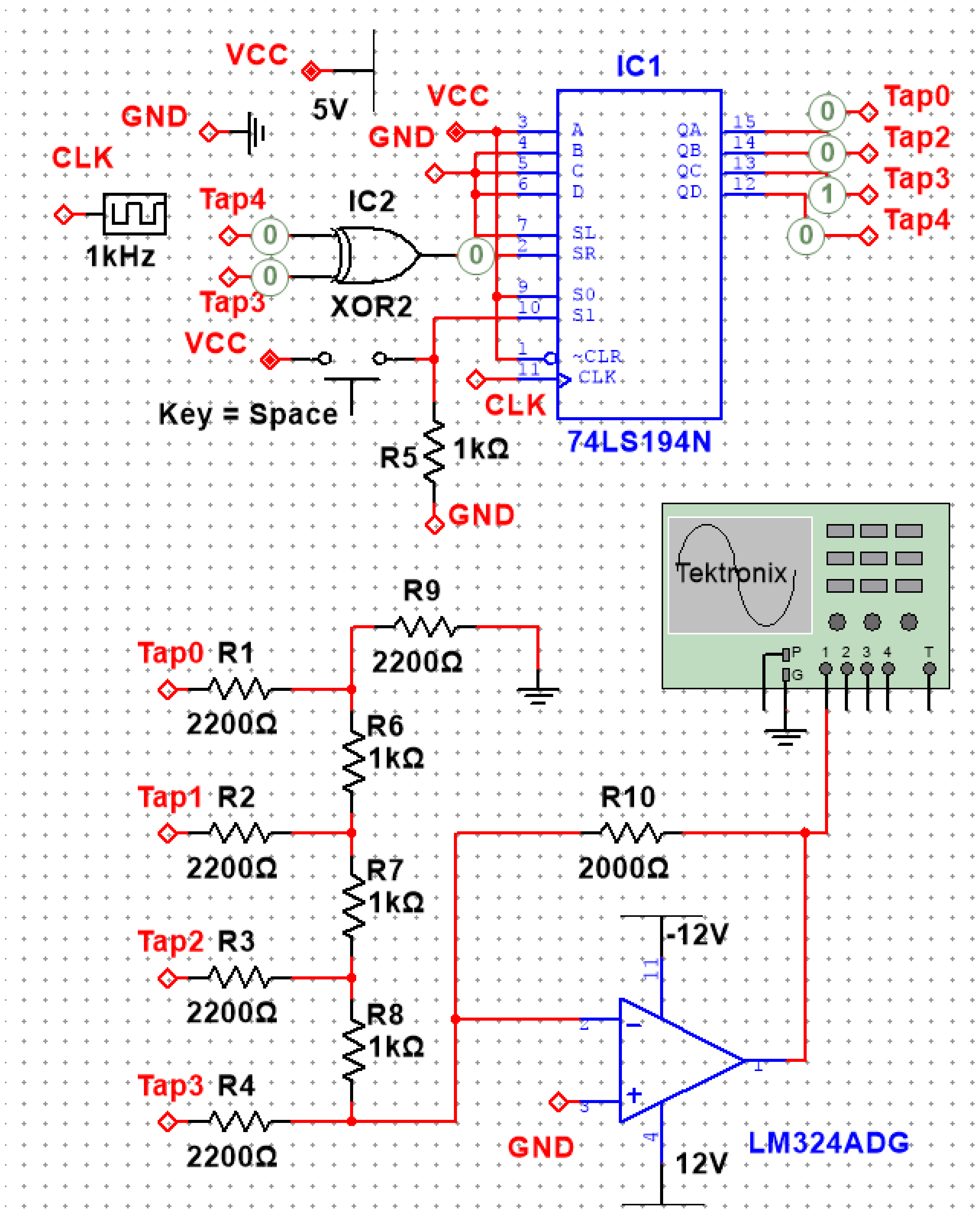


Figure 13: 4-bit Digital Pseudo-Random Noise Generator (DPRNG) ported from KiCad

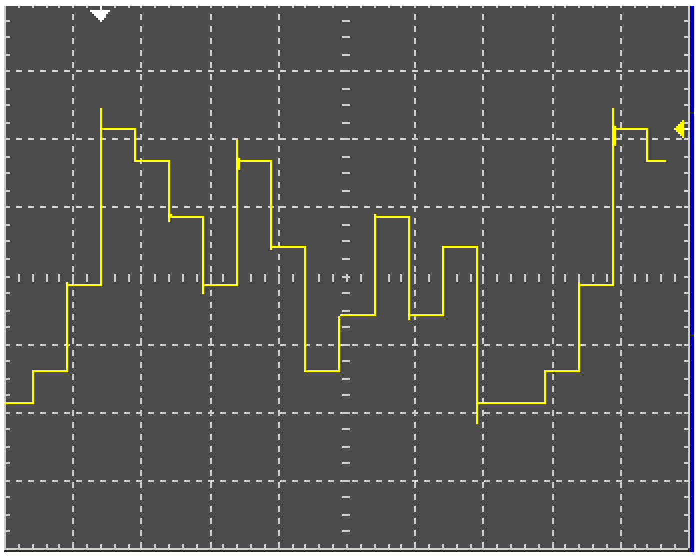


Figure 14: Output Results 4-bit DPRNG

9. In procedure 9, we are tasked with expanding the design from procedure 8, into a 8-bit LFSR using two 74194s. The schematic can be seen in Figure 15 and the output results in Figure 16.

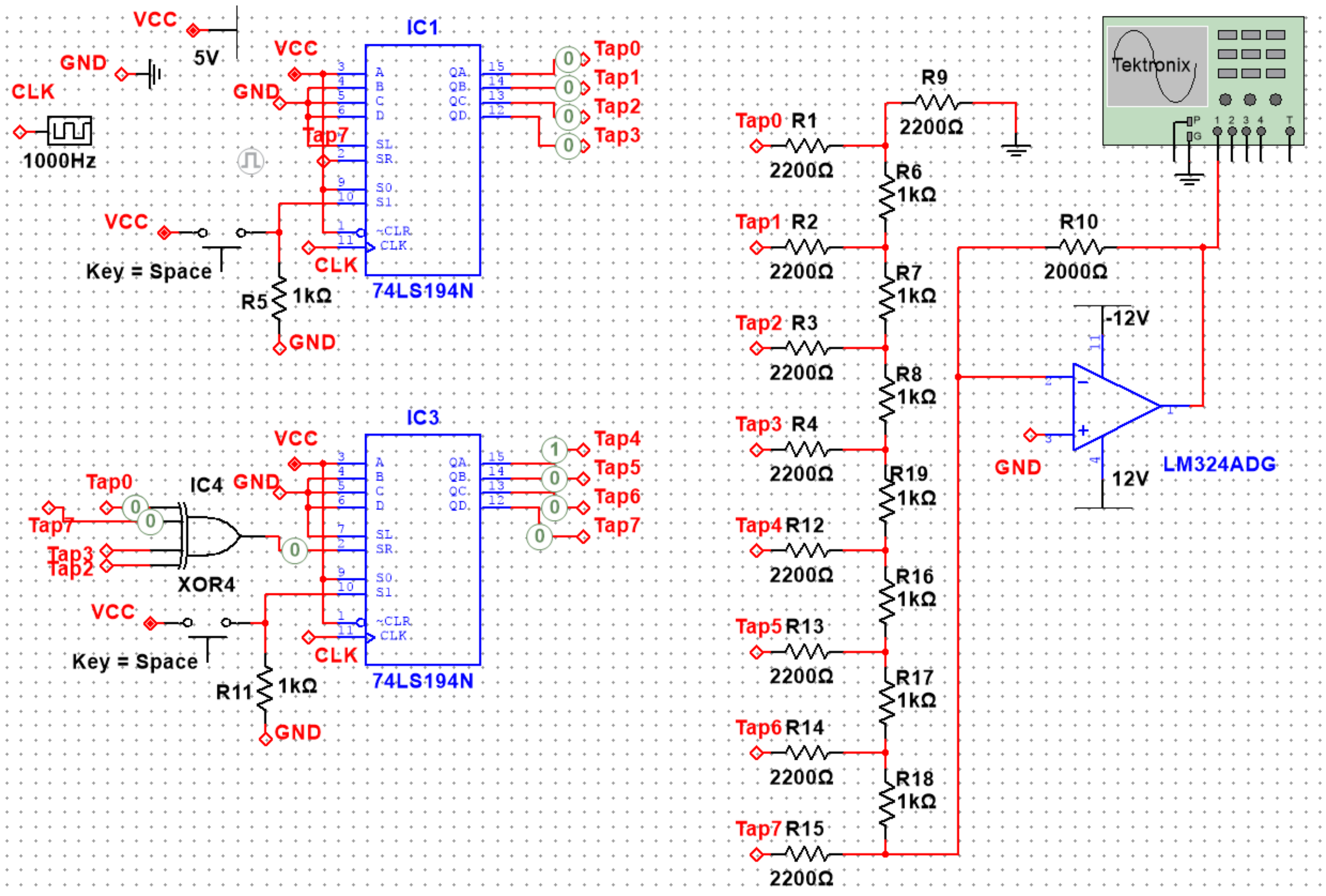


Figure 15: 8-bit DPRNG schematic

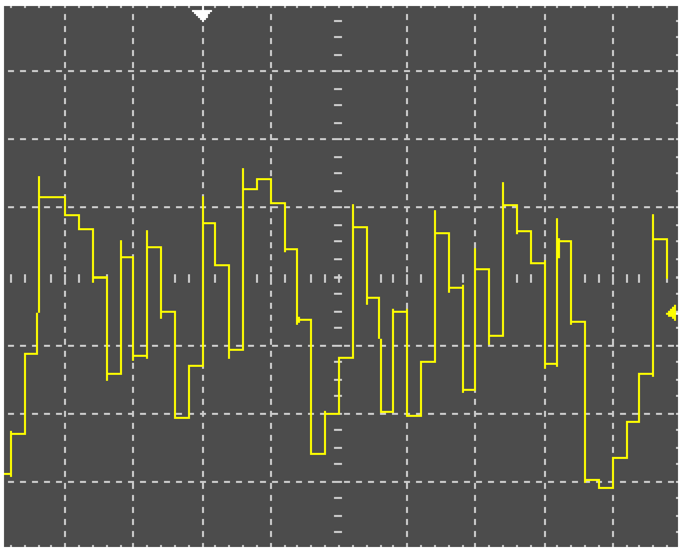


Figure 16: Output Results 8-bit DPRNG

10. We are tasked with analyzing the DPRNG from the last two procedures that use the R-2R DAC.

    a. In this section, we are asked why the two values 1k and 2.2k for the resistors instead of 1k and 2k that's seen in the previous lab where we built R-2R DACs. This is because when using a R-2R DAC, we consider a voltage divider across these two resistors. With the 1k + 2.2k, we have a voltage divider factor of 0.66 and with the 1k + 2k, we have a voltage divider factor of 0.5. With the 0.66 factor, it causes the voltage differences to be larger than the 0.5 factor, which results in each voltage level being more defined (farther apart) and less risk of bubble errors occurring when we increase the number of bits.

    b. In this section, we are tasked with replacing the R-2R DAC with a binary-weighted DAC. Then we are asked to compare the two different circuits. In Figure 17, we can see the schematic with a 8-bit binary weighted DAC. In Figure 18, we can see the waveform output. Please note that due to an 8-bits having 256 outputs ($2^8 = 256$), we can only display part of the waveform. We selected the Single Sequence option on the oscilloscope for our waveform.

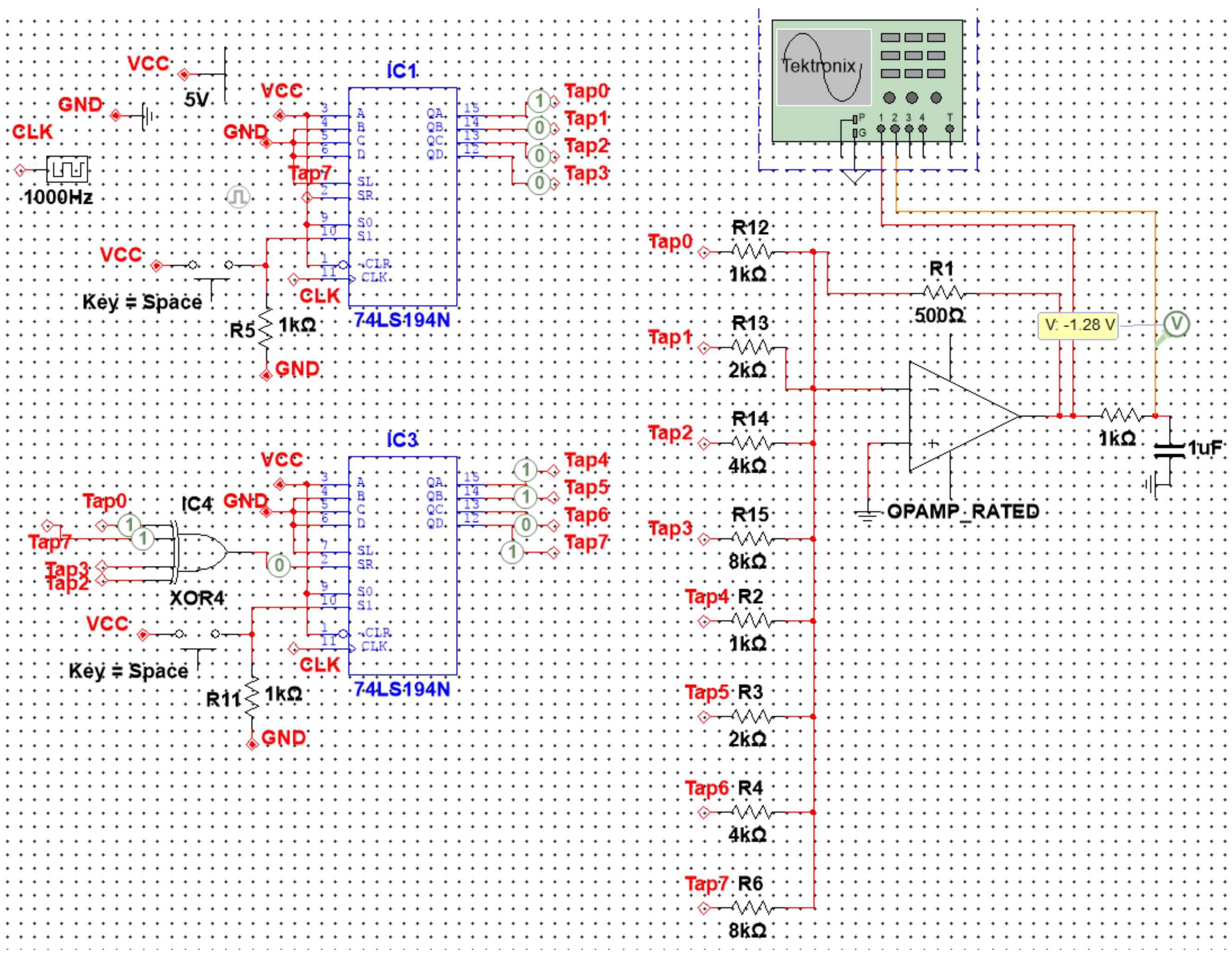


Figure 17: 8-bit LFSR with 8-bit Binary Weighted DAC (BWDAC) schematic

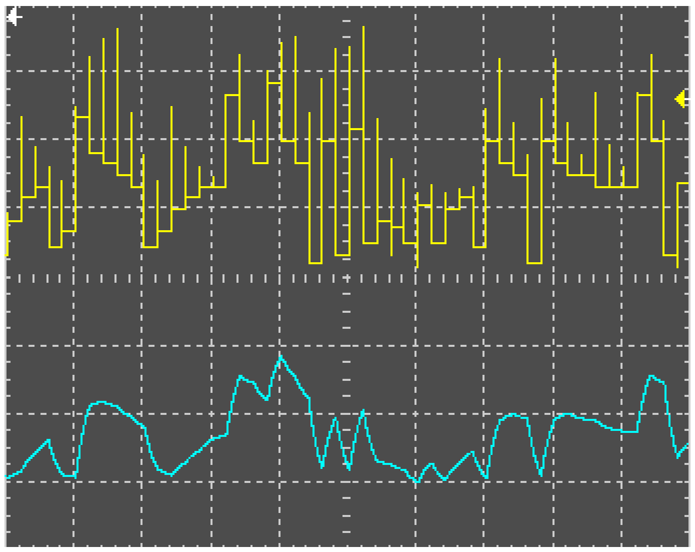


Figure 18: 8-bit LFSR with 8-bit BWDAC Output Results

When we compare the output results in Figure 18 and 16, we can see that the BWDAC has more consistent output levels, rather than output levels that are “random and all over the place”. We can see there are more anomalies in Figure 18, with the random up and down spikes happening more often. We expected some sort of anomalies, with the anomalies we had in the R2-R DAC version, so we added in a filter to the output to attempt to pull those out and it did not work. This means that the BWDAC is better than the other.

We can also say that the BWDAC is more practical, because of the more defined voltage levels, the fact it uses half the amount of resistors and it uses a simple OP AMP instead of a LM324 IC that requires +/- 12V rails. The BWDAC version will also take up less space and cost less to manufacture, which makes it ideal to use.

Conclusion

This lab was very interesting to see how Shift Registers work to produce a random sequence of outputs. This could potentially be incorporated into hashing, like SHA-254. It is the same concept, but for SHA-254, we would use python to code and decode a string of characters or numbers.

If we were tasked with expanding on this lab, I would focus on getting the output waveform on the 8-bit LFSR with the BWDAC to have no anomalies. We would potentially do this by having a more complex filter on the output before the oscilloscope. We could also attempt to make a SHA-254 hashing unit, but we feel this is more advanced than what is expected in this class. A LFSR could also be used to build a strobe light that is random in the flashing sequence. There are quite a few applications that this circuit would be implemented in.

We are specifically interested in SHA-254 hashing because that is what is used in crypto currencies and why they are so secure. There are 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936 possible outputs for SHA-256, so getting the exact key to unlock it via hacking, is nearly impossible. Cryptography is becoming more and more important, with the increase in hacking and adversarial behavior from people, countries and entities who want to do us harm. Cryptography allows us to secure communication and data, and this type of circuit is a similar encryption style.