

---

# Pseudo-Random Number Generator

---

**Project Report**  
**ELL201**

**Author**

Krishna Sharma - 2022MT11263

April 27, 2024

# Contents

<b>1</b>	<b>Pseudo-Random Number Generators</b>	<b>3</b>
1.1	Linear Feedback Shift Registers . . . . .	3
1.2	Cellular Automata Shift Registers . . . . .	3
<b>2</b>	<b>Design</b>	<b>3</b>
2.1	Verilog Code . . . . .	3
2.2	Testbench Code . . . . .	5
2.3	Circuit Diagram . . . . .	6
<b>3</b>	<b>Output</b>	<b>6</b>
3.1	Numbers . . . . .	6
3.2	Waveform . . . . .	7
<b>4</b>	<b>Testing</b>	<b>7</b>
4.1	NIST Statistical Test Suite . . . . .	7
4.2	Result . . . . .	9
<b>5</b>	<b>Applications</b>	<b>9</b>
5.1	Cryptography . . . . .	9
5.2	Monte-Carlo Simulations . . . . .	10
5.3	Testcases Generation . . . . .	10
5.4	Game Simulations . . . . .	10

# 1 Pseudo-Random Number Generators

A **pseudo-random number generator (PRNG)**, also referred to as a **deterministic random bit generator (DRBG)**, is an **algorithm** designed to generate a sequence of numbers that mimic the properties of truly random sequences. Unlike true random sequences, however, the sequence produced by a PRNG is determined by an initial value known as the **seed**. Moreover, PRNG-generated sequences find extensive usage due to their speed in number generation and their reproducibility.

## 1.1 Linear Feedback Shift Registers

A **linear-feedback shift register (LFSR)** is a shift register whose input bit is a linear function of its previous state.

The most commonly used linear function of single bits is **exclusive-or (XOR)**. Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value.

The initial value of the LFSR is called the **seed**, and because the operation of the register is deterministic, the sequence of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

## 1.2 Cellular Automata Shift Registers

A **cellular automata shift register (CASR)** is a type of shift register that operates based on principles inspired by cellular automata. It consists of a grid of cells, each representing a storage element (e.g., a bit). The state of each cell, typically binary (0 or 1), is updated based on the states of neighboring cells and predefined transition rules, similar to the rules governing cellular automata. During a shift operation, the contents of the grid, representing the shift register's data, are shifted, with each cell's state being updated according to the transition rules and neighboring cell states. The output of the CASR is usually derived from specific cells in the grid, providing a dynamic and adaptable behavior influenced by the principles of cellular automata.

# 2 Design

LFSR and CASR, individually, may not introduce significant randomness into the sequence of generated numbers. Therefore, our approach is to use the bits generated by LFSR and CASR separately, and then XOR these bits together to produce the final random bit of the sequence. All our codes and outputs can be found [here](#).

## 2.1 Verilog Code

```
1 module pro(clk, reset, number_o);  
2   input clk;  
3   input reset;
```

```

4  output reg[7:0] number_o;
5
6  reg[12:0] lfsr_reg;
7  reg[10:0] casr_reg;
8  reg[12:0] lfsr_var;
9  reg outbitlfsr;
10
11 always @(posedge clk or negedge reset)
12 begin
13     if (!reset)
14         begin
15             lfsr_reg = 13'b0100100100100;
16         end
17     else
18         begin
19             lfsr_var = lfsr_reg;
20             lfsr_var [12] = lfsr_var[11];
21             outbitlfsr = lfsr_var[11];
22             lfsr_var[11] = lfsr_var[10]^outbitlfsr;
23             lfsr_var[10] = lfsr_var[9];
24             lfsr_var[9] = lfsr_var[8]^outbitlfsr;
25             lfsr_var[8] = lfsr_var[7]^lfsr_var[11];
26             lfsr_var[7] = lfsr_var[6]^outbitlfsr;
27             lfsr_var[6] = lfsr_var[5];
28             lfsr_var[5] = lfsr_var[4];
29             lfsr_var[4] = lfsr_var[3]^lfsr_var[6];
30             lfsr_var[3] = lfsr_var[2]^outbitlfsr;
31             outbitlfsr = lfsr_var[5];
32             lfsr_var[2] = lfsr_var[1]^outbitlfsr;
33             lfsr_var[1] = lfsr_var[0]^outbitlfsr;
34             lfsr_var[0] = lfsr_var[12];
35             lfsr_reg = lfsr_var;
36         end
37     end
38
39     reg [10:0] casr_var, casr_out;
40     always @(posedge clk or negedge reset)
41     begin
42         if (!reset)
43             casr_reg = 11'b00010101010;
44         else
45             begin
46                 casr_var = casr_reg;
47                 casr_out[10] = casr_var[9]^casr_var[0];
48                 casr_out[9] = casr_var[8]^casr_out[10];
49                 casr_out[8] = casr_var[7]^casr_out[9];
50                 casr_out[7] = casr_var[6]^casr_out[8];
51                 casr_out[6] = casr_var[5];
52                 casr_out[5] = casr_var[4]^casr_out[6];
53                 casr_out[4] = casr_var[3]^casr_out[5]^casr_var[4];
54                 casr_out[3] = casr_var[2]^casr_out[4];
55                 casr_out[2] = casr_var[1]^casr_out[3]^casr_var[6];
56                 casr_out[1] = casr_var[0]^casr_out[2];
57                 casr_out[0] = casr_var[1]^casr_out[10]^casr_var[8];
58                 casr_reg = casr_out;
59             end
60         end
61
62     always @(posedge clk or negedge reset)
63     begin
64         if (!reset)
65             begin
66                 number_o = 8'b0;
67             end
68         else
69             begin
70                 number_o = (lfsr_reg [7:0] ^ casr_reg [7:0]);

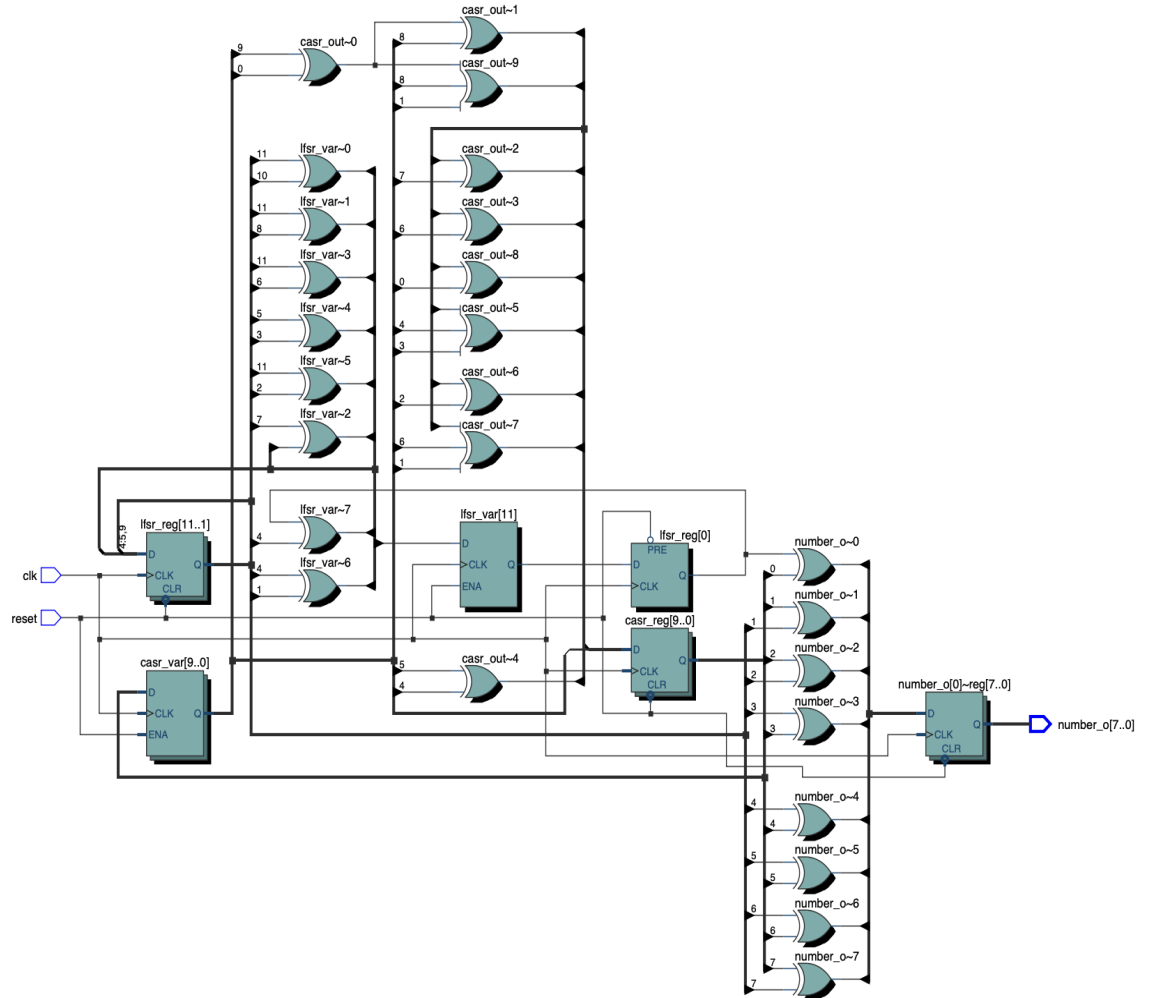
```

```
71         end
72     end
73 endmodule
```

## 2.2 Testbench Code

```
1  `timescale 1ns / 1ps
2
3  module testbench();
4
5      // Inputs
6      reg clk;
7      reg reset;
8
9      // Outputs
10     wire [7:0] number_o;
11
12     pro UUT(
13         .clk(clk),
14         .reset(reset),
15         .number_o(number_o)
16     );
17
18     // Clock generation
19     always #1 clk = ~clk; // Toggle clock every 1 time unit
20
21     // Initial reset assertion
22     initial begin
23         clk = 0;
24         reset = 0;
25         #5 reset = 1;
26     end
27
28     integer f;
29     // Monitor and write to file
30     initial begin
31         f = $fopen("out.txt", "w");
32         repeat (100000) begin
33             @(posedge clk or negedge reset);
34             $fwrite(f, "%d\n", number_o);
35         end
36
37         $fclose(f);
38         $finish;
39     end
40
41 endmodule
```

## 2.3 Circuit Diagram



## 3 Output

In our testbench code, we generated a random sequence consisting of a total of 100k numbers and wrote these numbers to a *.txt* file.

### 3.1 Numbers

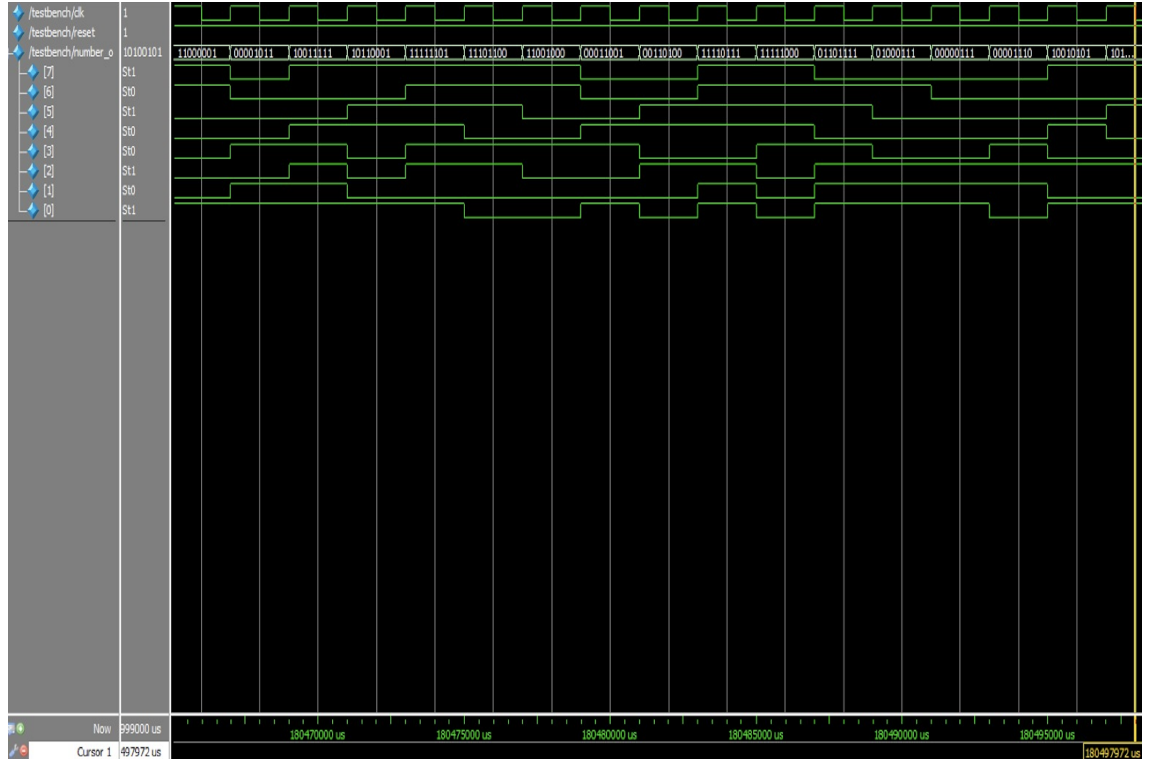
Some of the initial numbers of the generated sequence are provided below:

output
0, 0, 0, 54, 94, 30, 186, 69, 136, 63, 196, 14, 156, 25, 23, 9, 62, 200, 62, 225, 85, 14, 26, 150, 35,
80, 35, 112, 126, 203, 186, 104, 96, 219, 154, 190, 192, 128, 3, 163, 92, 182, 246, 80, 14, 152, 154,
149, 10, 52, 249, 207, 48, 220, 190, 72, 178, 248, 202, 54, 245, 121, 106, 66, 37, 219, 29, 48, 217, 29,
19, 140, 54, 125, 75, 145, 43, 79, 152, 146, 10, 149, 160, 82, 4, 12, 144, 175, 77, 151, 165, 91, 145,
41, 70, 37, 252, 99, 127, 111, 77, 58, 72, 180, 251, 231, 121, 194, 12, 145, 143, 29, 184, 67, 137, 179,

89, 158, 63, 227, 216, 188, 102, 248, 203, 55, 216, 189, 103, 213, 15, 58, 79, 151, 134, 135, 168, 101, 213, 44, 111, 100, 249, 65, 32, 254, 97, 220, 149, 42, 105, 203, 25, 177, 82, 163, 119, 212, 45, 99, 221, 31, 61, 79, 23, 162, 241, 119, 240, 243, 90, 26, 182, 214, 13, 60, 78, 27, 27, 23, 11, 159, 26, 153, 55, 84, 12, 151, 173, 77, 28, 159, 185, 64, 5, 44, 225, 253, 64, 174, 194, 35, 216, 185, 228, 222, 30, 18, 42, 200, 146, 13, 151, 160, 217, 62, 238, 65, 131, 39, 83, 164, 120, 64, 132, 40, 199, 13, 51, 90, 150, 38, 210

## 3.2 Waveform

The output waveform for the generated random sequence is provided below:



## 4 Testing

How can we ensure that the generated sequence is truly random? Our sequence of random numbers passed **9 out of 16** standard tests for randomness listed in NIST statistical test suite. The tester can be found [here](#).

### 4.1 NIST Statistical Test Suite

The **NIST Statistical Test Suite** is a collection of statistical tests developed by the **National Institute of Standards and Technology (NIST)** for evaluating the randomness of random number generators (RNGs) and cryptographic algorithms. These tests are designed to assess the quality and randomness of sequences of binary data produced by RNGs or cryptographic algorithms.

The NIST Statistical Test Suite includes a comprehensive set of tests that evaluate various statistical properties and characteristics of the data, such as:

- **Frequency Test:** Check if the proportion of ones and zeros in the sequence is approximately equal.
- **Frequency Test within a block:** Examine the frequency of predefined patterns or blocks of bits within the sequence.
- **Runs Test:** Assess the presence of runs or sequences of consecutive ones or zeros in the sequence.
- **Test for the Longest Run of Ones in a Block:** Determine the longest consecutive sequence of ones in the sequence.
- **Binary Matrix Rank Test:** Evaluate the rank of matrices derived from the sequence.
- **Discrete Fourier Transform (Spectral) Test:** Analyzes frequency distribution to assess randomness.
- **Non-overlapping Template Matching Test:** Detect the presence of specific patterns or templates within the sequence.
- **Overlapping Template Matching Test:** Similar to non-overlapping template matching, but with overlapping templates.
- **Maurer’s “Universal Statistical” Test:** Assess the proportion of ones within certain subsequences of the sequence.
- **Linear Complexity Test:** Measures predictability based on shortest feedback shift register length.
- **Serial Test:** Examine correlations between adjacent bits or sequences of bits within the sequence.
- **Approximate Entropy Test:** Measure the randomness of the sequence by assessing its entropy.
- **Cumulative Sums Test:** Check for deviations from expected cumulative sums of the sequence.
- **Random Excursions Test:** Analyze the number of excursions from a predetermined baseline in the sequence.
- **Random Excursions Variant Test:** Similar to random excursions tests, but with additional criteria.



## 4.2 Result

The final output displaying the results passed (and failed) along with the respective **p-values** is provided below. It should be noted that the threshold p-value for the “Test for the Longest Run of Ones in a Block” is 0.01, and the value we received is 0.00934. Therefore, this test failed by a narrow margin; otherwise, we would have passed **10 out of 16** standard tests.

Type of Test	P-Value	Conclusion		
01. Frequency (Monobit) Test	0.9127525207914535	Random		
02. Frequency Test within a Block	1.4024826890989044e-36	Non-Random		
03. Runs Test	0.8457652178254071	Random		
04. Test for the Longest Run of Ones in a Block	0.009338716697701553	Non-Random		
05. Binary Matrix Rank Test	0.0	Non-Random		
06. Discrete Fourier Transform (Spectral) Test	0.0	Non-Random		
07. Non-overlapping Template Matching Test	0.8794434173445763	Random		
08. Overlapping Template Matching Test	0.011725941459423392	Random		
09. Maurer's "Universal Statistical" Test	0.31817046032235374	Random		
10. Linear Complexity Test	0.0	Non-Random		
11. Serial Test:			0.0	Non-Random
			0.0	Non-Random
12. Approximate Entropy Test	0.0	Non-Random		
13. Cumulative Sums Test (Forward)	0.9941787759566996	Random		
14. Cumulative Sums Test (Backward)	0.9962003297145519	Random		
15. Random Excursions Test:				
State	Chi Squared	P-Value	Conclusion	
-4	1.6969935952964255	0.8892782974180536	Random	
-3	3.1887695817490496	0.6709091906581793	Random	
-2	4.215180960428109	0.518869403439787	Random	
-1	12.453738910012675	0.029072604537161813	Random	
+1	5.916349809885931	0.3144437464444382	Random	
+2	4.420723215822497	0.4905599452692553	Random	
+3	7.769149809885934	0.16942554362305548	Random	
+4	3.652526487432096	0.6004443240451137	Random	
16. Random Excursions Variant Test:				
State	COUNTS	P-Value	Conclusion	
-9.0	1569	0.969005806068061	Random	
-8.0	1642	0.7686448409353488	Random	
-7.0	1664	0.6711437697170246	Random	
-6.0	1667	0.6328872918418516	Random	
-5.0	1647	0.6822381849056846	Random	
-4.0	1632	0.7163740172053483	Random	
-3.0	1617	0.756208208338158	Random	
-2.0	1630	0.5930582058703868	Random	
-1.0	1604	0.6434989459129046	Random	
+1.0	1562	0.7757915939008643	Random	
+2.0	1505	0.45311691241123786	Random	
+3.0	1480	0.43530844226198684	Random	
+4.0	1504	0.618577092058646	Random	
+5.0	1529	0.7712493872923101	Random	
+6.0	1544	0.8552064425677152	Random	
+7.0	1527	0.8012071853421796	Random	
+8.0	1497	0.7096836304974316	Random	
+9.0	1483	0.6817042766860951	Random	

## 5 Applications

PRNGs are versatile tools used in various applications, where randomness and unpredictability are essential for achieving desired outcomes. Some of them are mentioned below:

### 5.1 Cryptography

PRNGs ensure the creation of cryptographic keys, initialization vectors, and nonces, critical for secure communication. They provide the randomness needed for encryption, decryption, and cryptographic hashing, enhancing the security of data transmission and storage.

## 5.2 Monte-Carlo Simulations

PRNGs generate random inputs essential for Monte Carlo simulations, enabling the modeling and analysis of complex systems in various fields such as finance, engineering, and physics. They facilitate the estimation of probabilities, solution of optimization problems, and simulation of stochastic processes, contributing to decision-making and problem-solving in diverse domains.

## 5.3 Testcases Generation

PRNGs are indispensable for generating diverse sets of test cases, ensuring thorough evaluation of software functionality, performance, and robustness. They help in simulating real-world scenarios and edge cases, improving test coverage and identifying potential issues or vulnerabilities in software systems.

## 5.4 Game Simulations

In game development, PRNGs are essential for generating random events, outcomes, and environments, enhancing gameplay dynamics and replayability. They enable the creation of unpredictable gameplay experiences, including random enemy behavior, procedural level generation, and random item drops, enriching player immersion and engagement.

## References

- [1] [https://en.wikipedia.org/wiki/Pseudorandom\\_number\\_generator](https://en.wikipedia.org/wiki/Pseudorandom_number_generator)
- [2] [https://en.wikipedia.org/wiki/Linear-feedback\\_shift\\_register](https://en.wikipedia.org/wiki/Linear-feedback_shift_register)
- [3] <https://www.yumpu.com/en/document/read/41498041/a-random-number-generator-in-verilog>