Lab 03: Pseudo- and True Randomness

Jasmin Kaur and Alec Braynen
University of South Florida
Tampa, FL 33620

I. Introduction

Random number generators (RNGs) see many applications today as devices of generating randomness in computing systems. They are often critical in areas where it is important to have repeatable non-deterministic implementations such as randomizing the neural network parameters, introducing unpredictibility in gambling/gaming systems, cryptographic systems, and so on [1]. Depending upon the *components* and how the random bits are generated by an RNG, it can be categorized into three different classes - pseudo-random number generators (PRNG), cryptographically secure PRNG (CPRNG), and true random number generators (TRNGs) [1].

The PRNGs make use of mathematical algorithms to generate random bits/numbers. PRNGs are not considered truely random as they rely on an initial *seed* value fed to a mathematical algorithm to generate random bits such as a linear feedback shift register (LFSR). PRNGs could be enhanced to be cryptographically secure to be used in cryptographic applications (CPRNGs). TRNGs are the only RNGs that are truly random because they rely on the inherent randomness of physical phenomenon such as thermal noise, variations in temperatures, quantum interactions, and so on to generate random numbers. Additionally, TRNGs could be used to generate an initial random *seed* for CPRNGs [1].

In this work, the authors experiment with implementing two different kinds of TRNGs and PRNGs and determining whether they are truly random by testing the generated random bits using NIST test suite. The NIST test suite a statistical testing framework consisting of 15 tests used to determine randomness of a given bit stream [1]. For this experiment, two different bit streams consisting of 1 million (1M) and 10 million bits (10M) are first generated using an FPGA programmed with a ring-oscillator LFSR TRNG (RO_LFSR_TRNG), and then a Galios ring-oscillator TRNG (GARO). Another set of 1M and 10M random bits are generated using Numpy's random library as well as Python's in-built random number generator "os.urandom". The NIST tests are then performed for all the generated random bit streams and the test results are plotted and annotated using Python's "matplotlib.pyplot" library. Results show that the LFSR TRNG only pass 4 of the 15 NIST tests while the GARO TRNG passes 13 of the 15. The Numpy and Python RNGs however pass all of the tests.

II. Reading Check

Question 1: What kinds of applications are TRNGs used for?

Answer: TRNGs are typically used in applications where unpredictibility is incredibly important such as random sampling, gambling softwares, cryptographic systems, and so on. One of the main applications include communications systems in IoT where TRNGs are required to securely send/receive data using cryptographic systems. [1].

Question 2: What are the 3 major components of a good TRNG design?

Answer: The three main components of a good TRNG are the entropy source, the harvesting mechanism, and the post-processing step [1] . The entroy source could be any physical phenomenon such as clock jitter, temperature variations, etc. that are considered unpredictable and aperiodic to be used to harvest randomness. The harvesting mechanism should be designed in a manner that would not disturb the entropy of the systems it's collecting the information from. Finally the post-processing step is to mask imperfections and/or provide security measures against fault tolerance.

Question 3: How can we evaluate whether or not a TRNG is truly random?

Answer: Most of the time the standardized tests to determine the randomness heavily depend upon sound mathematics and statistical tools and their optimizations. The NIST Statistical Test Suite is one such standard used to determine the randomness of a binary sequence. It comprises of 15 statistical tests which test different features of a binary sequence. The tests performed in NIST test suite are - Frequency monobit, frequency test within a block, runs test, longest run of 1's in a block, binary matrix rank, Discrete Fourier Transform (Spectral), non-overlapping template matching, overlapping template matching, Maurer's "Universal Statistical" test, linear complexity, serial test, approximate entropy, cumulative sums, random excursions, and random excursion variants. [1].

III. Methods

For this experiment 1M and 10M random bits were generated using four different types of RNGs. The first two sets of 1M and 10M random bits were harvested from RO_LFSR_TRNG and GARO_TRNG implemented on CMOD S7 FPGA using PySerial and Diligent software. The second two sets of 1M and 10M random bits were genereted using two PRNGs available in Python - Numpy's random library and Python's built-in RNG "os.urandom". The generated random bits were writen as raw bytes to a binary (.bin) files.

A. Software Setup

The software setup for the experiment consists of a virtual machine running Ubuntu OS on Virtualbox. The Python code is implemented via Jupyter notebooks on this virtual machine. PySerial and Diligent software were used to program the FPGA and to send the number of bytes to generate and receive the random bytes. The libraries numpy, os were used to generate random bits for the PRNGs.

The directory structure of the files in the Jupyter notebook is structured as follows:

```
PHS-Lab-03
├─ Code
│   ├─ Lab-03-Acquisition.ipynb
│   ├─ Lab-03-Analysis.ipynb
│   ├─ Lab-03-Acquisition-PRNG.ipynb
│   ├─ Lab-03-Analysis-PRNG.ipynb
│   ├─ NIST.py
│   └─ tests
├─ Data
│   ├─ RO_1M.bin
│   ├─ RO_10M.bin
│   ├─ GARO_1M.bin
│   ├─ GARO_10M.bin
│   ├─ NUMPY_1M.bin
│   ├─ NUMPY_10M.bin
│   ├─ OS_1M.bin
│   └─ OS_10M.bin
├─ Figures
│   ├─ RO_1M.pdf
│   ├─ RO_10M.pdf
│   ├─ GARO_1M.pdf
│   ├─ GARO_10M.pdf
│   ├─ NP_1M.pdf
│   ├─ NP_10M.pdf
│   ├─ OS_1M.pdf
│   └─ OS_10M.pdf
└─ Hardware
    ├─ RO_LFSR_TRNG.bit
    └─ GARO_TRNG.bit
```

1) The ”Code” directory hold the jupyter notebook files where the python code is executed along with the NIST.py file for the NIST tests and a test folder holding the py files for the NIST.py tests.
2) The ”Data” directory holds the random response bits received from the FPGAs, NumPy, and Python as output.
3) The ”Figures” directory holds the generated figures of the analyzed data.
4) The ”Hardware” directory holds the RO TRNG and GARA TRNG bitfiles.

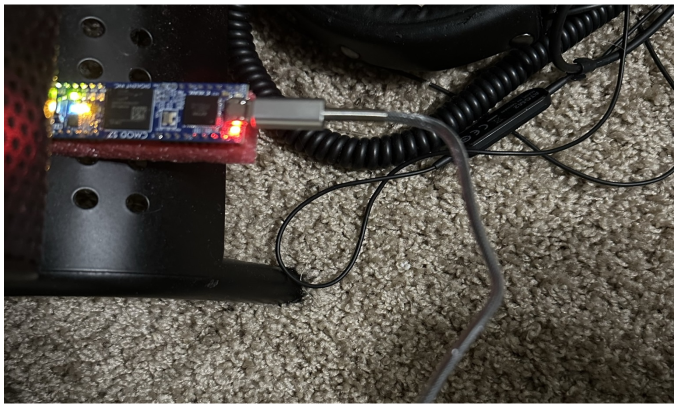


Fig. 1: FPGA Connected to Desktop

B. Hardware Setup

The hardware setup consisted of the FPGA connected via a microUSB cable to a desktop machine running Windows (Fig. 1).

C. Experimental Procedure

Part A: Sampling Random Bit Responses

Harvesting random bits from the FPGA:

- The FPGA is programmed with the relevant bitfile (RO_LFSR_TRNG.bit OR GARO_TRNG.bit)
- 1M and 10M bit responses of random data are sampled from it using Pyserial (This is indicated by sending an integer converted to 3 bytes as input to the FPGA requesting an amount of random responses).
- The bit responses are saved as bin files in the data folder.

Harvesting random bits from ”numpy.random” and Python’s ”os.urandom” PRNGs:

- 1M and 10M bit responses of random data are sampled using ”numpy.random.default_rng.bytes()” and ”os.urandom” in Python.
- The bit responses are saved as bin files in the data folder.

Part B: Data Analysis

The analysis of the bit responses are done by:

- Using the TRNGtester from the NIST.py script in the code folder, along with the NumPy and Matplotlib libraries.

- Instances of the TRNGTester are created with the .bin file location of the generated data in A passed as arguments.
- The run_nist_tests() method is called on the instances of TRNGTester which outputs the RNG's results on the 15 tests.
- The minimum p-values are extracted from each test for analysis and plotting.

IV. Results

The results of our work is shown in Figures 2-5 respectively. The checkmarks on each bar on the bar plot indicates that a test was passed. The 'X' marks indicate that a test was failed. T0 through T14 on the figures refer to the NIST tests [2]:

1) The Frequency (Monobit) test
2) The Test for Frequency within a Block test
3) The Runs test
4) The Test for the Longest-Run-of-Ones
5) The Random Binary Matrix test
6) The Discrete Fourier Transform (Spectral) test
7) The Non-overlapping Template Matching test
8) The Overlapping Template Matching test
9) Maurer's "Universal Statistical" test
10) The Linear Complexity test
11) The Serial test
12) The Approximate Entropy test
13) The Cumulative Sums
14) The Random Excursions test
15) The Random Excursions Variant test

As the figures show, the RO LFSR RNG passed 4 out of 15 tests with both its 1 million and 10 million generated random bits. The GARO RNG passed 13 of the 15 tests with both its 1 million and 10 million generated random bits. The NumPy and Python OS RNGs however passed all 15 of the NIST tests as also shown in the figures.

These results show that the PRNG implementations of NumPy and Python are well implemented RNGs. They passed every single NIST RNG test. The RO LFSR RNG was the worst of all of our tested RNGs, only passing 4 of the NIST tests. The GARO RNG, while not passing every NIST test, shows that its implementation is pretty good for small embedded devices. It passed 13 of the 15 NIST tests which is good with its limited source of entropy and computational power.

V. Discussion

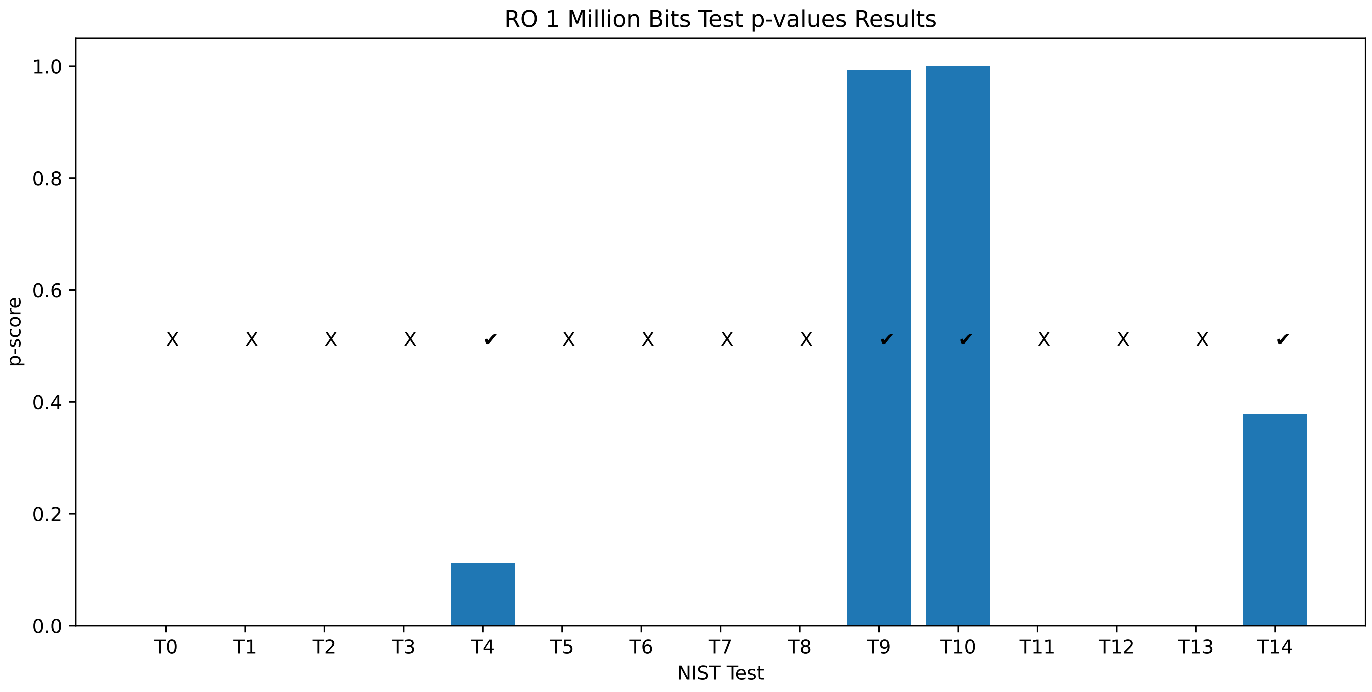
These results show generally that the complex PRNGs used by modern software such as Python and NumPy, outperform the RO and GARO RNGs used on our FPGA devices. Not only do the Python and NumPy iterations pass every NIST test, they generate the random data at a much faster output as well.

Nevertheless, this shows the necessity of research of RNGs for smaller embedded devices such as FPGAs. FPGAs and other embedded devices are typically low power and do not have the high clock speeds and the sources of entropy that the more powerful machines have and that more complex RNG implementations may require. This means that designing good RNGs for FPGAs has different challenges that must be overcome.

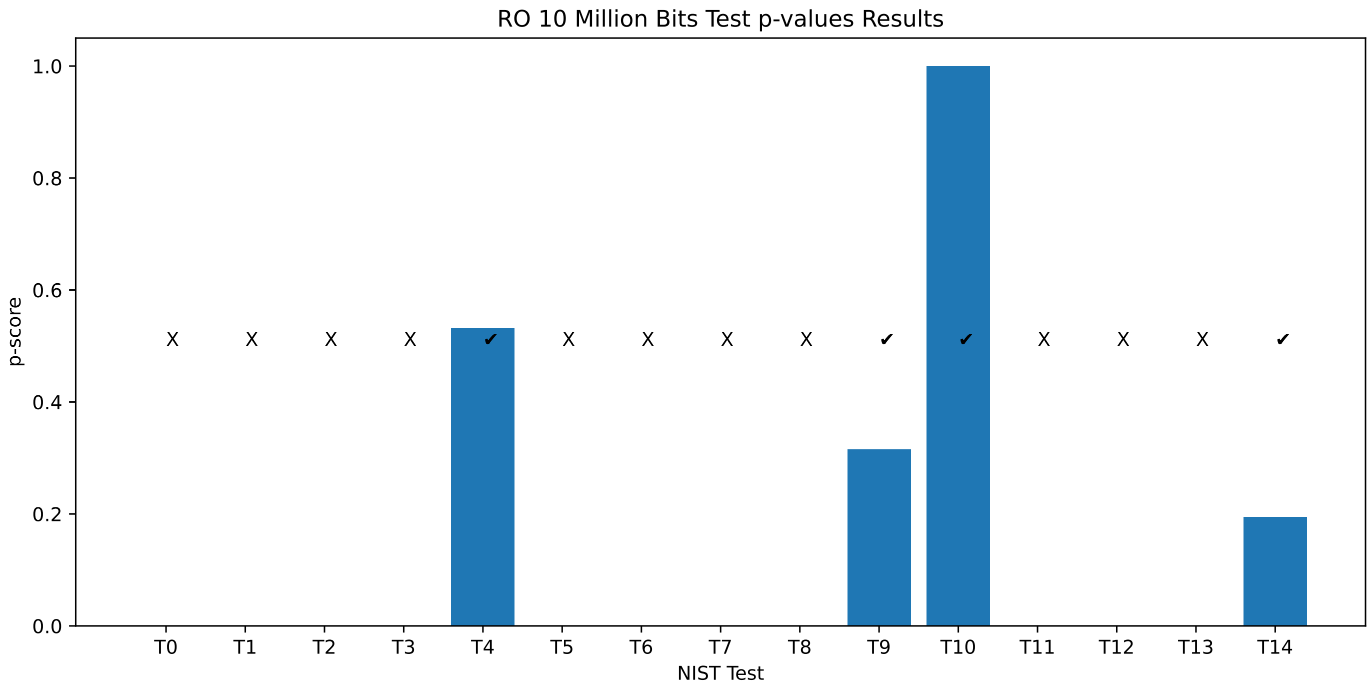
Despite this, our GARO RNG still showed good performance on the NIST tests. It passed 13 of the 15 tests, failing only the random excursions variant and the non-overlapping template matching tests. So we recommend, if deciding between implementing a RO RNG and a GARO RNG, to implement the GARO RNG for a higher amount of randomness.

During the experiment, we had issues properly extracting all of the random bits from the FPGA. We had to use Pyserial's in_waiting() method to wait for the FPGAs buffer to fill and then read bytes from the FPGA buffer 1 byte at a time. This was to ensure we didn't miss or double read any bytes.

All in all in future work, we would investigate contributing additional sources of entropy into the RO and GARO RNGs. We tested the FPGA in an isolated environment; however, testing it in some context where the environment may be able to contribute entropy as a source may show better results.

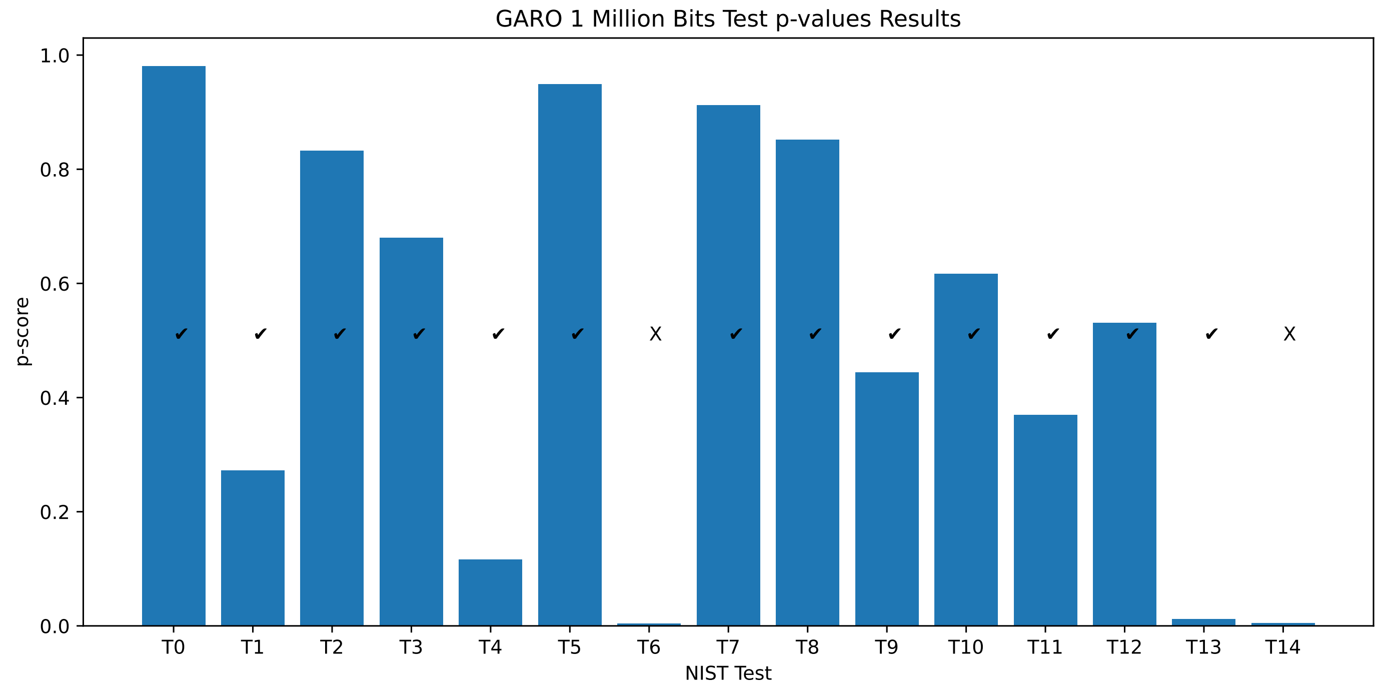


(a)

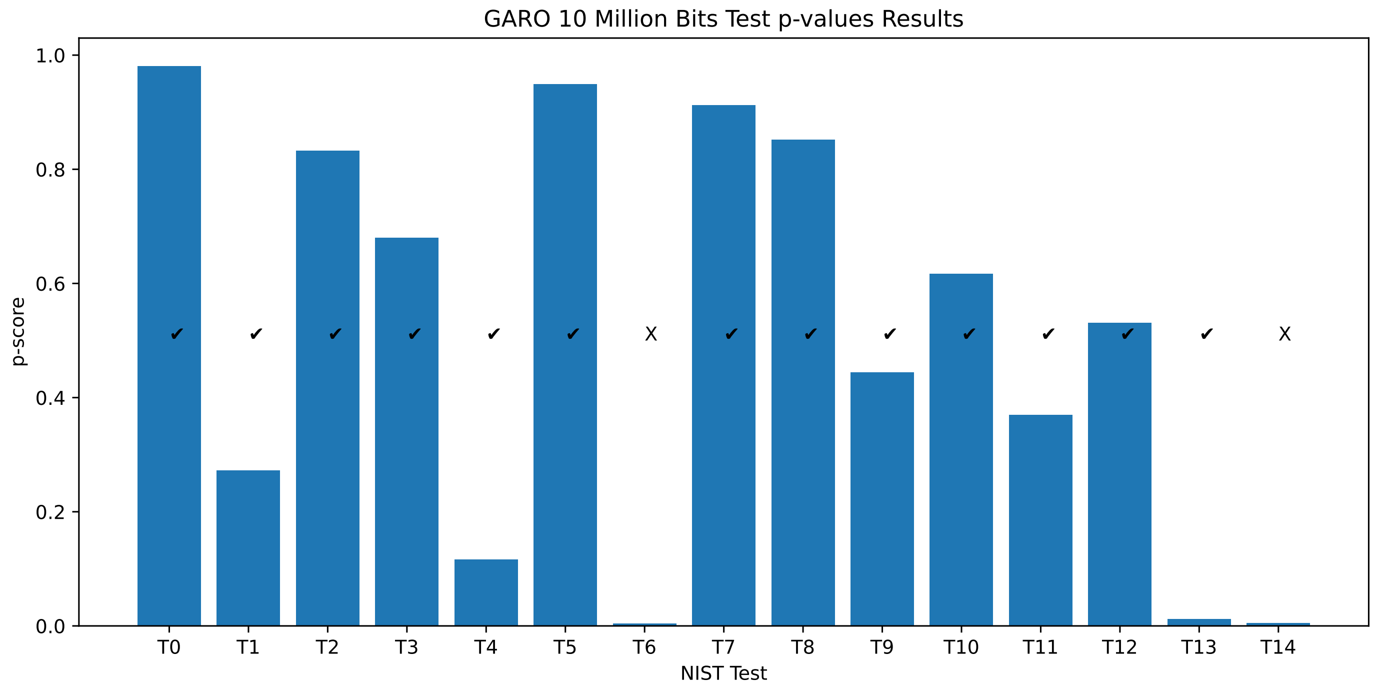


(b)

Fig. 2: NIST test results a) 1M and b) 10M random bits generated using RO LFSR RNG

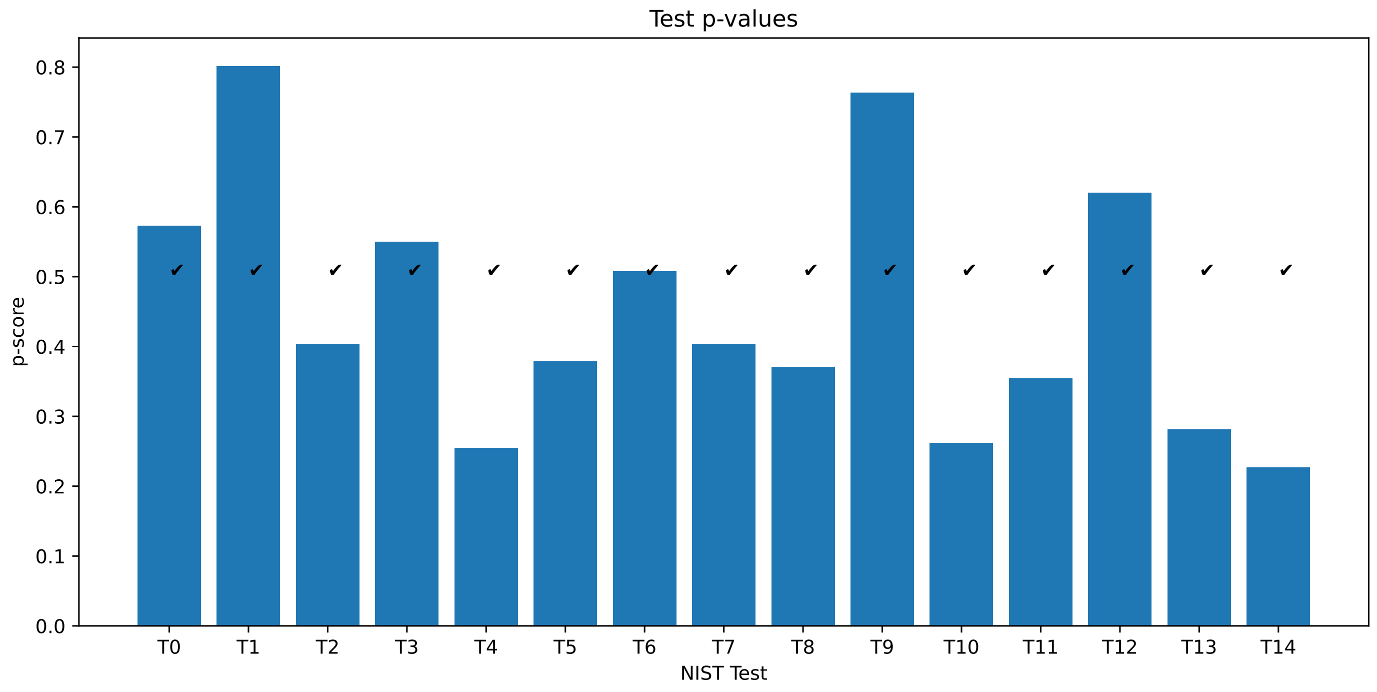


(a)

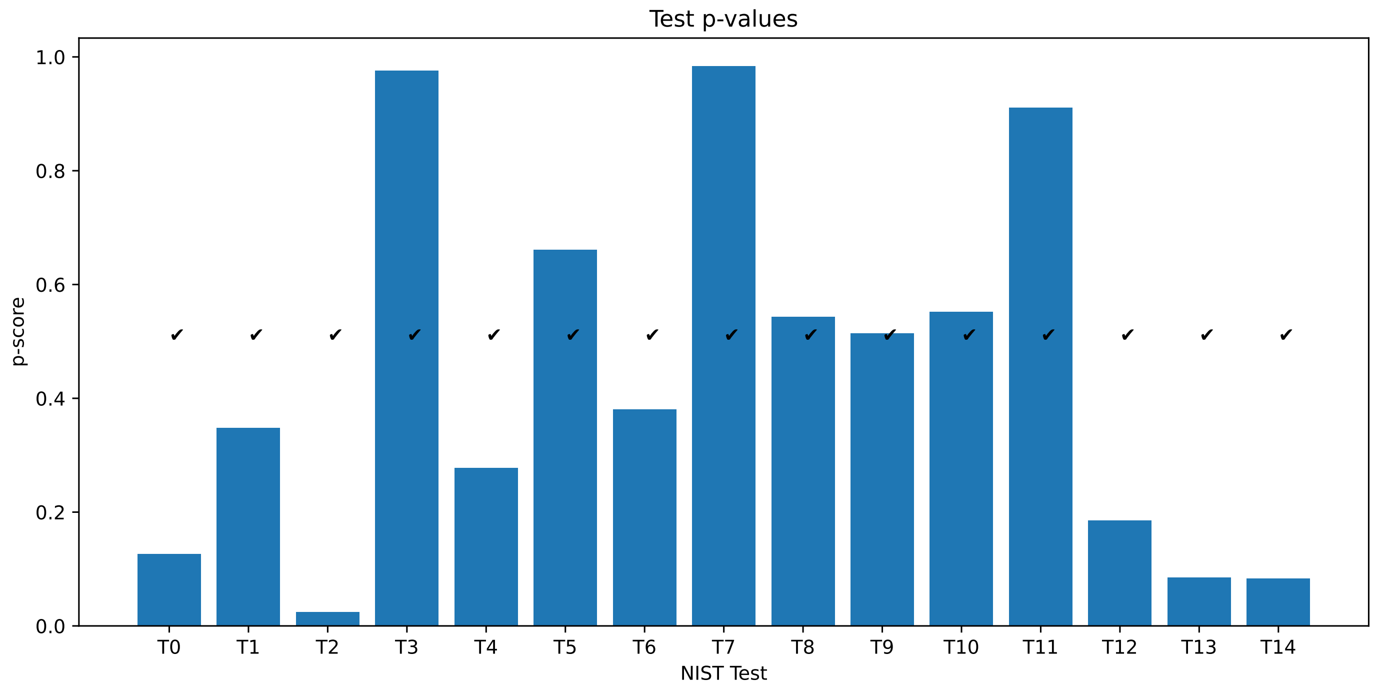


(b)

Fig. 3: NIST test results a) 1M and b) 10M random bits generated using GARO RNG

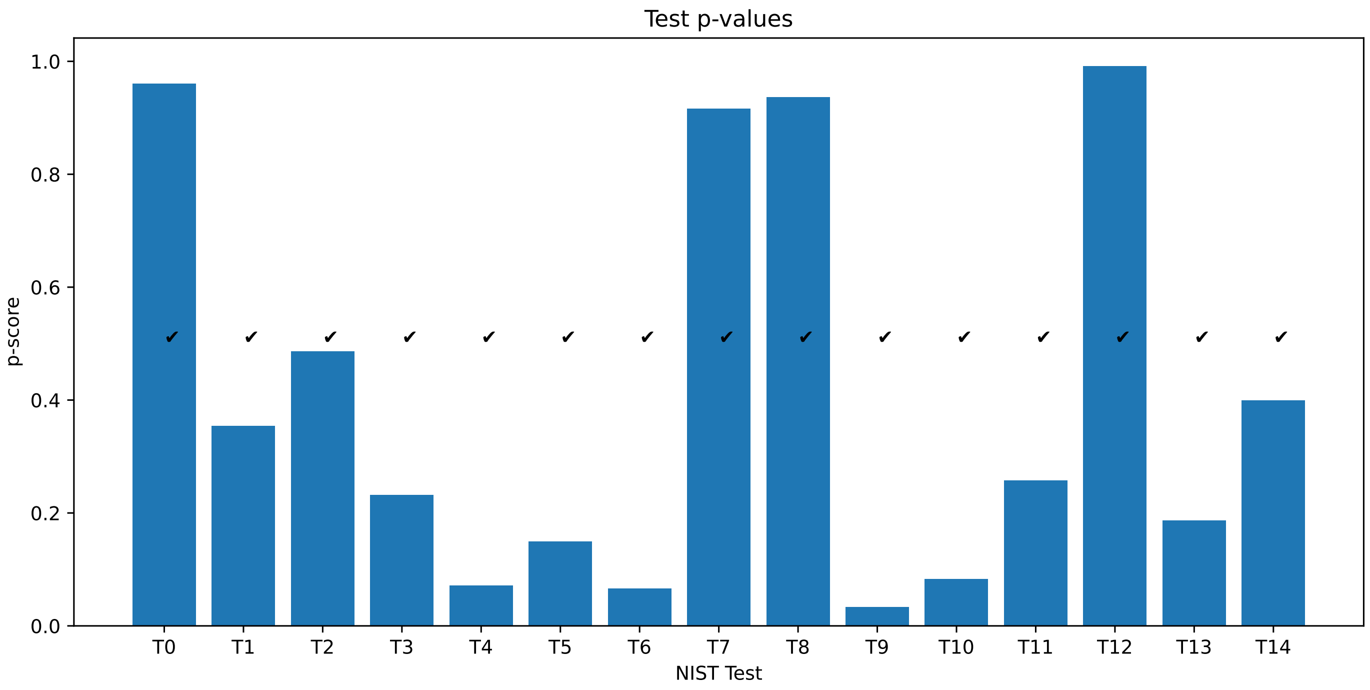


(a)

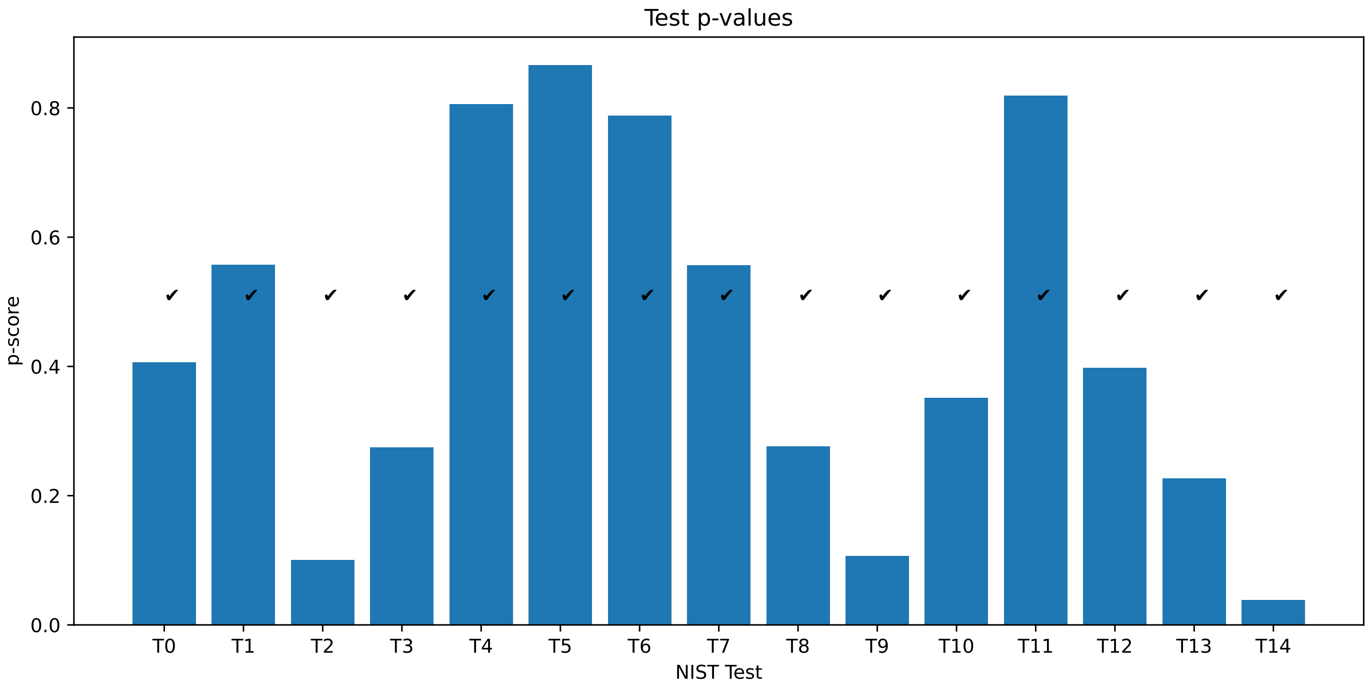


(b)

Fig. 4: NIST test results a) 1M and b) 10M random bits generated using numpy random library



(a)



(b)

Fig. 5: NIST test results a) 1M and b) 10M random bits generated using Python's built-in os.urandom function

VI. Conclusion

In this lab, the randomness of a RO RNG, a GARO RNG, NumPy's RNG implementation and Python's RNG implementation were evaluated for 1M and 10M generated random bits using the NIST test suite. A CMOD S7 FPGA was used to generate 1M and 10M random bits for the RO LFSR RNG and the Galois based GARO RNG. Similarly, 1M and 10M random bits were generated using the built-in ”random” functions of the Numpy and OS python libraries. The results showed that the FPGA RNG implementations were outperformed by the Python and NumPy implementations. Probably due to the entropy sources for the FPGA RNGs not being entropic enough.

Additionally, through this lab we learned how to give bytes as inputs, read bytes as outputs and write these outputs as bin files. Additionally, we learned how to generate random bits using NumPy and Python. For further work, we'd like to learn how to add additional entropy to the FPGA/improve the RO RNG's and GARO RNG's performance on the NIST tests.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, “Experiment 3: Pseudo- and True Randomness,” in *Practical Hardware Security Course Manual*. University of South Florida, Aug 2022.

[2] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” Booz-allen and hamilton inc mclean va, Tech. Rep., 2001.