NOVEL MEMRISTOR BASED TRUE RANDOM NUMBER GENERATOR

by

Scott Stoller

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

December 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Scott Stoller

Thesis Title: Novel Memristor Based True Random Number Generator

Date of Final Oral Examination: 09 October 2020

The following individuals read and discussed the thesis submitted by Scott Stoller, and they evaluated their presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Kristy A. Campbell, Ph.D.	Chair, Supervisory Committee
Elisa H. Barney, Ph.D.	Member, Supervisory Committee
Kurtis Cantley, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Kristy A. Campbell, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

DEDICATION

I would like to dedicate this work to my wife Gabrielle. I want to thank her for her patient encouragement and support throughout the course of my research. She was always willing to listen, talk through issues with me, and for patiently allowing me to process my thoughts out loud to her.

ACKNOWLEDGMENTS

I would like to thank Dr. Campbell for her encouragement, guidance, support, and patience throughout the course of this research. I would also like to thank Dr. Barney and Dr. Cantley for their support and willingness to be a part of my supervisory committee.

Special thanks to Alex Nugent from Knowm for his support, knowledge, and recommendation of memristor devices to use for this thesis project as well as providing devices to use for the project.

Finally, I would also like to thank my employer, Micron Technology, for providing the flexibility and support that allowed me to pursue my M.S. while working full time, as well as the financial support they provided in the form of tuition reimbursement.

ABSTRACT

Random numbers are an important, but often overlooked part of the modern computing environment. They are used everywhere around us for a variety of purposes, from simple decision making in video games such as a coin toss, to securing financial transactions and encrypting confidential communications. They are even useful for gambling and the lottery.

Random numbers are generated in many ways. Pseudo random number generators (PRNGs) generate numbers based on a formula. True random number generators (TRNGs) capture entropy from the environment to generate randomness. As our society and our devices become more connected in the digital world, it is important to develop new ways to generate truly random numbers in order to secure communications and connected devices.

In this work a novel memristor-based True Random Number Generator is designed and a physical implementation is fabricated and tested using a W-based self-directed channel (SDC) memristor. The circuit was initially designed and prototyped on a breadboard. A custom Printed Circuit Board (PCB) was fabricated for the final circuit design and testing of the novel memristor-based TRNG. The National Institute of Standards and Technology (NIST) Statistical Test Suite (STS) was used to check the output of the TRNG for randomness. The TRNG was demonstrated to pass 13 statistical tests out of the 15 in the STS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AD2	Analog Discovery 2
BSU	Boise State University
CSV	Comma-Separated Value
DFT	Discrete Fourier Transform
DRC	Design Rule Check
GCC	Gnu C++ compiler
LCG	Linear Congruential Generator
LFSR	Linear Feedback Shift Register
NIST	National Institute of Standards and Technology
PRNG	Pseudo-Random Number Generator
RNG	Random Number Generator
RTN	Random Telegraph Noise
SDC	Self Directed Channel
STS	Statistical Test Suite
TRNG	True Random Number Generator

CHAPTER ONE: INTRODUCTION

Random numbers have a variety of uses in modern computing and information security. Uses can range from a simple decision making or branching in a video game to providing the basis for keeping banking transactions secure and encryption of secure and secret documents.

1.1 Methods of Generating Random Numbers

While there are many methods to generate random numbers, there are two main types of random number generators: Pseudo Random Number Generators (PRNGs) which generate sequence of numbers based on a formula, and True Random Number Generators (TNRGs) which generate numbers from entropy and cannot be predicted.

1.1.1 Pseudo Random Number Generators

Pseudo Random Numbers Generators generate a deterministic sequence of numbers that are nearly random but are not truly random. PRNGs begin with a seed value and generate a sequence of random numbers based on a formula. A PRNG will generate the exact same sequence of random numbers if it starts with the same seed in each run. PRNGs will eventually repeat and follow the same sequence again. In this sense, they generate a string of numbers that are difficult to predict and have the appearance of being random yet are not truly random. The output from PRNGs however can be predicted. This can be useful for many applications, specifically software applications where a repetitive but “random” behavior is useful, such as Monte Carlo simulations and low-stakes games. This behavior is not desirable for other applications, such as security applications, that demand that a random number must be truly unpredictable, not merely difficult to predict. PRNGs can be implemented in a variety of ways in either hardware or software, but the most common method is to use a liner congruential generator (LCG) or a linear feedback shift register (LFSR).

1.1.2 True Random Number Generators

A True Random Number Generator generates numbers that are truly random in nature. Unlike a PRNG, there is no way to predict the sequence of numbers that a true RNG will generate. The same sequence of random numbers will never be generated twice by a TRNG. True random number generation is important in data security applications, such as securing banking transactions, or encrypting communications where it is essential that the sequence of random numbers can never be predicted or guessed. True RNGs generally utilize a hardware component that captures entropy to generate truly unpredictable results. Examples of true hardware RNGs could be a fair 6-sided die, a lava lamp [1], or thermal or quantum noise (shot noise, random telegraph noise, etc.) in a circuit component.

Capturing entropy and generating true random numbers is a very difficult process. Many TRNGs require a significant amount of post-processing of the data they generate in order to output sequences of numbers that are truly random in nature. For example, certain natural sources of entropy such as Random Telegraph Noise (RTN) in NAND Flash Memories require post-processing to capture the randomness of the output [2]. Other TRNGs require post-processing as well, such as Intel’s RDRAND [3] and the Linux TRNG Entropy Pool [4], both of which will be discussed later in this thesis. It is challenging to capture entropy at high speed and many TRNGs capture entropy slowly. In many cases a truly random number is generated from an entropy source and used to seed a PRNG in order to improve the speed and efficiency of the RNG. The PRNG seed is normally updated at a regular interval to ensure unpredictability in the sequence.

1.1.3 Capturing Entropy in a Memristor

A memristor is a two-terminal device that changes resistance as charge flows through the device. The memristor is often referred to as the 4th circuit element [5-8]. Resistors can often be used in TRNGs to capture entropy as thermal noise. In many cases, a memristor can be better suited for use in a TRNG because other sources of noise besides just thermal noise are present in the device. In this thesis entropy is captured using a Tungsten Self Directed Channel (W-SDC) device [9,10].

1.2 Topics Covered in this Thesis

This thesis introduces the concept of random numbers and randomness in Chapter 1. In Chapter 2, the National Institute of Standards and Technology (NIST) methods for testing random numbers are introduced and discussed. Chapter 3 describes the data collection hardware and methodology. Chapter 4 describes six PRNGs and TRNGs that have been previously presented in the literature by other researchers. These designs are examined in this thesis through breadboard testing. The results of these tests were obtained using the procedures and hardware described in Chapters 2 and 3. A novel memristor-based TRNG designed for this thesis, including steps to fabricate and test the proposed design are described in Chapter 5. Tests were first performed at the breadboard level, and upon demonstration of proof of concept, were then performed at the printed circuit board level. Results of the breadboard and PCB TRNG tests are presented in Chapter 6 and compared with the other PRNG and TRNG designs tested in this work.

Chapter 7 contains a discussion of future work, shortcomings of, and improvements to the design. The Appendix contains additional information such as the code that was written and use for data collection and analysis.

CHAPTER TWO: OVERVIEW OF NIST STATISTICAL TEST SUITE

2.1 NIST Statistical Test Suite

NIST (the National Institute of Standards and Technology) has made software available to test the randomness of random number generators [9,10]. The Statistical Test Suite (STS) version 2.1.2 was downloaded, compiled, and used to analyze binary bitstreams of numbers generated by RNGs. The NIST statistical test suite contains a series of 15 statistical tests to determine the randomness of a series of generated numbers. Some are as simple as verifying that the number of ones and zeros is an even 50/50 split. Other tests look for repeating patterns or continuous runs of the same bit within the sequence. In the paragraphs that follow, a complete list and description of each test is included. The following summary of these tests was written after careful reading and understanding of the NIST STS guide.

2.2 Description of NIST Tests

2.2.1 Frequency (Monobits) Test

A truly random number should have a 50/50 distribution of ones and zeros. The frequency test analyzes the number of ones and zeros in the test sequence. The test will fail if there are significantly more ones than zeros, or significantly more zeros than ones. The number of ones and zeros should be approximately equal. None of the other NIST randomness tests are valid if this test does not pass. A case of all zeros or all ones in a sequence is not random and would fail the frequency test.

2.2.2 Frequency Test Within a Block

This test is equivalent to the frequency test described above. This test simply looks at the frequencies of ones and zeros in an M-bit sample block. For a truly random number, the number of ones and zeros should be approximately equal. It may be possible for a series of 100 zeros followed by 100 ones to pass the frequency test, but the frequency test within a block would catch and fail this case.

2.2.3 Runs Test

The purpose of the runs test is to evaluate the total number of runs in the sequence of random numbers. A run consists of a sequence of repeating bits, bounded on each end by an opposite bit. The length of the run is the number of bits that are identical in the run. For example, the sequence “100001” is a run of zeros of length 4. This test checks if the number of runs of various lengths matches the expectation for a random sequence. Specifically, this test can find oscillations in the random sequence. A sequence of four zeros followed by four ones repeating over and over might pass both frequency tests above, but would fail the runs test because there are no runs of length one, two, or three.

2.2.4 Longest Runs Test

The longest runs test finds the longest run of ones in each block and tabulates the frequencies into categories. Each category is the length of the run. Analysis of the frequency of the occurrence of longest runs in a block can predict if a number is random. Runs of ones that are longer than expected indicate a non-random number generator. Runs of zeros are not checked in this test. This test operates under the assumption that runs of zeros are similar to runs of ones.

2.2.5 Binary Matrix Rank Test

The purpose of this test is to look for linear dependence among the fixed length substrings of the original sequence. The binary sequence test calculates the rank of a matrix formed by arranging the input sequence in the rows and columns. The test fails if a linear dependence is detected among the fixed length substrings of the original sequence. This sequence is also included in other randomness test software.

2.2.6 Discrete Fourier Transform (Spectral) Test

The Discrete Fourier Transform (DFT) Spectral test analyzes the random sequence and looks for periodic patterns in the sequence. This test will detect if repetitive patterns are near each other. A perfectly random sequence should have a spectral analysis that is flat (should look like noise). The test will fail if there are peaks in the spectrum that exceed the 95% threshold. The binary representation of a sine wave will fail the DFT tests due to a single large spectral peak.

2.2.7 Non-overlapping Template Matching Test

The purpose of this test is to analyze the rate of occurrence of pre-specified target lists of numbers. This test searches for matches between the binary templates provided (located in the “templates” folder) and the random input sequence. If a match is not found the sequence is shifted by 1 bit and a search for the next sequence begins from that sequence. If a sequence is found, then the search for the next sequence starts from the end of the current sequence. The test fails if too many occurrences of the same pattern are found in the sequence. For example, the test sequence mentioned above in the frequency block test of four zeros followed by four ones repeating would fail the non-overlapping template matching test.

2.2.8 Overlapping Template Matching Test

The overlapping template match sequence is very similar to the non-overlapping template matching test described above. The main difference from the non-overlapping test is that the overlapping template test will scan for new sequences starting at an increment of one from the start of the last sequence instead of starting at the end of the previous sequence. The test fails if too many occurrences of the same pattern are found in the sequence.

2.2.9 Maurer's "Universal Statistical" Test

The purpose of this test is to search for repetition in the input sequence. The first portion of the sequence is turned into a set of patterns. The rest of the sequence is analyzed for repetition of these patterns. The purpose of this test is to check if the sequence is easily compressible without loss of information. If there are too many occurrences of these patterns in the rest of the sequence then the sequence is non-random. This test is similar to the overlapping and non-overlapping template tests.

2.2.10 Linear Complexity Test

The purpose of this test is to compute the length of LFSR required to generate the input pattern. The Berlekamp-Massey algorithm is used to determine the minimum length LFSR needed to generate a pattern for each block in the sequence. A pattern with short length LFSRs, or a pattern that lacks linear complexity is considered non-random. In many cases, PRNGs cannot be detected by this test because most modern PRNGs have an extremely long period. The well-known Mersenne Twister has a period of 2^{19937}-1.

2.2.11 Serial Test

The serial test analyzes the frequency of occurrence of all possible overlapping patterns in the input sequence. A random pattern should have a similar occurrence rate of all other patterns if it is random. The input is considered non-random if some patterns have a higher than expected rate of occurrence. Note that for a pattern length of 1, the serial test is the same as the frequency test,

2.2.12 Approximate Entropy Test

This test focuses on the frequency of all possible overlapping patterns across the entire sequence. This test compares the frequency of occurrence of m-bit patterns with m+1-bit patterns in the sequence. The test will fail if the frequency of overlapping blocks is not what is expected for a random sequence.

2.2.13 Cumulative Sums Test

The cumulative sums test analyzes the cumulative sum of the sequence to test that the sequence never deviates too far from the expected value. Ones add one to the cumulative sum and zeros subtract one from the cumulative sum. If the sequence is random, the cumulative sum should never deviate too far from zero. The cumulative sums test will fail if the excursion from zero is too large or too small. From the previous example of a sequence of 100 ones followed by 100 zeros, the cumulative sum will reach a value of 100 steps away from zero which is not probable to occur in a series of random numbers of length 200 that is passing the frequency tests.

2.2.14 Random Excursions Test

The random excursions test is a different take on the cumulative sums test. Like the cumulative sums test, a one adds one to the cumulative sum and a zero take one away from the cumulative sum. Sequences are selected such that they start and end at zero. An analysis of the number of visits to a particular state (i.e. -4, -3, -2, -1, +1, +2, +3, or +4) in the sequences is performed. If the number of visits to a particular state does not match that expected of a random sequence, the sequence is not random. From the previous example of a series of four ones followed by four zeros repeated, the cumulative sums test will consistently reach a cumulative sum of +4 and 0 with the same rate of occurrence. For a truly random sequence a cumulative sum of +4 should occur less often than a cumulative sum of 0.

2.2.15 Random Excursions Variant Test

Just like the cumulative sums and random excursions test, the random excursions variant test analyzes the total number of times each state is visited. The states in this test range from -9 to +9. If the number of visits to each state deviates from that expected of a random sequence, the test fails.

2.3 Downloading, Compiling, and Using the NIST Statistical Test Suite

The NIST Statistical Test Suite can be downloaded from the NIST website [12]. Once the source code is downloaded the application must be compiled. For this thesis project Ubuntu 18.04 LTS Sever was installed in a virtual machine using VirtualBox [13] running as a guest OS on a Windows 7 PC. This allowed for the Waveforms application to run on the Windows environment and STS 2.1.2 code to run in a Linux environment. A shared drive that could be accessed from both the host Windows OS and the guest Linux OS was used to share data between the two operating systems. Instructions for sharing a drive between a guest OS and Windows 7 were found online [14].

To make the STS application a little more friendly and convenient to use, several modifications were made to the STS Assess application throughout the course of this project. Print statements to show the progress of the statistical analysis were added, as well as code to print the number of lines read in the case that there the input file does not contain a sufficient amount of data. Due to the simplicity and small amount of modification this code is not shared in the appendix. The modifications to the source code are contained in the appendix section of this thesis.

2.3.1 NIST Statistical Test Suite Code Location

Campbell Research group folder at BSU or Scholarworks:

\sstoller_thesis_final_writeup\STS\sts-2.1.2

2.3.2 Compiling the NIST Statistical Test Suite Application

The following instructions detail how to install and compile the STS application on a Ubuntu Linux machine. Instructions for other Linux machines will be similar, but may require the use of a different package manager for installing `make` or `gcc`.

1. Install make (“`sudo apt-get install make`”)
    a. make is required to read the makefile and properly compile and link the application. Depending on the version of Linux used, make may already be installed.
2. Install gcc (“`sudo apt-get install gcc`”)
    a. `gcc` is the Gnu C++ compiler and must be installed in order to compile the application. Depending on the version of Linux used, `gcc` may already be installed.
3. Clean (“`make clean`”)

   a. Executing `make clean` will remove any previously compiled files and executables from the current directory. This is an optional step, but certainly a best practice when compiling source code to an executable.
4. Compile and link the executable (“`make`”)
   a. Executing `make` uses the makefile in the current directory to compile and link the source code to create an executable. The “assess” executable will be created when `make` runs.

<u>2.3.3 Testing Randomness with the STS Assess Application</u>

This section provides a quick overview of how to use the STS assess application once it has been compiled by following the steps in the previous section.

1. Execute assess (“`./assess 1000000`”)
   a. Assess is an executable application that was compiled from the STS source code. The number following the application is the length of each bitstream to be tested. A value of 1,000,000 (1M) bits was used for testing random numbers whenever possible.
2. The application prompts the user to select a Generator/Source
   a. To analyze an input file, select option 0
3. The application prompts the user to select the desired Statistical Tests to run
   a. To run all tests, enter a value of 1. Entering a value of 0 allows the user to select which tests to run or not run. For this thesis project all the statistical tests were applied.
4. Next the application prompts the user if any changes to the Parameters are desired.

   a. To keep all Parameter Adjustments at their default values, enter a value of 0. To modify parameters, select the parameter to adjust by entering a number for that test. For this thesis project none of the parameters were adjusted.

5. The application prompts the user for the number of bitstreams to test.
   a. For this test the user must divide the length of the binary file generated by the size of the bitstream passed as an input parameter to the assess application when the application was executed. In general, the more bitstreams that can be tester the better. 100 sequences of 1M bits was the target bitstream and test length for this thesis project. This was not always possible.

6. Finally, the application prompts the user for the input file format.
   a. Enter a value of 0 for a file in an ascii format (e.g. each bit is an ascii ‘1’ or ‘0’ read in a text editor. Enter a value of 1 for a file in binary format (the Perl scripts written to process the Waveforms output write a binary file).

7. Alternatively, the following command string may be used to input a series of predefined options: “`time echo "0 <path to binary file> 1 0 49 1" | ./assess 1000000`”
   a. The echo command and a pipe are used to input the options (in the order prompted by the application) into the assess application.
   b. The time command is used to record and print the execution time it takes the assess application to execute.

c. The ./ executes the assess command and the 1000000 following specifies the length of each bitstream input.

2.3.4 Understanding the Output from the NIST STS Application

The NIST statistical test suite generates a text document report in the “\experiments\AlgorithmTesting\” path titled “finalAnalysisReport.txt”. This report contains a report summary for all statistical tests. Three types of information are provided for each test in this report: columns C1 through C10 show the distribution of p-values for each test (each column represents a range or “bucket” of p-values). The P-value column is a summary of the p-values in columns C1 through C10 via application of a chi-square test. According to the NIST STS guide, the P-value is the “probability (under the null hypothesis of randomness) that the chosen test statistic will assume values that are equal to or worse than the observed test statistic value when considering the null hypothesis.” [11].

The proportion column is the pass rate of the sequences tested. A value of “96/100” indicates that 96 sequences of the 100 sequences tested are passing the specific test. An asterisk indicator is displayed next to any p-value or proportion that is failing for a specific test. The default alpha value of 0.96 was used for this thesis project. This means that in order to be considered a random sequence, 96% of sequences must be passing each test.

It is worth noting that some tests run multiple times (i.e. Cumulative Sums or Non-Overlapping Template tests). For these tests a single row is displayed in the results table showing the sum proportion for all of these tests. Only the summary P-value and proportion for each randomness test are shared in the results for each RNG tested in this thesis. An asterisk is shown next to each proportion or P-value that is not passing.

CHAPTER THREE: DATA COLLECTION USING THE DIGILENT ANALOG DISCOVERY 2 BOARD

3.1 Introduction

This section describes the methods, hardware, and software used to test and characterize the hardware TRNGs that were implemented. A brief description of the Analog discovery 2 board is provided. Design of the data collection circuit that enables faster data collection is reviewed. Software and scripts that aided in data collection are shared and a workflow for collecting and characterizing the random data is presented.

3.2 Digilent Analog Discovery 2 Board

The Digilent Analog Discovery 2 (AD2) board was used to collect all of the random data samples from the hardware random number generators tested. The AD2 interfaces with the Digilent Waveforms software that runs on either a Windows or Linux PC (in this case a PC running Windows 7 was used). The AD2 board was chosen because it is a small, cost-effective, and versatile board that is easy and convenient to use. The AD2 has two power supply outputs, two function generator outputs, two oscilloscope inputs, and 16 digital I/Os. Figure 3.1 shows a block diagram of the TRNG, Digilent AD2, and PC Waveforms software interface. Figure 3.2 shows a pinout diagram of the AD2, from the Digilent AD2 documentation [15].

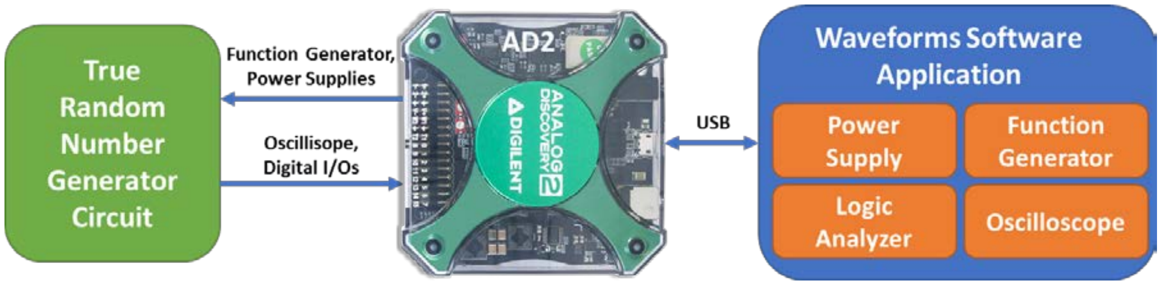

True Random Number Generator Circuit
Function Generator, Power Supplies
Oscillisope, Digital I/Os
AD2
USB
Waveforms Software Application
Power Supply
Function Generator
Logic Analyzer
Oscilloscope

Figure 3.1 TRNG, Digilent AD2, and PC Waveforms block diagram

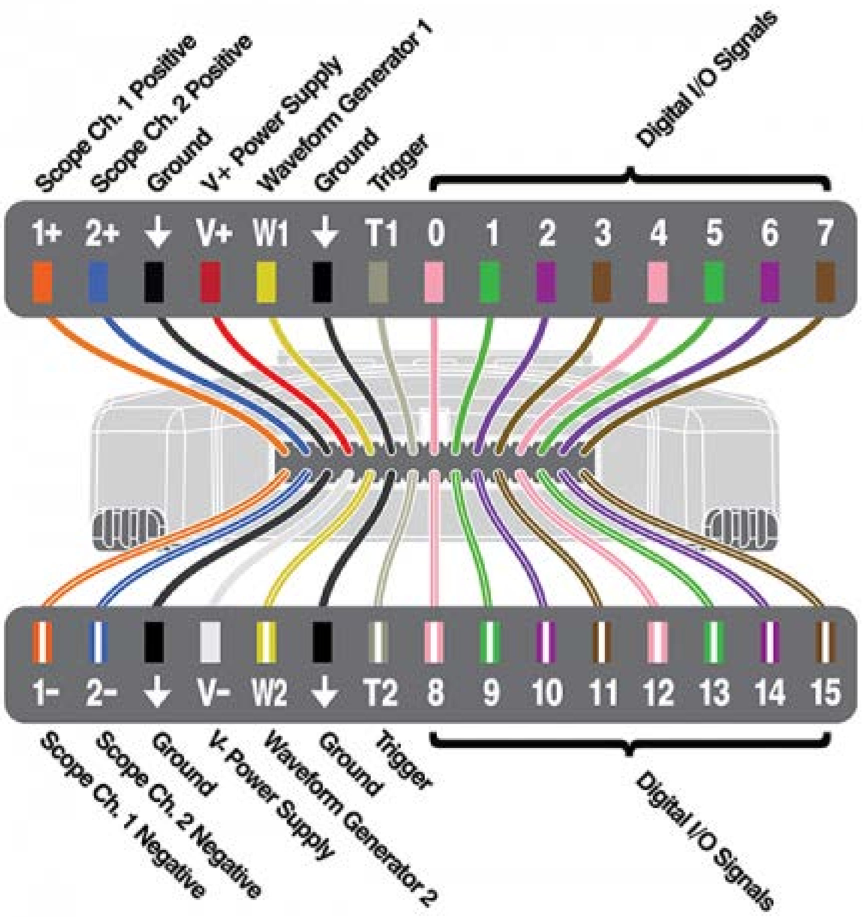


Figure 3.2 Pinout diagram of the Digilent AD2 [15]

3.3 Design of Data Collection Circuit

In order to maximize the efficiency of data collection on the AD2, a data collection circuit was designed and built that was able to sample and collect 8 bits in a single clock cycle on the AD2. This allows for 8 times more data to be collected in each capture with the Digilent Waveforms software [16] by reading data in parallel using 8 bits instead of serially one-by-one. The x8 data collection circuit consists of a binary counter and a serial-in-parallel-out shift register. The slow oscillator is input to a binary counter. The fast oscillator is input into a shift register. The 3rd bit from the binary counter circuit is connected to the AD2 DIO 15 to act as the output clock. The 3rd bit toggles every 8 cycles of the input slow clock. The Digilent AD2 samples the output of the shift register every 8th cycle of the slow clock. The parallel outputs from the shift register are connected to DIOs 0-7 on the AD2. For every 8 cycles of the slow clock, the binary counter toggles a master clock for the AD2 to sample DIOs 0-7. As a result, the sample frequency of the AD2 can be reduced by a factor of 8, allowing it to save more data. Figure 3.3 shows a simplified schematic of the data collection circuit. There are some ways that the data collection circuit is not ideal. The design and implementation of the data collection circuit are described in depth in Chapter 5. Suggestions for improving the data collection circuit design are discussed in Chapter 6 where future work is discussed.

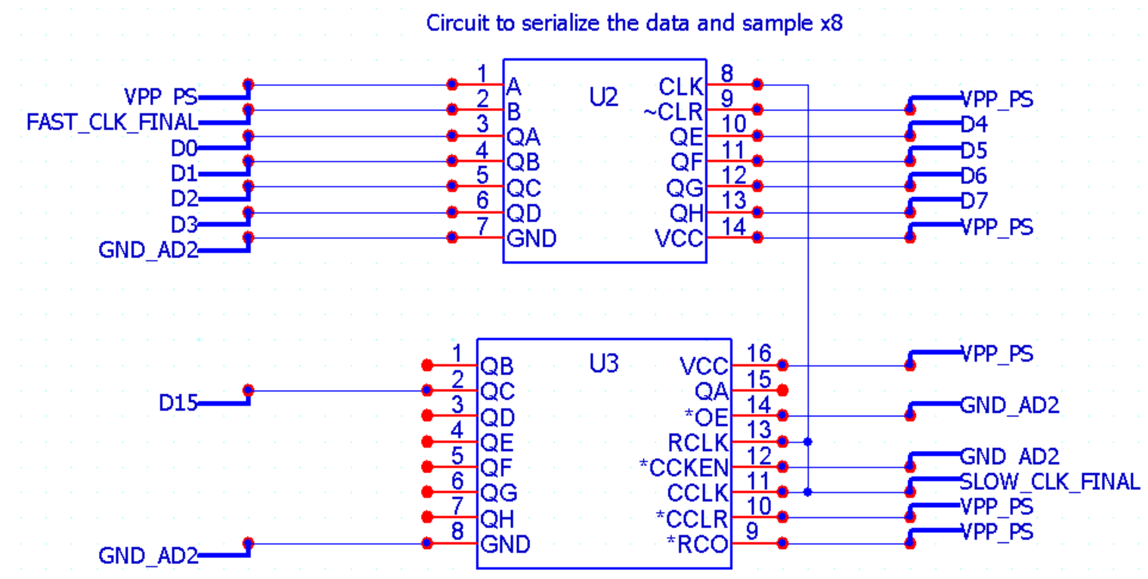


Figure 3.3 Schematic of the data collection circuit

The SLOW_CLK_FINAL net is the output from the slow memristor oscillator circuit. This signal is connected to both Binary Counter (U3) and the Shift Register (U2). The 3rd bit from the binary counter is connected to D15 (DIO 15) on the AD2 board. The FAST_CLK_FINAL net is the clock generated by the fast resistor oscillator circuit. This is the input to the shift register. Each time the slow clock transitions from low to high, the state of the fast oscillator is sampled and shifted in the shift register. The 8 parallel outputs of the shift register are connected to the inputs D0-D7 (DIO 0- DIO 7) on the AD2 board.

Figure 3.4 shows the shifting output behavior of the shift register on the DIO 0 through DIO 7 signals. In this case the sample rate on the AD2 is much faster than the slow clock. RV Out (Resistor Multivibrator, or fast oscillator output) can be seen as a very fast oscillator. Both MV 1 Out (Memristor Multivibrator 1) and MV 2 Out (Memristor Multivibrator 2) outputs can be seen operating at a slower speed. The Clock (input clock to the Digilent AD2) can be seen sampling the data on the Data 15Clk data bus. This output is saved to a CSV file for data processing and conversion to a binary bitstream later.

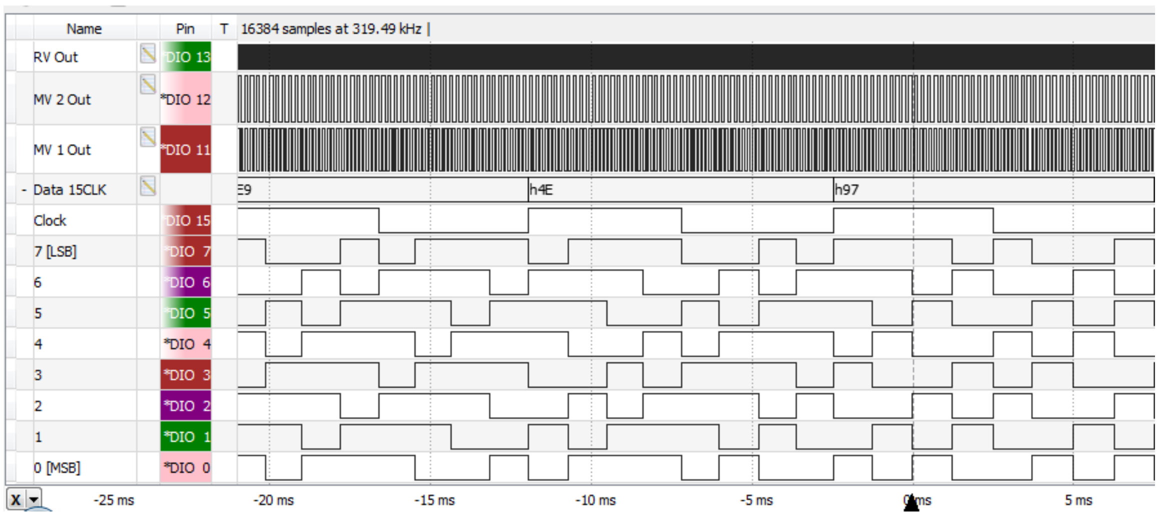


Figure 3.4 Sample output from the data collection circuit

The primary advantage of using the shift register is that 8 bits of data can be collected in one sample. The Digilent AD2 logic analyzer capture is limited to 100M samples. Using an 8 bit bus to capture output allows for up to 800M bits to be captured in a single run. The downside of implementing the circuit is additional components, complexity, and power.

3.4 Importance of Proper Sampling Rate of Data and Other Considerations

When sampling the output of the random number generator it is important to consider the frequency of the slow clock and the sample rate of the logic analyzer used to capture the data. In order to maximize the number of data samples taken at once, a slow sample rate is desirable. However, the sampling rate must be fast enough to ensure that the data is all properly captured. The Nyquist sampling theorem states that in order to properly sample a signal, the sample frequency must exceed the signal frequency by a factor of 2 [17]. With the data collection circuit this means that a sample rate twice that of the output of the binary counter can be used. This rate is 1/8th the frequency of the slow oscillator. The Nyquist sampling theorem states that the sample rate must be twice the frequency, resulting in a minimum sample frequency of 1/4th the frequency of the slow oscillator clock,

$$\begin{aligned} Minimum\ Sample\ Frequency &= \frac{Frequency\ Slow\ Oscillator}{8} * 2 \\ &= \frac{Frequency\ Slow\ Oscillator}{4}. \end{aligned} \tag{3.1}$$

The minimum sample rate is equal to twice the frequency of the input signal into DIO 15. The frequency of DIO 15 is the frequency of the slow oscillator divided by 8.

3.5 Debiasing the Output of the TRNG

In order to debias (or whiten) the output of the TRNG, a simple whitening strategy proposed by Jon von Neumann was used. Whitening is required when there is bias in the output of the TRNG, for example when there are a disproportionate number of 0’s and 1’s in the output. Many TRNGs require debiasing in order to pass even just the frequency test. Fine-tuning of the output voltage swing of the fast oscillator and the reference voltage of the data collection circuit could be done to ensure proper biasing of the TRNG, but achieving a level of debiasing that is very close to 50/50 is extremely difficult. In addition, it was found that the circuit can quickly drift with time, temperature, and other factors. For this reason, Jon von Neumann’s whitening algorithm was applied to the circuit [18].

Von Neumann whitening was implemented in software after the data collection was performed, but it can also be easily implemented in hardware with basic logic gates. An XOR gate can be used to determine whether two samples are the same value of different values. Based on this output both samples can be thrown out or the first sample can be kept. The debiasing algorithm does not change the random properties of the output stream. In many TRNG implementations (i.e. those that rely on long delays between asynchronous events, or circuits that may have a built-in DC bias) it is mandatory to apply a whitening algorithm. One example of this is seen when generating random numbers based on the output for a Geiger counter where there is a very long time delay between events [19]. It should be noted that the debiasing algorithm is simple and efficient to implement and does not change the entropy characteristics of the sequence.

Jon von Neumann's whitening algorithm is very simple. It searches for the transition from a 0 to a 1 or a 1 to a 0 in the string of numbers. Any time two bits in a sequence are the same, there is no output. If the sequence transitions from a 0 to a 1 then the output is simply a 0. If the sequence transitions from a 1 to a 0 then the output is a 1. This reduces the length of the random number sequence by a factor of about four for a sequence that does not have a significant bias. The more the sequence is biased the more the length of the sequence is reduced. The Perl script "csv_to_datastream_von_neumann.pl" applies the von Neumann whitening algorithm to the sequence of bits when it converts the data output from the Waveforms CSV file to binary file for the Statistical Test Suite. Table 3.1 shows the method to debias output samples and return a nearly perfect 50/50 distribution of bits in a truly random stream of bits regardless of any bias the input datastream may have. The source code can be found in Appendix A.

Table 3.1 Von Neumann Whitening Scheme

Even Bit Value	**Odd Bit Value**	**Result**
0	1	0
1	0	1
0	0	Sample is ignored
1	1	Sample is ignored

It was observed in testing of all TRNGs in this thesis (Chapters 4 and 6) that almost all sequences still had a strong bias after being processed using von Neumann's whitening algorithm. This is often the result of a sequence where a bit may have a strong correlation to the previous bit. In this case the sequence is by definition not random.

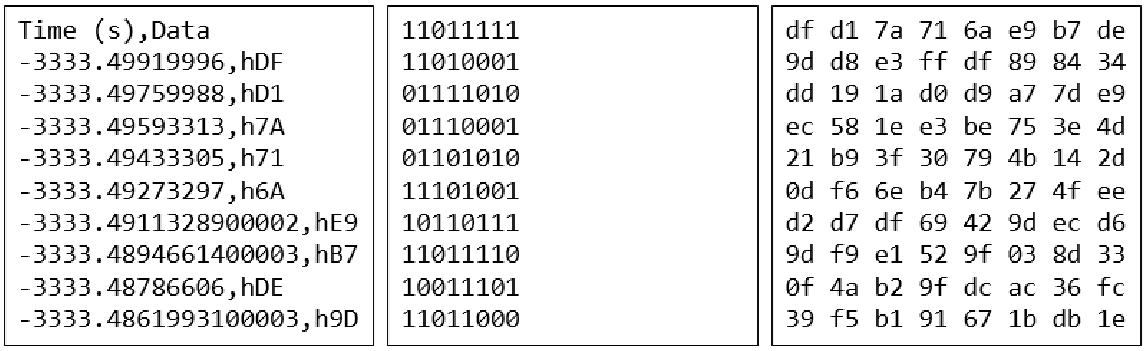
The downside of using this algorithm is that many samples may be lost in the debiasing effort. The more biased the output of the circuit is, the more often samples are thrown out or ignored. In many cases this is a necessary post-processing step, as all other NIST statistical tests are invalid if the frequency (monobits) test is not passing. There are other potential methods to debias the circuit, such as a feedback loop to adjust the reference voltage of the fast oscillator or using a method to convert the output of the fast oscillator multivibrator to a square wave to minimize bias etc. The downside of using any type of feedback loop is finding the fine balance between the circuit converging on a DC voltage that has no bias and the potential of such a circuit to oscillate. Oscillations could lead to non-random output in the TRNG.

3.6 Software and Scripts to Aid in Data Collection

The Waveforms software can save the collected data output as a text Comma-Separated Value (CSV) file format. In order to use the Assess tool in the NIST

Statistical Test Suite to analyze the output file it is necessary to convert the file to a format that can be used by the tool. The STS tool can only read in files that are in a text format that contains only ‘1’ or ‘0’ characters, or a binary format. Example output of the Digilent Waveforms software is shown in figure 3.4, along with two sample outputs that can be read by the STS software. Note that the text input (B) does not require newlines every 8 characters, or at all. The output in (C) is the hexdump output of the binary file that converts the raw binary data to a human-readable hex format. For this thesis project, only binary files were processed due to the smaller file size.

Figure 3.5 shows the output from the Waveforms logic capture (A) and sample text input (B) and binary input viewed with the Linux hexdump tool (C) that may be analyzed by the STS assess application.



```
Time (s),Data
-3333.49919996,hDF
-3333.49759988,hD1
-3333.49593313,h7A
-3333.49433305,h71
-3333.49273297,h6A
-3333.4911328900002,hE9
-3333.4894661400003,hB7
-3333.48786606,hDE
-3333.4861993100003,h9D
```

```
11011111
11010001
01111010
01110001
01101010
11101001
10110111
11011110
10011101
11011000
```

```
df d1 7a 71 6a e9 b7 de
9d d8 e3 ff df 89 84 34
dd 19 1a d0 d9 a7 7d e9
ec 58 1e e3 be 75 3e 4d
21 b9 3f 30 79 4b 14 2d
0d f6 6e b4 7b 27 4f ee
d2 d7 df 69 42 9d ec d6
9d f9 e1 52 9f 03 8d 33
0f 4a b2 9f dc ac 36 fc
39 f5 b1 91 67 1b db 1e
```

(A) CSV (B) Binary Text (C) Binary (Hexdump)

Figure 3.5 Examples of different types of random data formats

Several Perl scripts were written to aid in data post-processing. The primary purpose of the Perl scripts is to convert the text (CSV) output from the Waveforms application to a binary file that the STS Assess application can read in and test for randomness. Perl was selected because it is an easy-to-learn and cross-platform scripting language. Perl excels at parsing text files and it can read and write binary files with relative ease. Perl is installed by default on most Unix/Linux systems. The author was also very practiced in writing Perl code. Perl's execution time is slow compared to a compiled language like C or C++, but the scripts written and used for this project are simple and are generally able to parse the CSV files to binary outputs more quickly than the STS Assess application can test the binary sequence for randomness.

The collection of Perl scripts used for data collection, debiasing, and analysis can be found in appendix A. Source code and example usage for all scripts can be found in appendix A.

All of the data processing and analysis scripts follow the same general usage. All input arguments are positional arguments. Scripts converting a CSV text file to binary generally take a list of input files (separated by a space). One additional file name for the output file must be provided, and the file must use the .bin extension. Some of the scripts also have an additional switch to enable whitening by exclusive oring output data with the RANDU PRNG. A final argument of '1' will enable this function, or a '0' will disable.

Scripts converting a binary stream to another binary stream generally only take a single input file, although some may allow multiple file inputs to be read. Input files should be binary. A single output file must also be specified, also of .bin extension.

There are also several other scripts that are simply used to generate a random output string of bits. These scripts generally do not have any input options. Output file name or length of file generated are determined by variables that are instantiated within the script.

3.7 Workflow for Collecting and Analyzing Data

The following workflow was used for collecting, processing, and analyzing data output from the TRNG. Figure 3.6 shows a block diagram of the workflow.

1. Digilent AD2 and Waveforms software was used to collect the output from the TRNG. The output was collected using the logic analyzer window and saved to an output file (in CSV format) using the logging window. If needed, multiple runs were collected depending on sample rate and amount of data collected with the AD2 software. The text CSV files were saved to a hard disk location that was accessible by both the Windows PC and Linux VM guest OS.
2. The "csv_to_datastream_von_neumann.pl" Perl script was executed on the Linux VM guest OS to convert the text CSV data file(s) to a binary output stream. In this project, unless otherwise specified, output data was never XOR'd with the RANDU PRNG output.
3. The STS Assess tool was used to analyze the randomness of the output. In general, the bitstream size used was 1 Million. The number of bitstreams tested depended on the number of bits in the binary bitstream file. In general, it is a good rule of thumb to test at least 100 sequences. Final results for random number generators were tested once 100M bits had been collected unless otherwise noted.

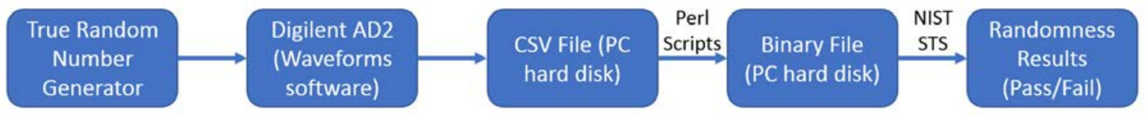
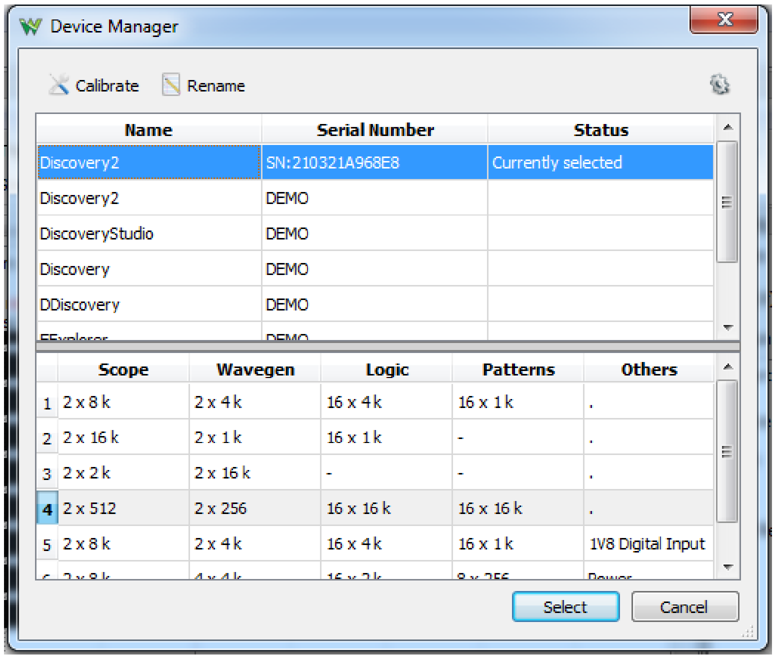

True Random Number Generator
Digilent AD2 (Waveforms software)
CSV File (PC hard disk)
Perl Scripts
Binary File (PC hard disk)
NIST STS
Randomness Results (Pass/Fail)

Figure 3.6 Entire data collection workflow and process

3.7.1 Tips, Tricks, and Problems Found Using the AD2 for Data Collection

It is best to use the proper configuration of the AD2 depending on the data being collected. The Device Manager (in the “Settings” dropdown menu) can be used to configure the AD2 device in different configurations. When recording data using the Logic Analyzer, configuration 4 should be used because it gives the largest amount of space to the Logic hardware. When using the oscilliscope, configuration 2 should be used because it allocates the largest amount of space to the Scope hardware.

Figure 3.7 shows a screenshot from the Waveforms software. There are several different hardware configurations for the AD2, each optimized for different functions.



Name	Serial Number	Status
Discovery2	SN:210321A968E8	Currently selected
Discovery2	DEMO	
DiscoveryStudio	DEMO	
Discovery	DEMO	
DDiscovery	DEMO	
EExplorer	DEMO	

	Scope	Wavegen	Logic	Patterns	Others
1	2 x 8 k	2 x 4 k	16 x 4 k	16 x 1 k	.
2	2 x 16 k	2 x 1 k	16 x 1 k	-	.
3	2 x 2 k	2 x 16 k	-	-	.
4	2 x 512	2 x 256	16 x 16 k	16 x 16 k	.
5	2 x 8 k	2 x 4 k	16 x 4 k	16 x 1 k	1V8 Digital Input
6	2 x 8 k	4 x 4 k	16 x 2 k	8 x 256	Power

Figure 3.7 Waveforms Device Manager options

The scope probes are not ideal voltage probes. It is prudent to keep this in mind when measuring circuits that have a high resistance node and/or a low drive strength. This can be worked around by using a high-impedance input op-amp as a unity gain buffer to probe the desired circuit node. The same is also true of the DIOs when used as inputs. Sampling analog signals with the DIOs can impact these circuits by pulling them towards a DC bias with a low (for this type of equipment) impedance.

The wavegen function does not have a very low output impedance. The wavegen are able to power simple circuits that do not draw significant current, but the power handling is limited. For the purposes of creating an additional power supply for the fast oscillator circuit the power handling of the wavegen was sufficient.

The AD2 may experience issues when running only on USB power, especially when powering a high-powered circuit or when the host PC has a USB port that does not supply enough power. An external power supply is highly recommended to be used with the AD2 for best results. The AD2 requires a 5V DC power supply with a 5.5x2.1mm adapter plug.

It is highly recommended to use a version of the waveforms software that is the same or newer than the beta version 3.13.1 used for this thesis project. This version of the software was found to be more stable and contained a crucial bugfix to prevent a bad current reading on the USB bus from shutting off the power supplies on the AD2 when an external power supply was powering the board. Several features, such as the system monitor function were also included in the beta version.

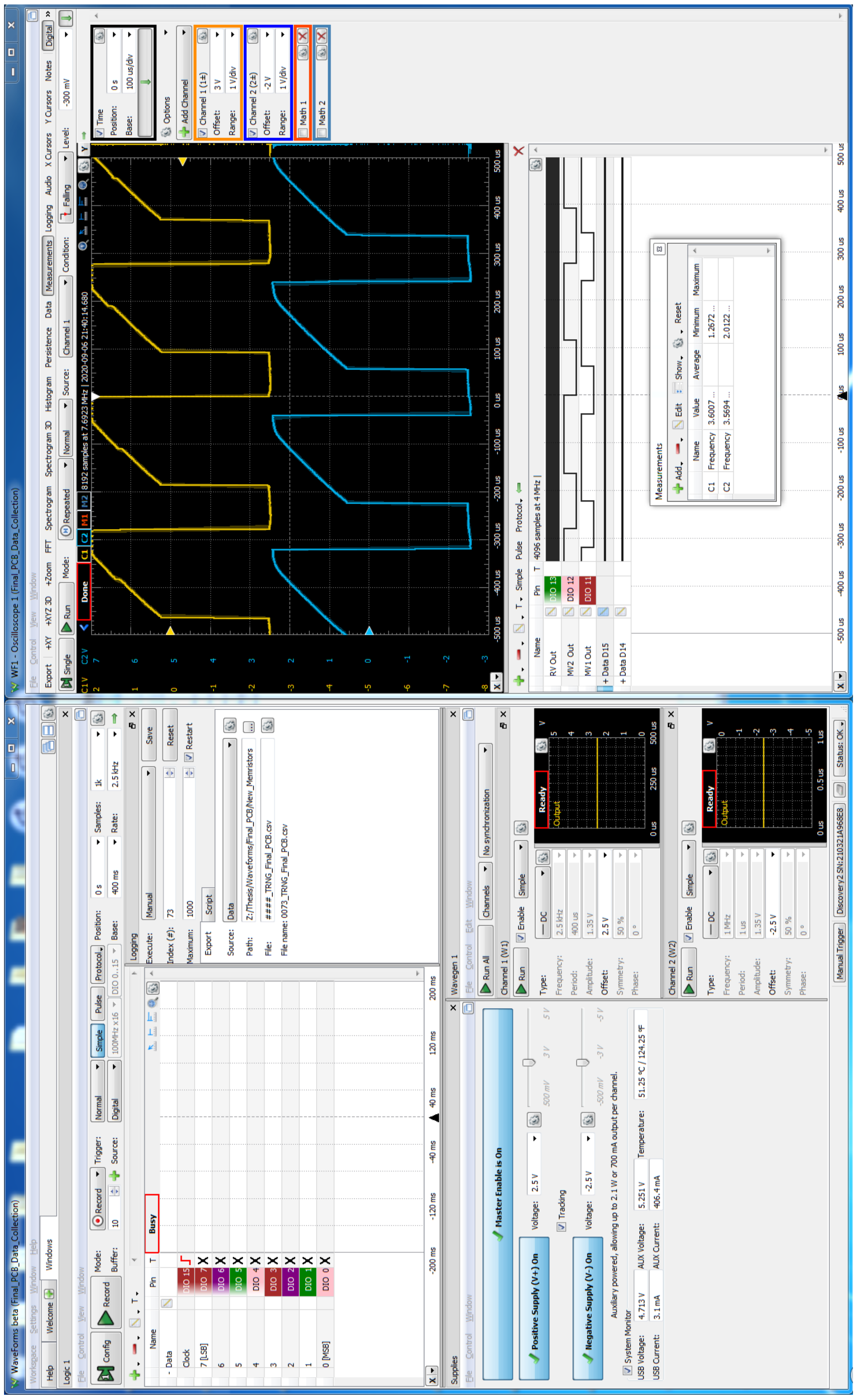
The Waveforms logic analyzer software can be configured to allow the user to collect and save up to 100 million samples of digital information in a single run. Up to 16 digital I/O signals can be saved at once. In the case of this thesis project, data was saved on an 8 bit bus. Sampling at the Nyquist frequency would allow one to collect as many as 50 million samples of the 8 bit bus, or 400 million total bits of output from an RNG per run.

The Logging window on the Waveforms software can be used to log the output of any signal desired to a CSV text file. Logging can be triggered automatically or manually after each acquisition.

The Supplies window of the Waveforms software also displays a system monitor for the AD2. Other relevant information such as the USB current and voltage, and aux voltage and current is displayed in this window.

The Logic Analyzer function of the Digilent AD2 is set to operate for 3.3V or 5V logic by default. However, it can be configured to operate for 1.8V logic (~0.5V logic threshold) by changing the configuration in the device manager.

Figure 3.8 shows the layout of the Digilent AD2 Waveforms software that was primarily used for data collection. This layout can be found by opening the Final_PCB_Data_Collection.dwf3work file located in the documentation uploaded to BSU Scholarworks for this thesis within the "sstoller_thesis_final_writeup\Novel_TRNG\Waveforms" directory. In this view, the oscilloscope and logic views can be viewed at the same time. The oscilloscope measurements window and a small preview of the logic are available in the oscilloscope view. The logging, supplies, and wavegen windows are available in the logic view.


WaveForms beta (Final_PCB_Data_Collection)

WF1 - Oscilloscope 1 (Final_PCB_Data_Collection)

Mode: Repeated | Source: Channel 1 | Condition: Falling | Level: -300 mV

8192 samples at 7.6923 MHz | 2020-09-06 21:40:14.680

Time — Position: 0 s, Base: 100 us/div

Channel 1 (1±) — Offset: 3 V, Range: 1 V/div

Channel 2 (2±) — Offset: -2 V, Range: 1 V/div

Measurements

	Name	Value	Average	Minimum	Maximum
C1	Frequency	3.6007 ...		1.2672 ...	
C2	Frequency	3.5694 ...		2.0122 ...	

Logic 1 — Mode: Record, Trigger: Normal, Buffer: 10, Source: Digital, Samples: 1k, Rate: 2.5 kHz, Base: 400 ms

Logging — Execute: Manual, Index (#): 73, Maximum: 1000

Path: Z:/Thesis/Waveforms/Final_PCB/New_Memristors

File: ####_TRNG_Final_PCB.csv

File name: 0073_TRNG_Final_PCB.csv

Supplies — Master Enable is On

Positive Supply (V+) On — Voltage: 2.5 V

Negative Supply (V-) On — Voltage: -2.5 V

Auxiliary powered, allowing up to 2.1 W or 700 mA output per channel.

System Monitor — USB Voltage: 4.713 V, AUX Voltage: 5.251 V, Temperature: 51.25 °C / 124.25 °F, USB Current: 3.1 mA, AUX Current: 406.4 mA

Wavegen 1 — Channel 1 (W1): Type: DC, Frequency: 2.5 kHz, Period: 400 us, Amplitude: 1.35 V, Offset: 2.5 V, Symmetry: 50 %, Phase: 0 °

Channel 2 (W2): Type: DC, Frequency: 1 MHz, Period: 1 us, Amplitude: 1.35 V, Offset: -2.5 V, Symmetry: 50 %, Phase: 0 °

Discovery2 SN:210321A968E8 | Status: OK

Figure 3.8 Screenshot of the Digilent AD2 Waveforms application

CHAPTER FOUR: CHARACTERIZATION OF OTHER CIRCUITS

4.1 Introduction

The Intel RDRAND TRNG, Digilent AD2 PRNG, RANDU PRNG, Linux Random Entropy Pool TRNG, TRNG from Jiang et al., and TRNG from Rai et al. were all tested [20-27]. To characterize these RNGs 100 million bits of data were collected and analyzed from each. The data was split into 100 bitstreams with length of 1 Million bits each. Scripts or applications were written to capture output from the various RNGS and their output was analyzed to assess the effectiveness of the RNGs.

4.2 Intel RDRAND

4.2.1 Brief Description of RNG

The Intel RDRAND is an on-chip TRNG that Intel has included on their modern desktop (and server) processors. The RDRAND instruction is part of the Intel 64 and IA-32 instruction sets and is included on Intel’s Ivy Bridge and newer processors. Modern AMD processors also have a built-in TRNG and support the RDRAND instruction. A variety of cryptographic standards were used by Intel in the development of their RDRAND hardware, including NIST SP800-90, FIPS-140-2, and ANSI X9.82 [11].

The Intel RDRAND TRNG is composed of several parts. The first and most important part is the entropy source itself. The entropy source is a self-timed, self-oscillating digital circuit known as a dual differential jamb latch with feedback. At its core the circuit consists of two cross-coupled inverters that form a latch. The inverters are placed in a metastable state (that is, a state where both inverters are neither a digital 0 or

1) and then allowed to settle. There is a 50/50 chance that the latch settles in either one direction (output of 0) or the other (output of 1), based on thermal noise. Intel's circuit is self-clocked, meaning that once a stable state has been reached, the circuit triggers itself back into the metastable state. There is a feedback mechanism in the circuit in order to ensure that a metastable state is reached (and to compensate properly to ensure perfect 50/50 chance of the circuit settling into either stable state) before the circuit is allowed to settle into a stable state again. The output from this part of the circuit is clocked into a 256 bit shift register [20].

The next part of the TRNG is a Health and "Swellness" Testing portion of the circuit. This portion of the circuit performs some simple testing on the output from the Entropy Source to ensure that the bitstream generated has random characteristics. If the bitstream is passing the Health and Swellness testing it is passed on to the next part of the circuit that performs more post-processing and conditioning on the bitstream, finally outputting bits into 4x128 bit output buffers as high quality nondeterministic random seeds [20].

There are two instructions that can access the output of the TRNG on Intel processors: RDRAND and RDSEED. The RDRAND instruction returns a string of random numbers generated by an on-chip PRNG that is periodically re-seeded by the entropy source. The numbers generated by the RDRAND instruction are not considered to be truly random and are thus not suitable for cryptographic usage, despite being more unpredictable than a traditional PRNG alone. Because the outputs of the RDRAND instruction rely on a PRNG it may be possible (however unlikely) to guess the next part of the sequence by knowing a previous part of the sequence. The RDRAND instruction is more suitable for use in applications that need high quality random numbers capable of being generated at a high throughput, such as running Monte-Carlo simulations. Throughput of the RDRAND command is very high due to the fact that it's primary source of randomness is through a PRNG.

The RDSEED instruction on the other hand returns a truly random, cryptographically secure key from the entropy source. The RDSEED is truly random and is suitable for cryptographic applications or seeding other software PRNGs or hardware PRNGs on the computer system. For cases like this, high throughput is not needed since secure keys are normally only generated once. The RDSEED instruction is significantly lower throughput than the RDRAND instruction [21].

4.2.2 Process/Script used to Collect Data

C code was adapted from an example usage for RDRAND provided by Intel [22]. The code was modified to simply write a random string of numbers to a text file. The code was converted from text to a binary bitstream. Code used to generate the random bitstream may be found in Appendix A. 100 bitstreams of length 1 million bits each were tested using the STS assess application.

4.2.3 Generator Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\rdrand\rdrand\drng_samples\testdrng.c

sstoller_thesis_final_writeup\RNGs\rdrand\rdrand\perl\rdrand_to_datastream.pl

4.2.4 Results Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\rdrand\rdrand\data\short\rand_out.bin

4.2.5 Results

Table 4.1 Results of Intel RDRAND TRNG.

Test	**P-Value**		**Proportion**		**Pass/Fail**
Frequency	0.026948		100/100		PASS
BlockFrequency	0.657933		99/100		PASS
CumulativeSums	0.262249, 0.637119		199/200		PASS
Runs	0.759756		100/100		PASS
LongestRun	0.366918		97/100		PASS
Rank	0.401199		98/100		PASS
FFT	0.978072		99/100		PASS
NonOverlappingTemplate	-		14645/14800		PASS
OverlappingTemplate	0.304126		97/100		PASS
Universal	0.075719		95/100	*	FAIL
ApproximateEntropy	0.366918		99/100		PASS
RandomExcursions	-		537/544		PASS
RandomExcursionsVariant	-		1208/1224		PASS
Serial	0.275709, 0.851383		198/200		PASS
LinearComplexity	0.779188		100/100		PASS

All tests are passing except the Universal test with a value of 95/100. A proportion of 96/100 or better was needed to consider this test passing.

4.3 Digilent AD2 RNG

4.3.1 Brief Description of RNG and Data Collection Process

The Digilent AD2 Wavegen and Patterns functions have the ability to generate a series of random signals. The source of the randomly generated data is not known, but it is highly unlikely that the source is a TRNG due to the high rate at which random bits or analog voltages can be generated. It is likely that an LFSR or LCG is implemented in the Waveforms software or the AD2 hardware since these are simple and effective methods for generating random data in an application like this.

A data bus consisting of 8 random signals and a single clock was captured on the Digilent AD2. The Patterns function was used to set up DIO 0 thru DIO 7 to generate random peak-to-peak digital signals at a frequency of 50 kHz. DIO 15 was set to a 50 kHz clock signal. 100 million samples (50 million bytes, or 400 million bits of data were collected) of the clock signal were captured by the Waveforms Logic Analyzer software sampling at a frequency of 100 kHz. The output of the RNG passes all statistical tests.

4.3.2 Process/Script used to Collect Data

The output from the Digilent AD2 was saved to a .csv file. Data was collected on an 8-bit bus, in the same format used for data collection for the memristor TRNG implemented in this thesis. A Perl script was written to convert the CSV file to a binary bitstream. 100 sequences of 1 million bits each were tested using the STS assess application.

4.3.3 Script Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing\csv_to_datastream.pl

4.3.4 Results Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\Digilent_AD2_Random\out.bin

4.3.5 Results

Table 4.2 Results of Digilent AD2 RNG.

Test	**P-Value**		**Proportion**		**Pass/Fail**
Frequency	0.574903		97/100		PASS
BlockFrequency	0.678686		99/100		PASS
CumulativeSums	0.249284, 0.153763		194/200		PASS
Runs	0.171867		100/100		PASS
LongestRun	0.678686		100/100		PASS
Rank	0.23681		98/100		PASS
FFT	0.12962		99/100		PASS
NonOverlappingTemplate	-		14624/14800		PASS
OverlappingTemplate	0.224821		99/100		PASS
Universal	0.514124		100/100		PASS
ApproximateEntropy	0.202268		97/100		PASS
RandomExcursions	-		521/528		PASS
RandomExcursionsVariant	-		1184/1188		PASS
Serial	0.534146, 0.419021		199/200		PASS
LinearComplexity	0.026948		99/100		PASS

All statistical tests are passing.

4.4 RANDU

4.4.1 Brief Description of RNG

RANDU is a poor PRNG that was implemented on early mainframes in the Fortran programming language [23]. RANDU is a linear congruential generator (LCG) and can be easily implemented in programming languages other than Fortran. A LCG takes the form of a linear equation with a modulus of the result (discontinuous piecewise linear equation). An example is shown in equation 4.1. To assess the randomness of the RANDU LCG, it was implemented in Perl and tested using the NIST STS.

For years it was assumed that RANDU was a good PRNG and was suitable for many uses, including research that utilized it for Monte-Carlo simulations. In 1968 George Marsaglia at Boeing analyzed the RANDU PRNG and showed that it was a poor PRNG. When triplets from RANDU's output are plotted in a 3-dimensional space it was found that the resulting points all landed in a set of 15 planes [24]. A TRNG or good PRNG should produce an output that has no patterns in a three-dimensional plot (appearance should be the general shape of a cloud). Unfortunately, RANDU was widely used in many computer systems around the world for many years. A significant amount of research (specifically, research that used RANDU for Monte-Carlo simulations) was invalidated or had to be redone once the flaws with RANDU were discovered. The LCG used for RANDU is

$$R_{n+1} = 65539 * R_n \ mod \ 2^{31} \text{ [23].} \tag{4.1}$$

One major advantage of using multiplicative congruential generators like equation 4.1 to generate PRNGs is that they can generate many numbers extremely quickly. For example, it took less than 2 seconds to generate 100 million bits of binary data. By comparison, testing 100 million bits generated by the RANDU PRNG took approximately 23 minutes on the same machine. There are many modern multiplicative congruential PRNG generators that output much better strings of pseudo random numbers than RANDU. Many modern systems and software use multiplicative congruential generators today. In many cases (e.g. Intel's RDRAND), a TRNG is used to periodically re-seed a PRNG to give nearly TRNG output with the efficiency, speed, and throughput of a PRNG.

4.4.2 Process/Script used to Collect Data

The RANDU PRNG was implemented in a Perl script, randu.pl. A brief description of the script and source code can be found in Appendix A. In this script the parameters for output file, sequence length, seed, multiplier, and modulus are set as variables. When the script runs it creates a binary output file and writes the resulting string of binary numbers generated by the sequence. A starting seed of 31 was used for this analysis, but any seed could be chosen and would yield similar results.

4.4.3 Generator Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\RANDU\randu.pl

4.4.4 Results Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\RANDU\output.bin

4.4.5 Results

Table 4.3 Results of RANDU PRNG.

Test	**P-Value**		**Proportion**		**Pass/Fail**
Frequency	0	*	0/100	*	FAIL
BlockFrequency	0	*	0/100	*	FAIL
CumulativeSums	0, 0	*	0/200	*	FAIL
Runs	0	*	0/100	*	FAIL
LongestRun	0	*	14/100	*	FAIL
Rank	0	*	0/100	*	FAIL
FFT	0	*	0/100	*	FAIL
NonOverlappingTemplate	-		7834/14800	*	FAIL
OverlappingTemplate	0	*	0/100	*	FAIL
Universal	0	*	45/100	*	FAIL
ApproximateEntropy	0	*	0/100	*	FAIL
RandomExcursions	-		0/0		N/A
RandomExcursionsVariant	-		0/0		N/A
Serial	0, 0	*	0/200	*	FAIL
LinearComplexity	0.366918		100/100		PASS

The Frequency fails for all 100 bitstreams. No other test is considered valid when the frequency test does not pass. Regardless, no other tests passed with the exception of the Linear Complexity test.

4.5 Linux RANDOM Entropy Pool

4.5.1 Brief Description of RNG

Linux maintains an entropy pool from which truly random data can be extracted. Two files /dev/random and /dev/urandom are special character files which provide an interface to the Linux Kernel's random number generator [12, 25]. This random number generator collects noise from many different entropy sources of such as user input (mouse and keystroke inputs), external time delays (network latencies, HDD or SSD access times), mechanical sensors (HDD or fan rotation speed), onboard TRNGs (Intel RDRAND) etc. All this information is collected and pooled together in the entropy pool. Up to 4096 bits are stored in the entropy pool. The entropy pool may be directly read using the /dev/random character device, or used to seed a PRNG and to generate nearly truly random output at a very high bandwidth by reading from the /dev/urandom character device.

1. /dev/urandom returns a string of random numbers from a PRNG that is seeded by the entropy pool.
2. /dev/random returns random bytes directly from the random entropy pool. Note that the function will delay and block (wait) for new entropy to be created in the entropy pool if it is exhausted by the function call.

4.5.2 Process/Script used to Collect Data

Since the random entropy pool /dev/random only has a size of 4096 bytes, an output from the RNG was generated from the urandom utility. The Linux command line may be used to dump 12,500,000 bytes (100 million bits) into a binary file.

4.5.3 Data Collection Process

The following command was executed to collect data from the Linux URANDOM generator “`head -c 12500000 /dev/urandom >| output.bin`”

4.5.4 Results Location

Campbell Research group folder at BSU or Scholarworks:

System without rdrand: sstoller_thesis_final_writeup\RNGs\urandom\nordrand\urandom_test_nordrand.bin

System with rdrand: sstoller_thesis_final_writeup\RNGs\urandom\withrdrand\urandom_test_withrdrand.bin

4.5.5 Results

Table 4.4 Results of Linux uRandom RNG on system without RDRAND

Test	**P-Value**		**Proportion**		**Pass/Fail**
Frequency	0.595549		99/100		PASS
BlockFrequency	0.678686		100/100		PASS
CumulativeSums	0.202268, 0.897763		198/200		PASS
Runs	0.066882		99/100		PASS
LongestRun	0.145326		99/100		PASS
Rank	0.739918		98/100		PASS
FFT	0.834308		100/100		PASS
NonOverlappingTemplate	-		14648/14800		PASS
OverlappingTemplate	0.102526		99/100		PASS
Universal	0.935716		100/100		PASS
ApproximateEntropy	0.99425		99/100		PASS
RandomExcursions	-		508/512		PASS
RandomExcursionsVariant	-		1146/1152		PASS
Serial	0.798139, 0.366918		196/200		PASS
LinearComplexity	0.030806		100/100		PASS

All tests are passing for uRandom on a Linux system with a CPU that does not support the Intel RDRAND instruction. An additional sequence was tested (location listed above) from a system that does support the Intel RDRAND command and all tests were found to pass for that system as well (table of results is not shown for this case).

4.6 True RNG Circuit from Jiang et al.

4.6.1 Brief Description of RNG

Jiang et al. have proposed, and implemented in hardware, a TRNG using a Ag:SiO_2-based diffusive memristor device [26]. A pulse train is sent through a memristor that is wired in series with a resistor. Entropy is captured in the memristor device as a variability in the time it takes for the device to transition from a high resistance state to a low resistance state. When the memristor device is in a high resistance state, the resistor pulls the signal down, resulting in the output of a low voltage into the comparator. Once the resistance of the memristor transitions to a low resistance state, it can override the resistor and pull the output voltage to a high value, switching the signal output from the comparator.

When the output of the memristor-resistor ladder is low (the memristor is a high resistance), below the V_{ref}, the comparator output can be high. The comparator used in the hardware implementation had both an output and inverse output to allow the counter to be enabled for either the memristor in a low or high resistance state simply by selecting one output or the other and adjusting the comparator input voltage accordingly. When the comparator output is low, the AND gate output will always be a logic 0 and the counter will not count. When the comparator output is high, the AND gate output will be the clock input and the counter will count. When the memristor transitions from a high resistance to a low resistance the output of the comparator switches from a 1 that allows the clock to pass through the AND gate and the counter count, to a 0 and the counter stops counting because the clock is no longer passed through the AND gate. In my implementation of this circuit, the output of the comparator was fed straight into the enable input on the counter. When the enable input is a 0 (active low enable) the counter counts. When the enable input transitions to a 1 the counter stops counting. In addition, I also included a brief pulse to clear the counter state with each cycle in order to reduce the chances of the output of the previous clock cycle affecting the output of the next clock cycle.

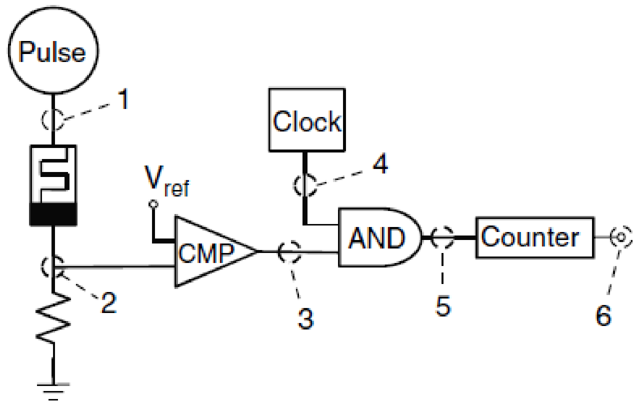


Figure 4.1 Proposed TRNG Design by Jiang et al. [26]

The TRNG proposed by Jiang et al. consists of a pulse generator, memristor, resistor, comparator, and gate, and counter. The circuit can easily be implemented on a breadboard.

4.6.2 Process/Script used to Collect Data

The circuit was implemented on a breadboard with a 1 kHz pulse frequency and a 2 MHz clock frequency (both generated by the Digilent AD2 Wavegen). V_{ref} was generated using a potentiometer between the Digilent AD2 V+ power supply and V- power supply. This allowed the V_{ref} voltage to be easily adjusted for varying pulse input voltages.

Data was saved using a 2 kHz sample rate. This was found to be optimal because it is a multiple of 1 kHz and the minimum frequency to properly sample the 1 kHz clock by the Nyquist sampling theorem. The data was collected using a data bus sampled on the falling edge of the input data clock and post-processed using a generic Perl script to convert the single bit samples to a binary format. A 22 kΩ resistor was used in series with a W-based self-directed channel (SDC) memristor devices. Figure 4.2 shows the schematic of the circuit used in the test of the Jiang et al. circuit [26].

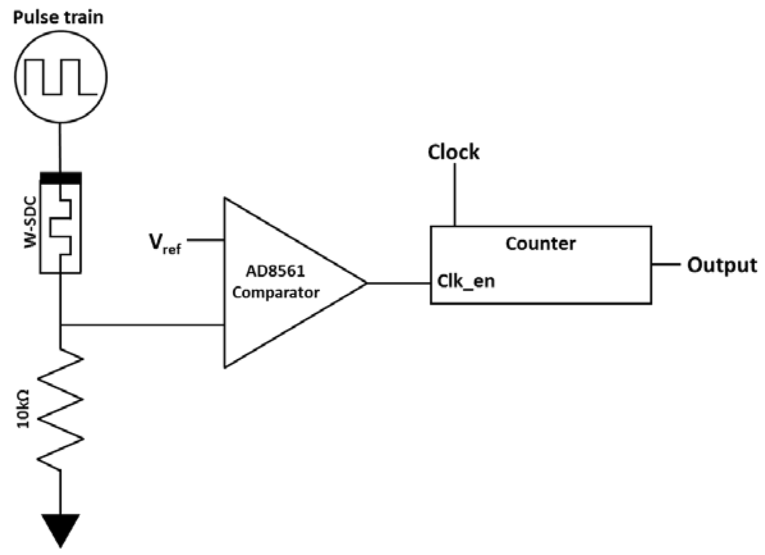


Figure 4.2 Schematic of Jiang et al. tested in this Thesis

The Digilent AD2 Logic Analyzer data was saved to a csv data file and post-processed using a Perl script “perl\CSV_to_Datastream\csv_bin_to_datastream.pl” to convert the csv to a binary output file that could be tested by the NIST test suite. This script does not perform any debiasing on the raw data. A second script was used to apply von Neumann whitening, “datastream_to_datastream_von_neumann.pl”.

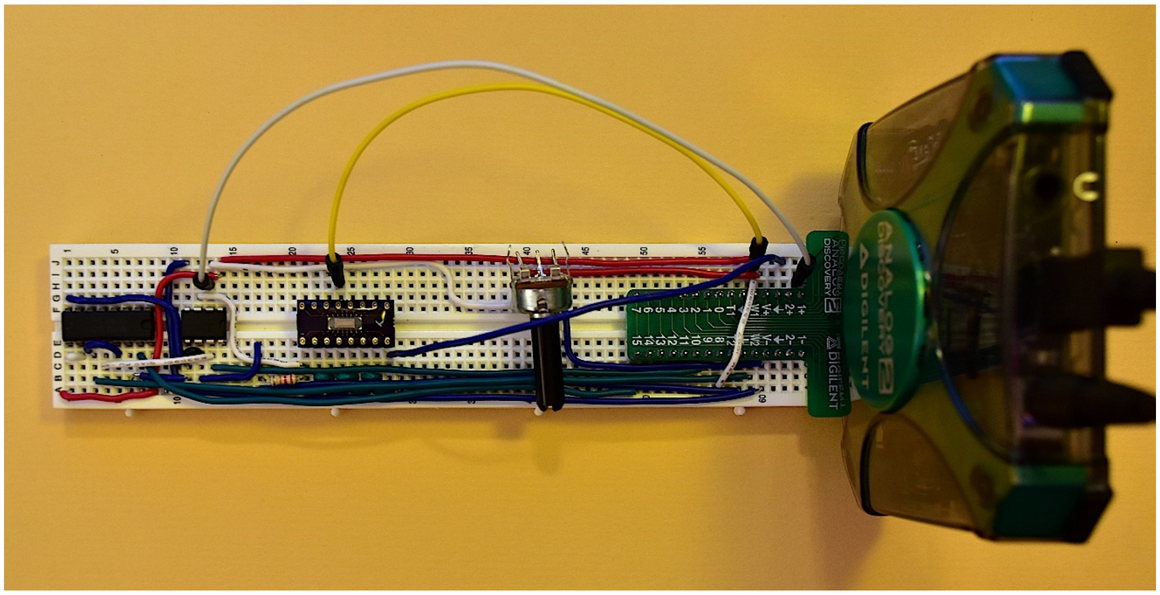


Figure 4.3 Implementation of Jiang et. al design on a breadboard

Figure 4.4 shows an oscilloscope output and data bus capture of the TRNG in action. The blue oscilloscope trace is the output of the memristor-resistor ladder. The switching action of the memristor from low to high resistance can be clearly seen. The yellow trace is the analog voltage output of the potentiometer used to set the comparator voltage threshold (V_{ref}). In the logic section, the output of the data bus can be seen. Output data is clocked on the falling edge of the clock. The MSB value of the Bus (DIO15) can be seen counting with the input clock when the output from the comparator (DIO9) is low. The output from the comparator only enables the counter when the output of the memristor-resistor ladder is above the yellow V_{ref} voltage.

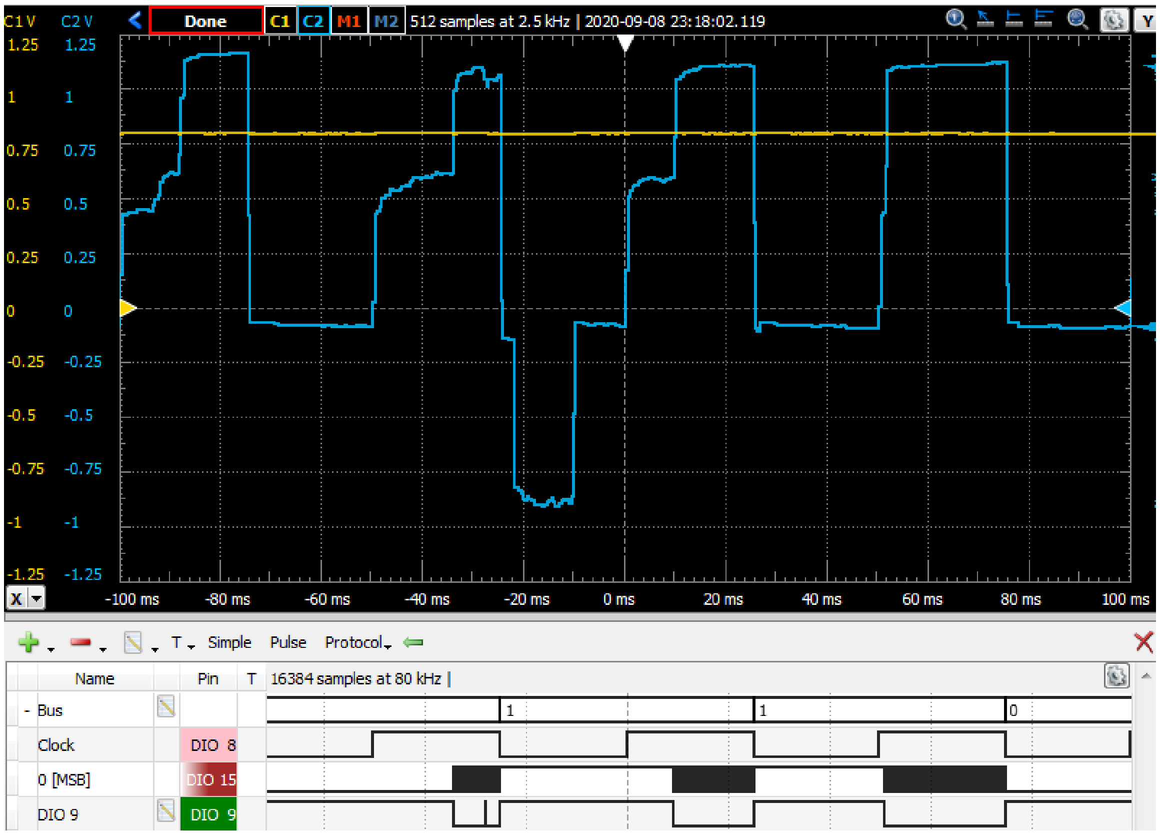


Figure 4.4 Data Collection of Jiang et. al. design in Digilent AD2

4.6.3 Data Collection Script

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\csv_bin_to_datastream.pl

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing\datastream_to_datastream_von_neumann.pl

4.6.4 Results Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\Jiang

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\Jiang

4.6.5 Results

Table 4.5 Results of TRNG proposed by Jiang et. al.

Test	P-Value		Proportion		Pass/Fail
Frequency	0.637119		10/11	*	FAIL
BlockFrequency	0.637119		11/11		PASS
CumulativeSums	0.437274, 0.162606		21/22		PASS
Runs	0.637119		9/11	*	FAIL
LongestRun	0.162606		10/11	*	FAIL
Rank	0.275709		11/11		PASS
FFT	0.162606		11/11		PASS
NonOverlappingTemplate	-		1546/1628		PASS
OverlappingTemplate	0.637119		11/11		PASS
Universal	0.834308		10/11	*	FAIL
ApproximateEntropy	0.437274		10/11	*	FAIL
RandomExcursions	-		47/48		PASS
RandomExcursionsVariant	-		108/108		PASS
Serial	0.437274, 0.437274		20/22	*	FAIL
LinearComplexity	0.437274		11/11		PASS

Multiple tests are failing with only a single failure. The data in table 4.5 is for Jiang’s TRNG with von-Neumann whitening applied to the output bitstream. 11 sequence of length 1M bits were tested. Proportion to consider passing was 11/11 (96% or greater). Without von Neumann whitening the TRNG returned an 18% to 82% ratio of 0’s to 1’s and was thus not tested. More sequences need to be tested for this TRNG before making a final judgement if the Jiang TRNG is passing of failing the NIST STS.

4.7 True RNG Circuit from Rai et al.

4.7.1 Brief Description of RNG

The TRNG proposed by Rai et al. is a design that is similar in concept to the dual inverter oscillator design mentioned earlier. Rai et al. simulated and analyzed several different TRNGs before settling on a design with two series of oscillators, each with a memristor in series with the inverters [27]. A TiO_2 memristor model from [28] was used in the simulations performed by Rai et al. It is important to note that the results shared in this Thesis are from a true implementation of the circuit. Figure 4.5 shows a representation of the design.

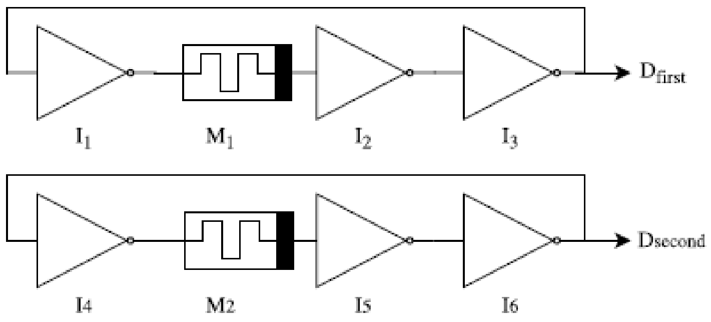


Figure 4.5 Rai’s proposed circuit design [27].

The final design settled upon by Rai et al. was not implemented as a true ring oscillator, instead receiving a clock input from another circuit. The final design consisted of two identical inverter-memristor string pairs in the order of Inverter, Memristor, Inverter, Inverter. The input of the first inverter was a clock signal from another circuit. The outputs of the two inverter strings are both sampled and the output of the TRNG is based on which output switches first. If the D_{first} output switches first the output is 0. If the D_{second} output switches first, then the output is a 1.

In this thesis writeup, this TRNG circuit was implemented with discrete components on a breadboard and the output was collected using the Digilent AD2 and Perl scripts to parse the result. TI SN74AHCT04N inverter chips were used along with the W-based SDC memristors. The circuit input was driven by a square wave clock generated from the Digilent AD2 Wavegen. An Analog Devices AD8561 comparator was used to capture which device switched first by using the latch input on the comparator.

One delay path was connected to the input of the comparator and the other delay path was connected to the latch input on the comparator. If the first delay path was faster, the output of the comparator would be driven high before the output was latched by the second delay path. If the second delay path was faster, the output of the comparator would be latched at a low output. Figure 4.6 shows a schematic of the circuit tested in this thesis.

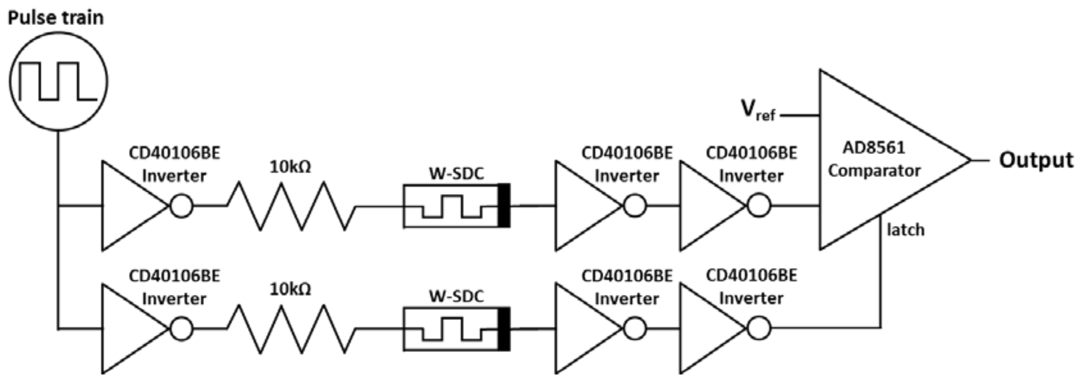


Figure 4.6 Circuit schematic tested for Rai et al. TRNG

The variability between Memristor devices was a challenge in testing this circuit. In many cases one delay path was consistently much faster than the other delay path. It was challenging to find two devices that switched at approximately the same time. In addition, the devices sometimes showed a tendency to drift and change with time. In order to try to compensate for some of this effect, von Neumann debiasing was applied to the outputs and tested as well. The circuit does not pass the NIST STS tests without von Neumann whitening applied.

Several other methods were investigated for collecting data from this TRNG. All of these methods involved oversampling the logic output of the AD2 at a very high frequency in order to capture which delay path was quicker. Several different data processing methods were utilized, from looking at which delay path was faster, to looking at whether the number of samples between the input clock and output clocks were even or odd. These methods showed potential to also produce random numbers, but very few samples could be collected due to the very high sample rate needed to collect this information. For this reason only results collected with the latched comparator output are saved.

4.7.2 Process/Script used to Collect Data

Figure 4.7 shows an image of the test setup and memristor-based TNRG proposed by Rai et al. Two inverter ICs and two Memristor ICs are shown on the breadboard along with the comparator IC. 2.2 kΩ resistors were placed in series with the memristor devices to ensure that current through the devices was limited.

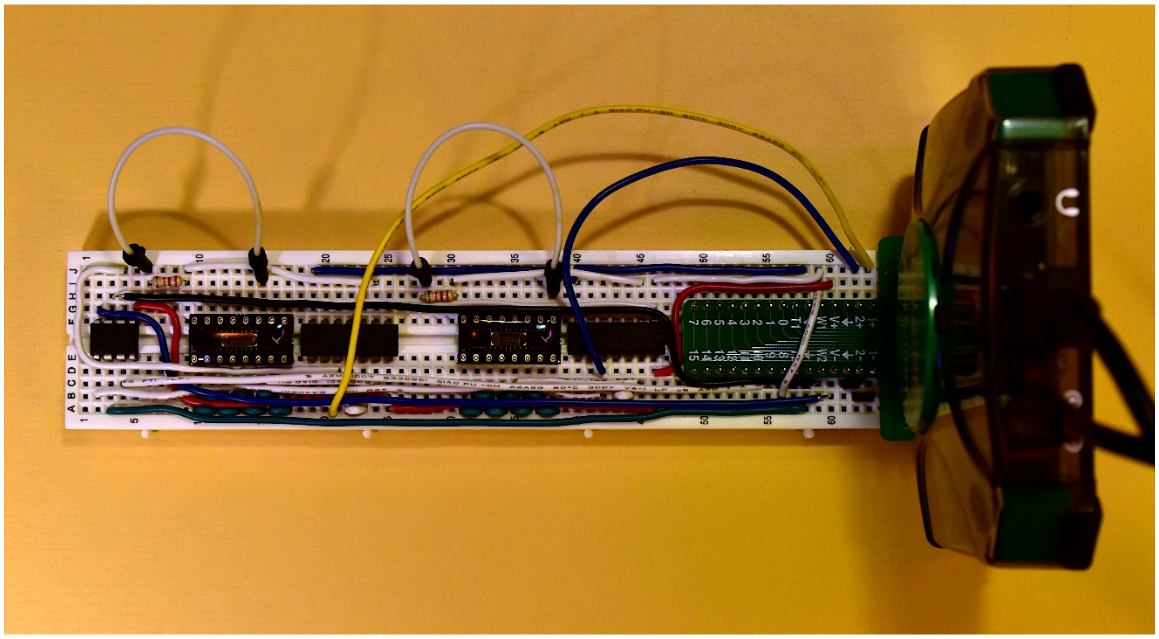


Figure 4.7 Image of the test setup and circuit to test Rai et al.

4.7.3 Data Collection Script

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing\csv_bin_to_datastream.pl

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing\datastream_to_datastream_von_neumann.pl

4.7.4 Final Results Path

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\RNGs\Rai

4.7.5 Results

The TRNG was tested with a squarewave clock frequency of 1kHz into the memristor-inverter chain. The output was sampled with a frequency of 8kHz in order to ensure that the latched output from the TRNG could be captured. Approximately 62 million bits were captured. Due to a 60/40% distribution of ones and zeros in the output the raw output was not tested through the STS because it would certainly have failed the frequency test. With von Neumann whitening applied 12 sequences of length 1 million bits were produced. The results in table 4.6 show all NIST STS tests passed for the TRNG proposed by Rai et al [27]. More sequences need to be tested for this TRNG.

Table 4.6 Results of TRNG proposed by Rai et al [27].

Test	P-Value		Proportion		Pass/Fail
Frequency	0.213309		12/12		PASS
BlockFrequency	0.350485		12/12		PASS
CumulativeSums	0.739918		24/24		PASS
Runs	0.017912		12/12		PASS
LongestRun	0.350485		12/12		PASS
Rank	0.213309		12/12		PASS
FFT	0.739918		12/12		PASS
NonOverlappingTemplate	-		1760/1776		PASS
OverlappingTemplate	0.350485		12/12		PASS
Universal	0.122325		12/12		PASS
ApproximateEntropy	0.122325		12/12		PASS
RandomExcursions	-		69/72		PASS
RandomExcursionsVariant	-		158/162		PASS
Serial	0.122325		23/24		PASS
LinearComplexity	0.122325		12/12		PASS

CHAPTER FIVE: DESIGN OF FINAL CIRCUIT

The primary purpose of this Master's thesis project was to implement a True Random Number Generator (TRNG). The following sections discuss the final circuit design, testing, and implementation of the memristor based TRNG.

5.1 Design Concept

The core concept of the novel memristor-based TRNG circuit is not much different from many other TRNGs. The circuit consists of two oscillators. One oscillator runs at a fast speed and the other oscillator runs at a slow speed. The slow oscillator acts as a clock to sample data from the fast oscillator. Entropy is captured in the slow oscillator as jitter [29].

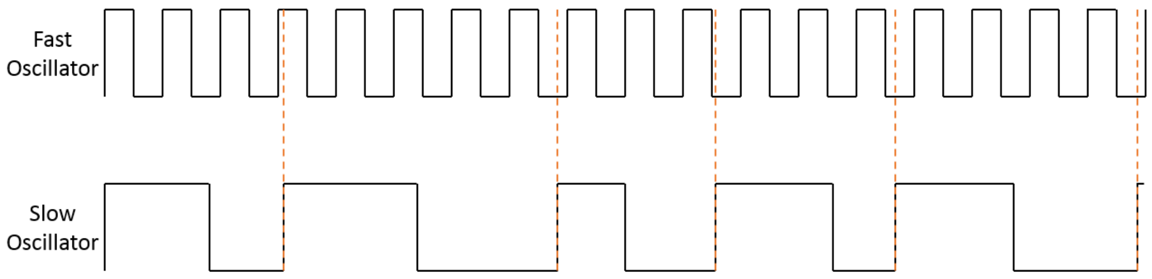


Figure 5.1 A slow clock sampling a fast clock

Figure 5.1 shows Jitter (emphasized) on a slow clock that is sampling a fast clock. Jitter is emphasized. It is desirable to have many oscillations of the fast clock for each oscillation of the fast clock.

5.2 How a Multivibrator Circuit Works

The multivibrator circuit shown in figure 5.2 is an astable type of multivibrator. The circuits in figure 5.2 are shown with both a resistor and with a memristor in place of a resistor as was used in the TRNG presented in this thesis. Figure 5.2 shows the circuit schematic for the memristor-based multivibrator circuit used in the TRNG design presented in this thesis.

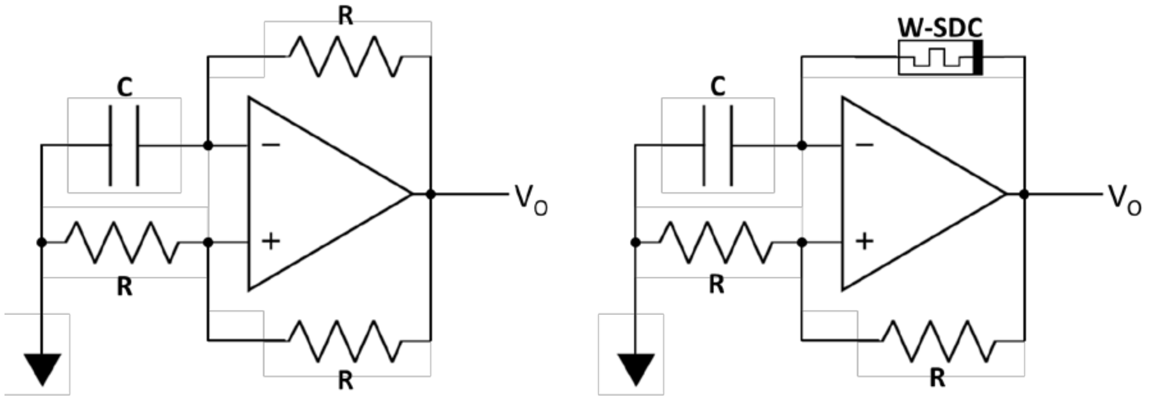


Figure 5.2 Multivibrator circuits implemented with resistor and memristor

At its core, the multivibrator consists of a voltage divider and an RC circuit network. The voltage divider between the resistors is the input to the positive leg of the op-amp. The RC network is the input to the negative leg of the op-amp. The multivibrator works because the voltage divider always places a supply voltage of ½ V_O on the positive input of the op-amp and the RC network always lags behind the output switching on the negative leg.

When the voltage output is 1V on V_O, the voltage input on the positive leg is 0.5V. In order for V_O to rail its output at the positive supply, the voltage across the capacitor must be less than 0.5V. With time, the RC circuit will eventually charge up the capacitor to a voltage of 0.5V. Once the supply voltage of 0.5V is reached (negative and positive inputs into the op-amp are the same), V_O will drop to 0V. Once V_O drops to 0V, this changes the voltage at the positive input of the op-amp. As a result, the positive voltage is now lower than the negative voltage of the opamp, leading to a V_O that will rail at -1V, the negative supply voltage. This cap will slowly discharge until a voltage of -0.5V is reached and V_O will again rail at the high supply voltage. This cycle will repeat and the circuit will oscillate as long as the op-amp has a sufficiently high gain and very little hysteresis or other types of feedback that could cause the circuit to settle.

The maximum voltage across the memristor in a multivibrator circuit can be easily calculated. The maximum or minimum voltage at the negative input of the op-amp is one half of the negative supply or positive supply voltage. In this case, the output of the op-amp is 100% of the positive or negative supply voltage. If the supply voltages are 100% negative of each other (e.g. VCC is equal to -VDD) the maximum voltage across the memristor devices is 150% of VCC or VDD supply voltage as calculated by

$$V_{MAX} = 0.5\,VCC - VDD = 0.5\,VCC - -VCC = 1.5\,VCC. \tag{5.1}$$

It was found that replacing the resistor with a memristor achieved the same type of oscillation but with significantly more jitter and noise in the frequency of the multivibrator output. A memristor operates well in this circuit because it is continuously cycled between a high resistance and low resistance state. Depending on the orientation of the memristor device in the circuit and the oscillation speed of the circuit (dependent on the memristor device and capacitor), the circuit can have a skewed duty cycle (that is it will spend significantly more time in one state than the other). The significant difference in the operation of the circuit with a memristor device is that the memristor will change resistance from low to high or high to low in the different states of the circuit.

As the memristor changes resistance (it has been shown by Jiang et al. that the memristor resistance change happens at a truly random time), the RC time constant of the circuit changes. Entropy is captured in the delay that it takes for the memristor device to switch from low to high or high to low resistance. The circuit may be modified by placing multiple memristor devices in parallel or in series with each other or with a resistor.

Figure 5.3 shows the measured and plotted IV curve for a W-Type SDC memristor in a multivibrator circuit. A 1 kΩ resistor was placed in series with the memristor in the multivibrator circuit in order to measure current. Voltage was measured across both the memristor and the resistor. Current was calculated using the voltage across the resistor and Ohm’s law. The IV measurement device response of the W-SDC memristor-based multivibrator oscillator is consistent with the IV curve device response from other publications [9-10].

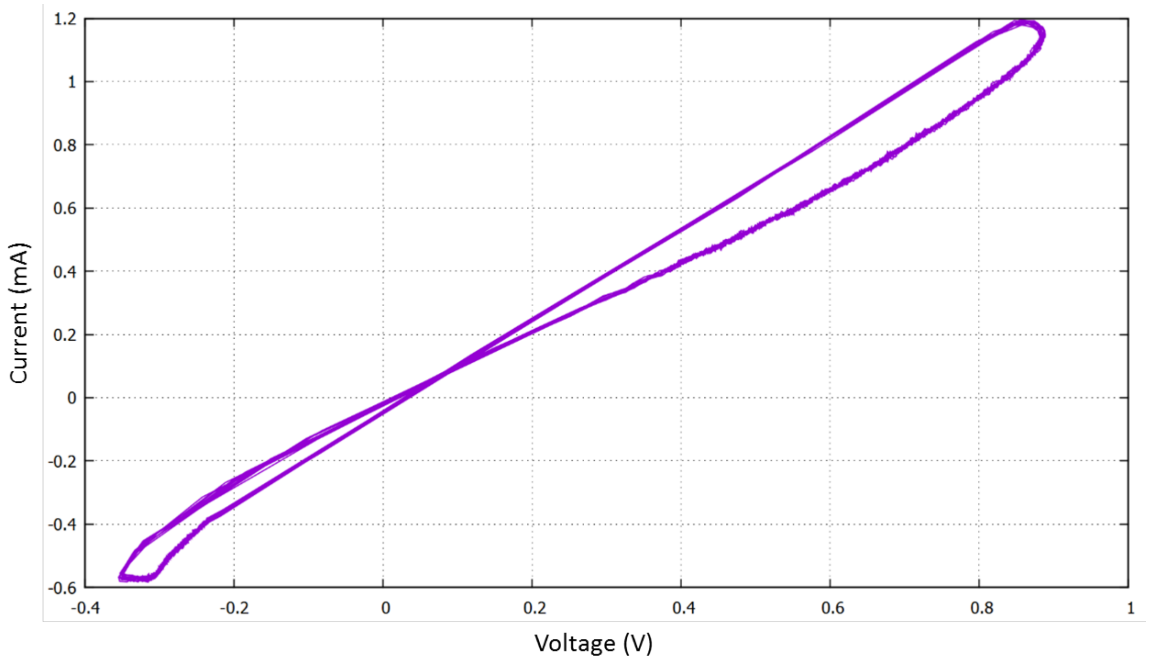


Figure 5.3 Measured IV curve for a W-SDC Memristor in a multivibrator circuit

5.3 Multivibrator Oscillator Simulations and Measurements

This thesis project is unique in the fact that a memristor is incorporated in a multivibrator circuit for the slow oscillator. None of the other papers and RNGs that were reviewed as part of a literature survey for this topic utilized this type of circuit design to capture entropy and generate random numbers. Two other TRNG circuit designs were discussed earlier. Both utilized the unique characteristics of memristor devices to capture entropy in unique ways, but both ultimately relied on an external circuit to generate a square wave clock. This project is unique because the memristor device itself is a part of the circuit that creates oscillations.

After implementing and testing the randomness of many different circuit designs and topologies a final circuit design was settled upon and is shown in figure 5.4. It was found through rigorous testing and trial and error that a memristor works well in place of a resistor in a multivibrator circuit and allows the circuit to oscillate as desired.

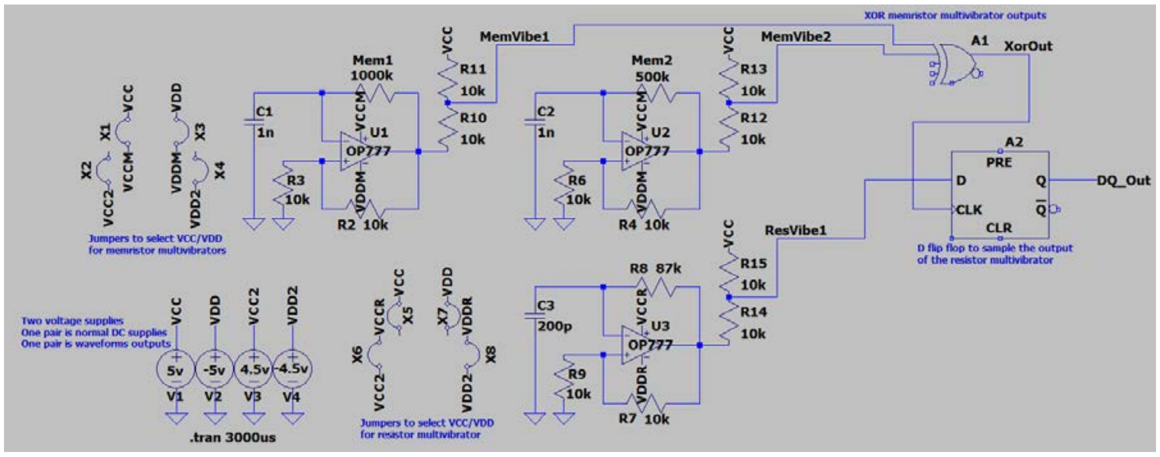

XOR memristor multivibrator outputs
MemVibe1
MemVibe2
XorOut
A1
Mem1
1000k
Mem2
500k
R11 10k
R10 10k
R13 10k
R12 10k
C1 1n
C2 1n
U1 OP777
U2 OP777
R3 10k
R2 10k
R6 10k
R4 10k
Jumpers to select VCC/VDD for memristor multivibrators
A2
PRE
D
Q
CLK
CLR
DQ_Out
D flip flop to sample the output of the resistor multivibrator
ResVibe1
R8 87k
R15 10k
R14 10k
C3 200p
U3 OP777
R9 10k
R7 10k
Two voltage supplies
One pair is normal DC supplies
One pair is waveforms outputs
5v
-5v
4.5v
-4.5v
V1
V2
V3
V4
.tran 3000us
Jumpers to select VCC/VDD for resistor multivibrator

Figure 5.4 Simplified schematic of the final circuit

The characteristic of the memristor (mainly the changing resistance and the variability in the delay and amount of resistance changes) lend them well to capturing significant amounts of entropy. The comparison of the output from a multivibrator implemented with a memristor compared to a resistor is shown in figures 5.5 and 5.6. In both figures approximately 1000 samples have been captured from the oscilloscope on the Digilent AD2 and overlaid to show the jitter produced in both circuits. It can be easily seen that the circuit with the memristor has significantly more jitter, which lends itself to being useful for capturing entropy as the slow clock signal in this TRNG.

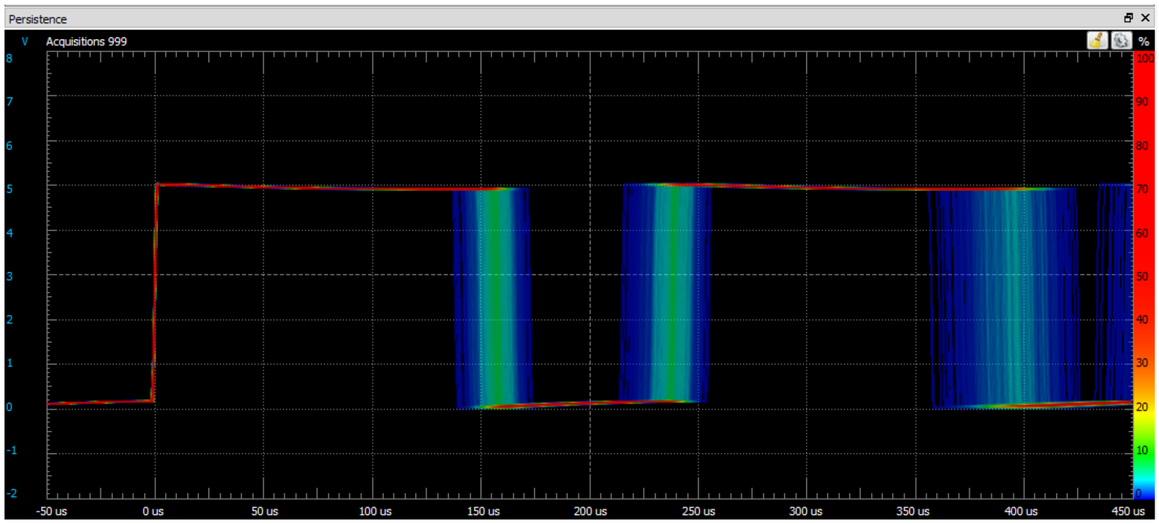


Figure 5.5 Persistence plot of the output from a memristor based multivibrator

Figure 5.5 shows a persistence plot that demonstrates the jitter in 999 samples captured from a memristor based multivibrator circuit. Note the significant amount of jitter present in the output waveform.

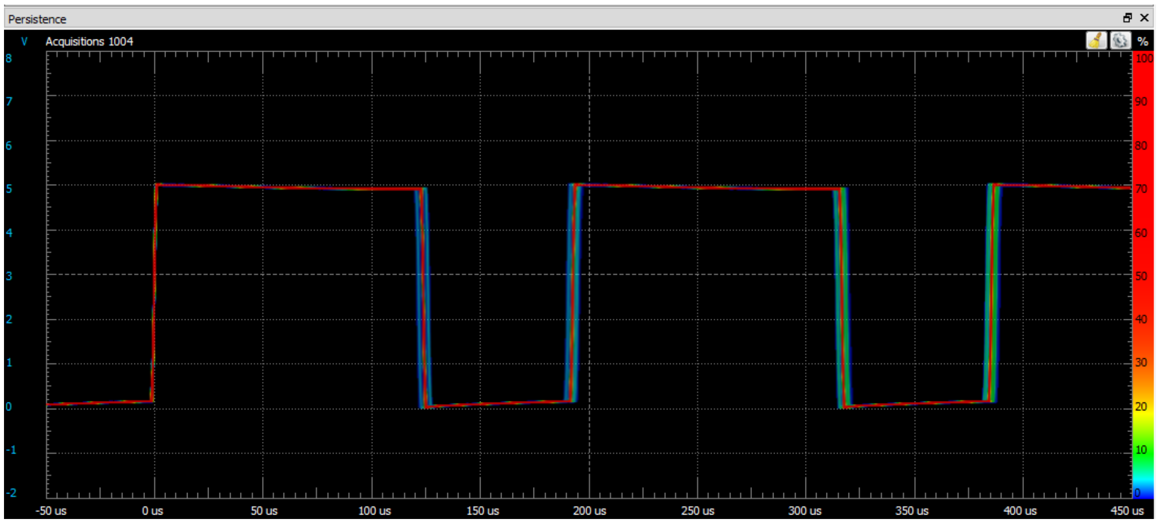


Figure 5.6 Persistence plot of the output from a resistor based multivibrator

Figure 5.6 shows a persistence plot that demonstrates the jitter in 1004 samples captured from a resistor based multivibrator circuit. Note that there is significantly less jitter shown in this circuit implemented with a resistor. Both circuits were tuned to operate at approximately the same frequency.

The final circuit design settled upon (shown in figure 5.7) consists of three oscillators. The fast oscillator design consists of a multivibrator implemented with a resistor. In reality this could be any simple circuit that oscillates at a high frequency, such as a 555 timer, ring oscillator, or square wave generated by a function generator. The primary purpose of this oscillator is to be sampled by the slow oscillator. The frequency of the oscillator can be modulated by changing the values of the capacitor or resistor in the circuit to change the RC time constant and thus the frequency of oscillation. A smaller capacitor or lower value resistor will increase the oscillation speed. A larger capacitor or more resistive resistor will slow down the oscillation speed. In order to optimize the power of the circuit, the smallest capacitor (and thus largest resistor) should be selected.

The slow oscillator final design consisted of two multivibrators, each implemented with a memristor. As shown in figures 5.5 and 5.6 the memristors implemented in this circuit create a very significant amount of jitter. The oscillation speed of these circuits can be controlled by changing the value of the capacitors in the multivibrators. In addition, a different memristor device could be used. While only a single type of memristor was used and characterized for this project, it was noted that there was some device-to-device variability. In addition, several experiments were done with multiple devices placed in series and in parallel with each other. It was also noted that increasing the supply voltage of the circuit also resulted in faster oscillation of the memristors.

The output of the two multivibrator oscillator circuits were XOR'd together. There are several advantages of using this topology: the frequency of the slow clock is increased by a factor of two, allowing random data to be generated twice as fast. In addition, different types of memristors may be mixed, or a memristor may be mixed with a resistor etc. in the two different oscillators. The circuit also provides flexibility to allow for only one of the oscillators to be active.

Figure 5.7 shows a simplified schematic of the circuit along with simulated SPICE output. For SPICE simulations, resistors were used in place of the memristor devices.

There are many different parts of the circuit. The two slow memristor multivibrator oscillator outputs go into an XOR gate that effectively increases the entropy captured in the circuit. This also results in a faster clock rate for the circuit (or allows for more clock division later on in the circuit to capture even more entropy). The XOR circuits output goes into an 8 bit binary counter with jumpers that are used to divide the slow clock to allow for more oscillations of the multivibrators for each slow clock cycle. This allows for more entropy to be captured for every oscillation of the slow clock that samples the fast clock in the circuit. The output of the binary counter (with clock division selected by jumper) goes into the binary counter and the serial-in-parallel-out shift register. The 3rd bit of the binary counter here is connected to the Digilent AD2. This signal toggles one time for every 8 bits that are shifted through the shift register. The fast oscillator (resistor-based multivibrator circuit) goes directly into the serial-in-parallel-out shift register where it is sampled as the slow oscillator toggles.

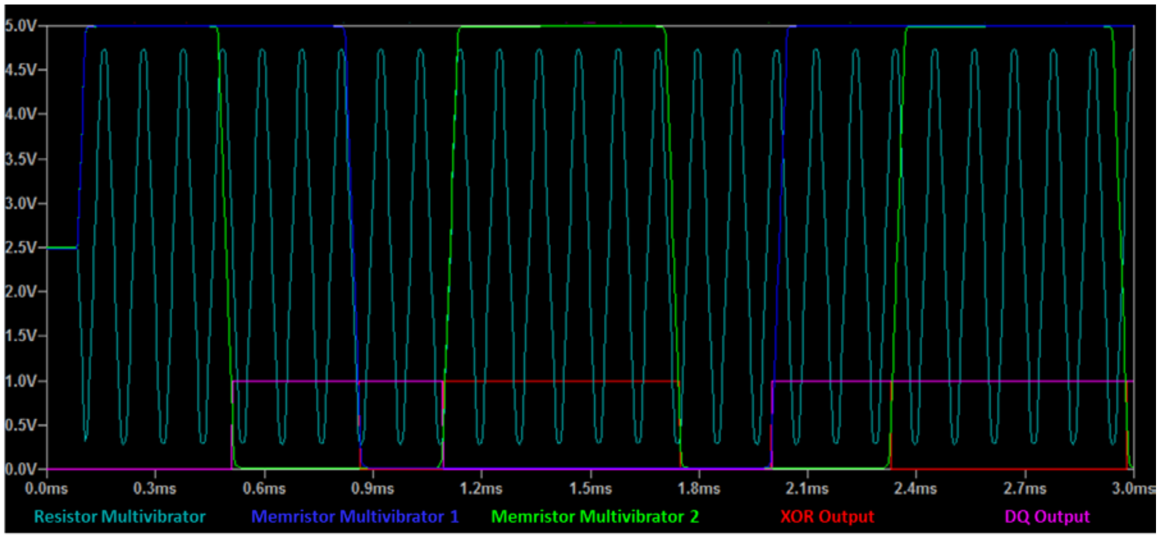


Figure 5.7 Voltage outputs from a transient simulation of the circuit

The output of the XOR (Red) samples the input to the DQ flip flop (Teal). The output of the DQ Flip flop (Pink) shows how "fast clock" is sampled by the "slow clock". Final implementation of the circuit also included a binary counter to allow the "slow clock" to be divided by a factor of 2, 4, 8, or up to 256. This allows more oscillations of the slow clock to accumulate extra jitter and noise for each time that the fast clock is sampled.

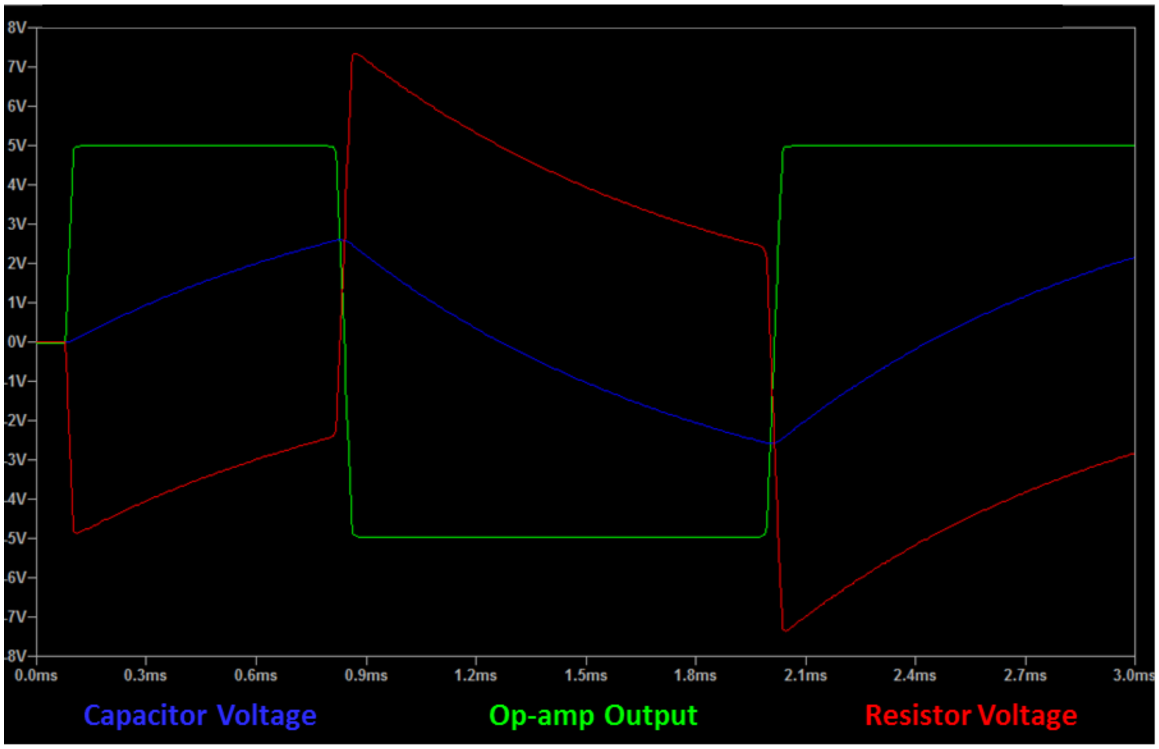


Figure 5.8 Transient Simulation Results for multivibrator with resistor

Figure 5.8 shows transient simulation results for a multivibrator circuit implemented with a resistor. Green shows the output of the op-amp. Blue is the voltage across the capacitor, which is also the voltage at the inverting input of the op-amp. Red is the voltage across the resistor in lieu of a memristor in the multivibrator.

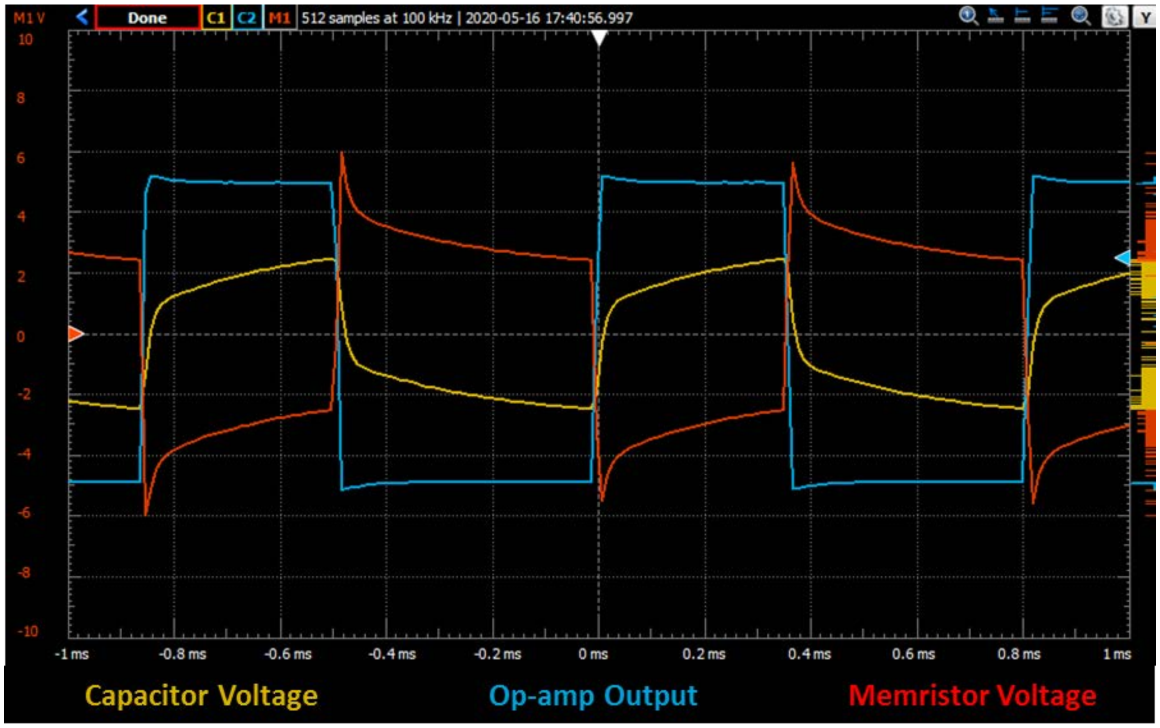


Figure 5.9 Measured output for multivibrator implemented with a W-SDC memristor

Figure 5.9 shows measured output for a multivibrator circuit implemented with a W-type SDC memristor. Blue shows the output of the op-amp. Yellow is the voltage across the capacitor, which is also the voltage at the inverting input of the op-amp. Red is the voltage across the memristor. The op-amp supply voltage in this case is +/- 5V.

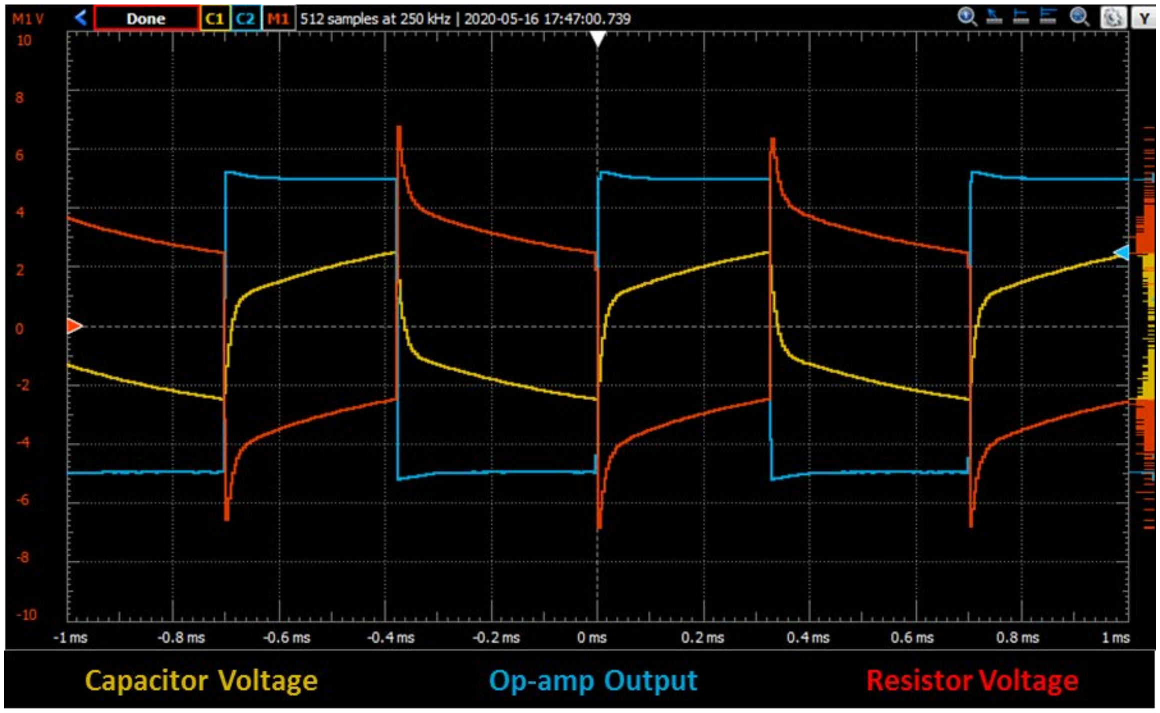


Figure 5.10 Measured output for multivibrator implemented with a resistor

Figure 5.10 shows measured output for a multivibrator circuit implemented with a resistor in place of the memristor. Blue shows the output of the op-amp. Yellow is the voltage across the capacitor, which is also the voltage at the inverting input of the op-amp. Red is the voltage across the resistor taking the place of the memristor. The op-amp supply voltage in this case is +/- 5V.

5.4 Circuit Design and Development

Many different circuit iterations were prototyped and tested before settling on the final circuit design. These circuits ranged from simple ring oscillators to the memristor based multivibrator, to combining the input of multiple multivibrators. Many different circuit tweaks and modifications were tested.

5.4.1 Summary of Breadboard Implementation

Initially the final circuit design was implemented and prototyped on a breadboard. Figure 5.11 shows an image of the breadboard with different parts of the circuit noted.

The entire circuit was implemented on breadboard before designing and fabricating the PCB.

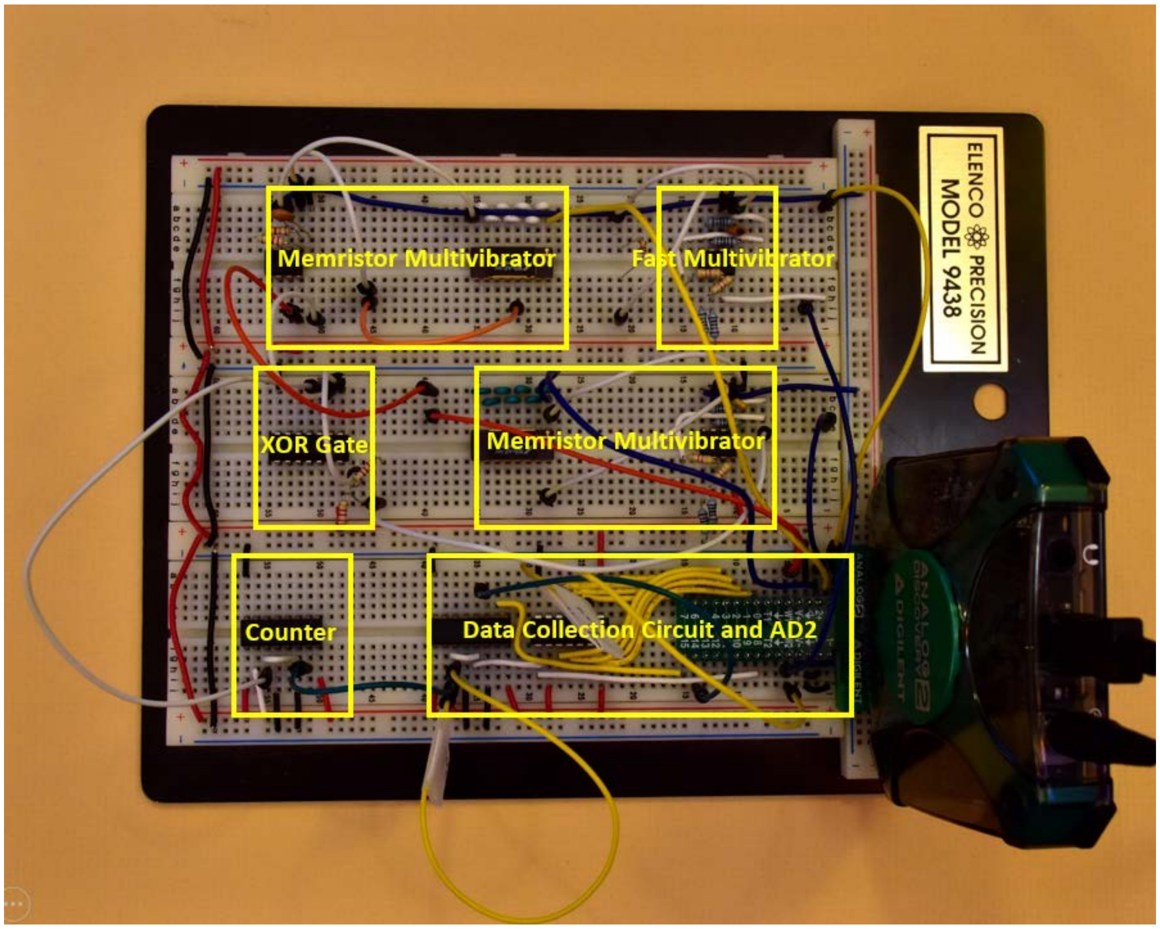


Figure 5.11 Breadboard with the TRNG prototype

In the circuit both memristor multivibrators can be seen, along with the XOR gate, the binary counter that is used as the clock divider for the slow clock, and the binary counter and serial-in-parallel-out shift register used for the data collection. The bus from the data collection circuit to the Digilent AD2 can also be seen. The fast oscillator multivibrator circuit is also shown. There are additional wires used in the circuit, such as the Digilent AD2 oscilloscope inputs which are being used in this image to probe the outputs of the memristor multivibrators.

5.4.2 PCB Design

The TRNG was implemented on a printed circuit board (PCB) once the design was prototyped and finalized on the breadboard. The PCB was designed to contain the entire TRNG circuit on a nine square inch (3”x3”) layout and to plug directly into the Digilent AD2 for quick, efficient, and easy data collection. The PCB was fabricated by Sunstone Circuits [31]. Sunstone Circuits provides their own circuit design and layout application called PCB123 [32]. This software was easy to learn and intuitive to use. The software also includes a very large parts library. The PCB123 software allows for the creation of custom footprints for any parts that are not in the parts library.

5.4.3 PCB Schematics

PCB schematics are shown in figures 5.12, 5.13, 5.14, and 5.15. These figures show the memristor multivibrator circuits, the fast resistor multivibrator circuit, the data collection circuit, and the binary counter used to divide the output of the slow clock circuit.

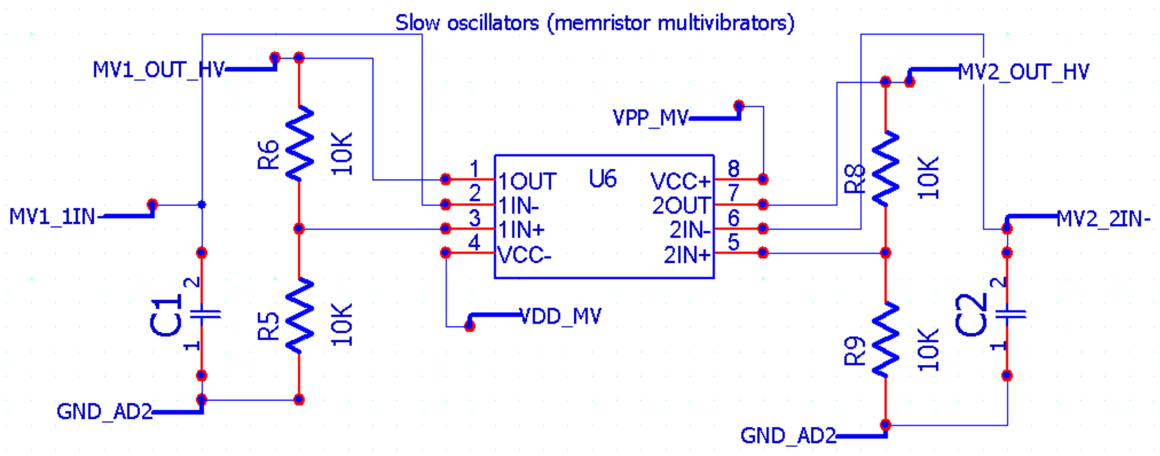


Figure 5.12 PCB circuit schematic for the two memristor multivibrator oscillators

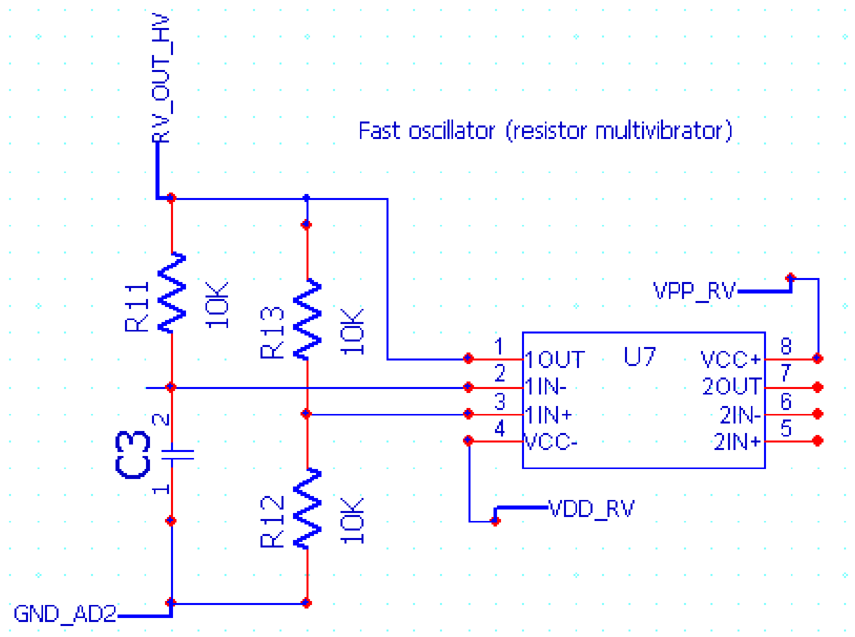
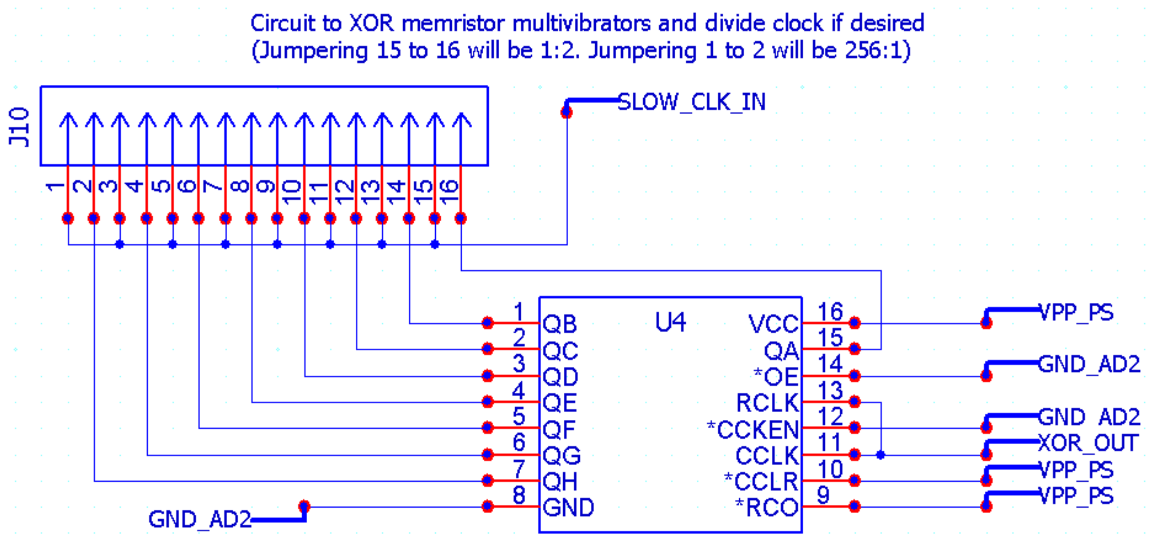


Figure 5.13 Multivibrator circuit schematic for the fast oscillator.



Circuit to XOR memristor multivibrators and divide clock if desired
(Jumpering 15 to 16 will be 1:2. Jumpering 1 to 2 will be 256:1)
J10
SLOW_CLK_IN
U4
QB
QC
QD
QE
QF
QG
QH
GND
VCC
QA
*OE
RCLK
*CCKEN
CCLK
*CCLR
*RCO
VPP_PS
GND_AD2
XOR_OUT

Figure 5.14 Binary counter slow clock divider schematic

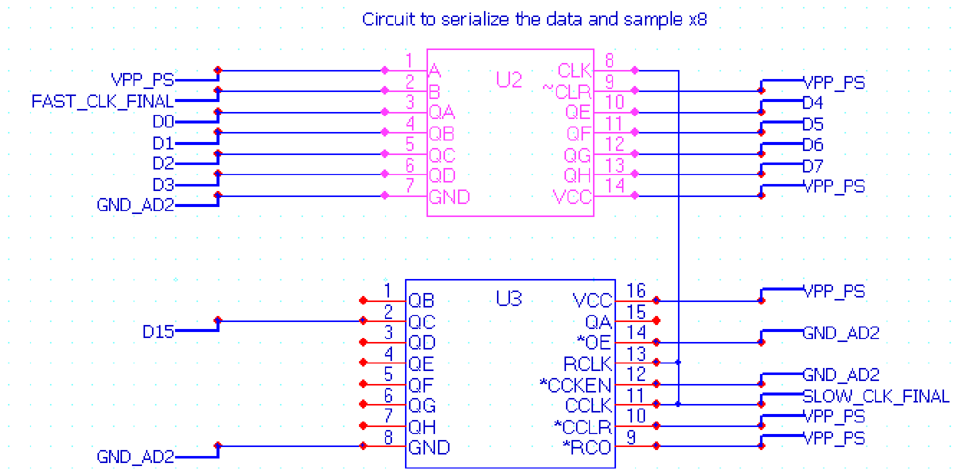


Figure 5.15 Serial-to-parallel data collection circuit schematic

5.5 PCB Layout

Great care was taken in the PCB design and layout. Circuits were kept compact and modular in order to help improve trace routing. When designing and laying out the PCB good design practices were followed. The PCB layout was done completely by hand to ensure optimal placement of all components and optimal routing. Initially the auto place and auto routing tools were used, but they yielded suboptimal results. In order to fit the final circuit in the desired 3”x3” footprint manual placement and routing was required for the entire board.

In order to reduce IR droop and power loss in PCB traces, routing for power traces was sized up to a width of 0.016 inches (twice the width of signal traces). In order to provide the shortest routing of power, all power traces were routed before any signal traces. Whenever possible, power wires for different supplies were not routed parallel to each other to minimize crosstalk. The data collection circuit (specifically the serial-in-parallel-out shift register) was placed as close as possible to the 30 pin connector for the AD2 to minimize routing complexity of the 8-bit output bus. Working backwards from the data collection circuit was found to be the most effective method of routing to ensure that routing was kept simple, short, and efficient. This worked well because the oscillators only required power routed to them. There are not circuit connections directly from the oscillators to the 30 pin AD2 connector. However, some connections were added on to the PCB after the fact by soldering wires to jumper outputs from each of the oscillators to additional digital inputs that were not used on the AD2 30 pin connector. The additional wires can be seen in figure 5.16 that shows the back side of the PCB final modified PCB.

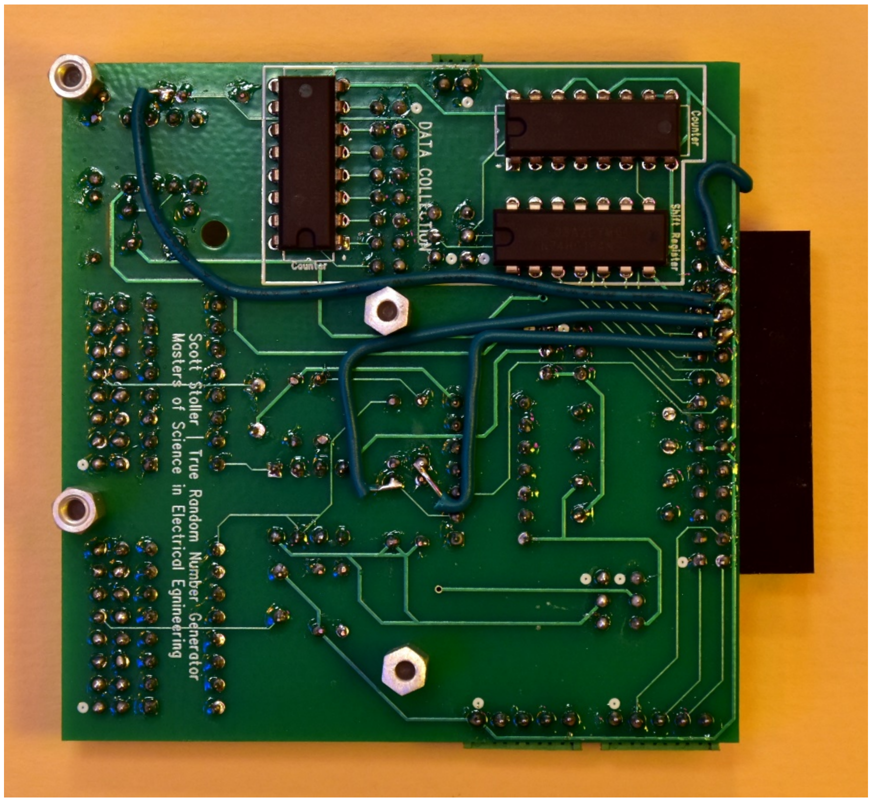

DATA COLLECTION
Counter
Shift Register
Counter
Scott Stoller | True Random Number Generator
Masters of Science in Electrical Egineering

Figure 5.16 Backside image of finished, modified PCB

In an effort to improve the layout of the PCB and routing efficiency, it was found that routing could be made easier by flipping over several components in the data collection circuitry, including the serial-in-parallel-out shift register and both binary counters. Not only did this improve the board layout, it also gave extra room around several of the jumpers and the terminal for external clock inputs. The slow multivibrator oscillators shared the same dual op-amp chip. These circuit layouts were mirrored as much as possible to make routing and layout easier and yield a cleaner finished look to the PCB.

All components used in the PCB design were through-hole components in order to keep soldering easy. This was also convenient when the aforementioned modifications were made to help validation and testing of the circuits.

In some cases, not all components were available in the built-in libraries provided by the Sunstone PCB123 software. Several components, such as the screw terminals, standoffs, and resistor/capacitor sockets were not readily available or easy to find in the software or online for download and import into the PCB software. In addition, models were not readily available only for direct import into the PCB123 software libraries. For these components, different components with the same footprint were used. After careful searching and measurement, adequate substitutions were found for these layouts that had the same hole spacing and hole size.

In order for the PCB to sit on the tabletop when connected to the AD2 and provide clearance from between the table top and the PCB for components that were placed on the bottom of the board, five holes for standoffs were included in the final design. These holes were added last and as a result there was very little room to place the standoffs. Not all standoffs could be placed in ideal locations at the corners of the board. Standoff height was estimated based on measurements of other boards that plug into the Digilent AD2. The aluminum standoffs were sanded slightly (less than 1/16th of an inch) to reduce their height for final fitment and to reduce any rocking due to a very slight but noticeable difference in the standoff height and height of the AD2 board/connector.

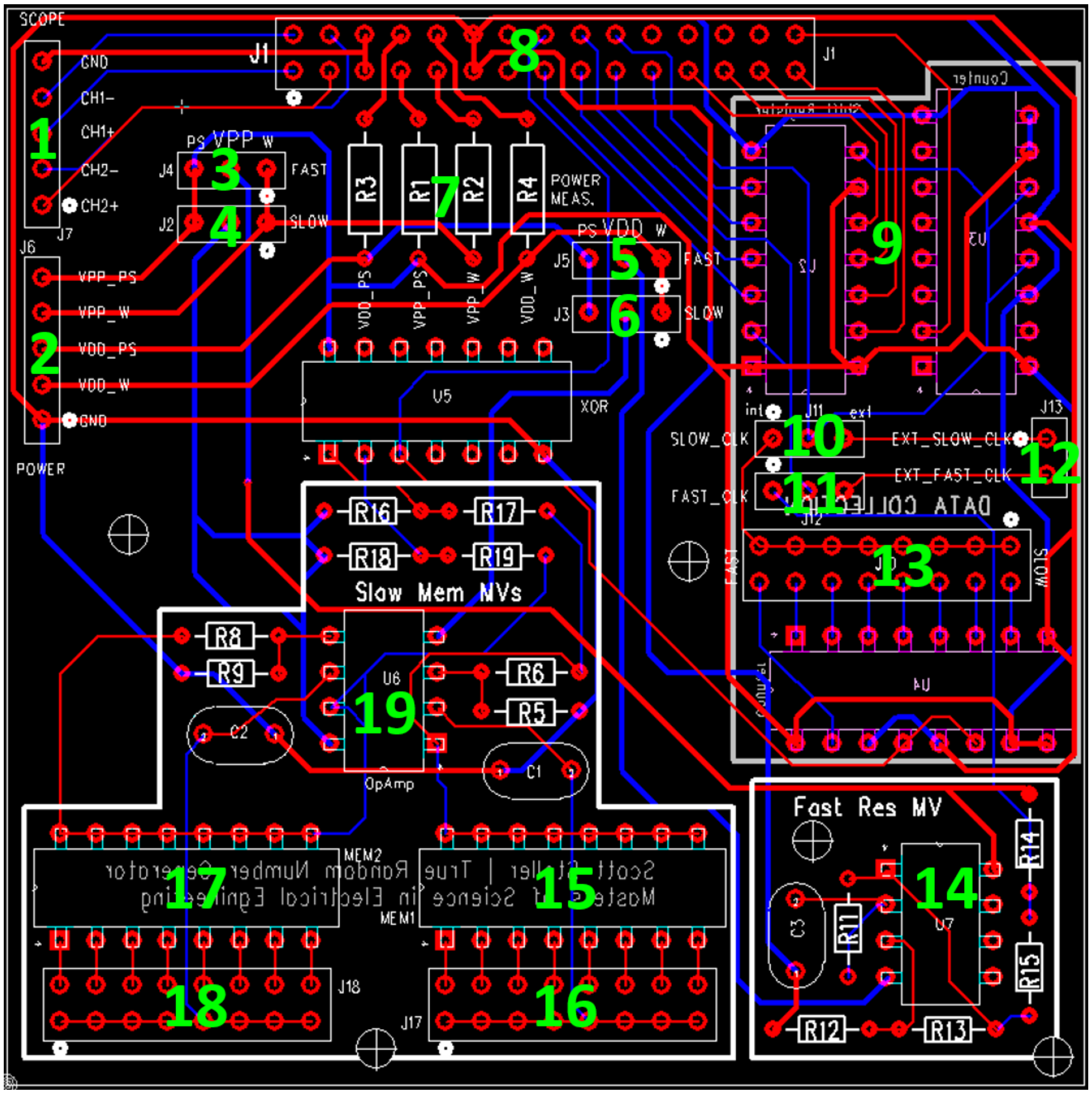


Figure 5.17 Final PCB Layout.

Figure 5.17 shows the final PCB layout. Different sections of the PCB are highlighted in the figure. These sections are described below, starting at the top left corner of the board and moving clockwise around the board. In many cases a 2-pin jumper may be connected to different header pins to change the configuration of the board for added flexibility.

1. **Scope Inputs:** GND, Scope CH1-, Scope CH1+, Scope CH2-, and Scope CH2+ signals are routed from the AD2 to a 5 port screw terminal. This is useful for probing and debugging circuits on the board.
2. **Power Outputs:** VPP Power Supply (V+), VPP Waveforms (W1), VDD Power Supply (V-), VDD Waveforms (W2) signals are routed from the AD2 to a 5-port screw terminal. These signals are useful for the case that an external circuit is implemented, and the external slow and fast clocks are input directly to the data collection circuit, bypassing the oscillators on the board.
3. **VPP Voltage Source Selector for Fast Oscillator Circuit:** A jumper is used to select between W1 or V+ as the positive power supply for the fast multivibrator oscillator.
4. **VPP Voltage Source Selector for Slow Oscillator Circuits:** A jumper is used to select between W1 or V+ as the positive power supply for the memristor multivibrator oscillators.
5. **VDD Voltage Source Selector for Fast Oscillator Circuit:** A jumper is used to select between W2 or V- as the negative power supply for the fast multivibrator oscillator.
6. **VDD Voltage Source Selector for Slow Oscillator Circuits:** A jumper is used to select between W2 or V- as the negative power supply for the memristor multivibrator oscillators.

7. **Resistors in Series with Power Supplies:** Low resistance resistors or wire jumpers may be soldered in place here to measure the power consumption in various parts of the circuit.
8. **Digilent AD2 Header:** 30 pin header for the Digilent AD2. A right-angle header female connector is used.
9. **Data Collection Circuit:** Binary counter and serial-in-parallel-out shift register circuit portion of the data collection circuit. Located as close as possible to the AD2 30 pin header.
10. **Internal or External Slow Clock Selector:** A jumper is used to select between the internally generated slow clock (from the memristor multivibrators) or an external slow clock generated by another circuit with input provided to the external slow clock screw terminal.
11. **Internal or External Fast Clock Selector:** A jumper is used to select between the internally generated fast clock (from the resistor multivibrator) or an external fast clock generated by another circuit with input provided to the external fast clock screw terminal.
12. **External Clock Terminal:** 2 port screw terminal for input of slow and/or fast external clock oscillators. Jumpers (10) and (11) must be jumpered properly to use this input.
13. **Clock Divider Jumper:** 8 option jumper to divide the clock down from the slow oscillators circuit. Left jumper is fasts, right option is slowest.
14. **Fast Multivibrator Oscillator circuit:** Full circuit for fast oscillator, including output level shifting resistors.

15. **Memristor 1 Socket:** Memristor socket for first slow oscillator multivibrator circuit. A memristor in standard DIP package is placed in the socket.
16. **Memristor 1 Device Select Jumper:** The jumper selects which memristor device is used for data collection. Multiple devices may be selected. A resistor also fits in the socket as an alternative to a memristor.
17. **Memristor 2 Socket:** Memristor socket for first slow oscillator multivibrator circuit. A memristor in standard DIP package is placed in the socket. The gap between the sockets for the memristors is spaced such that a single 16 DIP package may be used in both sockets at the same time.
18. **Memristor 2 Device Select Jumper:** The jumper selects which memristor device is used for data collection. Multiple devices may be selected. A resistor also fits in the socket as an alternative to a memristor.
19. **Slow Oscillator Circuit:** The rest of the slow oscillator circuit including resistors and op-amp.

PCB design and layout files can be found in the Campbell Research group folder at BSU at the location sstoller_thesis_final_writeup\PCB Design. These files can also be found in BSU Scholarworks. An Excel spreadsheet with the bill of materials and other notes is also included at this location.

5.5.1 Interface to the Digilent AD2

The PCB interfaces to the Digilent AD2 through a 30 port right angle female header connector. The scope channels are routed directly to a header on the board. The scope channels have their own ground signal. The power supply and waveform generator signals are routed to resistors/jumpers that allow for power measurements of the circuit.

Digital I/Os 0-7 are the 8 bit input sampled from the data collection circuit, with digital I/O 15 being the clock. Digital I/Os 11, 12, and 13 were not a part of the original PCB design, but were soldered onto the design later with a wire jumper to enable output to the digital I/Os from each oscillator in order to aid in circuit validation, debug, and testing. Digital I/O 14 was also added later. This signal was soldered in place to enable the readout of every 16th byte instead of every 8th byte from the data collection circuit as a potential means to increase randomness of the circuit even more.

Table 5.1 contains a list of the power supply, analog, and digital pins on the Digilent AD2 board. There is also a column that list the net that each pin is attached to in the PCB schematic. A '*' in the notes section denotes a pin that was not part of the PCB design. A wire jumper was soldered on later for one of the PCBs in order to capture these signals for debug output or other purposes.

Table 5.1 Pin usage of the Digilent AD2 board connector

Pin Number	Pin Name	Description	Net	Notes
1	1+	Scope CH1 Positive	SC_CH1P	
2	1-	Scope CH1 Negative	SC_CH1N	
3	2+	Scope CH2 Positive	SC_CH2P	
4	2-	Scope Ch2 Negative	SC_CH2N	
5	GND	Ground	GND_Scope	
6	GND	Ground	GND_Scope	
7	V+	V+ Power Supply	PS_VP	VPP PS
8	V-	V- Power Supply	PS_VN	VDD PS
9	W1	Waveform Generator 1	W1_P	VPP W
10	W2	Waveform Generator 2	W2_N	VDD W
11	GND	Ground	GND_AD2	
12	GND	Ground	GND_AD2	
13	T1	Trigger 1	N/C	
14	T2	Trigger 2	N/C	
15	0	Digital I/O 0	D0	Data out
16	8	Digital I/O 8	XOR_OUT	Slow clk out of xor
17	1	Digital I/O 1	D1	Data out
18	9	Digital I/O 9	FAST_CLK_FINAL	Final fast clk
19	2	Digital I/O 2	D2	Data out
20	10	Digital I/O 10	SLOW_CLK_FINAL	Final slow clk
21	3	Digital I/O 3	D3	Data out

22	11	Digital I/O 11	N/C	*Mem 1 Osc Out
23	4	Digital I/O 4	D4	Data out
24	12	Digital I/O 12	N/C	*Mem 2 Osc Out
25	5	Digital I/O 5	D5	Data out
26	13	Digital I/O 13	N/C	*Fast Osc Out
27	6	Digital I/O 6	D6	Data out
28	14	Digital I/O 14	N/C	*For sampling I/Os slower
29	7	Digital I/O 7	D7	Data out
30	15	Digital I/O 15	D15	Clock for sampling I/Os

5.5.2 PCB Screw Terminal Signal Breakouts

There are three sets of terminals on the board. Two sets of terminals are located on the top left of the board. One set of terminals is located halfway down the right side of the board. The screw terminals require a fine jeweler’s screwdriver to tighten and accept various wire gauges and jumper sizes.

The purpose of the terminals on the top left corner of the PCB (labelled “scope”) is to break out signals from the Digilent AD2. The first terminal is a breakout of the scope signals for the AD2 board. There is a ground signal. To improve signal integrity and reduce noise this screw terminal is the only net connected to this ground net on the AD2. This allows for breakout of the oscilloscope inputs for the AD2 in order to probe and measure voltages and signals from various locations on the PCB. These signals were useful to initially verify functionality of the PCB after fabrication and assembly. They were also used to capture various waveforms and IV curves for the oscillators. The screw terminals are each labelled “GND”, “CH0-“, “CH0+”, “CH1-“, “CH1+”.

The second terminal (labelled “Power”) is a breakout of the power supplies and waveforms outputs from the Digilent AD2. There are two power supply outputs and two waveform generator outputs as well as another ground. These outputs are connected to the output sides of the power measurement resistors to help make power measurements easier. These outputs can also be used to power an external oscillator circuit. The output of the Waveforms generator are also made accessible at this terminal for use as an additional power supply in an external circuit or to generate a wave to stimulate an external circuit. To collect data for and evaluate the random waveform generator for the Digilent AD2 waveform output, both a clock and random waveform were generated from these terminals and connected to the fast clock input and slow clock input terminals on the other side of the board. A ground terminal (ground supply used for the rest of the circuits) is also supplied to this terminal. The screw terminals are labelled “VPP_PS”, “VPP_W”, “VDD_PS”, “VDD_W”, and “GND”.

The third screw terminal (labelled “J13”) is for an external oscillator input to be used. This allows for any external circuit for either the fast or slow oscillator to be used for data collection using the PCB’s x8 data collection circuit and interface to the Digilent AD2. There are two terminals, one labelled “EXT_SLOW_CLK” for an external slow clock/oscillator signal and one labelled “EXT_FAST_CLK” for an external fast clock/oscillator signal. Either one or both of these signals may be used in combination with the slow or fast oscillators on the PCB. Immediately to the left of this terminal is a pair of jumpers to switch the circuit over from using the internal oscillator signals to an external input.

5.5.3 PCB Jumpers

There are several sets of jumpers located on the PCB. These jumpers give a high level of configurability and flexibility to the circuits on the PCB. Starting from the top left corner of the PCB and moving clockwise, the jumpers are J4, J2, J5, J3, J11, J12, J10, J17, and J18. The jumpers serve different purposes from selecting power supplies for various circuits, to selecting devices to be used in circuit etc.

J4, J2, J5, and J3 are all jumpers to select which power supplies are connected to the oscillator circuits. These jumpers are 3 pin headers. Each jumper should connect the center pin to either the left or right pin. For all four of these jumpers, connecting the center pin to the left pin will connect the oscillator to the AD2 power supply. Connecting the center pin to the right pin will connect the oscillator to the AD2 waveforms output (used as an additional power supply). J4 and J5 are for the positive and negative supply voltages of the fast oscillator (resistor multivibrator). J2 and J3 are for the positive and negative supply voltages of the slow oscillators (memristor multivibrators). Isolating the oscillator power supplies from each other will reduce the chances of noise from one oscillator affecting the other oscillator. In addition, the waveforms output could be used to inject supply noise to the oscillator circuits. The importance of power supply noise oscillation is discussed in further detail in Chapter 6.

Jumpers J11 and J12 are used to toggle between the internal oscillators and external oscillator inputs. Each jumper is a 3 pin header. Connecting the center pin to the left pin on either jumper will use the internal oscillators. Connecting the center pin to the right pin on either jumper will configure the board to take sample the oscillator input to the header just to the right of these jumpers. The top jumper (J11) is for the slow clock. The bottom jumper (J12) is for the fast clock.

Jumper J10 (label not visible on assembled PCB) is a set of 8 pairs of jumpers (2x8 set of headers). The purpose of this jumper is to allow for the selection of various clock divisions of the slow oscillator. The outputs of a binary counter can be selected as the slow clock oscillator to sample the fast clock. Connections between pins should only be made vertically from a top pin to bottom pin. Each pin oscillates at ½ of the frequency of the pin to the left of it. The fastest option can be chosen by jumping the leftmost vertical pair of pins. The slowest option can be chosen by jumping together the rightmost vertical pair of pins. In hindsight, a ninth option should have been made available in order to allow a bypass of the binary counter altogether. The fastest output of the binary counter is only half of the frequency of the input clock frequency. To prevent shorting outputs of the binary counter together, jumpering multiple pairs of pins together at once should be avoided.

Jumpers J17 and J18 are located at the bottom center and bottom left corners of the PCB. The purpose of these jumpers is to act as a selector for the memristor devices. Different devices may be selected by jumping different pins together. The vertical jumper jumping two vertical pins together corresponds to select the device in the socket directly above it. This also allows for multiple devices could be jumped together to function in parallel.

5.6 PCB Fabrication and Soldering Process

The PCB design, layout, and routing were all manually checked multiple times over the course of several days to ensure that there were no errors in the final design. Several issues were found and corrected in the first and second checks of the PCB. The third check found no errors and the design was submitted for fabrication. The PCB123 software has a built-in Design Rule Check (DRC) and netlist checking tool to ensure that there are no violations of design rules (e.g. traces too close together) and that the final layout and routing matches the circuit schematic. Four dual layer PCBs were ordered from Sunstone circuits at a total cost of $169.00.

Components were ordered from Digikey. All components used were through-hole components which made PCB assembly easy. No surface mount or other types of components were used. The downside of using through hole components is that they generally require more space on the PCB. Figure 5.18 shows the front and back sides of the final PCB.

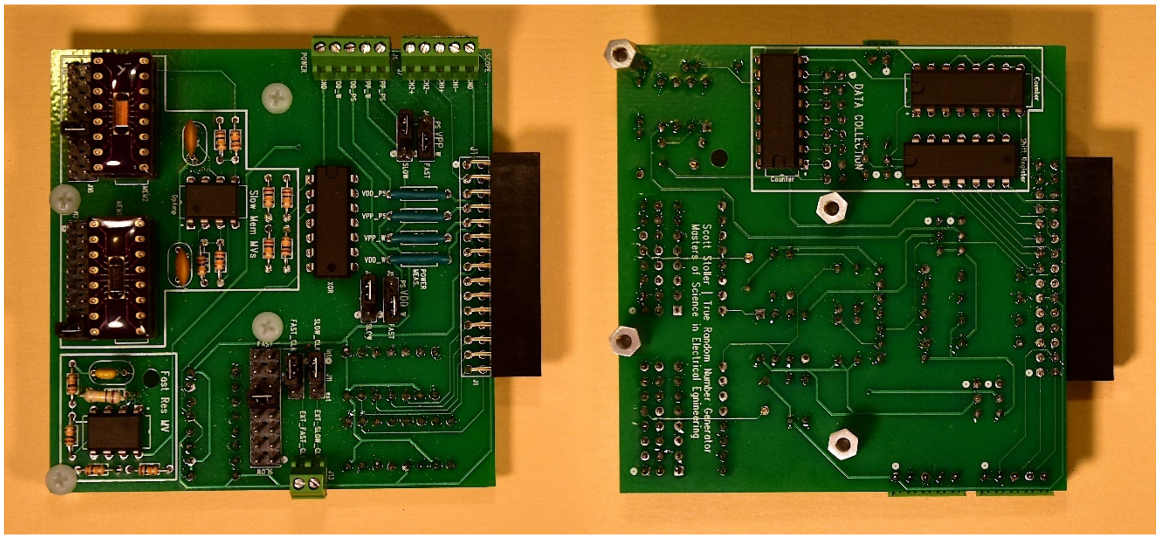


Figure 5.18 Final PCB front side (left) and back side (right)

5.7 Project Bill of Materials

The final bill of materials for PCB components is listed in table 5.2. A complete table of components with comments can be found in the Bill_Of_Materials.xlsx file located in “sstoller_thesis_final_writeup\PCB Design” on the Campbell Research group folder at BSU or Scholarworks. In some cases the part number of the part that was actually ordered (actual Digi-Key Part # column) was different from the part used on in the PCB123 software (PCB Part # column). The quantities below are for a single PCB. Total cost of components to for two PCBs was $71.95 with shipping, tax, and tariff from Digikey [33]. SDC memristors were purchased from Knowm [10].

Note that through-pin sockets were soldered in place of some components (various resistors and capacitors) so that components of different values could be swapped in to change the characteristics of the circuits (e.g. swap resistors and capacitors in the oscillator circuits to change the frequency of oscillation). The column titled “Actual Digi-Key Part #” is the actual part that was ordered and soldered to the board for final implementation. The column titled “PCB Part #” is the part number that was substituted for the final PCB design.

Table 5.2 List of all components used on the Final PCB Design

Component Identifier	**Description**	**Quantity**	**Actual Digi-Key Part #**	**PCB Part #**
J1	30 pin connector	1	S5568-ND	SAM1037-15-ND
J2, J3, J4, J5	3 pin jumper	4	SAM1099-03-ND	Same
Jumper	2 pin jumper connector	25	S9337-ND	Same
R1, R2, R3, R4	10 Ohm Resistors	4	10.0XBK-ND	Generic Resistor
J6	5 port wire terminal	1	1528-1975-ND	SAM1222-05-ND
J7	5 port wire terminal	1	1528-1975-ND	SAM1222-05-ND
S1, S2, S3, S4	Hex Standoff	5	1772-1887-ND	0.1160 Hole
U2	Shift register	1	296-8248-5-ND	Same
U3	Binary Counter	1	296-1599-5-ND	Same
U4	Binary Counter	1	296-1599-5-ND	Same
J10	16 pin jumper	1	SAM1097-08-ND	Same
J11	3 pin jumper	1	SAM1099-03-ND	Same
J12	3 pin jumper	1	SAM1099-03-ND	Same
J13	2 port wire terminal	1	1528-1974-ND	SAM1222-02-ND
U5	XOR Gate	1	296-4627-5-ND	Same
U6, U7	Dual opamp	2	296-34316-5-ND	296-1775-5-ND
Resistor	Resistor Pin Thing	16	ED1256-ND	Generic Resistor
MEM1	Memristor	1	2057-ICS-316-T-ND	Generic DIP

J17	16 pin jumper	1	SAM1097-08-ND	Same
MEM2	Memristor	1	2057-ICS-316-T-ND	Generic DIP
J18	16 pin jumper	1	SAM1097-08-ND	Same
R5, R6	10k resistors	2	10KEBK-ND	Generic Resistor
R8, R9	10k resistors	2	10KEBK-ND	Generic Resistor
R12, R13	10k resistors	2	10KEBK-ND	Generic Resistor
R11	10k resistors	1	10KEBK-ND	Generic Resistor
C1, C2, C3	1000pf capacitors	3		Generic Capacitor
R14, R15, R16, R17, R18, R19	10k resistors	6	10KEBK-ND	Generic Resistor

5.8 QuickStart Guide to Using the PCB

The previous sections (specifically sections on the screw terminals and jumper settings) thoroughly describe how to reconfigure and change certain options on the PCB. The purpose of this section is to describe how to set up and configure the Digilent AD2 and Waveforms software to capture data. It is essential to set up the Waveforms software properly in order to capture data.

5.8.1 Power Supply Setup

In order for the circuits to operate properly, the power supplies must all be set up. Both the V+ and V- supplies are used, as well as the W1 and W2 outputs. Jumpers J2, J3, J4, and J5 may be set up to apply V+ and V- to the slow oscillator power rails and set to apply W1 and W2 to the fast oscillator power rails. Both the Supplies window and the Wavegen window should be open. Up to +/- 5V may be applied to the positive and negative power supplies. It is best to keep the “Tracking” box checked so that the supplies both track together. The Wavegen waveform outputs should be set to “DC”. W1 should be set to a positive voltage and W2 should be set to a negative voltage. All supplies should be enabled in the software. In order to verify that power is being supplied and the oscillators are operating, it is useful to have the oscilloscope window open and the probes hooked up to the Scope screw terminals to probe different areas on the board at the VCC and VDD locations.

5.8.2 Logic Setup

The logic window is used to capture the data output from the circuit. The logic analyzer should be set up to collect data from a data bus. The data bus needs to be set up with data on DIO 7 (LSB) through DIO 0 (MSB). The clock for sampling the data should be DIO 15. Up to 100M data samples can be collected and saved using the Logging function of the Logic window. In order to set this up, the record mode must be used. An example workspace for data collection is saved and located on Boise State University Scholarworks and the Campbell Research group shared folder in the sstoller_thesis_final_writeup\Novel_TRNG\Waveforms directory.

CHAPTER SIX: FINAL CIRCUIT CHARACTERIZATION

6.1 PCB and Hardware Verifications

6.1.1 Verification of PCB

Once the finished PCBs were received a quick validation was performed. A multimeter was used to ensure that proper connections were made, and that the final PCB product matched the initial schematic netlist. Throughout the soldering process, care was taken to check each net and verify that the solder joint was properly soldered and was not in contact with any surrounding nets.

6.1.2 Verification of circuits

Once soldering was complete, the PCB was plugged in to the Digilent AD2 and the board was powered on. With the data collection circuit disconnected from power by turning off the power supplies and using the wavegen only, the oscillators were tested individually and confirmed to work properly. Finally, the data collection circuit was enabled and verified to be working properly.

6.2 Power Measurements

100 Ohm resistors were placed in series with both the V+ and V-, and the W1 and W2 power supplies for the circuit in order to measure the amount of power consumed by different parts of the circuit. By measuring the voltage across the resistors, current can be calculated using Ohms law. Supply voltage is known, and power can be calculated using

$$Power = V_{SUPPLY} * V_{RESISTOR} / R_{RESISTOR.} \tag{6.1}$$

Power for each part of the circuit was characterized by isolating the different portions of the circuit using the power supply jumpers on the PCB. Table 6.1 shows the power consumption of each part of the circuit.

Table 6.1 Power Measurements of the TNRG circuit

Standby	Supply Voltage	Current (Ave)	Power
Oscillator VPP	2.7 V	11.885 mA	32.09 mW
Oscillator VDD	-2.7 V	2.4134 mA	6.52 mW
Data Collection VPP	2.7 V	1.7071 mA	4.61 mW

Average power consumption was measured for the memristor multivibrator circuit and was calculated based on equation 6.1. Current was measured by placing a 100 Ω resistor in series with the positive (VPP) and negative (VPP) supplies for a single oscillator circuit. A tungsten memristor was used with 100 nF capacitor yielding an oscillation frequency of about 2.5 kHz. Circuit oscillations are obvious in the current consumption of the negative power supply but not in the positive supply. Power measurements were taken with VPP at 2.7V and VDD at -2.7V (the minimum supply voltages for the op-amp used).

Figure 6.1 shows the power supply measurement of the oscillator circuit. It is interesting to note that the positive supply current is the same regardless of the oscillation state of the multivibrator circuit, however the negative supply is variable with the oscillation of the multivibrator circuit. This behavior was observed with only the op-amp connected to the power supply and changing the input voltage to the op-amp to cause the output voltage to switch from one voltage supply rail to the other. It was concluded that this behavior is due to the output stage design of the op-amp and not the circuit design of the multivibrator.

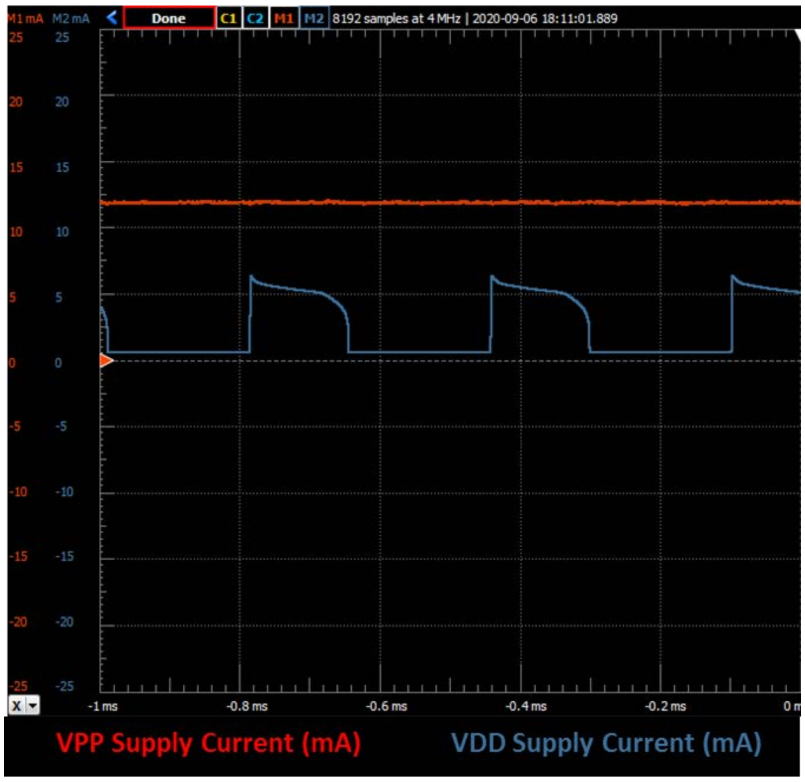


Figure 6.1 VPP and VDD Current measurements in W-SDC memristor oscillator

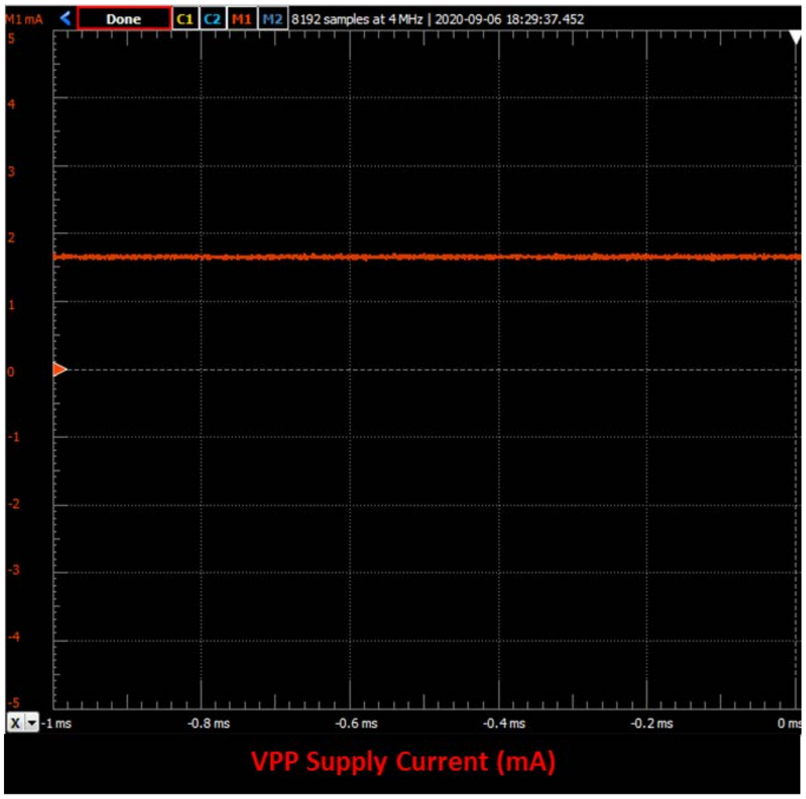


Figure 6.2 VPP current measurement in data collection circuit

Power of the data collection circuit was also measured using the same method of placing a 100 Ω resistor in series with the VPP power supply. Figure 6.2 shows the power supply measurement of the data collection circuit.

Power consumption of both circuits is shown in table 6.1. Total circuit power consumption will be the sum of the oscillator VPP and VDD, tripled because two memristor oscillators are used and one resistor oscillator is used. Finally, the data collection power consumption must also be added. This yields a total estimated power consumption of about 120.44mW. Most of this power consumption is driven by the use of op-amps in the circuit.

6.3 Randomness Characterization of Final Circuit

The final circuit was tested for randomness. 100 million bits were collected over the course of several runs of the Digilent Waveforms logic analyzer data capture function. The data was post-processed using only von Neumann whitening. The NIST Statistical Test Suite was used to test the randomness of 100 sequences each of length 1 million bits.

Table 6.2 contains the NIST STS test results for the memristor multivibrator TRNG. All tests passed except for the runs test, which failed with a proportion of 82/100, and the Approximate Entropy test, which barely failed with a proportion of 94/100. A proportion of 96/100 or better was needed to pass each test. The final circuit characterization shows that the TRNG is passing 13 of 15 NIST STS randomness tests. A 96/100 proportion or greater is required to pass the randomness tests. The Runs test only passesd 82/100 sequences. The Approximate Entropy test barely failed the randomness tests with a proportion of 94/100 sequence. The runs test also failed with a P-Value of 0.

6.3.1 Data Collection Script

Campbell Research group folder at BSU or Scholarworks: sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

6.3.2 Results Location

Campbell Research group folder at BSU or Scholarworks: sstoller_thesis_final_writeup\Novel_TRNG\Final_Data_Collection

6.3.3 Results

Table 6.2 Results of Memristor Multivibrator TRNG

Test	**P-Value**		**Proportion**		**Pass/Fail**
Frequency	0.102526		98/100		PASS
BlockFrequency	0.319084		98/100		PASS
CumulativeSums	0.122325, 0.001112		194/200		PASS
Runs	0	*	82/100	*	FAIL
LongestRun	0.202268		100/100		PASS
Rank	0.153763		98/100		PASS
FFT	0.319084		97/100		PASS
NonOverlappingTemplate	-		14614/14800		PASS
OverlappingTemplate	0.494392		98/100		PASS
Universal	0.350485		100/100		PASS
ApproximateEntropy	0.090936		94/100	*	FAIL
RandomExcursions	-		424/432		PASS
RandomExcursionsVariant	-		962/972		PASS
Serial	0.739918, 0.595549		195/200		PASS
LinearComplexity	0.304126		100/100		PASS

6.4 Randomness Characterization of Final Circuit Seeding a PRNG

It is often common to find a TRNG seeding a PRNG as a method to improve the output of quality, speed, and efficiency of a TRNG. In this case an LCG PRNG proposed by Lewis, Goodman, and Miller was used [34]. The LCG that was used to generate PRNG is

$$I = 16807 * I \,\%\, 2147483647. \tag{6.2}$$

To seed the PRNG 32 bits were read from a binary bitstream of random data. The value of I was set to this value. The PRNG equation runs for 128 iterations. Each iteration the 16 least significant bits of I are written to the output file. After 128 iterations I is re-seeded with the next 32 bits from the input file. For each 32 bits (4 bytes) of input data, 256 bytes of output are generated.

It is worth noting again that this output is not considered to be truly random and is not suitable for use in cryptographic applications.

6.4.1 Data Collection Script

Campbell Research group folder at BSU or Scholarworks:

datastream_to_datastream_prng.pl

6.4.2 Results Location

Campbell Research group folder at BSU or Scholarworks:

sstoller_thesis_final_writeup\Novel_TRNG\Final_Data_Collection\RNG_Seeding _PRNG

6.4.3 Results

Table 6.3 Results of Memristor Multivibrator TRNG seeding PRNG

Test	P-Value		Porportion		Pass/Fail
Frequency	0.275709		100/100		PASS
BlockFrequency	0.494392		98/100		PASS
CumulativeSums	0.096578, 0.040108		199/200		PASS
Runs	0.191687		98/100		PASS
LongestRun	0.678686		98/100		PASS
ank	0.798139		97/100		PASS
FFT	0.122325		98/100		PASS
NonOverlappingTemplate	-		14665/14800		PASS
OverlappingTemplate	0.153763		100/100		PASS
Universal	0.366918		98/100		PASS
ApproximateEntropy	0.574903		99/100		PASS
RandomExcursions	-		515/520		PASS
RandomExcursionsVariant	-		1166/1170		PASS
Serial	0.213309, 0.145326		195/200		PASS
LinearComplexity	0.739918		100/100		PASS

6.5 Comparison to other RNGs

Table 6.4 shows a comparison of all RNGs tested compared to the TRNG designed and implemented in this Thesis project. A pass rate of 96% or better was required to pass. Failing tests are shown with bold text.

Table 6.5 shows a comparison of the number of sequences tested for each RNG, whether debiasing was necessary for each RNG, the approximate time that it took to collect the random data sequences for each RNG.

Table 6.4 Results Comparison of all RNGs

Test	RANDU [24] (PRNG)	Digilent AD2 (PRNG)	uRandom [4] (TNRG + PRNG)	RDRAND [3] (TNRG + PRNG)	Stoller RNG (TRNG)	Jiang et al. [26] (TRNG)*	Rai et al. [27] (TRNG)*
Frequency	**0%**	97%	99%	100%	98%	**91%**	100%
BlockFrequency	**0%**	99%	100%	99%	98%	100%	100%
CumulativeSums	**0%**	97%	99%	100%	97%	**95%**	100%
Runs	**0%**	100%	99%	100%	**82%**	**82%**	100%
LongestRun	**14%**	100%	99%	97%	100%	**91%**	100%
Rank	**0%**	98%	98%	98%	98%	100%	100%
FFT	**0%**	99%	100%	99%	97%	100%	100%
NonOverlappingTemplate	**53%**	99%	99%	99%	99%	**95%**	99%
OverlappingTemplate	**0%**	99%	99%	97%	98%	100%	100%
Universal	**45%**	100%	100%	**95%**	100%	**91%**	100%
ApproximateEntropy	**0%**	97%	99%	99%	**94%**	**91%**	100%
RandomExcursions	**0%**	99%	99%	99%	98%	98%	96%
RandomExcursionsVariant	**0%**	100%	99%	99%	99%	100%	98%

Serial	**0%**	100%	98%	99%	98%	**91%**	96%
LinearComplexity	100%	99%	100%	100%	100%	100%	100%

Table 6.5 Additional Comparisons of all RNGs

	RANDU [24] (PRNG)	Digilent AD2 (PRNG)	uRandom [4] (TNRG + PRNG)	RDRAND [3] (TNRG + PRNG)	Stoller RNG (TRNG)	Jiang et al. [26] (TRNG)*	Rai et al. [27] (TRNG)*
Number of sequences assessed (length 100M bits)	100	100	100	100	100	11	12
Debiasing Necessary?	No	No	No	No	Yes	Yes	Yes
Time to collect random samples	2 seconds	2 minutes	2 seconds	2 seconds	8 hours	8 hours	8 hours

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

7.1.1 A TRNG was successfully designed and implemented

The conclusion of this research shows that memristors may be used in a self-oscillating multivibrator circuit to successfully generate truly random numbers.

7.1.2 Importance of Power Supply Isolation

Throughout the extensive development and testing of the circuit it was found that one primary factor that can lead to non-random output from the TRNG is supply noise. Power supply ripple can be observed due to the instantaneous high current observed when the multivibrator circuit switches from one mono-stable state to the other. This supply ripple from one oscillator switching can lead to other oscillators that share the same supply switching at the same time. Figures 7.1 and 7.2 show this phenomenon occurring between two multivibrator circuits.

In Figure 7.1 persistent samples of the output of two different memristor multivibrator oscillators which normally oscillate at different frequencies are shown correlating with one another because of a 100Ω resistor placed in series with the power supply. The small power supply voltage droop caused by the 100Ω resistor is enough to cause the oscillators to often switch together with each other.

Figure 7.2 shows the exact same two oscillators with no resistor in series with the power supply. It can be clearly seen that there is practically no visible correlation between the two oscillators in the fact that the second oscillator does not consistently switch when the first oscillator switches.

In both images, the oscillators are shown vertically separated. The scope is always triggered on the falling edge of the top oscillator. Also note the reduced peak-to-peak swing of both oscillators when the 100Ω resistors are placed in series with the power supplies due to the IR droop caused by the current draw of the op-amp.

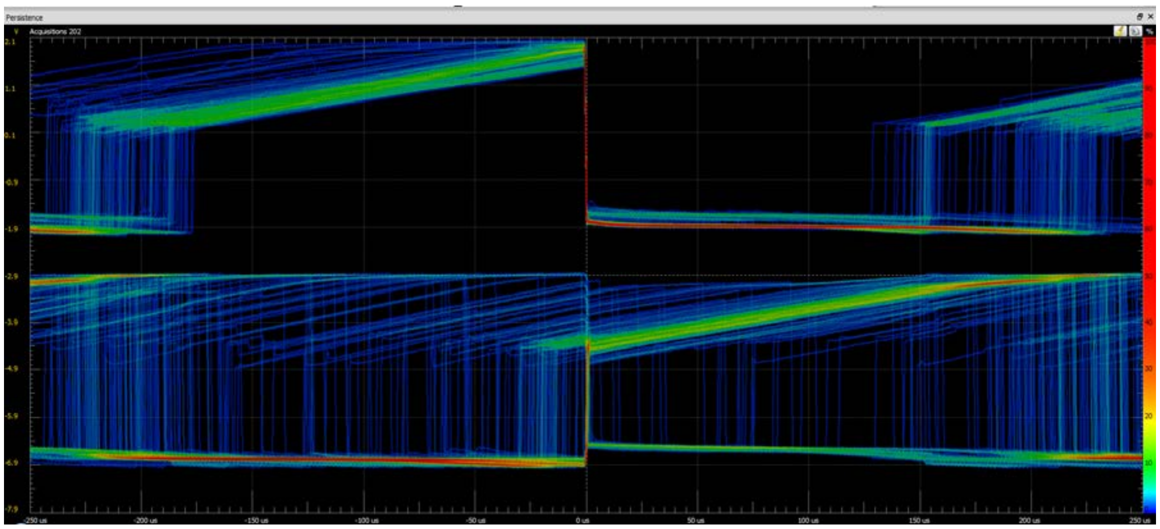


Figure 7.1 Persistence plot with poor power supply for both oscillators

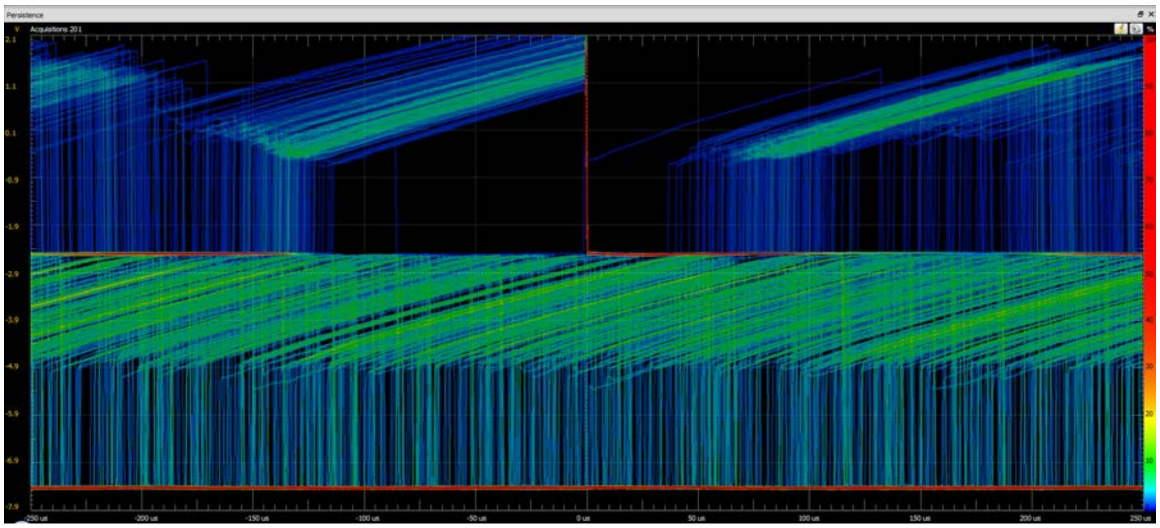


Figure 7.2 Persistence plot with clean supplies for both oscillators

This issue was solved on my final circuit design by using the Waveforms outputs to power the slow oscillators and the AD2 V+ and V- power supplies to power the fast oscillator for the TRNG. This minimized the chances of any correlation between the slow and fast oscillators and significantly improved the randomness of the TRNG.

This exact same problem is present on other TRNG circuits that utilize oscillators to capture entropy. For these designs, careful consideration must be taken to ensure a clean power supply, especially when multiple supplies are not available, or when implementing the design as an integrated circuit. While supply noise could be another source of entropy (for example a TRNG implemented on an integrated circuit with a CPU), sensitivity to power noise can also be a liability, especially if the noise is repetitive. It may be possible for an attacker to compromise the TRNG through a side-channel attack by either injecting noise on the power supply that may cause the oscillator to oscillate in a non-random fashion, or by measuring the power supply noise and using this information to predict the output of the TRNG.

The concerns of power supply noise apply to any oscillator or set of oscillators used to implement a TRNG, however not all types of oscillators may be impacted as severely as a multivibrator circuit. A multivibrator circuit may be impacted more severely than other types of oscillators specifically because there is a long RC time delay that causes the circuit to switch from one state to the other state. When the RC voltage at the negative input of the op-amp is close to the switching threshold (0.5 V_{SUPPLY}) input at the positive input of the op-amp, a slight amount of supply noise can cause the threshold to droop and prematurely cause a transition to the other state.

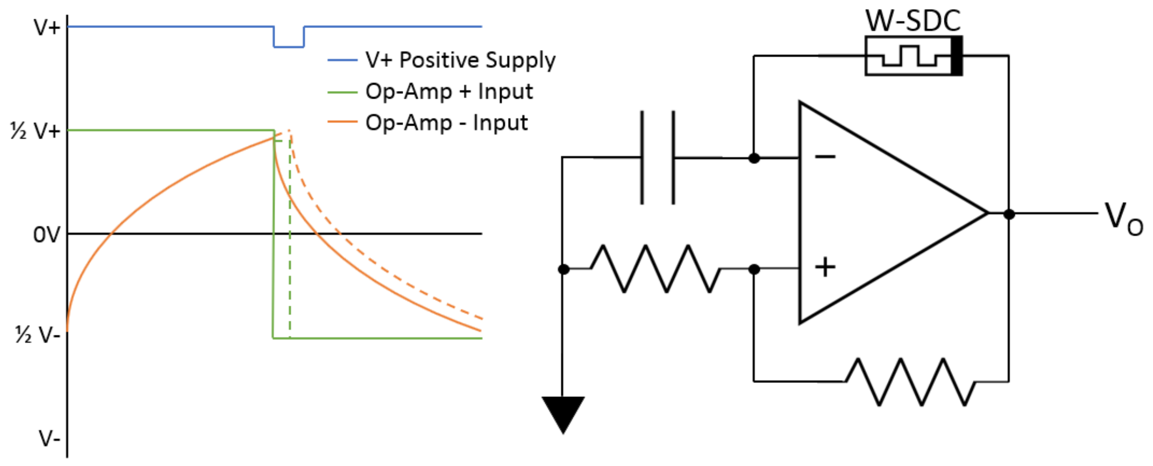


Figure 7.3 Power supply noise impact on Multivibrator

Figure 7.3 shows how voltage noise on the positive power supply can cause the output of the multivibrator oscillator to switch early. Power supply noise created by one multivibrator can inject noise on another oscillator. In the worst case this can cause the two oscillators to sync perfectly.

7.1.3 SPICE Simulations Alone are not Sufficient for Characterizing a TRNG

For many reasons such as using an ideal power supply it is possible for SPICE simulations of TRNGs to output a truly random string of numbers. In simulations there are many factors that are ideal, from the noise in the circuit elements themselves (noise generation in a SPICE simulation is an example usage of a PRNG), to the ideality of power supplies and lack of other types of interactions, deterministic noise, and distortions that are present in a non-ideal circuit. It was observed in the literature survey that often a TRNG implementation will perform flawlessly in simulation but will fail to pass the NIST randomness tests once implemented in hardware. In addition, it is very inefficient to run analog simulations that capture an output of sufficient length for testing. When possible in this paper, 100 samples of length 1 million bits were tested (total length 100 million bits). This requires a lot of time or a really powerful machine to simulate an analog circuit long enough to generate 100 million bits of output.

7.2 Future Work

7.2.1 Potential Improvements to Final RNG Circuit

There are several potential improvements to the final RNG circuit. One optimization would be to use a ring oscillator with an RC delay element instead of a multivibrator for the fast oscillator. This has many advantages. The main advantage is that an op-amp is not required for a ring oscillator. There are many benefits of not having an op-amp such as simplicity of the circuit design and most likely a faster output slew rate (output will be closer to a square wave and not a sawtooth or triangle wave at high frequencies that you get with an op-amp). A ring oscillator design is also probably lower power than a multivibrator design that uses an op-amp.

Another potential improvement in the design would be to provide better power supply isolation between the two memristor multivibrators. It is not as much of a concern for one of the memristor multivibrators to interfere with the other, as correlation between these two oscillators is not as detrimental to the randomness of the design as a correlation between either of these oscillators and the fast resistor multivibrator. There are a few options to fix this. One method may be to use a device that has multiple power supply rails. Another option would be to design in proper power supply isolation between the two multivibrators. In order to accomplish this each multivibrator needs to have its own discrete op-amp integrated circuit. Both solutions increase design complexity. There is a large penalty for doing this on a PCB, but this would not be as complicated if implemented in silicon as an integrated circuit.

7.2.2 Improvements to Data Collection Circuit Design

The purpose of the data collection circuit is to increase the amount of data that can be collected by taking a serial stream of data and parallelizing it. The data collection circuit design has a design flaw in that there is not a latch to hold the output from the serial to parallel data converter. Instead, the parallel output from the shift register continues to shift data throughout the entire clock period of the slow clock that toggles data into the AD2.

There are several risks as a result of this design oversight. The first issue is that the output from the shift register continues to toggle as the Digilent AD2 board is sampling the data. In this case, it may be possible that some bits are missed by the data collection circuit, even if the Nyquist sample rate is used. This could also lead to a violation of setup or hold times for the AD2. Figure 7.4 shows how some bits may be missed if data clock has jitter that causes it to operate at a significantly different frequency than the AD2 sample clock. It can be seen that some bits are wasted in the 3rd sample of the AD2 clock as a result of the data clock running faster than the Nyquist frequency. Similarly, some bits are wasted in the 4th sample of the AD2 clock as a result of the AD2 clock not sampling exactly on the rising edge of the data clock.

It is unlikely for the randomness of the output bitstream to change significantly as a result of this phenomenon. If anything, it should increase the randomness of the resultant datastream as some bits or samples will randomly be thrown out. In order to test this, one could oversample the output at a very high frequency and test that sequence for randomness. Then take the oversampled output and write software to convert it to a lower sample rate that and compare the results. This is a little difficult to test because the under sampled results will generate a shorter sequence of bits.

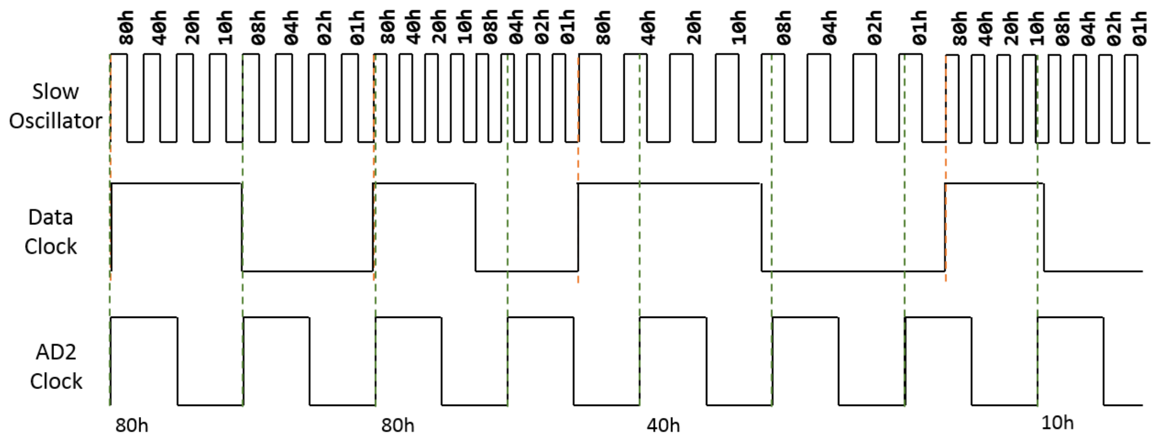


Figure 7.4 Missed samples with current data collection circuit design

In addition, a latch can be timed to latch data on the falling edges of the slow oscillator clock in order to improve data integrity by giving sufficient margin for setup and hold times. In the current circuit there is not much thought given to the setup and hold times of the components in the circuit. At slow data collection frequencies this is not a large problem, but in order to make the circuit more robust, reliable, and have the ability to operate at a higher speed it would be prudent to spend some additional time here.

The proper way to implement this circuit is to use a x8 latch circuit (such as the Texas Instruments TPIC6B237) could be attached to the outputs of the serial-in-parallel-out shift register. This circuit would sample the output of the shift register exactly at the rising edge of the clock and hold the same value until the next rising edge.

7.2.3 Characterization of the Circuit with Different Types of Devices

More potential future work would be to characterize the TRNG circuit with different types of devices (memristors or other devices). All of the characterization for this thesis project was done on a W-Based device. There are many other types of memristor devices readily available (SDC devices such as C, Sn, Cr-based devices), among many others. Some devices may not be as tolerant to high voltages and currents as the W-based devices used in this thesis project. The basic circuit design should still suffice, but devices will need to operate at a lower power supply voltage or have a resistor in series with the memristor to limit current. A resistor could also be used in the multivibrator oscillator in place of a memristor.

7.2.4 Fabrication of the Circuit as a Custom IC Design

Future work could be done to fabricate this circuit as a fully custom designed chip. In order to fabricate this design as a chip several things must be considered:

1. Op-amp design and linearity is not crucial to the operation of the circuit, however design needs to be good enough to eliminate significant amounts of ringing of other types of distortion. There is no need to design a complex and low-distortion op-amp output stage. In fact, distortion could add or amplify noise in this type of circuit.
2. An additional benefit of using a custom-design op-amp is the fact that the design can be optimized for low power. For example, a design that supports a lower supply voltage can be implemented. In addition, rail-to-rail voltage output swing could be sacrificed to improve power efficiency in the op-amp circuit because it is not needed for this circuit to function properly.
3. Power supply design is very crucial to successfully designing the circuit. If the same voltage source is to be used for the supplies for both the slow and fast oscillators, then careful attention must be payed to the design of regulator circuits

to ensure that there is no noise or interference between the two supplies. Alternatively, two different voltage sources could be provided to the chip or generated on the chip.

7.2.5 Alternative Multivibrator Circuit Topologies

Figure 7.5 shows three additional alternative circuit topologies. In reality, there are many different potential topologies as any resistor or combination of resistors could be replaced with memristors.

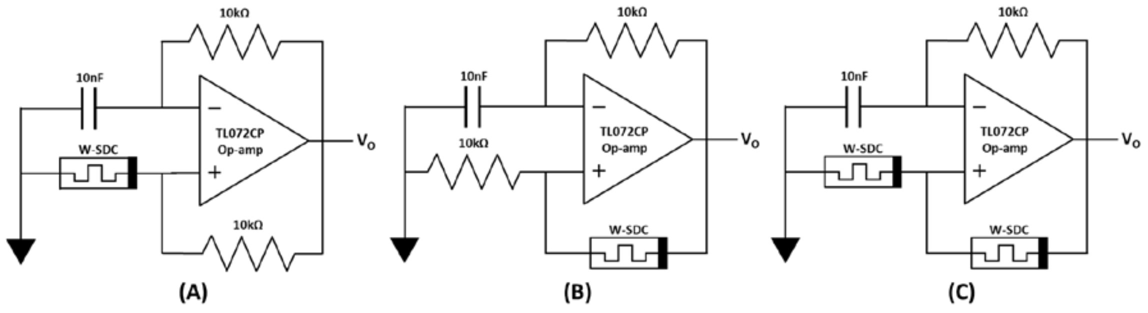


Figure 7.5 Three alternative circuit topologies for TRNG

There are several potential advantages of replacing the resistors attached to the noninverting input of the op-amp as opposed to the circuit topology used in the TRNG presented in this paper.

One advantage is that less voltage is applied over one or both memristors when they are on the noninverting input of the op-amp. When the multivibrator switches, the capacitor is further from ground and leads to a higher potential than simply V_O over the memristor. In the designs shown in figure 7.5, the voltage is split between the memristors (each memristor will see about ½ V_O), or current is limited if a memristor plus resistor is used. This could allow or necessitate the usage of a higher supply voltage, which could be an advantage or disadvantage. The circuits (A) or (B) would capture entropy in a very similar manner to the design proposed by Jiang et al.

Another potential advantage is that as more memristor devices are added to the final circuit, there is the potential to capture more noise.

One thing that must be carefully considered during circuit design is the frequency at which the circuit operates. Resistor and capacitor values mush be chosen such that the circuit operates at a frequency that is similar to the typical transition delay between states of the memristor. This is crucial to capture entropy of the circuit. If the circuit is very slow or very fast, then the memrsitor will always be in the same state by the time the capacitor is charged or discharged and there will be less entropy generated by the circuit.

Another thing which must be considered in this circuit is that the previous state of the circuit (the voltage on the capacitor) can impact the next state of the multivibrator. If the previous state switched quickly because the memristor changed voltage quickly, then the voltage on the capacitor will be lower, which will make the transition to the state after quicker too. This could cause a correlation (and thus a loss of randomness) from one oscillation of the multivibrator to the next.

REFERENCES

[1] C. N. Landon, R. G. Mende, and S. Sisodiya, “Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system,” 24-Mar-1998.

[2] Y. Wang, W.-K. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, “Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints,” 2012 IEEE Symposium on Security and Privacy, 2012.

[3] G. Cox, C. Dike, and D. J. Johnston, “Intels digital random number generator (DRNG),” 2011 IEEE Hot Chips 23 Symposium (HCS), 2011.

[4] “urandom(4) - Linux man page,” urandom(4): kernel random number source devices - Linux man page. [Online]. Available: https://linux.die.net/man/4/urandom. [Accessed: 23-Sep-2020].

[5] L. Chua, “Memristor-The missing circuit element,” IEEE Transactions on Circuit Theory, vol. 18, no. 5, pp. 507–519, 1971.

[6] L. Chua, “Everything You Wish to Know About Memristors But Are Afraid to Ask,” Radioengineering, vol. 24, no. 2, pp. 319–368, 2015.

[7] L. O. Chua, “The Fourth Element,” Proceedings of the IEEE, vol. 100, no. 6, pp. 1920–1927, 2012.

[8] L. Chua, “If it’s pinched it’s a memristor,” Semiconductor Science and Technology, vol. 29, no. 10, p. 104001, 2014.

[9] K. A. Campbell, “Self-directed channel memristor for high temperature operation,” Microelectronics Journal, vol. 59, pp. 10–14, 2017.

[10] "M SDC Memristor 8 Discrete 16 DIP," Knowm Inc. [Online]. Available: https://knowm.com/collections/frontpage/products/m-sdc-memristor-8-discrete-16-dip. [Accessed: 23-Sep-2020].

[11] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[12] I. T. L. Computer Security Division, "NIST SP 800-22: Documentation and Software - Random Bit Generation: CSRC," CSRC. [Online]. Available: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software. [Accessed: 23-Sep-2020].

[13] "Welcome to Box.org!," Oracle VM VirtualBox. [Online]. Available: https://www.virtualbox.org/. [Accessed: 23-Sep-2020].

[14] "Mounting VirtualBox shared folders on Ubuntu Server 16.04 LTS," Gist. [Online]. Available: https://gist.github.com/estorgio/1d679f962e8209f8a9232f7593683265. [Accessed: 23-Sep-2020].

[15] M. Dabacan, "Analog Discovery 2 Reference Manual," Analog Discovery 2 Reference Manual [Digilent Documentation]. [Online]. Available: https://reference.digilentinc.com/reference/instrumentation/analog-discovery-2/reference-manual. [Accessed: 23-Sep-2020].

[16] "WaveForms - Digilent" Digilent. [Online]. Available: https://store.digilentinc.com/waveforms-download-only/. [Accessed: 23-Sep-2020].

[17] H. Nyquist, "Certain topics in telegraph transmission theory," Proceedings of the IEEE, vol. 90, no. 2, pp. 280–305, 2002.

[18] J. von Neumann, "Various Techniques Used in Connection with Random Digits," in Monte Carlo Method, vol. 12, Washing, DC: US Government Printing Office, 1951, pp. 36–38.

[19] P. Crowley, “Generating random binary data from Geiger counters,” ciphergoth.org: Generating random binary data from Geiger counters. [Online].

[20] M. Hamburg, P. Kocher, and M. E. Marson, Analysis of Intel's Ivy Bridge Digital random Number Generator, Cryptography Research, Inc., 2012.

[21] “Intel® Digital Random Number Generator (DRNG) Software Implementation...,” Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html. [Accessed: 23-Sep-2020].

[22] “The DRNG Library and Manual,” Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/the-drng-library-and-manual.html. [Accessed: 23-Sep-2020].

[23] IBM application package: System/360 scientific subroutine package; version III; programmers manual; program number 360A-CM-03X. White Plains, NY: IBM Technical Publications Dept., 1968. P 77.

[24] G. Marsaglia, “Random Numbers Fall Mainly In The Planes,” Proceedings of the National Academy of Sciences, vol. 61, no. 1, pp. 25–28, 1968.

[25] random(4) - Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man4/random.4.html. [Accessed: 23-Sep-2020].

[26] H. Jiang, D. Belkin, S. E. Savel’Ev, S. Lin, Z. Wang, Y. Li, S. Joshi, R. Midya, C. Li, M. Rao, M. Barnell, Q. Wu, J. J. Yang, and Q. Xia, “A novel true random number generator based on a stochastic diffusive memristor,” Nature Communications, vol. 8, no. 1, 2017.

[27] V. K. Rai, S. Tripathy, and J. Mathew, “Memristor based Random Number Generator: Architectures and Evaluation,” Procedia Computer Science, vol. 125, pp. 576–583, 2018.

[28] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams. “The missing memristor found,” Nature, 2008.

[29] S. Robson, “A Ring Oscillator Based Truly Random Number Generator,” 2013. [Online]. Available: https://uwspace.uwaterloo.ca/bitstream/handle/10012/7911/Robson_Stewart.pdf?sequence=1. [Accessed: 23-Sep-2020].

[30] S. P. Adhikari, M. P. Sah, H. Kim, and L. O. Chua, “Three Fingerprints of Memristor,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 60, no. 11, pp. 3008–3021, 2013.

[31] “Sunstone Circuits Printed Circuit Boards,” Sunstone Circuits Printed Circuit Boards | Sunstone.com. [Online]. Available: https://www.sunstone.com/. [Accessed: 23-Sep-2020].

[32] “PCB123® ,” PCB123LP. [Online]. Available: https://www.sunstone.com/pcb123-lp. [Accessed: 23-Sep-2020].

[33] “World's Largest Selection of Electronic Components Available for Immediate Shipment!®,” DigiKey Electronics - Electronic Components Distributor. [Online]. Available: https://www.digikey.com/. [Accessed: 23-Sep-2020].

[34] P. A. A. Lewis, A. S. Goodman, and J. M. Miller, “A pseudo-random number generator for the System/360,” IBM Systems Journal, vol. 8, no. 2, pp. 136–146, 1969.

APPENDIX A

This section of the appendix contains the Perl scripts used to aide in data collection.

csv_bin_to_datastream.pl

The purpose of this script is to convert a CSV file output with a single bit binary output per line from the Digilent AD2 to a binary bitstream file that can be analyzed by the STS assess application. This script has the option to whiten the output data by XORing with the RANDU PRNG.

Location

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

Usage

Perl csv_bin_to_datastream.pl <infiles> <outfile.bin> <whitening>

Code

```
#!/usr/bin/perl
use strict;
use warnings;

die "Not enough arguments. Script requires a list of input files and one output file
and a 0/1 toggle for RANDU whitening." if @ARGV < 3;

my $outfile = $ARGV[-2];
my $whitening = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -2 );

die "Error: Specificed output file is not .bin file" unless $outfile =~ '.bin';

# fix issue where csv data file is overwritten when I forget to give a unique output
file name.
foreach my $infile (@infiles){
	die "ERROR: Infile is same as outfile.\n" if $infile eq $outfile;
}

open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

# Variables for RANDU whitening
my $seed = 31;
my $b = 65539;
my $mod = 4294967296;

# Variables for manipulating random sequence
my $num1 = 0;
my $count = 0;
my $out = 0x00;
my $pos = 0;
my $numlines = 0;
my $fc = 0;

# variables for deserializing stuff
my $b0 = 0;
my $b1 = 0;
my $c = 0;

# Iterate over all input files and convert to binary and single output file
foreach my $infile (@infiles)
{
	open(my $FI, '<', $infile) or die "Could not open infile $infile $!";

	while(my $line = <$FI>) {
		last if $line =~ ',Data';
	}

	$fc++;
	print ("Working on file $infile\n");
	while(my $line = <$FI>) {
		if($line =~ ',1') {
			$b0 = 1;
		}
		else {
			$b0 = 0;
		}
		$out = $out + ($b0 << $pos);
		$pos = $pos + 1;

		# If an entire 8-bit number has been generated print it out and count
the 1's and 0's
		if($pos == 8) {
			if($whitening) {  # Whiten sequence with RANDU?
				$seed = $seed*$b % $mod;  # RANDU whitening
				$out = $out ^ ($seed & 0xFF);  # RANDU whitening
			}
			print $FO pack('C', $out);
			# printf("0x%2x\n",$out);
			$count++;
			if(($out & 0x01) == 0x01){$num1++;}
			if(($out & 0x02) == 0x02){$num1++;}
			if(($out & 0x04) == 0x04){$num1++;}
			if(($out & 0x08) == 0x08){$num1++;}
			if(($out & 0x10) == 0x10){$num1++;}
			if(($out & 0x20) == 0x20){$num1++;}
			if(($out & 0x40) == 0x40){$num1++;}
			if(($out & 0x80) == 0x80){$num1++;}
			$pos = 0;
			$out = 0;
		}
	}
	$numlines += $.;
	print "$. lines Parsed!\n";
	close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
if($et == 0) {$et = 1};  # Make sure elapsed time is not 0
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all files in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";

close $FO;

exit(0);
```

csv_to_datastream.pl

The purpose of this script is to convert a CSV file output with 8 bits (1 byte) of data per line from the Digilent AD2 to a binary bitstream file that can be analyzed by the STS assess application. This script has the option to whiten the output data by XORing with the RANDU PRNG.

Location

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

Usage

Perl csv_to_datastream.pl <infiles> <outfile.bin> <whitening>

Code

```
#!/usr/bin/perl
use strict;
use warnings;

die "Not enough arguments. Script requires a list of input files and one output file
and a 0/1 toggle for RANDU whitening." if @ARGV < 3;

my $outfile = $ARGV[-2];
my $whitening = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -2 );

die "Error: Specificed output file is not .bin file" unless $outfile =~ '.bin';

# fix issue where csv data file is overwritten when I forget to give a unique output
file name.
foreach my $infile (@infiles){
	die "ERROR: Infile is same as outfile.\n" if $infile eq $outfile;
}
open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

# Variables for RANDU whitening
my $seed = 31;
my $b = 65539;
my $mod = 4294967296;

# Variables for manipulating random sequence
my $clk = 1;
my $data = 0xFF;
my $num1 = 0;
my $count = 0;
my $numlines = 0;
my $fc = 0;
my $out = "0xQQ";
```

```
foreach my $infile (@infiles)
{
	open(my $FI, '<', $infile) or die "Could not open infile $infile $!";
	$fc++;
	print ("Working on file $infile\n");
	while(my $line = <$FI>) {

		if($line =~ /,h(\w\w)/) {
			$out = hex($1);
			if($whitening) {  # Whiten sequence with RANDU?
				$seed = $seed*$b % $mod;  # RANDU whitening
				$out = $out ^ ($seed & 0xFF);  # RANDU whitening
			}
			print $FO pack('C', $out);
			$count++;
			if(($out & 0x01) == 0x01){$num1++;}
			if(($out & 0x02) == 0x02){$num1++;}
			if(($out & 0x04) == 0x04){$num1++;}
			if(($out & 0x08) == 0x08){$num1++;}
			if(($out & 0x10) == 0x10){$num1++;}
			if(($out & 0x20) == 0x20){$num1++;}
			if(($out & 0x40) == 0x40){$num1++;}
			if(($out & 0x80) == 0x80){$num1++;}
		}
	}
	$numlines += $.;
	print "$. lines Parsed!\n";
	close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
if($et == 0) {$et = 1};  # Make sure elapsed time is not 0
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all $fc file(s) in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";

close $FO;

exit(0);
```

csv_to_datastream_von_neumann.pl

The purpose of this script is to convert a CSV file output from the Digilent AD2 to a binary bitstream file that can be analyzed by the STS assess application. This script also applies von Neumann debiasing to the output. This script has the option to whiten the output data by XORing with the RANDU PRNG.

Location

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

Usage

Perl csv _to_datastream_von_neumann.pl <infiles> <outfile.bin> <whitening>

Code

```
#!/usr/bin/perl
use strict;
use warnings;

die "Not enough arguments. Script requires a list of input files and one output file
and a 0/1 toggle for RANDU whitening." if @ARGV < 3;

my $outfile = $ARGV[-2];
my $whitening = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -2 );

die "Error: Specificed output file is not .bin file" unless $outfile =~ '.bin';

# fix issue where csv data file is overwritten when I forget to give a unique output
file name.
foreach my $infile (@infiles){
       die "ERROR: Infile is same as outfile.\n" if $infile eq $outfile;
}

open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

# Variables for RANDU whitening
my $seed = 31;
my $b = 65539;
my $mod = 4294967296;

# Variables for manipulating random sequence
my $num1 = 0;
my $count = 0;
my $out = 0x00;
my $pos = 0;
my $numlines = 0;
my $fc = 0;
```

```
# Iterate over all input files and convert to binary and single output file
foreach my $infile (@infiles)
{
	open(my $FI, '<', $infile) or die "Could not open infile $infile $!";
	$fc++;
	print ("Working on file $infile\n");
	while(my $line = <$FI>) {
		if($line =~ /,h(\w\w)/) {
			# Look at every pair of bits (4 pairs per byte)
			for(my $bit = 0; $bit < 8; $bit += 2) {
				my $c = chr(hex($1));
				my $b0 = (ord($c) >> $bit & 0x01);
				my $b1 = (ord($c) >> ($bit + 1) & 0x01);
				# only put a bit in the sequence in B0 and B1 are 
different
				if($b0 != $b1) {
					$out = $out + ($b0 << $pos);
					$pos = $pos + 1;
				}

				# If an entire 8-bit number has been generated print it 
out and count the 1's and 0's
				if($pos == 8) {
					if($whitening) {  # Whiten sequence with RANDU?
						$seed = $seed*$b % $mod;  # RANDU whitening
						$out = $out ^ ($seed & 0xFF);  # RANDU 
whitening
					}
					print $FO pack('C', $out);
					$count++;
					if(($out & 0x01) == 0x01){$num1++;}
					if(($out & 0x02) == 0x02){$num1++;}
					if(($out & 0x04) == 0x04){$num1++;}
					if(($out & 0x08) == 0x08){$num1++;}
					if(($out & 0x10) == 0x10){$num1++;}
					if(($out & 0x20) == 0x20){$num1++;}
					if(($out & 0x40) == 0x40){$num1++;}
					if(($out & 0x80) == 0x80){$num1++;}
					$pos = 0;
					$out = 0;
				}
			}
		}
	}
	$numlines += $.;
	print "$. lines Parsed!\n";
	close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
if($et == 0) {$et = 1};  # Make sure elapsed time is not 0
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all files in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";
```

```
close $FO;

exit(0);
```

datastream_to_datastream_prng.pl

The purpose of this script is to convert binary datastreams to a binary datastream with the input stream seeding a PRNG that generates the output datastream.

Location

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

Usage

Perl datastream_to_datastream_prng.pl <infiles.bin> <outfile.bin>

Code

```
#!/usr/bin/perl
use strict;
use warnings;

# README!!! ########################################
# DATASTREAM TO DATASTREAM PRNG
# The purpose of this perl script is to take a binary datastream and use it to seed a
PRNG. The PRNG used
# is an LCG proposed by Lewis, Goodman, and Miller. I = a*I % m. a = 7^5 = 16807. M =
2^31-1 = 2147483647.
# 32 binary bits are read and used to initially seed the RNG. The 16 LSBs are written
to the output file
# for 128 iterations before the PRNG is re-seeded. 64 bytes are written to the output
file for every byte
# read from the input file.

die "Not enough arguments. Script requires a list of input files and one output file."
if @ARGV < 2;

my $outfile = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -1 );

die "Error: Specificed output file is not .bin file" unless $outfile =~ '.bin';

# fix issue where csv data file is overwritten when I forget to give a unique output
file name.
foreach my $infile (@infiles){
       die "ERROR: Infile is same as outfile.\n" if $infile eq $outfile;
}

open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

# Variables for manipulating random sequence
my $num1 = 0;
my $count = 0;
my $numlines = 0;
my $fc = 0;

# Variables for PRNG (Lewis, Goodman, Miller)
```

```
my $a = 16807; # 7^5
my $m = 2147483647; # 2^31 - 1

# Variables for save_data sub;
my $dchar = 0;
my $dcount = 0;

# Iterate over all input files and convert to binary and single output file
foreach my $infile (@infiles)
{
       open(my $FI, '<:raw', $infile) or die "Could not open infile $infile $!";
       $fc++;
       print ("Working on file $infile\n");
       while(1) {  # read in the file
              my $bytes_read = read $FI, my $bytes, 4; # read 1 byte at a time

              last if($bytes_read < 4);  # until there are no more bytes to read
              my $data = unpack('L', $bytes);  # unpack the 4 bytes into a long

              # Seed the PRNG and generate some number (128) of samples
              for(my $num = 0; $num < 128; $num++)
              {
                     $data = ($a * $data) % $m;
                     my $temp_data = $data;  # create new var to preserve current seed

                     # shift out every bit and write it to the file
                     for(my $shift = 0; $shift < 16; $shift ++)
                     {
                            my $bit = $temp_data & 0x00000001;
                            &save_data($bit);
                            $temp_data = $temp_data >> 1;
                     }
              }
       }
       close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
if($et == 0) {$et = 1};  # Make sure elapsed time is not 0
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all files in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";

close $FO;

exit(0);


# sub to save data to a file.
sub save_data()
{
       my ($data) = @_;
       # print "$data\n";  # for debug
```

```
		# add the number to the char;
		$dchar += $data * (0x01 << $dcount);
		$dcount += 1;

		if($dcount == 8) {
			# printf ("dchar=0x%2x\n", $dchar);  # for debug
			print $FO pack('C', $dchar);
			$count++;
			if(($dchar & 0x01) == 0x01){$num1++;}
			if(($dchar & 0x02) == 0x02){$num1++;}
			if(($dchar & 0x04) == 0x04){$num1++;}
			if(($dchar & 0x08) == 0x08){$num1++;}
			if(($dchar & 0x10) == 0x10){$num1++;}
			if(($dchar & 0x20) == 0x20){$num1++;}
			if(($dchar & 0x40) == 0x40){$num1++;}
			if(($dchar & 0x80) == 0x80){$num1++;}
			$dcount = 0;
			$dchar = 0;
		}
	}
```

datastream_to_datastream_von_neumann.pl

The purpose of this script is to convert binary datastreams to a binary datastream with von Neumann whitening applied.

Location

sstoller_thesis_final_writeup\RNGs\AD2_Post-Processing

Usage

Perl datastream_to_datastream_von_neumann.pl <infiles.bin> <outfile.bin> <whitening>

Code

```
#!/usr/bin/perl
use strict;
use warnings;

# README!!! ########################################
# DATASTREAM TO DATASTREAM VON NEUMANN
# The purpose of this perl script is to take a binary datastream and perform Von
Neumann debiasing on it.

die "Not enough arguments. Script requires a list of input files and one output file."
if @ARGV < 2;

my $outfile = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -1 );

die "Error: Specificed output file is not .bin file" unless $outfile =~ '.bin';

# fix issue where csv data file is overwritten when I forget to give a unique output
file name.
foreach my $infile (@infiles){
	die "ERROR: Infile is same as outfile.\n" if $infile eq $outfile;
}

open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

# Variables for manipulating random sequence
my $num1 = 0;
my $count = 0;
my $numlines = 0;
my $fc = 0;

# Variables for save_data sub;
my $dchar = 0;
my $dcount = 0;

# Iterate over all input files and convert to binary and single output file
```

```
foreach my $infile (@infiles)
{
	open(my $FI, '<:raw', $infile) or die "Could not open infile $infile $!";
	$fc++;
	print ("Working on file $infile\n");
	while(1) {  # read in the file
		my $bytes_read = read $FI, my $bytes, 1; # read 1 byte at a time
		last if($bytes_read == 0);  # until there are no more bytes to read
		my $data = unpack('C', $bytes);
		# print "$data\n" if($data != 0);  # for debug

		# unpack it into bits 0 or 1
		my $b0 = ($data & 0x01) >> 0;
		my $b1 = ($data & 0x02) >> 1;
		my $b2 = ($data & 0x04) >> 2;
		my $b3 = ($data & 0x08) >> 3;
		my $b4 = ($data & 0x10) >> 4;
		my $b5 = ($data & 0x20) >> 5;
		my $b6 = ($data & 0x40) >> 6;
		my $b7 = ($data & 0x80) >> 7;

		# print("$data, $b7, $b6, $b5, $b4, $b3, $b2, $b1, $b0\n");  # for debug

		# write the value of the 1st bit if the bits are not the same (e.g.
there is a transition)
		if($b0 != $b1) {&save_data($b0);}
		if($b2 != $b3) {&save_data($b2);}
		if($b4 != $b5) {&save_data($b4);}
		if($b6 != $b7) {&save_data($b6);}
	}
	close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
if($et == 0) {$et = 1};  # Make sure elapsed time is not 0
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all files in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";

close $FO;

exit(0);


# sub to save data to a file.
sub save_data()
{
	my ($data) = @_;
	# print "$data\n";  # for debug

	# add the number to the char;
	$dchar += $data * (0x01 << $dcount);
	$dcount += 1;
```

```
        if($dcount == 8) {
                # printf ("dchar=0x%2x\n", $dchar);  # for debug
                print $FO pack('C', $dchar);
                $count++;
                if(($dchar & 0x01) == 0x01){$num1++;}
                if(($dchar & 0x02) == 0x02){$num1++;}
                if(($dchar & 0x04) == 0x04){$num1++;}
                if(($dchar & 0x08) == 0x08){$num1++;}
                if(($dchar & 0x10) == 0x10){$num1++;}
                if(($dchar & 0x20) == 0x20){$num1++;}
                if(($dchar & 0x40) == 0x40){$num1++;}
                if(($dchar & 0x80) == 0x80){$num1++;}
                $dcount = 0;
                $dchar = 0;
        }
}
```

randu.pl

The RANDU PRNG was implemented in a software Perl script. The script is very simple and writes a binary data file of 100M bits length. The script has variable that allow the user to change the multiplier, modulus, and starting seed for the PRNG.

Location

sstoller_thesis_final_writeup\RNGs\RANDU

Usage

perl randu.pl

Code

```
#!/usr/bin/perl
use strict;
use warnings;

my $outfile = "output.bin";
my $length = int(100000000/32); # generate 100M bits (each "seed" is 32 bits)
my $seed = 13;

my $multiplier = 65539;
my $modulus = 2147483648;

# open(my $FO, '>', $outfile) or die "Could not open file $outfile $!";
open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";
for(my $n=0; $n<$length; $n++)
{
	# printf $FO "%032b\n", $seed;
	print $FO pack('L', $seed);
	# printf "%08x\n", $seed;
	$seed = $seed * $multiplier % $modulus;
}
close $FO;
```

rdrand_to_datastream.pl

rdrand_to_datastream.pl was used to convert the text output from the rdrand generator to a binary bitstream that the STS assess application can read in and test for randomness. Multiple input files can be read. A single binary file is output.

Location

sstoller_thesis_final_writeup\RNGs\rdrand\rdrand\perl

<u>Usage</u>

Perl rdrand_to_datastream.pl <infiles > <outfile.bin>

<u>Code</u>

```
#!/usr/bin/perl
use strict;
use warnings;

die "Not enough arguments. Script requires a list of input files and one output file."
if @ARGV < 2;

my $outfile = $ARGV[-1];
my @infiles = @ARGV;
@infiles = splice( @infiles, 0, -1 );

open(my $FO, '>:raw', $outfile) or die "Could not open outfile $outfile $!";

my $clk = 1;
my $data = 0xFF;
my $num1 = 0;
my $count = 0;
my $numlines = 0;
my $fc = 0;

foreach my $infile (@infiles)
{
	open(my $FI, '<', $infile) or die "Could not open infile $infile $!";
	$fc++;
	print ("Working on file $infile\n");
	while(my $line = <$FI>) {

		# Example Line: 0x9b1f5269c6cfa127
		if($line =~ /0x(\w\w)(\w\w)(\w\w)(\w\w)(\w\w)(\w\w)(\w\w)(\w\w)/) {
			for my $c($1, $2, $3, $4, $5, $6, $7, $8) {
				print $FO pack('C', hex($c));
				$count++;
				if((hex($c) & 0x01) == 0x01){$num1++;}
				if((hex($c) & 0x02) == 0x02){$num1++;}
				if((hex($c) & 0x04) == 0x04){$num1++;}
				if((hex($c) & 0x08) == 0x08){$num1++;}
				if((hex($c) & 0x10) == 0x10){$num1++;}
				if((hex($c) & 0x20) == 0x20){$num1++;}
				if((hex($c) & 0x40) == 0x40){$num1++;}
				if((hex($c) & 0x80) == 0x80){$num1++;}
			}
		}
	}
	$numlines += $.;
	print "$. lines Parsed!\n";
	close $FI;
}

my $numbits = $count * 8;  # total number of bits found
my $num0 = $numbits - $num1;  # number of zeros
my $p0 = 100 * $num0 / $numbits;  # percent zeros
my $p1 = 100 * $num1 / $numbits;  # percent ones
my $et = time - $^T;  # elapsed time the script ran
my $lps = $numlines/$et;  # lines per second
print "\nFinished parsing all $fc file(s) in $et seconds!\n";
print "Parsing a total of $numlines lines or $lps lines per second!\n";
print "File size is $count bytes or $numbits bits\n";
print "Found $num0 0's and $num1 1's\n";
print "$p0 percent 0's and $p1 percent 1's\n\n";

close $FO;

exit(0);
```

testdrng.c

Testdrng.c was used to generate a string of random bits using the Intel RDRAND TRNG. Code was modified from the Intel RDRAND example code downloadable from Intel's website. The code was modified to simply write bits to a binary output file using the Intel RDRAND instruction. Code was compiled using the Make command. The code below has been condensed (large sections of comments and unused functions removed from the code).

<u>Location</u>

sstoller_thesis_final_writeup\RNGs\rdrand\rdrand\drng_samples\testdrng.c

<u>Usage</u>

Compile with make

./assess

<u>Code</u>

```
#include "drng.h"
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include "hexdump.h"

void test_rdrand();

int main (int argc, char *argv[])
{
	unsigned int drng_features;

	/* Determine DRNG support */

	drng_features= get_drng_support();
	if ( drng_features == DRNG_NO_SUPPORT ) {
		printf("This CPU does not support Intel(R) Data Protection with Secure
Key\n");
		return 1;
	}

	if ( drng_features & DRNG_HAS_RDRAND ) {
		printf("This CPU supports the RDRAND instruction\n");
	} else {
		printf("This CPU does not support the RDRAND instruction\n");
	}
	if ( drng_features & DRNG_HAS_RDSEED ) {
		printf("This CPU supports the RDSEED instruction\n");
	} else {
		printf("This CPU does not support the RDSEED instruction\n");
	}

	if ( drng_features & DRNG_HAS_RDRAND ) {
		test_rdrand();
	}

}

void test_rdrand()
{
	unsigned int i, n;
	uint16_t rand16;
	uint32_t rand32, rand32ar[18];
#ifdef __x86_64__
	uint64_t rand64;
#endif
	unsigned char data[1024] __attribute__ ((aligned (16)));
	unsigned char *dp;


#ifdef __x86_64__
	int a;
	for(a=0; a<625000000; a++) {
		if ( ! rdrand64_step(&rand64) ) {
			fprintf(stderr, "rdrand64_step: random number not available\n");
		} else {
			printf("rand64= 0x%016llx\n", (unsigned long long) rand64);
		}
	}
#endif

}
```