

**Abstract - This paper describes the process behind random number generation and our group's implementation of a true random number generator and its verification.**

**I.** Introduction

Encrypted data relies on an unpredictable key to keep information safe. Many schemes of creating keys involves random number generation, but true number generation is not often used. A Pseudo Random Number Generator (PRNG) relies on seed values and lookup tables to produce seemingly random values, but these values can be reproduced if the same seed is acquired. A True Random Number Generator (TRNG) is a type of Physically Unclonable Function (PUF) that can reliably produce statistically random numbers indefinitely.

A TRNG does not use a seed value to select a set of premade numbers like a PRNG. Because TRNG does not use lookup tables, it will produce a different stream of random numbers every time it starts, where a PRNG with the same seed will always produce the same sequence of numbers. These differences make a TRNG much more desireable for key generators. However, TRNG’s are not used very often because they need specific hardware to function.

TRNG’s need hardware dedicated to generate random numbers: our TRNG depends on jitter (deviation from an ideal periodic signal), while other circuits do not want this unpredictable behavior. This makes a TRNG a PUF, and it uses the physically unique characteristics of the hardware to produce the required function.

**II.** Project

For our PUF, we decided on making a TRNG. This was implemented on a Nexys 2 Field Programmable Gate Array (FPGA) board using the Verilog programming language on the Xilinx program. After creating this PUF we used the National Standard of Information and Technology (NIST) statistical test suite and Dieharder tests to ensure the generated stream was statistically random.

**III.** PUF Design

We decided to use the unique delays of ring oscillator hardware to create our PUF. We utilized 16 of the following ring oscillator circuits in our design. An odd number of not gates is needed for any oscillator, and an input was needed for initialization of the ring. For our convenience, we used an AND gate to make the feedback and initialization combine simply. We attempted to hard code an initialization value of 1, but this did not work. Instead we used a switch and always left it on, which worked. We inserted the flip flop at the end of each oscillator to sample each of the oscillators at the same time.

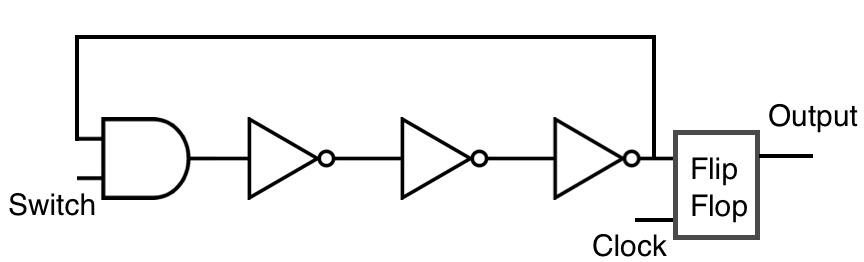


Figure 1: Ring Oscillator Design

The physical delays of each not gate and each oscillator are all different. By sampling these outputs and combining them, we were able to effectively generate a random number. We combined the outputs of all these oscillators by using an XOR tree. The XOR tree took the oscillator outputs and combined them down to one bit. The below figure is a simplified version of the XOR tree we used.

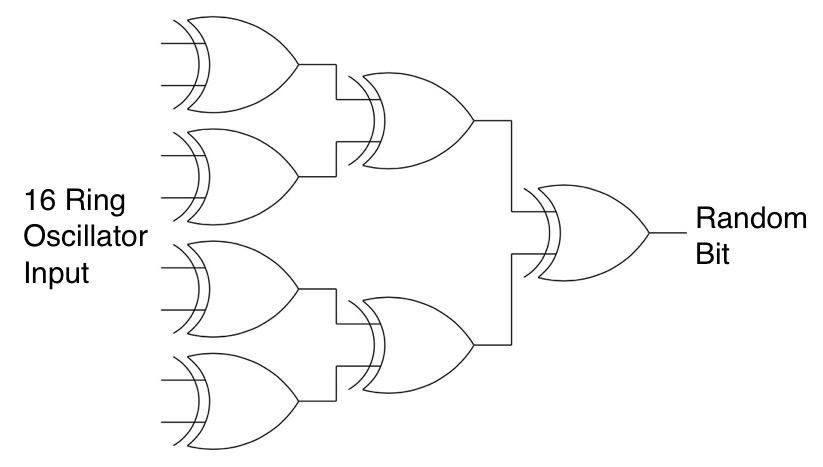


Figure 2: Simplified XOR Tree

Once a random bit was generated, we wrote that bit to RAM. We then used the program Digilent Adept (in the Memory tab) to read the RAM to a text file on a computer. We were then able to run this text file through the NIST and Dieharder test sets.

**IV** Verification

We ran both the NIST and Dieharder tests on the output of the the entire RAM which consisted of 16 million bits. We downloaded and compiled the NIST test suite onto a linux machine and copied our text output into the corresponding folder. We navigated to test results under the Experiments directory and recorded the results of each experiment. The Dieharder suite was also run on the random bit data and all the tests in both suites passed.

The restart test was also performed. We ran the program twice, and the generated numbers were very different, even from the start. The following results are how many bits were analyzed, and how manys zeros were generated compared to ones. The full output of the test can are submitted with this report.

BITSREAD = 16000000

0s = 8001681 1s = 7998319

**V.** Conclusion

True random number generation is more complicated than people think. The only true way of getting a random number is through the use of a PUF. Since a PUF is physically unclonable the same implementation on another FPGA will give random but different results. Even after a restart, the output will be completely different.

By using enough ring oscillators, there will always be jitter when sampling a bit. With only three ring oscillators, a pattern could be seen. However once a random sequence is achieved, adding more ring oscillators would not improve the performance as the stream is already random.