Exploration of SPAD Based CMOS QRNG Designs

Alex J.C. Janssen

August 2017

Abstract

In today's digital life, security and encryption are becoming more and more important. As random number generators are a fundamental block of security and encryption, it is crucial to guarantee that these devices operate securely. Random numbers are usually generated in two ways; pseudo random number generators (PRNGs) and true random random number generators (TRNGs). PRNGs output a sequence based on a seed and a mathematical function. The deterministic nature of pseudo RNG devices can result in the PRNG not being applicable for all protocols, in which case TRNGs are needed. Almost all strong cryptography requires TRNGs to generate keys. The difference is that these devices instead rely on real world physics in order to generate a random number. The TRNG implementation explored in this thesis makes use of the quantum mechanical properties of photons. TRNGs making use of this principle of elementary quantum mechanical decision making are called quantum random number generators (QRNGs). This source of entropy provided by photons can be extracted by utilizing SPADs. QRNG devices based on SPADs have been made before in different ways, however there are still large grounds undiscovered when concerning SPAD based designs. As SPAD based QRNGs can be completely produced using CMOS technology, a world of possibilities open, including integration with already existing designs. Different aspects of SPAD QRNG designs will be discussed in this thesis; size, speed, hybrid designs and QRNG test-benching. The first part of the exploration focuses on creating an as small as possible QRNG. This resulted in a QRNG design which is as small as just one flip-flop and one SPAD. This design has been nicely compared with existing QRNGs, being the smallest QRNG made so far to the authors knowledge. Simulations show that the device is able to run up to 25 Mb/s using a SPAD with a deatime in the range of 10 ns. This device has been produced using 130nm technology by STMicroelectronics. The second part of the exploration, delves into how fast a SPAD based QRNG can be. The main goals here were to make the fastest possible QRNG with good scalability characteristics. This resulted in a design proposition based on the difference in the time of flight of two photons. This design is simulated using Matlab, and can reach 70 Mb/s per SPAD-duo depending on the deadtime of the SPAD used. When using a SPAD with a deadtime of 1 μs, the scaled up design needs only 16 % of the SPADS needed by a state-of-the-art SPAD based QRNG design based on simulations. The amount of SPADs needed however schales almost linearly with a lower deadtime, having the potential to need only a fraction of the SPADs needed by the state-of-the-art in order to reach the same speeds. Then the concept of hybrid devices is explored, making use of a combination of PRNG and QRNG systems. The first hybrid design proposed is a design in which a very small QRNG is used to generate the key for multiple secure PRNG systems. The PRNG system used in this design is a trivium stream cypher. The design is completely written in VHDL except for the external QRNG. It then has been compiled and simulated using ModelSim, again using 130nm technology by STMicroelectronics. This resulted in a design which is able to reach speeds of 640 Gb/s, while using a total area of 99936 μm^2. The second hybrid device proposes a LFSR based design, which makes use of multiple very small QRNG devices to influence the function that the LFSR implements in order to increase the security. The last part of the exploration explores how it can be made easier and faster to test QRNG designs in an early design stage more accurately. As the source of entropy is quantum, the only risk of affecting the randomness is purely in which form the data of the photons is processed. By creating a chip which is able to extract the time of flight and the exact location of where the photon hit, testing potential QRNG devices can already be done in an early design stage with real-time data. A part of the chip that measures the time of flight of the photons arriving, has been designed in 40nm technology by STMicroelectronics. This part is a novel counter based on the principle of Gray counting, and is simulated extensively using extracted layout simulations. These simulations show that the device is able to run at speeds up to $3.6\,GHz$.

ACKNOWLEDGEMENTS

I would first like to thank Prof. Edoardo Charbon. Although he is the busiest man I have ever met, he always made time for me when I needed it. Furthermore I would like to thank dr. Francesco Regazzoni at ALaRI for being available daily to me for any questions. He allowed the thesis to be my own, but steered me whenever I needed it.

My sincere thanks also goes to Augusto Carimatto and Augusto Ximenes, who helped me setting up in the early stage of the thesis, and later gave me the honor to work together with them on the POLIS project.

I would also like to thank my friends Kees Kroep and Peter Stijnman for supporting me and helping by discussing my design choices with me throughout the year.

Finally, want to thank my parents and my girlfriend Wendy for continuously supporting and encouraging me throughout my years of study, research and writing this thesis. This would not have been possible without them.

Alex J.C. Janssen

Contents

1 Introduction

In our everyday digital life, digital data processing in computers, ATMs, mobile devices and more have a huge impact on our life. In all these applications and devices cryptography is an important aspect. Cryptography can simply be described as "the art and science of keeping messages secure" [1]. Random number generators (RNGs) are a very important aspect of cryptography, as a lot of processing is in need of RNG for security. Random number generation can be defined as the generation of a sequence of numbers, which cannot be predicted better than a random chance [2]. RNGs are not only used for security, but also put to use in gaming applications or Monte Carlo simulations. By having random numbers, systems can be secured and privacy can be guaranteed. Having however a RNG which is cracked, or appears to be influenceable, it can no longer be guaranteed to be secure. This could result in outsiders being able to decrypt passwords, influence game matchups or even generate TAN codes for bank accounts. This is why it is important to have reliable and cryptographically secure RNGs.

RNGs can be distinguished in two categories; pseudo-random number generators (PRNGs) and true-random number generators (TRNGs). Pseudo random number generators make use of mathematical implementations of functions whose statistical properties are the ones of a random distribution, while TRNGs make use of physical sources of entropy. All strong cryptography requires TRNGs to generate keys [3], which is why it is important to have true random number generators. In this thesis I will be making an exploration of TRNGs that generate random numbers based on photons, better known as quantum RNGs (QRNGs). The QRNGs explored in the thesis are all based on CMOS technology as this allows for reliable mass production on existing processes when using photon detectors [4, 5].

In this chapter the different techniques and state-of-the-art systems of pseudo and true random number generating will be discussed. Chapter 2 starts the exploration of different QRNG systems by first designing the smallest possible integrable QRNG system. The next Chapter 3 will take a look at a different aspect; reaching an as high as possible speed with a pure QRNG. Succeeding these two sections, two PRNG-TRNG hybrid systems will be discussed in Chapter 4. This section describes how the advantages of both PRNG and QRNG systems can be used to create more secure and faster RNG systems. The last area of the exploration, Chapter 5, will try to tackle a different aspect of QRNG designing. Here it will be explained how a chip, that is able to extract all the different properties of photons, can be used to quickly and efficiently test new potential QRNG designs. Part of this chip design will be explained in detail, which is a novel coarse counter used for measuring the time of flight of a photon based on the principle of Gray counting. The thesis will be concluded in Chapter 6, with a discussion on the results and what can be expected in the future of QRNG designs.

1.1 Pseudo randomness

A pseudorandom number generator (PRNG) outputs a sequence based on an initial seed [2]. This is expressed as shown below, where $s_0 = seed$.

$$s_{i+1} = f(s_i), i = 0, 1, ... \tag{1}$$

Thus the only thing that can be done is creating a function implementation which alters the output in a defined way [2]. There are of course a few requirements to these functions, and they have to pass a lot of tests in order to be used in cryptography, but that is out of the scope. The main point is that the bits should have good statistical properties.

When talking about RNGs in cryptography we are mostly talking about CSPRNGs, which means cryptographically secure pseudorandom number generators. This means that it is a PRNG which is not predictable, and that the next bit cannot be guessed better than a 50% chance of success. The second requirement is that the preceding bits should be impossible to calculate knowing the next bits. These requirements are unique to cryptography, in almost all other fields these are not needed. Nevertheless we would like to have systems that are unconditionally secure, however practical implementations do not meet this condition. In a world where we can assume infinite computing power, a CSPRNG is never secure. This is why we stick to computationally secure, which means we call it secure if the best known algorithm for breaking the system requires at least a certain amount of operations.

In this thesis only shift register based stream ciphers are discussed and used. This is because of their easy implementation in hardware, and many stream ciphers use different kinds of shift register based designs. Two systems will be discussed, the linear feedback shift register and the Trivium, of which the latter is a strongly secured design based of using multiple linear feedback shift registers.

1.1.1 Linear feedback shift register

A standard linear feedback shift register (LFSR) uses flip-flops, which are clocked storage elements combined with a certain feedback path. The amount of these flip-flops determine what degree the LFSR is. A simple 3rd degree LFSR is shown in Figure 1.

When using a seed where $FF_2 = 1$, $FF_1 = 0$ and $FF_0 = 0$, the sequence can be reconstructed at each clock tick, as seen in Table 1. Note that when using a different seed or feedback a different output is obtained.

As can be seen it starts repeating after clock cycle 6 and thus there is a period length of 7 bits. It is known that s_1, s_2 and s_3 are going reveal the seed that was put in the LFSR. The next bits can be computed easily following Equation 2.

$$s_{i_3} = s_{i+1} + s_i \bmod 2 \tag{2}$$

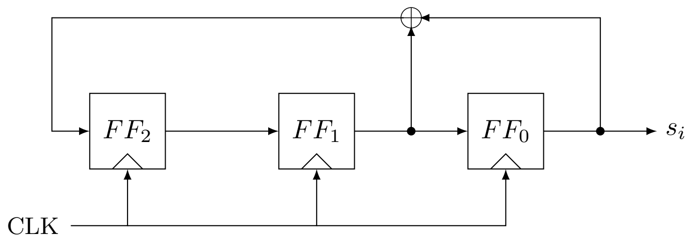


Figure 1: 3rd Degree LFSR.

Table 1: A sequence of the 3rd degree LFSR

clk	FF_2	FF_1	$FF_0(s_i)$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0
9	1	0	1

It is seen that this LFSR effectively uses all its available states, meaning it does not get "stuck" in a state. As soon a LFSR repeats a previous state, it will start repeating the whole sequence from that point. This means that an LFSR is not allowed to have the same internal state twice in a specific feedback state.

LFSRs are usually expressed using a polynomial equation. A LFSR which uses a certain feedback coefficient vector $(p_{m-1}, ..., p_1, p_0)$ can be expressed by the polynomial in Equation 3 [2].

$$P(x) = x^m + p_{m-1}x^{m-1} + ... + p_1x + P_0 \tag{3}$$

LFSRs that do not repeat the same internal state twice are called maximum-length LFSRs, which are recognized mathematically by so called "primitive polynomials". These are irreducible polynomials, which can be compared to prime numbers, which also only have the multiplication factors 1, and itself. There are a lot of primitive polynomials per state, as a LFSR of the degree $m = 31$ has $69,273,666$ unique primitive polynomials. This however means that only $69,273,666/2^{31} = 0.0323$ of the configurations are maximum-length LFSRs.

A single LFSR is however a very insecure system. A famous and efficient attack that even works on very large values of m is a Plaintext attack [2]. The attack itself is not complicated, however out of the scope of this thesis. The result is that as soon $2m$ output bits of an LFSR with a degree m are known, the polynomial of the LFSR can be discovered by solving a simple system of linear equations.

1.1.2 Trivium

A trivium is a hardware oriented stream cipher [6]. It makes use of multiple LFSRs in a complex scheme, which has as a result a simple but tested design. Its strength comes from the fact that although it uses linear elements as a basis, it uses nonlinear components to derive the output of each register.

Operation The trivium works with three shift registers, A, B and C with the lengths 93, 84 and 111 respectively, adding up to a 288-bit internal state. The key stream makes use of an iterative process which uses the values of 15 state bits, and uses these to update 3 bits of the state and compute one bit which is at the output of the key stream. The trivium can generate up to 2^{64} bits of unique key stream. This process is denoted by the pseudo code below, where t_m the output is of a register and z_i denotes an output state.

```
for $i = 1$ to $N$ do
    $t_1 \leftarrow s_{66} + s_{93}$
    $t_2 \leftarrow s_{162} + s_{177}$
    $t_3 \leftarrow s_{243} + s_{288}$

    $z_i \leftarrow t_1 + t_2 + t_3$

    $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
    $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
    $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$

    $(s_1, s_2, ..., s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$
    $(s_{94}, s_{95}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$
    $(s_{178}, s_{279}, ..., s_{288}) \leftarrow (t_2, s_{178,...,s_{287}})$
end for
```

From the pseudo code can be seen that the critical path is 1 flip-flop, an AND gate and an XOR gate. This process is visualized in Figure 2. The 3 AND gates, the 11 XOR gates and the 3 LFSRs can clearly be seen here, and the algorithm can be visualized.

One of the major downsides of this device is its initial Key and IV setup. First a 80-bit key and 80-bit initial value must be loaded into the 288-bit initial state.

$(s_1, s_2, ..., s_{93}) \leftarrow (K_1, ..., K_{80}, 0, ..., 0)$
$(s_{94}, s_{95}, ..., s_{177}) \leftarrow (IV_1, ..., IV_{80}, 0, ..., 0)$
$(s_{178}, s_{279}, ..., s_{288}) \leftarrow (0, ..., 0, 1, 1, 1)$

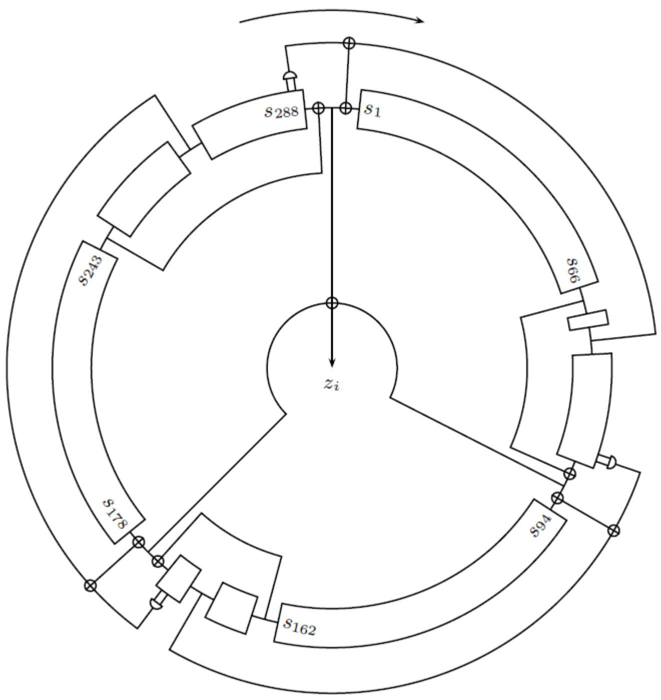


Figure 2: Visual representation of a trivium [6].

The first register is loaded with the key, and zeroes at the end. The second register is loaded with an initial value and also ended with zeroes. The third register is completely loaded with zeroes, except for the last 3 bits, which are initialized with ones. After the initialization the trivium needs to be "warmed up" by rotating it over 4 full cycles following the pseudo code for standard operation. During the first $4 \cdot 288 = 1152$ clock ticks the output of the trivium should be blocked in order to guarantee a secure output.

Scalability The trivium is a hardware oriented flexible design. It was designed to be compact, with restrictions on gate count, power-efficiency, and fast in applications that require high-speed encryption. It is a very scalable design as any state bit is not used for at least 64 iterations each time a bit is modified. This means that when the 3 AND gates and 11 XOR gates in the scheme from Figure 2 are duplicated accordingly, the design can be scaled up to 64 times. An estimation of the gate count for different degrees parallelization is listed in Table 2.

A 64 times increase in output only requires a $3488/5504 \cdot 100 = 63\%$ increase of hardware. Notice that this also speeds up the warm-up process, which now only requires $1152/64 = 18$ clock cycles. This shows that the trivium possesses great scalability characteristics.

Table 2: Estimated gate count for differently scaled implementations [6].

Components	1-bit	8-bit	16-bit	32-bit	64-bit
Flip-flops:	288	288	288	288	288
AND gates:	3	24	48	96	192
XOR gates:	11	88	176	352	704
NAND gate count:	3488	3712	3968	4480	5504

Security The trivium is thus a very high speed PRNG with great scalability characteristics. On top of that there are at the time of writing no known attack strategy's for it as long the the output is hidden during the warmup phase. However as this is still a relatively new stream cipher, it does not mean it will stay secure in the future. All in all, this is a very promising PRNG, which will be used later in the thesis for its strong security and fast hardware implementation.

1.2 True randomness

True random number generators, or just TRNGs have as a characteristic that their output cannot be reproduced. A PRNG can be reproduced, as when using for example the same LFSR and using the same seed, the results is the exact same bit stream. This is why TRNGs do not always share the same applications as PRNGs, as TRNGs are mostly used for session keys. Later in this thesis it will be shown how a TRNG can be used to generate a strong seed for PRNGs.

The output of a TRNG cannot be reproduced, as TRNGs are based of physical sources of entropy, this can be flipping a coin, thermal noise in resistors [7] or frequency jitter of electronic oscillators [8] and many more. Many of these kind of physical sources make use of huge and complex behaviour of many physical phenomena, which results in a close to non-predictable chaotic behaviour. These classical TRNG systems have in the essence a deterministic nature and external influences on these device may remain hidden. [9, 10]

Another good source of entropy, the one used in this thesis, is quantum mechanics. These effects can be used as the law of physics state that this process is intrinsically random. We do not know yet how to build devices capable of extracting randomness only from quantum mechanics, since current RNGs using this source often generate sequences where randomness comes also from other minor effects. Nevertheless, devices using quantum as a source are capable of producing sequences with strong random properties.

A TRNG using a quantum source is called a QRNG. Such a source could be the decay of a radioactive nucleus [11], however using radioactive substances is quite demanding in precautions. Another source which can be used for its elementary quantum mechanical decision making are photons, many systems have been using these as a source of entropy already, of which comparable state of the art devices are discussed in Section 1.3.

1.2.1 Light as source of entropy

Light will be the basic building block for all of the TRNG based architectures presented in this thesis. As is generally known, photons are a special kind of particles. They sometimes behave like a particle, sometimes a wave, they have no mass, yet do have momentum. Photons are thus a special case in all physical ways, its elementary quantum mechanical decision making can be used for random number generating. For the application of random number generating, two properties of photons are important, the arrival time and the arrival rate at a certain location. These can be modelled using a Poisson distribution, which is the probability of observing k events in an interval, which is given in Equation 4.

$$P(k) = \frac{\lambda^k e^{-k}}{k!} \tag{4}$$

In this case the chance of a photon having hit a certain area is k. λ Is the event rate, which is in this case the average amount of photons hitting per interval, this is the illumination in this case. λ Can be easily tuned, as this this would mean shining more or less light on average over a certain area over a certain amount of time. Using this, it is possible to give the Poisson distribution any form. From this it can be understood that photons can indeed be used as a QRNG seed, but what the eventual results is will still need to be measured in order to use it for generating numbers. This is done using single-photon detectors, which have some additional properties which contribute to the random behaviour of a QRNG.

1.2.2 Single-photon avalanche detectors

A single-photon detector is defined as a detector capable of measuring one or more characteristics of a single photon with no other present photons [4]. One type of single-photon detector is a single-photon avalanche detector (SPAD) which can be completely made in a CMOS process [4, 5]. Being able to detect photons using CMOS process allows for mass-production at reasonable costs and high yields [4]. This is a huge advantage for RNG designing, as it is possible this way to manufacture a QRNG without the need of an expensive custom process.

Basic operation SPADs are devices based on a p-n junction at a certain voltage that exceeds the breakdown voltage of the junction. At this bias the electric field is high enough that a single photon in the depletion layer has a certain chance to trigger a self-sustained avalanche. This current caused by a photon hit rises quickly up to a stable point somewhere in the mA range depending on the technology used. The photon induced current then continues until the the circuit is quenched, meaning that the bias voltage has to be lowered below the breakdown voltage of the junction. After lowering the voltage and stopping the avalanche current, the voltage needs to be brought back to the voltage point above the breakdown voltage again.

Quenching and recharge circuits A circuit that is able to make a SPAD operational is called a recharge circuit. This circuit is able to lower the bias voltage after avalanche down the breakdown voltage, called quenching, and then restore the junction back to operation level. Passive recharge is the simplest form of a recharge circuit [12, 4]. A passive quenching and recharge circuit is simply a resistor in series with the SPAD, as is depicted in Figure 3.

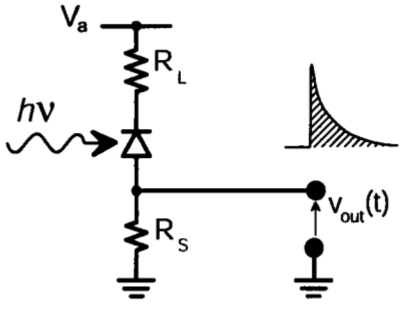


Figure 3: Passive recharge circuit [12].

The avalanche current causes a voltage drop across the resistance, which brings the voltage down the breakdown voltage again. This causes the avalanche to stop, which in turn reduces the voltage drop and brings the SPAD back to the operating point above the breakdown voltage. There are a couple of downsides to passive recharge, which have to do with the slow recharging time of the circuit. The biggest problem is that a SPAD may avalanche even before being completely recharged. This has as a result that afterpulsing is problematic with passive recharging [4], which is an avalanche that is not triggered by a photon but instead correlated to the last avalanche. This is unwanted in all applications, especially random number generation.

In order to combat this, active recharge circuits are used. A wide variety of active recharge circuits exist which are usually integrated on chip [13]. The basic idea of an active recharge circuit is to place a transistor which is turned on for a certain amount of time after an avalanche [4]. The big difference in using an active recharge circuit like this compared to passive recharge is that it is now possible to create a necessary amount of delay. In order to reduce the afterpulsing, an additional active quenching circuit can also be added. This helps the quenching process by detecting the avalanche onset and reducing the amount of free charge carriers flowing through the diode [4, 14].

Noise Noise is an unwanted property of every device, including SPADs. Especially when designing random number generators, correlated noise can be catastrophic to the device. There are two types of noise, uncorrelated and correlated noise. Starting with uncorrelated noise sources, there is noise caused by tunneling and strap-assisted noise [4]. Tunneling noise is mostly constant over temperature ranges, and is caused by particles "tunneling" through a potential wall that has more energy than the particle. Trap-assisted noise is due to Shockley-Read-Hall recombination, which is in turn caused by lattice defects. This noise is heavily temperature dependent, increasing with a factor two for every 10 degrees increase [4]. This can be reduced by reducing the amount of lattice defects, or by maintaining a lower temperature.

Correlated noise sources are the most dangerous to the entegrity of a QRNG. One of these sources has already been mentioned, which is afterpulsing. Afterpulsing is caused by electrons being trapped in states in the forbidden energy gap. During an avalanche a lot of electrons are in a state of conduction, being able to get trapped in an higher energy state than the valence band. When the device is quenched, and brought back to the bias in the breakdown voltage region, these electrons require much less energy to go back to the conduction band again. An electron receiving little energy for example from heat, will cause another avalanche, not based on a photon, but on an electron that got trapped during a previous avalanche. This thus creates a correlation between the last avalanche, and a new one caused by afterpulsing. By increasing the deadtime, it is possible to reduce the afterpulsing. It is thus important to set a deadtime long enough to reduce the chance of afterpulsing to virtually zero.

The second source of correlated noise is crosstalk, which is a correlation between two SPADs. There are two types of crosstalk, optical and electrical crosstalk. Optical crosstalk is caused by the photon emission of a SPAD when in avalanche, which in turn hits another SPAD. Optical crosstalk has been often observed, especially in dense packed SPAD arrays [15]. Electrical crosstalk less common [4] and is caused by another firing SPAD having an effect on the power line, which in turn makes another SPAD fire. Electrical crosstalk can also happen due to electrons tunneling from one to another SPAD. Crosstalk can generally be avoided by simply placing two SPADs far enough apart from eachother.

1.3 State-of-the-art photon based QRNG designs

There are different ways of utilizing photons in order to generate random numbers. A technique which has been used multiple times is a variety of systems using lasers to achieve high speeds. There is for example a high speed quantum random number generator with a speed of over $6.25\, Gb/s$, based on the quantum phase fluctuations of a laser operating near threshold [10]. Another system that is used is based on splitting a beam of photons using a polarizing beam splitter (PBS) [9]. This system makes use of a weak light beam which then passes the PBS which is set to polarize the incoming light at a 45 degree angle with respect to the PBS. Then two SPDs are used, where one is set to be a $'1'$ when a photon is detected, and the other a $'0'$. There are a few disadvantages though to using such a device, they require warmup time, calibration of the tube voltages of the photomultipliers and quite a setup size, namely $25x19x3\, cm^3$. Another device which operates in a similar way, by making use of a weak mirror which only reflects 50% of the photons, is a commercial QRNG by ID Quantique [3]. This device is able to reach speeds up to $16\, Mb/s$. The problem with these kind of devices is that they are fabricated in a custom process, and require extensive calibration. Furthermore it is hard with these devices to maintain a stable, high-quality stream of random numbers.

It is interesting to explore QRNG devices, which are producible in CMOS technology. Commercial QRNGs are usually build in very expensive custom processes, while CMOS processes have the potential to achieve the same, and even much higher speeds at a much lower cost. Creating a QRNG has been done in multiple occasions, but the parallelization has proven to be limited and this area of massively parallel QRNGs is still a largely unexplored area. A paper that did explore this area of CMOS QRNGs and parallelization of these devices, is the paper of reference [16]. Here is an already existing imager composed 512x128 SPADs being used to create a QRNG which is able to run up to $5\,Gb/s$. This shows that it is indeed possible to create a fast QRNG using CMOS process with parallelization. A point that would be interesting to explore, is to create a CMOS based QRNG which is made purely for quantum random number generating. Creating a device with this in mind from the start allows for more design options.

2 Small Integratable QRNG

Many state-of-the-art QRNG systems described in Section 1.3 rely on precise calibration or use a large area in order to reach higher speeds. In this section I want to step out of this design space, and explore largely undiscovered grounds; very small integratable QRNGs. Instead of focusing on high speed, high tech and large area based designs, I will be exploring how to make the smallest possible CMOS based QRNG for integrated designs. This comes with an advantage for the industry, as the device could easily be integrated in already existing designs which require RNG. On top of that a very small QRNG has the potential to open a whole new field of RNG designing, this will be discussed in Section 4.

2.1 Design goals

First of all the QRNG must be able to be produced in CMOS technology, so it can be universally used in different systems. Secondly, in order to make it easily integrable it is usefull to be as small as possible. Furthermore a simple interface benefits integrability. At last the system should have good entropy characteristics, in order to be able to connect it directly to another device. No requirements concerning the speed of the device are set, since the minimization of the area is the main goal. These requirements and goals can be confined to the following.

- The QRNG must be as small as possible.
- The QRNG must be able to be produced using CMOS technology.

2.2 Design overview

In order to explore the area of small QRNGs effectively, it is important to keep the design simple. The architecture exists by attaching a single SPAD with an active quenching and recharge circuit, to a T-flip-flop. These circuits are needed to make sure the deadtime can be set, and by using active quenching afterpulsing is reduced. The circuit diagram is depicted in Figure 4.

The operating principle of this architecture is to initialize the T-flip-flop at any value and invert its output at every pulse that arrives. This architecture relies on the fact that a photon can arrive at any given moment in time. The problem would seem that it is possible to predict the output of this device, based on a simple expected value calculation. If the average time of arrival of a photon is $100\,\mu s$, it could be expected that at $200\,\mu s$ the output of the flip-flop is in its initial state again. However it can be expected that when the wait time is long enough, the output of this flip-flop can no longer be estimated with an accuracy of more than 50%. This is due to the poisson distrubtion in the arrival time of the photons as described in Section 1.2.1. Because of this distribution it is expected that the illumination levels have a big influence on the expected value. If the SPAD used had a certain deadtime, and the illumination levels are infinite, the SPAD would fire as soon it is recharged. This would mean

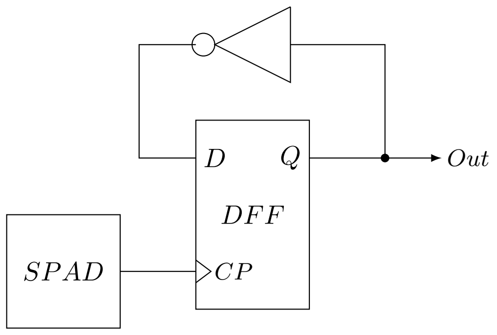


Figure 4: T-FF made using a D-FF and an invertor connected to a SPAD.

that the system will stay predictable as the correlation to the previous read out value will not dissapear. In this case a simple expected value calculation would be a good approximation. However if the illumination levels bare low enough, the time frame in which the flip-flop can switch becomes large compared to the deadtime. The expectation is that in that case it might be possible to undo the correlation to its last read out value over time. Furthermore it is expected that there is an optimum illumination level, that does not cause the SPAD to instantly fire when unquenched, but is also not so low that it takes a long time to converge the entropy to 1 Shannon.

2.3 Expected value Simulations

The architecture is dependent on a few parameters, the deadtime denoted by τ_{dead}, the light intensity hitting the SPAD, and the photon detection probability of the SPAD. These last two are combined for the simulations into a single parameter; pulses per second (PPS). This is the amount of pulses that the SPAD would generate on average per second when there would be no deadtime involved. There are two things that are interesting to find out using simulations of this device. The first point is to see at what values of PPS the device is able to converge to 1 Shannon, and how fast this can approximately go. The second important point is to find out what the effect of the deadtime of the SPAD is on the range of PPS that can be used, and on the maximum speed that can be reached. The simulation is done using Matlab of which the code can be found in Appendix A. The simulation assumes no external influences and that at time $t = 0$ the value of the T-flip-flop is known as well that at $t = 0$ the SPAD is not being quenched. The design is simulated by calculating the chance of an event occurring using the poisson distribution, and comparing it with a random generated number by the PC. If it is a hit, the SPAD is actively quenched and recharged, thus any further hits during this process do not create another inversion of the T-flip-flop. This process is repeated during the whole selected time frame for one single instance of the simulation. This whole simulation is then repeated a few thousands of times, and the results are averaged, which gives the probability mass function $P(t)$. The entropy in Shannon is calculated using Equation 5.

$$H(t) = -\sum_{n}^{i=1} P(t) log_b P(t) \tag{5}$$

The plot in Figure 5a depicts the probability mass function, while Figure 5b depicts the entropy on the Y axis, for the same PPS.

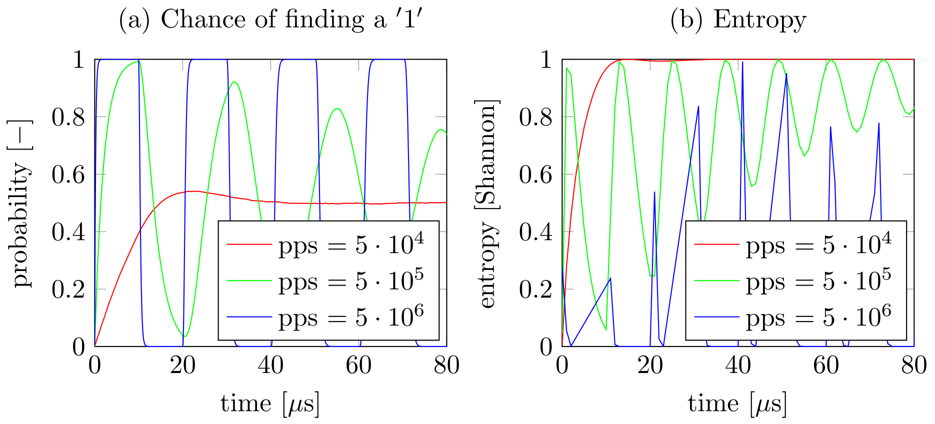


Figure 5: Expected value simulations for the deadtime of $10\mu s$ under different PPS conditions.

The plots in Figure 6 depict the results of the simulations for the different deadtimes: $1\mu s$, $100ns$ and $10ns$. Figure 6b makes use of a different amount of PPS than the other simulations, in order to see how the scaling in deadtime corresponds with scaling the PPS.

Simulation Results In Figure 5a, the effect of having too much illumination on a device with high deadtime is illustrated. The blue line behaves like a square wave which confirms the expectation. This is because under high illumination a photon will have a bigger chance to hit the area within a certain timeframe. If the deadtime is big, this timeframe will be very small compared to the deadtime. Having too much illumination on a device can cause that the device will not be able to converge within a reasonable timeframe. The green line in Figure 5 shows the effect of having a bit too much illumination, however the system is still converging to an entropy of 1 Shannon. The red line is somewhat optimized, and converges really fast to an entropy of 1 Shannon. This line confirms the operation of the device, as it is indeed possible to completely converge to an entropy of 1 Shannon. On top of that the line converges fast compared to the deadtime of $10\mu s$, around $40\mu s$ for this amount of PPS.

Figures 6a and b, look exactly the same, which must mean that the scaling of the PPS versus deadtime is linear. On top of that we can also see here that the

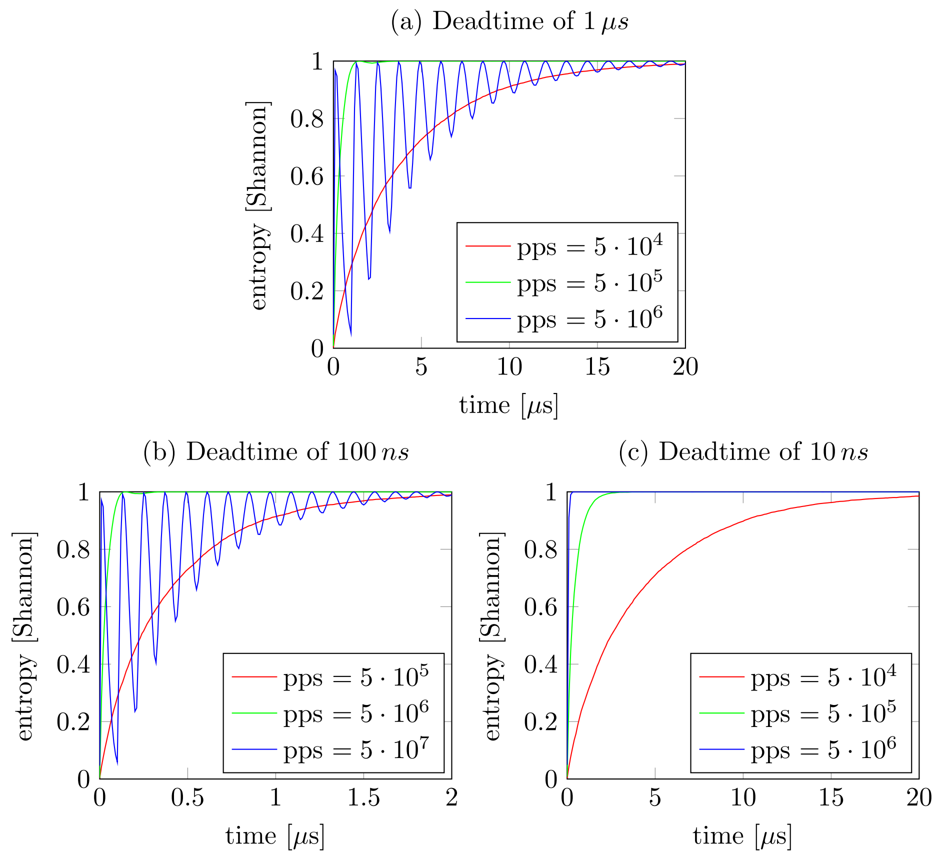


Figure 6: Entropy simulations for the deadtimes of $1\mu s$, $100ns$ and $10ns$ for different PPS conditions.

system converges in $4\mu s$ and $400ns$ for the plots in Figures 6a and b respectively. This is to be expected, as the scaling between deadtime and PPS is linear and the plots in Figure 5 show a converge time to 1 Shannon of around $40\mu s$ for a deadtime of $10\mu s$. This means that the convergence time can be optimized for one deadtime, and the optimum amount of PPS can then be calculated for all other SPAD deadtime values.

Figure 6 shows that there is a maximum converge speed that can be reached for a certain PPS. Figures 6a and 6c depict an equal convergence time for a PPS of $5 \cdot 10^4$.

From these simulations it can be concluded that it is indeed possible to remove the correlation of the previous value with the next value by using time. Furthermore having a lower deadtime scales linearly with the speed and the amount of PPS it can converge for. It is however important to restrict the maximum amount of PPS as the system will start oscillating when the PPS becomes too high. When this happens it will take much longer to converge to a entropy of 1 Shannon, or it will not be reached at all. On top of that quite high speeds can be reached, for a device of the size of only one SPAD and flip-flop. Assuming a 10 ns deadtime SPAD, speeds of $\frac{1}{40\,ns} = 25\,Mb/s$ can be reached!

2.4 Implementation

The system is implemented using 130nm technology provided by ST semiconductor. The chip is fabricated, and received however still needs to be tested. In this section the layout of the chip will be discussed.

2.4.1 Basic architecture layout

The base circuit itself is as simple as it can be, just one D-flip-flop and an inverter with sufficient delay to make the D-flip-flop toggle. The circuit and layout are shown in Figures 7 and 8 respectively. The T-flip-flop is able to toggle in approximately 130 ps in a typical process, which is fast enough, even when having a SPAD deadtime of only 10ns.

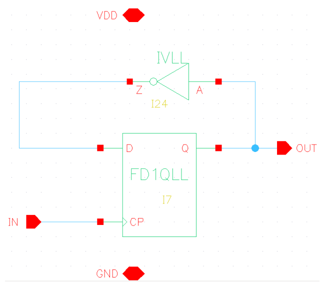


Figure 7: T-flip-flop schematic.

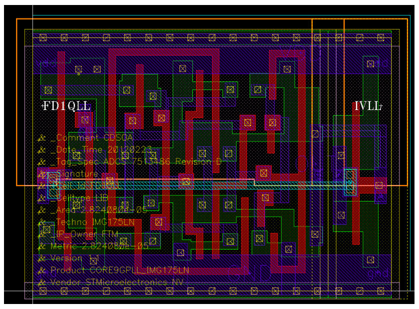


Figure 8: T-flip-flop layout.

In order to connect a SPAD to the T-flip-flop, it needs to be quenched and recharged, followed by a buffer, and in this case a pulse generator which generates a pulse long and strong enough to flip the T-flip-flop that will be connected to it. Figure 9 shows the schematic and the layout of the circuit which will quench and recharge the SPAD and generate a pulse when the SPAD fires.

As can be seen this is a passive quenching and recharge circuit instead of an active circuit which is preferable. This was however implemented this way as designing an active recharge circuit requires and is out of the scope of this exploration. The resistor is able to operate in the area of several mega ohms in its resistive operation region. The pulse generator uses the smallest available inverter at the input in order to minimize the load for the SPAD and generates a pulse of approximately 200 ps with the T-flip-flop attached to the output.

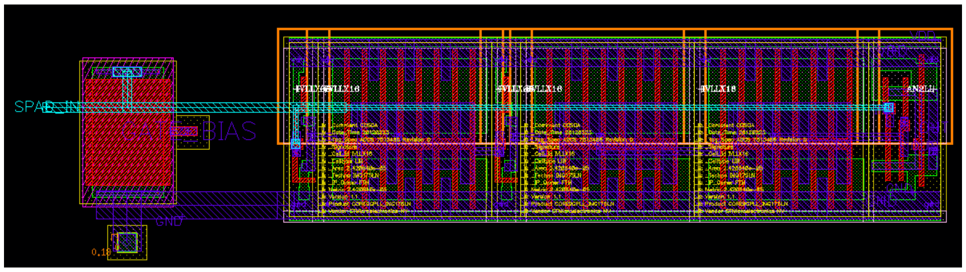


Figure 9: Quenching and pulse generation.

2.4.2 Chip layout

Figure 10 shows the whole chip layout, which is 814.070 x 725.235 μ^2 in size.

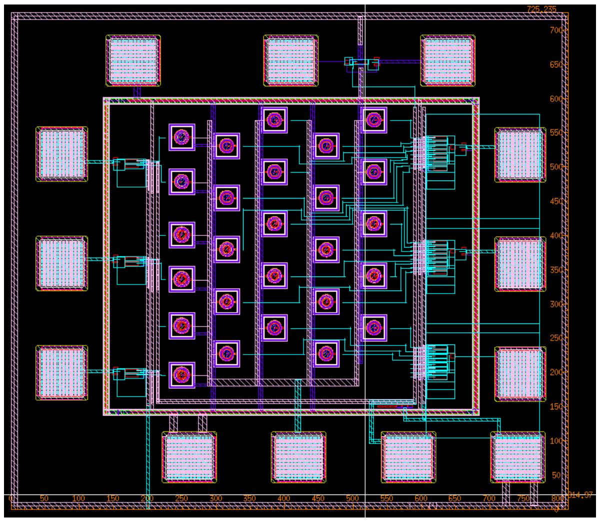


Figure 10: Visual representation of a trivium.

In the middle area of the chip 5 different types of SPADs can be seen, as there were at the time of creating the chip no available tested SPADs. This increases the chance of having a functional SPAD for testing the device. Due to the lack of library components, a simple PAD ring is made by hand. This exists out of two inner rings were made on two metal layers, one for the substrate ground, and one for the normal ground. The outer ring functions as the VDD line for the circuit. ESD protection is made by placing multiple hand-designed diodes in parallel.

In total 7 testing circuits are attached. Three identical circuits at the left side of the chip, connected to the left three PADs. Three circuits at the right side which have more SPADs connected to them for each circuit, and one circuit at the top side. The circuit at the top side is shown in Figure 11.

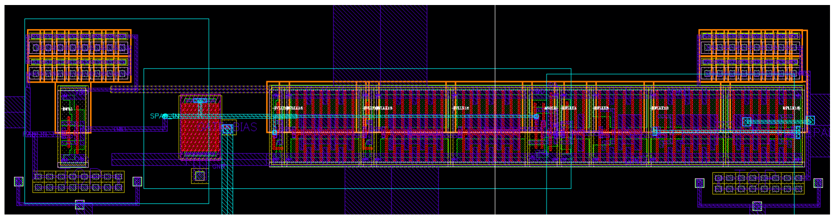


Figure 11: Main testing circuit where a SPAD of choice can be connected to.

This circuit contains a single passive quenching circuit and T-flip-flop, the SPAD is left out. This is the most basic, and most important circuit of the whole chip. In this way any SPAD can be connected to the circuit in case all the SPADs in this technology are non-functional. The top-middle PAD is meant to connect the external SPAD and the top-right PAD for the output of the QRNG. The next three testing circuits are identical and shown in Figure 12.

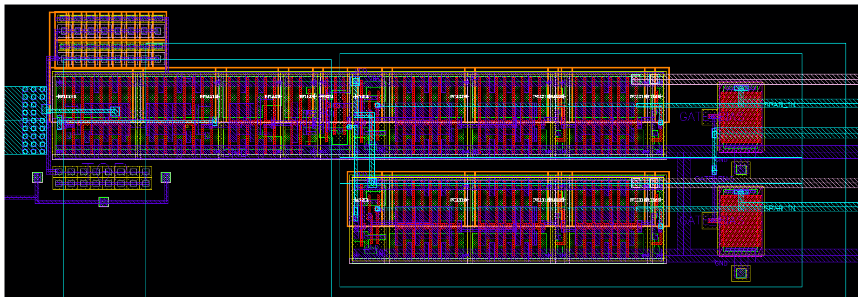


Figure 12: Testing circuit with two SPADs attached.

These circuits have two SPADs attached with the quenching and pulse generator, which are then connected together with an OR gate, which is then connected to the T-flip-flop. This is done to test the effect of scaling with SPADs for the Entropy. For each of the circuits a different type of SPAD is used, in order to have a greater chance of being able to test properties when using two SPADs instead of just one. The last three circuits are at the right, which are shown in Figure 13.

These 3 circuits are for testing the randomness when a lot of SPADs per QRNG. The top two at the right side have 8 SPADs connected with pulse generators, with again OR gates and a T-flip-flop at the output. Using this circuit, the effects of crosstalk can be investigated, while also investigating the effect of having multiple SPADs together has on the speed of the device. The last of these 3 circuits is the same, except it lacks ESD protection and has only 4 SPADs connected, in case there is something wrong with the custom made protection.

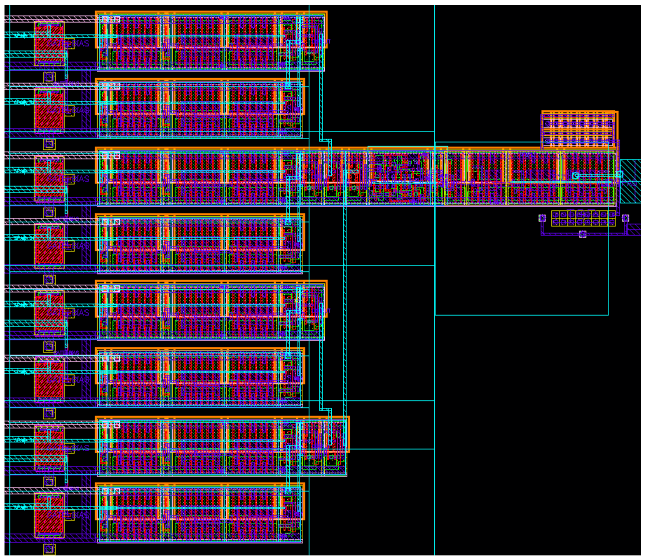


Figure 13: Testing circuit with eight SPADs attached.

2.5 Conclusion

The resulting design is very small and nicely compared with existing QRNGs, even the smallest QRNG made so far to the authors knowledge. On top of that the design is able to operate as QRNG on its own, where the speed is heavily dependent on the deadtime. When using SPADs with a deadtime as low as $10\,ns$, it is possible to reach speeds of up to $25\,Mb/s$, while using only one flip-flop and SPAD! At last the device is fully made in CMOS technology, with a very simple 1-output interface. These characteristics have as a result that the design can be easily integrated in already existing designs. An example of integration would be already existing PRNG architectures, as will be shown in Section 4, or other designs that are in need of RNG while sacrificing a minimum amount of area.

3 High Speed Scalable QRNG

3.1 Design goal

The second point in the design space in QRNG designing that will be explored is speed. This is a huge difference compared to the small QRNG in previous section, as the area size is no longer priority number one. More high speed QRNGs have been made in the past in CMOS process, most notably an image detector converted in a QRNG reaching up to $5\,Gb/s$ [16]. Building a QRNG in CMOS process however from the ground up opens up a lot of new possibilities and design choices. In this section a design is proposed with a focus on high speed and good scalability characteristics, while maintaining robustness. This design is therefore not aimed to be small in other designs, but rather to have a speed as high as possible with as main purpose a standalone QRNG. These requirements of this design can be confined to the following.

- The QRNG must be able to be produced using CMOS technology.
- The QRNG should be as fast as possible.
- The QRNG should have good scalability characteristics.

3.2 Design Overview

In order to have a fast design, it is important to be able to generate a number out of the first photon that arrives, which resulted in the architecture shown in Figure 14.

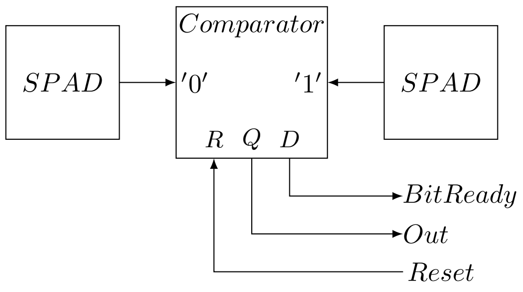


Figure 14: SPAD-Comparator-SPAD block.

The operation principle of this circuit is to have two identical SPADs connected to a comparator, where if the left SPAD fires first, the output of the comparator will be a $'0'$ and if the right SPAD fires first, the output will be $'1'$. An active quenching and recharge circuit is needed, where if one of the two SPADs goes in avalanche, both SPADs are quenched. The comparator will give a *BitReady* signal, when the output is ready, and will only recharge both SPADs when the system is read out with a *reset* signal. This is done to increase the speed of the device, the moment the signal is read out, the comparator can start generating a new number. Otherwise the one of the SPADs might receive another photon after the quenching time, which wastes power. Also the situation might arise where it is still quenching after it is read out, wasting potential time where a new bit could have been generated.

Throughout this chapter a few assumptions concerning this system will be made. The first assumption is that the SPADs are exactly identical. The second assumption is that as soon one of the SPADs fire, both are quenched and are unable to fire for a certain deadtime τ_{dead}, which is set in such a way that no afterpulsing is present. The last assumption is that at all time, both SPADs are illuminated with the same light intensity. If this is not the case, it will cause a bias towards $'1'$ or $'0'$.

3.3 Output codes

There are a few output possibilities for this architecture, and with the assumptions in place, 4 cases can be distinguished which are shown in Table 3. The two bits in the code show if it was the left SPAD that received the pulse first within a certain time range or the right SPAD. Code $'00'$ means that no pulse has arrived during the time. $'10'$ Means that the left SPAD got into avalanche first, $'10'$ that the right SPAD avalanched first. The $'11'$ code means that both photons arrived at the same time and caused an avalanche within a certain timeframe.

Table 3: Possible output codes.

Code	Result
00	?
10	"0"
01	"1"
11	?

When two pulses arrive at exactly the same time within the sample time this gives a "11" code, no new bit will be generated and the ready signal stays $'0'$. Of course the chance of two photons arriving at exactly the same time is almost infinitely small. The "11" code occurs within a certain timeframe, which is dependent on the speed in which the comparator can converge to a value, or the speed in which both SPADs can be quenched. In the case the SPADs can be quenched faster than the comparator can set the value, this will be the maximum timeframe in which a "11" code can occur after the first SPAD fired. In the case the comparator can converge faster than the quenching circuit can quench both SPADs, the comparators speed be the maximum timeframe. In this thesis I will assume that the quenching will determine the timeframe in which the "11" code can occur. A fast quenching circuit will approximately take $1\,ns$, an average circuit $2\,ns$ and a very slow quenching circuit $5\,ns$.

A "00" code means that no new number is generated within the time frame. This is the biggest problem with the system in its current state as it is asynchronous. A solution would be to simply be so certain of having a value in a timeframe so that other factors like SPAD tolerances weigh heavier than the chance of a $'00'$ or $'11'$ code appearing. This is however not reliable and the additional waiting time would slow down the system. A definitive solution to this however will be provided when scaling the system.

3.4 Speed simulation

Before looking how to scale the system, it is important to know the speed that the system can reach. It is expected that the speed is heavily reliant on the deadtime τ_{dead} of the SPADs. In order to find this out, the device has been simulated using Matlab. The source code for this simulation can be found in Appendix B. For the active quenching system is assumed; if within $2ns$ the second SPAD fires, it is regarded as a "11" code, and the bit is disregarded. Figure 15 shows the PPS, which is the amount of photons potentially causing an avalanche multiplied with the PDP, on the X-axis and speed of the device in Mb/s at the Y axis. The graph is plotted for three different deadtimes; $1\mu s$, $100ns$ and $10ns$. For these deadtimes is assumed that they are selected in such a way that no afterpulsing is present.

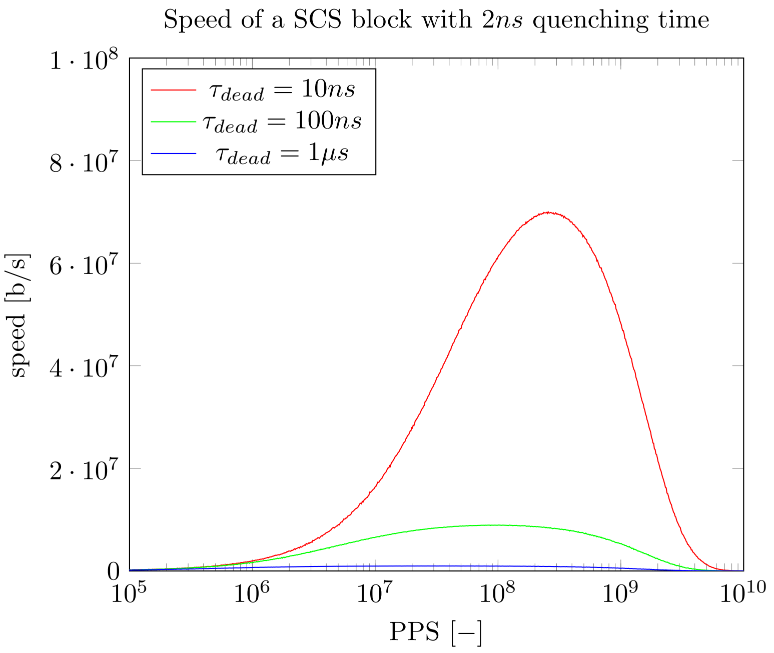


Figure 15: Expected value simulations for the deadtime of $10\mu s$ under different PPS conditions.

The first thing that is observed is that the deadtime of the SPAD used has a huge impact on the maximum speed of the device. This is to be expected, and it can also be observer that this scaling is almost linear, the reason that it is not however, has to do that at really low deadtimes, the quenching time becomes comparable in speed, and will have a larger influence on the total resulting speed. What can be further observed is that under low illumination levels, the QRNG has to wait for too long for a single photon to arrive, which has a huge impact on the speed. However as photons arrive faster, the chance of getting a "11" code also increases. After the peak, the amount of "11" code becomes so big that the speed starts decreasing again. The amount of Mb/s that is discarded because of "11" codes occuring is shown in Figure 16, which corresponds with the same deadtimes as Figure 15.

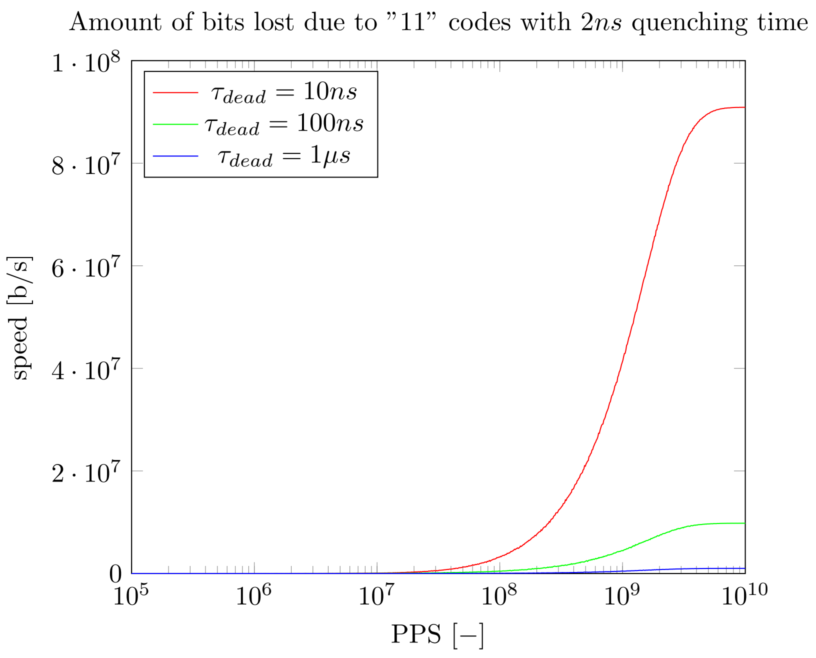
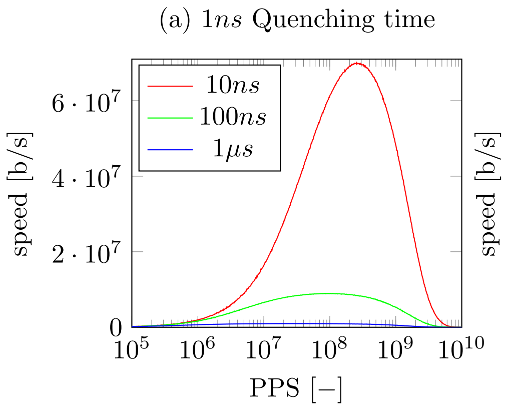


Figure 16: Expected value simulations for the deadtime of $10\mu s$ under different PPS conditions.

From this figure it can be seen the "11" code is an important factor. It is also clear that the speed of the QRNG should increase when the the timeframe in which this code can occur becomes smaller. It is thus interesting, to see in how far how the quenching speed relates to the speed of the QRNG. Figure 17a depicts the speed plotted against the PPS for fast quenching of $1ns$, while Figure 17b show this for slow quenching of $5ns$.

The results from the graphs are taken, and put together in a table. The maximum speeds that can be reached, with the according PPS, is shown in Table 4 for the different quenching speeds τ_q and deadtimes τ_{dead}.

From the table it is seen that there is a difference based on the time window in



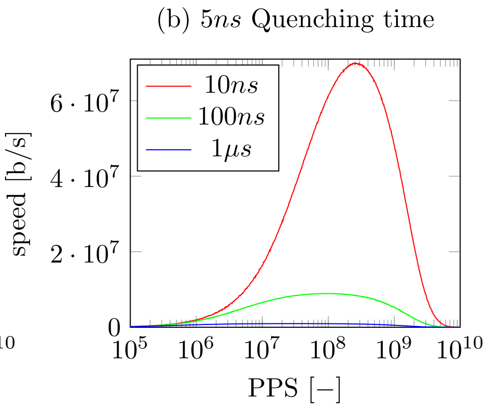


Figure 17: Speed simulations of SCS blocks for different quenching times.

Table 4: Counting sequence of a simple asynchronous counter.

	$\tau_{dead} = 10ns$	$\tau_{dead} = 100ns$	$\tau_{dead} = 1\mu s$
$\tau_q = 1ns$	$70.0\,Mb/s\,|\,2.7e8\,PPS$	$8.95\,Mb/s\,|\,8.8e7\,PPS$	$0.86\,Mb/s\,|\,3.0e7\,PPS$
$\tau_q = 2ns$	$64.0\,Mb/s\,|\,2.5e8\,PPS$	$8.9\,Mb/s\,|\,9.3e7\,PPS$	$0.97\,Mb/s\,|\,2.9e7\,PPS$
$\tau_q = 5ns$	$60.3\,Mb/s\,|\,2.5e8\,PPS$	$8.87\,Mb/s\,|\,9.3e7\,PPS$	$0.97\,Mb/s\,|\,2.5e7\,PPS$

which a "11" code can occur, however it has only an influence when working with high speeds and low deadtimes. No big influence is detected when concerning higher deadtime SPADs. This is as the deadtime is a much larger number than the quenching speed which is only a couple of nanoseconds. When compared to a deadtime of $10\,ns$, the quenching speeds become a comparable number, and has a more significant influence on the speed of the device. Thus when looking at the different speeds the device can reach, it is mainly dependend on the deadtime of the SPADs used. When using a low deadtime of $10\,ns$, the speed of a single SCS block can reach up to a $70\,Mb/s$!

3.5 SPAD tolerances

There is another potential problem when working with a system like this, which is the fact that due to production, not all SPADs are identical. The problem with differences in the SPADs is clear, the device is not as random as we would expect it to be as it will have a bias towards a $'0'$ or a $'1'$. This can be because one SPAD has more afterpulsing and is in need of a longer deadtime, or more likely a difference in the photon detection efficiency.

This problem is however not as big as it would appear as the differences in the PDP is usually small, and the afterpulsing can be regulated by simply introducing a longer deadtime to both SPADs. This system is also not meant for being directly connected to other systems as might be done with the QRNG from Section 2. This means that the biased output can be post processed in order to create a balanced output. However in the case that it is important to have a good as possible balanced output of ones and zeroes, a few solutions can be introduced. These solutions are mainly aimed at solving the difference in the PDP of the two SPADs used.

One solution would be using multiple SPADs at both sides connected together with OR gates, which would balance out the average PDP at each side. As soon one SPAD fires, everything else is quenched until the system is read out. It is important to use lower illumination conditions than normally optimal when implementing such a system. This is because the chance at having double pulses becomes bigger, up to the point that there are always two pulses within the given time limit of each other, no longer creating an output. This implementation thus drastically decreased the power/bit ratio, as the amount potential pulses thrown away becomes huge.

The best option is to make use of SPADs of which it is known what the characteristics are, and pair those up, in order to create a more balanced output. This can be done by placing the SPADs close to each other during production so the silicon characteristics do not change much, or by choosing and measuring SPADs that are already there. Mind that crosstalk in the case of placing the SPADs closely together can become a problem, as crosstalk can cause "11" codes to occur, and thus decrease the speed of the system.

In case the previous solution is not a possibility, looking at the illumination might be a good option. If the difference in the PDP is too big, it might be useful to tune the illumination on the two SPADs differently. By measuring the results over a long time, and adjusting when needed a balanced output can be produced. The big downside to this implementation is that two light sources are needed, both on a separate voltage. This means that it is possible to influence one of the two light sources, which then completely destroys the randomness of the device.

Concluding, it is possible to solve this problem in multiple ways, the most convenient way is to simply use post processing to maximize the entropy. However it is important that the system outputs $'0's$ and $'1's$ balanced to a certain degree, as that is the strength of the device. The simplest way to accomplish that is to design the system in such a way that the two SPADs are close to each other and the silicon does not differ too much during production, or simply picking two identical SPADs that are already there. The solution to make use of different light sources is also a very good option in case the previous is not possible, but one has to be careful with potential influences on these separate light sources. Using multiple SPADs per side is discouraged, as this lowers the efficiency of the device because of the non-linear increase in speed.

3.6 Scaling

Scaling the system up can be done in two dimensions, speed per bit and the amount of bits at the output. The first point that is going to be discussed is the speed per bit, and also making the system synchronous in the process of doing this. Figure 18 depicts a selector switch with multiple SCS blocks connected to it, and the small QRNG design from Section 2.

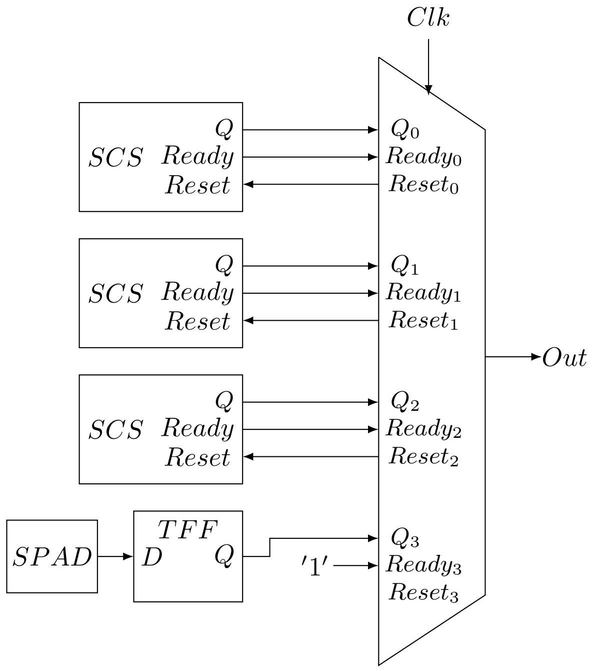


Figure 18: Selector with SCS blocks and a small T-flip-flop based QRNG attached.

Selector switch system The operating principle of this circuit is that the selector will check each time a bit is requested which input has set the $'R_x'$ signal to $'1'$ and give the bit back from that block. After it sent a bit the selector resets the block, so it can generate a new bit asynchronous.. Using this system, it is possible to synchronize the different asynchronous SCS blocks. There is however still always a chance that none of the SCS blocks has a new bit set ready. For the case none of the SCS blocks have a bit ready, the small QRNG design from Section 2 is attached. This block has its ready signal permanently set on $'1'$, so the selector can always output a value. Keep in mind that the T-flip-flop is not meant to increase the speed, just to make the system synchronous. The system now scales linearly with the amount of SCS blocks connected to a selector switch. The amount of bits can simply be scaled by adding more of these complete selector systems.

Potential problem The challenge with this device is to make sure that the T-flip-flop based design is only read out after its stabilization time. This cannot be guaranteed using this system, however the chances of that happening should be made so low by using enough SCS blocks or by lowering the clock readout frequency, that the chance of that happening becomes neglectable.

3.7 Conclusion

The goal was to make a highly scalable, fast as possible QRNG. By using a comparator system, a bit can be provided as soon a single photon lands on one of the two SPADs. This is mainly bottle-necked by the deadtime of the SPADs used, as was shown in the simulations. Using SPADs with a low deadtime, it is possible to reach high speeds up to $70Mb/s$ per SCS block. When comparing this to the small QRNG described in Chapter 2, it is $70/25 = 2.8$ times faster per single block, at the cost of more area. The SCS blocks also possess good scaling properties. By using a selector switch and the small QRNG from Section 2 the system can be made synchronous and multiple SCS blocks can be used to increase the speed per bit linearly. A state-of-the-art QRNG imager sensor is able to reach up to $5Gb/s$ using $512x128 = 65.536$ SPADs [16]. When assuming an average SPAD with a deadtime of $100ns$, this system would only need $\frac{5Gb/s}{8.9Mb/s} \cdot 2 = 1124$ SPADs! This is only a 1.7% of the SPADs that the mentioned system requires in order to reach the same speed. When assuming SPADs in need of a deadtime of $1\mu s$, it would need $\frac{5Gb/s}{0.97Mb/s} \cdot 2 = 10.310$ SPADs, which is still only 16% of the amount of SPADs needed in comparison with the state-of-the-art.

4 Hybrid PRNG-QRNG designs

Devices that make use of both a TRNG and PRNG in order to generate random numbers I call Hybrid devices. This can be only a QRNG seed as will be seen in Section 4.1, or a mixed up design using both elements as will be shown in Section 4.2. In the current status of security a computionally secure RNG is strong enough, as it cannot be deciphered within an achievable time limit [2]. Only using SPADs for making a pure QRNG limits the speed and efficiency, so it is worth looking at hybrid structures in order to utilize the upsides of both systems, while reducing or even solving the downsides of QRNG and PRNG systems.

4.1 Hybrid Trivium-TRNG

In Chapter 1.1.2 the trivium has been discussed. The main point of this design is that it has been safe for many years, and has proven to be still computionally secure. It is able to run at a high speed as the critical path is only 1 flip-flop and 2 gates. On top of that it can be easily scaled up to 64 bits, without any penalty on the security and only a relatively small increase in the power consumption and area. The big downside to this device, as with all PRNGs, is that it is in need of a seed. When one would be looking to influence the number generation of this device, it is the best idea to start at the seed for this. The main goal of this design is to create a fast as possible, scalable RNG, which is made more secure using a QRNG.

4.1.1 True RNG Seeding

The QRNG systems described earlier in the thesis can be used to generate a seed, more specifically the very simple T-flip-flop based architecture from Chapter 2. As the trivium already has a warm-up time, it is not meant for immediate operation, except when it is already warmed up. Using a system that increases the warm-up time does not bring a new penalty to the system. Using a T-flip-flop based design is a very area efficient, and most of all the most robust option for generating a seed. As the seed only needs to be generated once before starting the device, it thus only introduces additional warm-up time. The slow speed that a T-flip-flop based QRNG brings, has zero influence on the operation speed, covering the biggest weakness of that particular architecture, while its strengths cover the trivium's weakness.

The system works by having every clock cycle of a slower clock a value loaded from the T-flip-flop into another memory element. This clock has to be slow enough under the correct illumination in order to reach a stable random value with an entropy of 1 Shannon. A trivium needs a 80 bit initial seed, so there are a few ways to implement this principle in hardware. First of all by loading the seed one by one directly into the trivium, by blocking its feedback paths while being seeded. The second option is to have a 80 bit register, which loads one by one the value from a single T-flip-flop, and when reaching 80 it loads it all at once into the trivium. The last option is to have 80 T-flip-flop's and all load at once into the trivium. Using 80 T-flip-flops is unnecessary, as there is no need for decreasing the warm-up time, while it increases the area taken by the seed generation by a factor 80. As SPADs are quite big in this technology, and while the warm-up speed is not necessarily an issue, it is not the best solution for this implementation. The first solution to load the values one by one needs an additional gate to block the feedback. This increases the longest path, and thus reduces the maximum speed of the trivium. So the first and third solution are not desirable for this implementation.

The second option was to have a 80 bit shift register, it does not increase the total size significantly, but has one big point of advantage. Reseeding is done often in security, and by using a separate 80 bit shift register a new seed can be generated while the trivium itself is operating. This means that only the first and initial warm-up of the trivium is longer, but when reseeding, the system no longer has any drawbacks due to the seeding. On top of that it does not affect the speed of the trivium in any way. An enable input is needed on the register as it only needs to run, when a new seed can be produced.

4.1.2 Scaling

The goal is to have a RNG as fast as possible. As discussed in the paragraph before, the relatively slow generation of the seed does not impact the eventual output speed of the device. Using 130nm technology by STMicroelectronics, the longest path is $\tau_{FF} + \tau_{xor} + \tau_{and} \approx 100ps + 100ps + 100ps \approx 300\,ps$, one trivium will run at $\frac{1}{0.3\,ns} \cdot 64 = 213\,Gb/s$. This is when reading the worst case scenario from the data sheet. Experience has shown that in reality systems can be slower by a factor two. On top of that, this neglects all the connections that a control system would need to the Trivium which would introduce slower flip-flops. In this system 4 triviums will be run parallel, which will show how easy it is to increase the speed of the system. Using 4 triviums means of course that the 80 bit shift register now becomes a 320 bit shift register, and the startup time for the TRNG seed is increased by a factor 4.

4.1.3 Control system

The challenge with this system is to integrate the 4 triviums and the TRNG seed generation as one system using a control system. It is however useful to split the system up in two parts; the key control and the trivium control which combined form the whole control system. The reason to keep these two systems strictly apart is because they will both operate at a different clock speed. The key generation requires a slow clock because of the way a T-flip-flop based QRNGs operates. The triviums however need to operate as fast as possible, thus require a very fast clock. Furthermore the user must be able to reseed the device manually without having to reset and generate the seed all over again. This requires a *Reseed* input, and user feedback to indicate that a new key has been generated. For testing purposes it is not needed to block the output from the triviums during the 18 count warm-up cycle, however in the future this is needed. This is why a *TriviumReady* signal is also added to the output, as this signal will indicate if the outputs should be blocked, or passed through. The whole system overview with all the inputs, outputs and the two subsystems is shown in Figure 19.

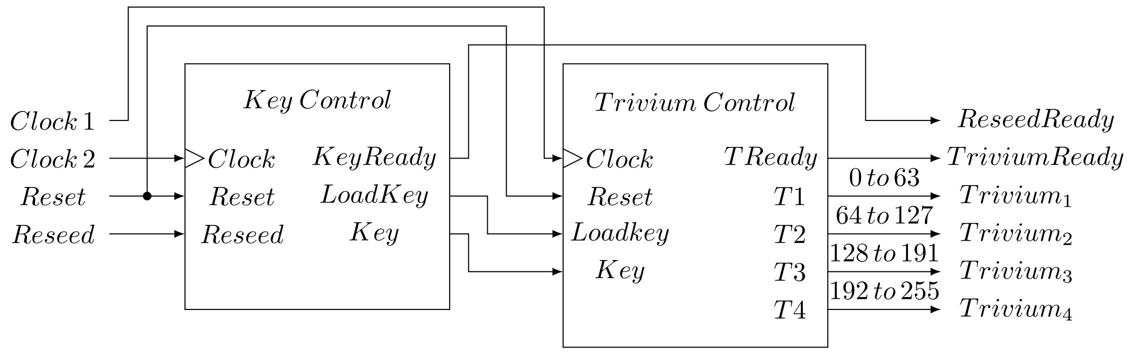


Figure 19: Control system overview.

Key control First the design of the key control will be discussed, which has a few requirements.

The system must be able to:

1. generate a 320 bit QRNG key.
2. start generating a key as soon the system is reset.
3. send a *LoadKey* signal to the Trivium control when the first key is generated.
4. generate a second reseed key after a key is loaded.
5. stop generating new bits when a reseed key is ready.
6. reseed the triviums on user command, when a key is ready.
7. provide a *KeyReady* signal to the user, when a reseed key is ready.

This system is thus in need of a few components, first of all a counter to keep track of the amount of bits loaded in the shift register. Secondly a 320 bit shift register to store the key in. A D-flip-flop to keep track if it is the first time the system is seeding, if so, it needs to send the *loadkey* signal by itself. Another D-flip-flop to keep track if a key has been generated, be it the first one or the reseed key. The total circuit for the key control is shown in Figure 20.

It must be kept in mind that the devices all use a negative reset. The counter is needed to determine if a key that is being generated is finished. When the

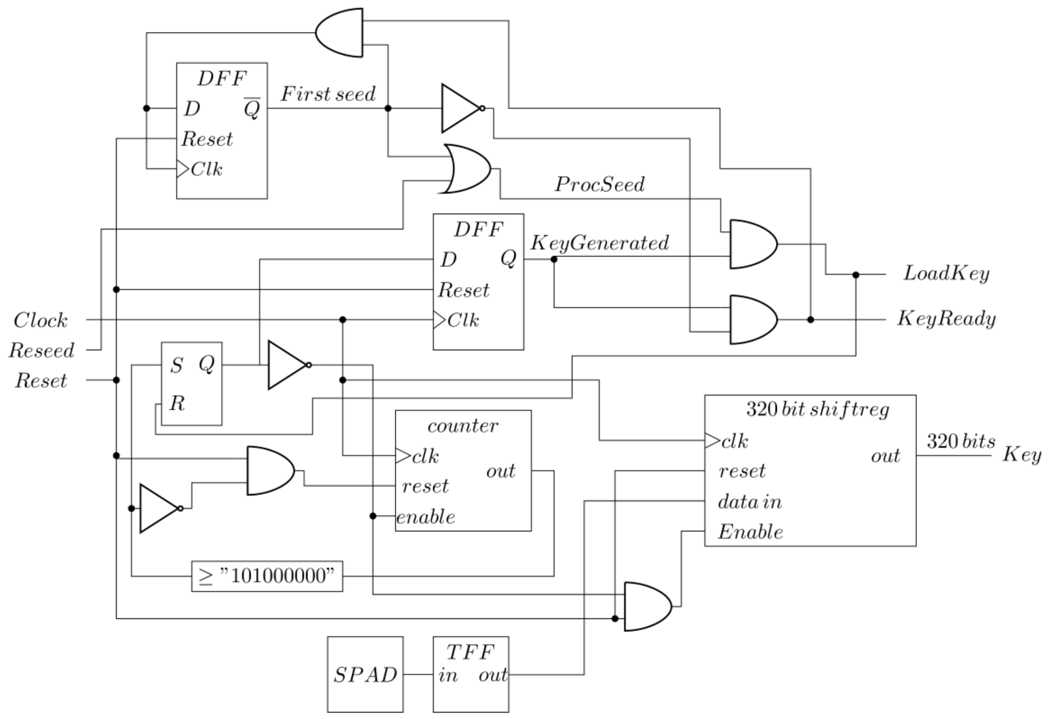


Figure 20: Key control circuit.

counter reaches a value of "101000000", a pulse is generated. This pulse will then reset the counter, and enter an SR-latch, which will hold the the value of $'1'$. This latch is connected to the enable input of the counter and the shift register, which will cause these device to pause.

The latch is also connected to a D-flip-flop which indicates if a key has been generated. If it is the first time seeding, the $KeyReady$ signal will not be send to the user, but the system will automatically sent the $LoadKey$ to the Trivium Controller. This will then cause the top D-flip-flop with the $First\,seed$ output to set its output to $'0'$, which means that the system has loaded its first key since its previous reset. As the $LoadKey$ signal is connected to the R input of the SR-latch, it will reset this latch, which in turn re-enables the counter and the 320 bit shift register.

This process starts generating a second key, while the Trivium Control loads the key in its next clock cycle. When the second key is generated, the counter and shift register are again disabled, and the D-flip-flop with the $KeyGenerated$ output is set to $'1'$. This time however the D-flip-flop which indicates if the system has seeded the first time already is set to $'0'$. The key will thus not load automatically, because of the AND-gate, until the user sets the reseed signal. If the reseed signal is permanently set on 1, it will have the same effect as if the system is doing its first reseed permanently.

The key control system in this design fulfills all requirements. It tracks if the system has seeded the first time and automatically seeds after a reset. Furthermore it stops the shift register and the counter when a reseed key is ready and provides the user feedback if a key is ready. It is also able to reseed on command whenever a reseed key is ready, or it can be done automatically if the user demands it.

Trivium control The second part of the hybrid Trivium-D-flip-flop RNG is the control part around the trivium. This part runs on a much higher clock frequency, and has the following requirements.

The system must be able to:

1. have 4 triviums running at the same time in parallel output.
2. load the 320 bit key from the Key Control when the LoadKey signal is given.
3. provide a signal to the user when the triviums are warmed up.
4. be able to reseed when a new LoadKey signal is given.

The schematic overview is shown in Figure 21.

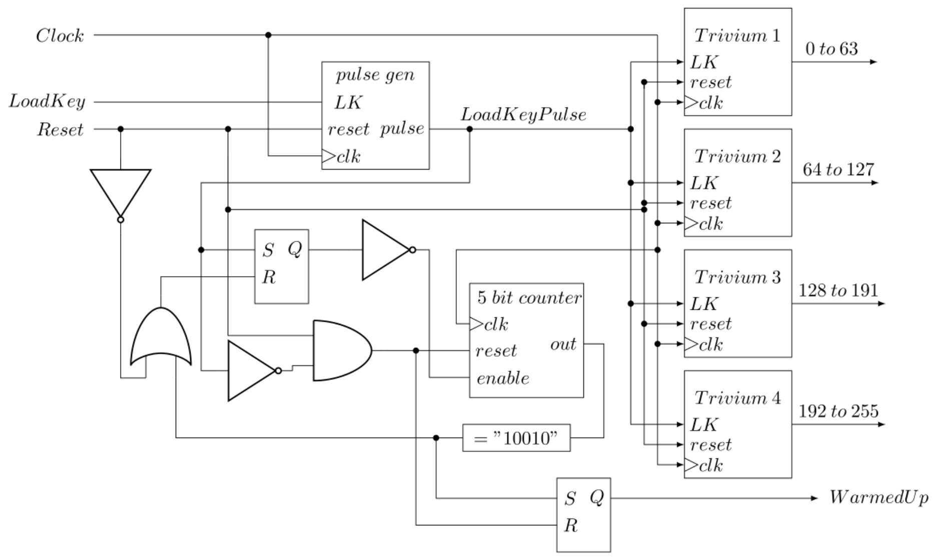


Figure 21: Trivium control circuit.

The trivium control uses a 5 bit counter to keep track of the warm-up phase, which sets a signal to 1 as soon "10010" or 18 in decimal is reached. The counter uses the exact same architecture as the one in the Key control, which is a standard synchronous counter. The output signal of the triviums is directly sent to the user, as this is the signal that will be used to block or pass the output of the triviums in the future.

The counter itself needs some feedback, as is seen in Figure 21, as it must not count when the reset is $'0'$ or when the counter has reached "10010". Furthermore it must reset to zero in case of a reseed, or when a reset signal is given. This is done by connecting the reset to an AND gate with the inverse of a $KeyReady$ signal on the other input, as the counter resets on a negative reset.

The triviums have a $LoadKey$ input which will take the values from the KeyControl its 320 bit shift register, and directly put the values into the flip-flops inside the triviums.

Concluded, this system is able to control multiple triviums, which can easily be scaled up by adding another one. As soon a $LoadKey$ signal is sent, the counter is reset, and the triviums load the values from the Key Control during the first clock tick. The next clock cycles the triviums will start warming up and the counter will keep track if they are warmed up. After the triviums are warmed up, the control system will set the $TReady$ output to $'1'$, which can be used in the future to block or pass the values of the triviums.

4.1.4 Results

The triviums and the control system are written in VHDL and compiled using DC compiler. For this the STMicroelectronics 140 nm technology is used. Each component has an individual code, as this helps with debugging during the synthetization process. The script used for DC compiler can be found in Appendix C. The VHDL code of each block of the design can be found in Appendixes D to N. Note that more blocks can be found in the appendix than in the figures. This is due to the fact that in order to make the whole design synthesizable, most code has to be written in structural, instead of behavioural. The counters for example are done are made using JK flip-flops written in structural code, as can be seen in Appendixes L, M and N. The whole design including the 4 triviums after compilation report an area usage of 99936 μm^2. Compiling a single trivium takes 20842 μm^2 of the area. Figures 22 and 23 show the ModelSim simulation of the compiled design, in order to check if the design operates as expected and check the delays. The VHDL code of the testbench can be found in Appendix O.

From Figures 22 and 23 it is extracted that it takes 400 ps from the start of the clock cycle to make the triviums out appear to the user. As is known from Section 1.1.2, the output of the trivium is made using 2 XOR gates. The glitch that is seen is thus the direct line passing through the XOR at the output, and the final value is when the output of XOR behind the XOR at the output has

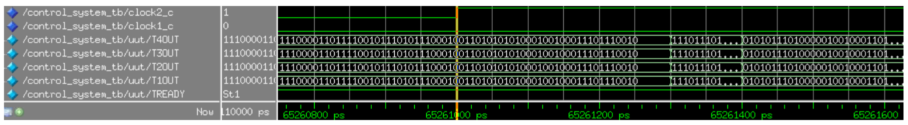

/control_system_tb/clock2_c
/control_system_tb/clock1_c
/control_system_tb/uut/T4OUT
/control_system_tb/uut/T3OUT
/control_system_tb/uut/T2OUT
/control_system_tb/uut/T1OUT
/control_system_tb/uut/TREADY
Now
65260800 ps
65261000 ps
65261200 ps
65261400 ps
65261600

Figure 22: Timing of the start of a clock cycle.

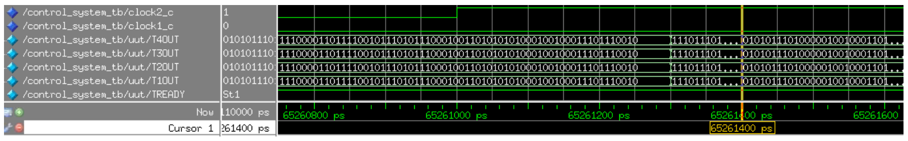

/control_system_tb/clock2_c
/control_system_tb/clock1_c
/control_system_tb/uut/T4OUT
/control_system_tb/uut/T3OUT
/control_system_tb/uut/T2OUT
/control_system_tb/uut/T1OUT
/control_system_tb/uut/TREADY
Now
Cursor 1
65260800 ps
65261000 ps
65261200 ps
65261400 ps
65261600
65261400 ps

Figure 23: Timing of a stable value at the output.

progressed through the line. As the propagation time of a single XOR-gate is $50\,ps$ in this simulation, it is calculated that the critical path of the triviums during operation is only $300\,ps$. This is what was expected from the calculations in Section 4.1.2, however there worst case values for gate delays were used there. This is now roughly compensated by the fact that additional connections to the trivium are needed, and the simulation uses normal case delay values. This ModelSim simulation is however still very optimistic, and does not take into account the wiring, connections, corners or any other factors other than the set gate delay. To stay on the safe side, it can be stated from this ModelSim simulation that each trivium can run at $160\,Gb/s$. This means for 4 triviums, that the whole system will run at $640\,GB/s$!

4.1.5 Conclusion

By using the advantages of the simple QRNG architecture, and combining it with a already known strong PRNG architecture, a strong implementation of a random number generator is made. The high speed of the Trivium is combined with a QRNG seed, which covers the weakness of this secured PRNG. Furthermore it can be seen that it is easy to get to very high speeds by using multiple Triviums. In this 130nm technology by STMicroelectronics it is possible to reach a speed of approximately $640\,Gb/s$ using 4 triviums! Concluding this design, it is shown that by using the vast speeds that a PRNG like a Trivium can reach can be used to create random numbers much faster as long the PRNG is secure. In this case the seed is secured by using a QRNG and as long the Trivium stays secure this is a very reliable high speed hybrid implementation.

4.2 Quantum Dynamic LFSR

As stated in section 1.1.1, the main problem with QRNGs thus far is the lack of speed. This was compensated last time using a Trivium, while only using the QRNG as a seed, as security was not a problem at the time of writing. A strong example of what incorporating QRNG elements in a PRNG can mean to the security of a PRNG design itself is using an LFSR, which is one of the simplest and weakest PRNG architectures. This has as a result a hybrid PRNG-QRNG, which is much stronger than the original design while covering the weaknesses of both PRNGs and QRNGs. This is not a fully fleshed out design, but an introduction to an idea with potential, which can inspire new ideas of how slow and simple QRNGs can be used to improve the security of PRNGs.

4.2.1 Incorporating TRNG

The problem with traditional LFSRs as explained in Section 1.1.1 is the speed with which these PRNGs can be cracked, which is only $2m$ where m the degree of the LFSR is. What creates an uncrackable LFSR is when we change the polynomial of the LFSR before $2m$ clock cycles. Figure 24 shows a traditional LFSR with but now with a Q block attached in the feedback. This Q block is a QRNG implementation, which will close the connection when it is $'1'$ and open the connection when it is $'0'$. A logical implementation for this would be the small QRNG design from Chapter 2, connected with an AND-gate.

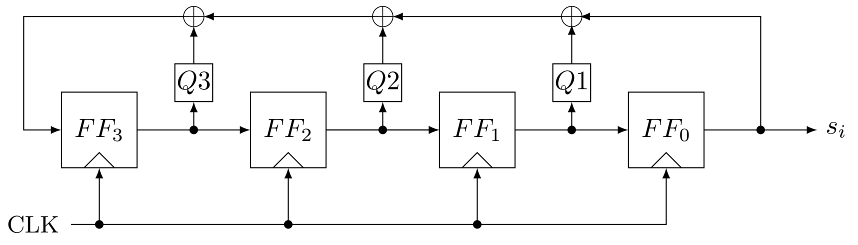


Figure 24: Dynamic 4th Degree LFSR.

What happens now is that at any given point in time, a feedback loop can close or open, changing the polynomial of the device, and if this happens before $2m$, it can no longer be cracked. The problem that occurs to someone trying to break the device is that the person does not know when it changed, it is a separate Poisson function per feedback loop. There is no longer a starting point for the $2m$, and if it can be guaranteed to a certain level that at least one feedback was always changed within $2m$ clock cycles, the system is no longer crackable. This way it is possible to significantly speed up a T-flip-flop based TRNG design using a simple PRNG like a LFSR.

4.2.2 Conclusion and future work

There are however some flaws with this system in the way it is now implemented. All polynomials are possible, including ones that are not primitive polynomials, or polynomials that can get the system "stuck". The best solution to this would be a mathematical solution. Instead of directly connecting the feedback to a QRNG, which has been done in this case, a control system around it could be made. This system should implement a certain mathematical function which would make the system only use primitive polynomials, or filter out the polynomials that make the system stuck and have a very low cycle length.

A simpler solution could be implemented by calculating the average cycle length of an LFSR with degree m. Then a very large value for m must be chosen, and a value lower than $2 \cdot m$ in which the system needs to have its polynomial changed, which is the time to crack the system taken. It can then be tuned how often the system gets a situation where it can be cracked within the limit of its cycle length to an acceptable percentage. The thing to keep in mind is that the attacker does not know when the cycle started, even if there is a certain small chance that the amount of cycles exceeds that the attacker could know what the LFSR's next output would be based on the LFSR repeating itself, or because the small chance occurs that the feedback has not changed yet, it is still a very hard nut to crack.

5 Universal QRNG Testbed

SPAD based QRNG designs make mainly use of two possible properties; time of arrival or the amount of arrivals. This can be done however in many different ways, two QRNG systems have been proposed before; the small T-flip-flop based design and the comparator based design. There are however many more possibilities to explore for example such as using a Time to Digital Converter (TDC) to measure the time of arrival, or perhaps designs that make use of a lot of SPADs and different arrival times to create high speed designs. As it has proven so far to be a quite intensive process to simulate a design, making the schematic, layout and produce the chip, it would severely help an exploration of designs if ideas could be accurately tested in an early design stage. That would mean to have a design with a lot of SPADs which can extract all the data about photons that a SPAD can give you, instead of making a new chip for every design idea. If this data could then be saved to a file from this device, it could be used to have much more accurate simulations, as all the data extracted from the SPADs are real measurement results. The device could then also be directly connected to an FPGA, for a more direct implementation testing approach and real time data. This way any possible SPAD based QRNG design idea can be tested almost directly after introducing a basic architecture.

5.1 POLIS project

A chip which could accomplish this would be a chip which is currently being developed at the TU Delft. It was originally meant as a versatile image detector, however in the scope of this project it was discovered that it could serve much more SPAD oriented fields. The idea of converting an image detector into a QRNG is not new, as it has already been done before in the paper of reference [16]. There is a big difference here however, as the aim here is to have a testing platform, and not anything performance related. The chip exists of a big array of SPADs from which all the important information about a photon can be extracted. With this chip it is possible to see exactly which SPAD fired at which location in the array, at which time. The comparator based design for example could easily be tested with extracted information by appointing two SPADs without having to produce the whole chip. Even two arrays of the POLIS chip could be used, one array would be pointed to be a $'0'$ code and another as a $'1'$ code in order to test scalability of the comparator based design. Also the efficiency of attaching multiple SPADs could be easily measured before producing the chip and deciding on how many SPADs to use. I will not go into details about the exact architecture of this chip, as it has not been published. Nevertheless I did work on the part of the chip that measures the time of flight of a photon of a SPAD. For this a coarse counter and a TDC are needed, where I designed the coarse counter.

5.2 High speed, high accuracy Gray counter

This section will be used to explain the design of the counter. First a short section is spend on why an ordinary counter does not suffice, after which the most important design choices for the counter and the design itself are explained. The section will be concluded with the results of the device.

5.2.1 Design goals

The counter of this system has a few requirements that call for a custom design approach instead of using a conventional counter. The system has the following set of constraints to be met.

- The counter must drive a 10 bit output.
- The counter must be able to run at a minimal clock speed of 2GHz.
- The counter must be able to be reset and start counting at 0.

The design has the following objectives in order of importance.

1. During count transitions there should be minimal to no glitches.
2. The time difference between each count transition should be as close to 0 as possible.
3. The resolution should be as high as possible.
4. $Area * Power/Speed$ ratio should be as low as possible.

5.2.2 Problems with conventional counters

A conventional synchronous counter counts in the usual bit convention. The problem with this convention is that multiple bits can flip at the same time. Table 5 shows the amounts of bits flipping per count.

Table 5: Bit flips in conventional counting system.

Count	Q_2	Q_1	Q_0	Bit flips
0	0	0	0	3
1	0	0	1	1
2	0	1	0	2
3	0	1	1	1
4	1	0	0	3
5	1	0	1	1
6	1	1	0	2
7	1	1	1	1

When the counter would be read out while it is flipping, the number read out can be nowhere close to what should have been read out because of the multiple bits flipping. When using a conventional synchronous counter while not running at maximum speed, the chance of encountering an error due to multiple bits flipping per count is minimal as the counter is most of the time waiting for a new clock tick. However when running it at a high speed to maximum speed, which is wanted in the POLIS project, this becomes a problem. When looking at normal synchronous counters, it is not possible to evade these multiple bit flips during a transition as it is the way traditional bit counts work. There is however a solution to the problem of these bits flipping, which makes use of a different way of counting. This method is called Gray counting, named after its inventor Frank Gray.

5.2.3 Gray counting

A Gray counter solves the main problem, which is the amount of bit flips per count. This is done by not following the conventional counting system, as is shown in Table 6. This table shows that when gray counting, indeed only one bit is flipped per count.

Table 6: Gray counting system.

Gray	Normal	Q_2	Q_1	Q_0	Bit flips
0	0	0	0	0	1
1	1	0	0	1	1
2	3	0	1	1	1
3	2	0	1	0	1
4	6	1	1	0	1
5	7	1	1	1	1
6	5	1	0	1	1
7	4	1	0	0	1

5.2.4 3-Bit gray counter

Gray counting however not a simple perfect solution as it means that the bits have to be forced to count in this way, instead of following the natural way of counting. When a lot of bits are needed, all these bits are dependent on each other. A direct implementation of a 10 bit gray counter would result in having feedback loops with a huge amount of gates, making the system very slow.

In order to make an efficient design it is important to start small and then scale it up. First an implementation of a 3-bit Gray counter will be explained. For a 3 bit counter, 3 flip-flops are needed, named A, B and C, where C is the LSB. As speed is an important parameter, the amount of feedback in the counter needs to be confined, preferably to just 1 gate plus a flip-flop in the longest path. A good feedback path can be determined by studying Table 6.

After carefully looking, it can be seen that when $C_s = 1$, $A_{s+1} = A_s$. Furthermore when $C_s = 1$ it is observed that $B_{s+1} = A'_s$. The next step is to see what can be found when $C_s = 0$, which is in this case $A_{s+1} = B_s$ and that $B_{s+1} = B_s$. Now only the feedback for C needs to be determined, it is seen that an XNOR gate can be used for C with A and B connected to the inputs. This XNOR can also be implemented with a MUX if needed. Nicely organizing the resulting feedback in the case statements below, starting with the feedback for A gives the following results.

$$A_{s+1} = \begin{cases} A_s & \text{if } C_s = 1 \\ B_s & \text{otherwise} \end{cases} \tag{6}$$

For the feedback of flip-flop B is defined below.

$$B_{s+1} = \begin{cases} B_s & \text{if } C_s = 0 \\ A'_s & \text{otherwise} \end{cases} \tag{7}$$

For the last feedback of flip-flop C which is simply a XNOR gate.

$$C_{s+1} = A_s \text{ XNOR } B_s \tag{8}$$

The result is a fast implementation of a 3 bit Gray counter, as there is only 1 flip-flop and 1 logical gate in the longest feedback path. In this technology a normal MUX is slower than a MUX with one of its inputs inverted. This requires some slight adjustments to the connections. The load has to be balanced as inverted input $D0$ of each MUX is twice as hard to drive compared to the inputs $D1$ and the selection input $S0$. For the XOR gate the inputs make no difference. The circuit diagram of the whole system using MUXes having one inverted input is shown in Figure 25. Here the load distribution connected to each output gate of the FF is optimized. The flip-flops used here are D-FF with a negative triggered reset, meaning that it will reset to '0' if a '0' is on the RN connection.

This basic system has a worst case delay of $\tau_{total} = \tau_{FF} + \tau_{MUX} = 250ps + 100ps = 350ps$, or simply put a worst case speed of $2,9GHz$. Comparing this implementation to the requirements, it can be concluded that this basic structure meets almost all of the constraints, except for the 10 bit implementation.

5.2.5 Basic architecture

As explained it is not possible to directly implement 10 bits, as it would cause a huge feedback and the counter will no longer be able to reach the required speed. This is why the previous 3 bit block is reused as a standard block, and being used as a part of the total 10 bit system. A system overview is depicted in Figure 26.

Each block is a 3-bit implementation of a Gray counter as shown in Figure 26. The C blocks are some implementation of control in order to make the parts behind the first block count. They will provide a pulse at count "111" to the next block. The extra block under the three blocks is the 10th bit.

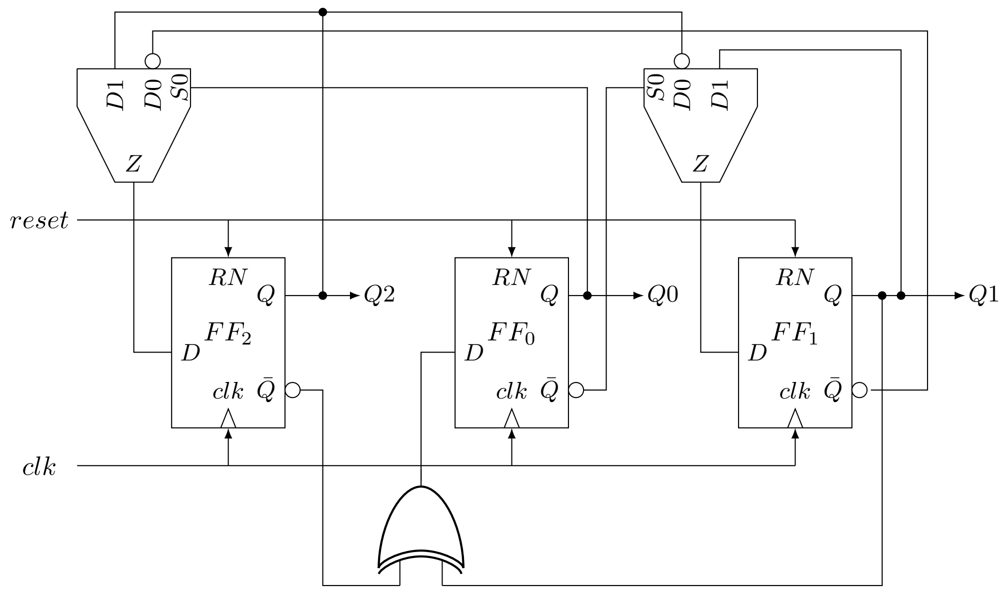


Figure 25: Circuit of the designed 3-bit Gray counter.

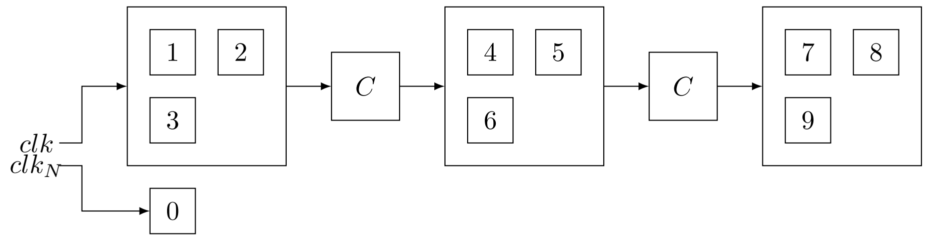


Figure 26: 10 bit implementation.

10th Bit There is a strict restriction of a 10 bit output, which could be solved by simply placing an extra FF behind the last 3-bit block, however there is a smarter method. With the use of what I call a "negative bit" the accuracy of the counter can be doubled, next to introducing the 10th required bit. This works because of the way this methodology is introduced, 3 different 3-bit counters. The addition of this bit count wise is shown in Table 7.

To make the circuit count like this is very simple; just connect a single flip-flop to the negative edge of the clock. The downside to this is that it might be possible to have a slight shift in the duration between each count. Because of this it is important to make a clock divider which has minimal delay between its negative and positive clock output. The upside is that the accuracy of the counter has been doubled, by only adding 1 single flip-flop! Furthermore the system is still a "true" Gray counter.

Table 7: Addition of the extra bit.

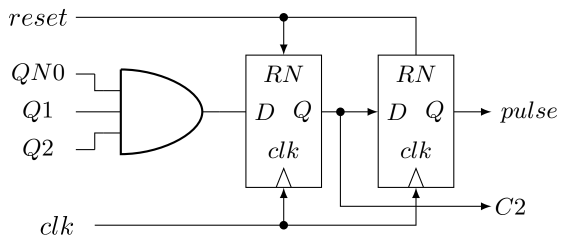
Count	$NegativeBit$	A	B	C	Count	$NegativeBit$	A	B	C
0	0	0	0	0	8	0	1	1	0
1	1	0	0	0	9	1	1	1	0
2	1	0	0	1	10	1	1	1	1
3	0	0	0	1	11	0	1	1	1
4	0	0	1	1	12	0	1	0	1
5	1	0	1	1	13	1	1	0	1
6	1	0	1	0	14	1	1	0	0
7	0	0	1	0	15	0	1	0	0

5.2.6 Implementation

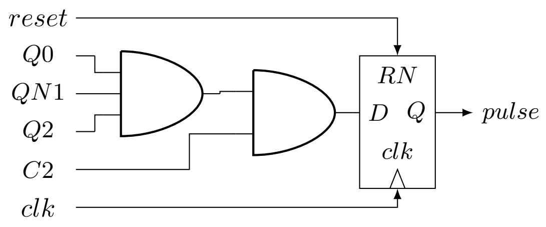
The implementation of the device seems straightforward, however some small changes have to be made in order to fill in the basic architecture. Again, a more elaborate explanation can be found in appendix A. First the connection block will explained, and then the changes that were made to the second and third counting blocks.

Connection blocks The connection blocks provide a pulse, which starts a clock cycle before the second and third 3-bit blocks should count. The pulse duration is exactly one count duration long, so the clock cycle afterwards, the second and third block will count. The first connection block is depicted in Figure 27a. It needs only 1 AND gate to start the pulse, but in order to make the whole design clocked and keep the longest path at just one flip-flop and one gate, additional flip-flops are needed. In between each gate a flip-flop is placed in order to guarantee that the longest path of the device stays 1 flip-flop and 1 gate. This means that the signal for when the next 3-bit block should count, already starts its propagation path at count 011. The second connection block is depicted in figure 27b. The third 3-bit block should only count if the the second block has to count too, which is the $C2$ connection depicted in both figures. As shown, an additional AND-gate is needed. As the 3-bit AND gate already has the value ready for a long time, the 2 AND-gates are not in need of another flip-flop in between, and the count will be set ready at 101.

2nd and 3rd counting blocks As can be expected, the second and third blocks operate a bit different, as they should count depending on the clock and a pulse, where the first 3-bit block is only dependent on the clock. Simply using an AND-gate with the clock and the pulse connected does not work in this case, as it will influence the length of the counts significantly as a delay in the clock is generated for two of the three 3-bit blocks that way. This is why I have opted to make use of enable-disable D-flip-flops. The pulse that comes out of the connection block connected to one of the 3-bit blocks will be connected to the enable input. The enable D-flip-flops have been made by simply adding a



(a) Connection block to 3-bit block 2.



(b) Connection block to 3-bit block 3.

Figure 27: The connection blocks (a) and (b) generating the pulses to 3-bit block 2 and 3 respectively.

MUX in front of the D-flip-flop with one of the connections connected to the current output, and one to the calculated next bit. This does not cause any problems concerning the longest path of the system, as the second and third counting blocks have at least 7 counts to get the value from the feedback MUX ready at the input of the enable-disable MUX. This thus creates a longest path from the enable-disable MUX to the flip-flop, thus only two gates.

Problems There is one problem with this system, as one in the eight counts there is a double bit flip. This can be easily solved by adding another flip-flop, or by calling two bits the same bit. This system however requires the full 10-bit scale, and cannot output 11 bits, as the connections for 10 bits are already there. This is thus a solvable problem, however there is not enough design room available in this particular application. The total system still meets the requirements as it stated that the system should have minimal to no glitches, where one in the eight counts a double bit flip can be called minimal glitches during transitions. And again, this problem can be easily solved in the future designs by providing an 11-bit output for a 10-bit counter, or by designing an on chip decoder which outputs 10 bits again.

5.2.7 Results

The schematic of the final design is depicted in Figure 28 and the layout is depicted in Figure 29.

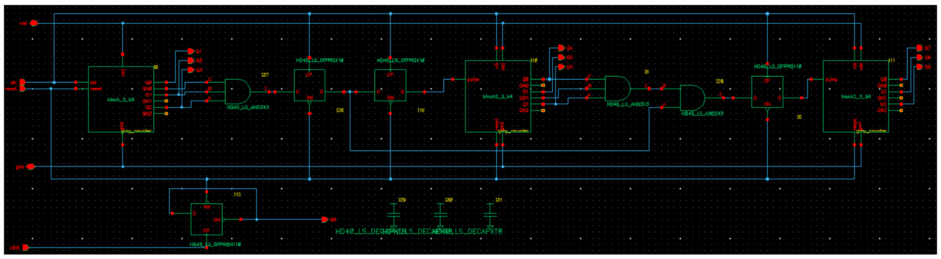


Figure 28: Final schematic of the 10-bit Gray counter.

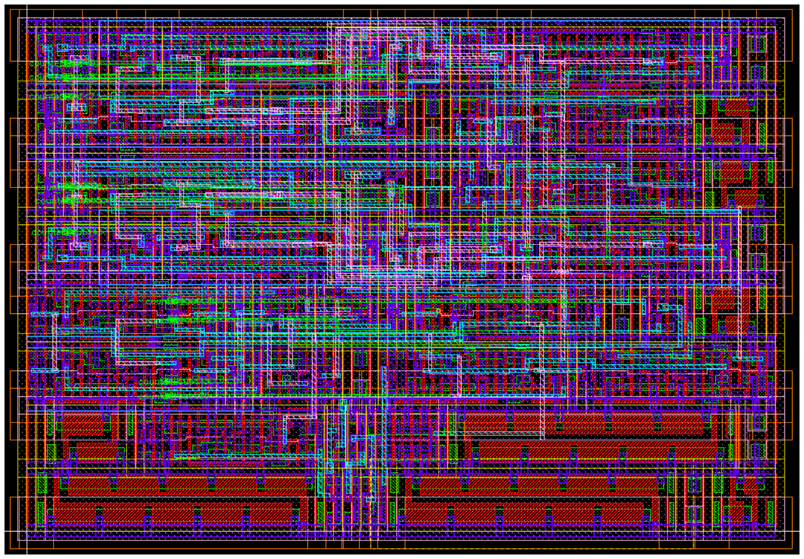


Figure 29: Implemented layout of the 10-bit Gray counter.

The design has been integrated in the rest of the POLIS project, and being taped out at the time of writing. The size of counter itself is $14.95\,x\,10.20\,\mu m^2$. Extracted simulations while under-volting at $1V$, using SS corners run completely stable at $2.25\,GHz$. These are the most pessimistic circumstances for a design in this technology. Running the extracted simulations of the device at normal conditions using $1.1V$ with TT corners result in a stable $3.6\,GHz$.

5.3 Conclusion

It can be concluded that the properties of the designed counter are for this specific system are a vast improvement over a normal synchronous counter for this, and more potential applications. While being more complicated than a normal synchronous counter, the Gray counter is not slow at all because of all the small design changes to keep the longest path the same as its simple counterpart. As it is able to run at $3.6\,GHz$ under normal conditions, and has only one double bit flip once in the 8 counts. The only future work that can be done on the system is to remove this double bit flip once in the eight counts, which can be done by using an additional flip-flop in the system.

When the whole system operates as designed, it will be easier for QRNG designs to be tested in an early stage of the process. This will significantly increase the speed in which ideas can be tested. Furthermore it will increase the results of the taped-out QRNGs as the device was can be accurately tested in advance on an FPGA with this chip attached.

Concluded it can be said that the coarse counter project was successful and an addition towards a multi-functional device. This device can then be used as an universal QRNG testbed system that will help in the design process of future QRNG designs.

6 Conclusion

There are a lot of ways to use SPADs for random number generation, of which different aspects have been explored. Having a very small QRNG opens up a lot of possibilities, mainly when looking to integrate a QRNG in already existing PRNG devices in order to create a device which can use the advantages of both systems. In this thesis a very small QRNG was made using only one T-flip-flop and one SPAD. Having run simulations in Matlab, it is shown that it is possible to undo the correlation to its last read out value over time. This however comes with the requirement that the illumination levels are adjusted to the deadtime of the SPAD. When this is done the system is able to reach a speed of $25\,Mb/s$, while only existing out of one SPAD and one flip-flop. The architecture has been made on a chip and has been produced using 130nm technology provided by STMicroelectronics.

The second part of the exploration explores how to design a SPAD based QRNG which is as fast as possible. This was not a new subject, however the area of building a QRNG from the ground up with the goals of speed and scalability in mind in CMOS process is a largely uncovered field. The fast scalable QRNG proposed in this thesis is based on a comparator system. Two SPADs are used, and the first one to receive a photon will determine if a $'1'$ or $'0'$ is set. The system is simulated, showing that the deadtime is the mainly dominant factor, next to the amount of overlapping pulses that could occur if both SPADs fired within a certain timeframe. The design is then made synchronous by the use of the small QRNG architecture as a backup bit. It is possible, depending on the deadtime, to run the comparator based system up to $70\,Mb/s$ per SPAD duo. When comparing the scalability of this design to the state-of-the-art using simulations, it is able reach the same speed of $5\,Gb/s$ as the CMOS imager, while only needing 16% of the amount of SPADs needed when assuming an average SPAD with a deadtime of $1\,\mu s$.

The next step of the exploration explores the effect a QRNG can have on already existing PRNG systems. These hybrid systems have essentially two ways in which a QRNG can be integrated in a PRNG. By generating a seed for a very strong PRNG, or by using a QRNG to actively change the function that is implemented by a QRNG. The first way of making hybrid systems is demonstrated by building a system in which one QRNG provides the key for 4 triviums. This has been implemented using VHDL and DC Compiler, assuming 130nm Technology by STMicroelectronics. This resulted in a design that is able to reach a speed of $640\,Gb/s$ while using an area of $99936\,\mu m^2$.

The second way a hybrid device can be implemented is by using a small QRNG to change the feedback path of a LFSR for a small as possible increase in area. If the feedback is c hanged before $2m$ outputs have been generated, the LFSR can no longer be cracked by a plaintext attack. With this design it is shown that using a QRNG to change the function implementation of a PRNG can improve the security of a PRNG to a point the PRNG is no longer vurnable to a plain text attack.

In order to be able to test all these systems, a lot of time is needed which makes the time needed for a bigger exploration out of bounds. The time that is needed would however be significantly reduced by having a chip that can extract all the information about photons. By extracting the location where the photons hit, and the time of flight, all of the previous systems could easily be tested with real data. Just by implementing the main architecture on an FPGA, except for the SPADs, very accurate measurements can be done before taping-out the new QRNG design. A chip that can do this has been made during the thesis, where I designed the coarse counter for measuring the time of flight as part of a team. The coarse counter is an implementation of Gray counting, as the system had to be as accurate as possible. Usually Gray counters are not used because of their size and slow speeds. However in this case novelties were introduced to enable the counter to reach speeds up to $3.6\,GHz$ under normal conditions with extracted simulations, while remaining much more accurate than an ordinary counter. The chip is currently being taped out in 40nm technology by STMicroelectronics.

Concluding, the exploration in the domain of SPAD based QRNGs has proven to be successful. It is shown how small a QRNG can still be quite fast and advantageous for other designs. The high-speed QRNG appears to be a big improvement compared to the state-of-the-art in terms of amount of SPADs needed. Furthermore it is shown how QRNGs can improve the security of already existing PRNG systems. It turned out to take a lot of time to completely test an architecture. This did however result in the idea to use an already being developed chip and exploit it for quantum random number generating. This way it will be much easier in the future to test SPAD based QRNG designs in an early design stage.

References

[1] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[2] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[3] ID Quantique. "White paper: Random number generation using quantum physics". In: *ID Quantique SA, Switzerland, Tech. Rep. Version* 3 (2010).

[4] Matthew W Fishburn. *Fundamentals of CMOS single-photon avalanche diodes*. fishburn, 2012.

[5] E Charbon. "Single-photon imaging in complementary metal oxide semiconductor processes". In: *Phil. Trans. R. Soc. A* 372.2012 (2014), p. 20130100.

[6] C Trivium De Canniere. "A stream cipher construction inspired by block cipher design principles". In: *Information Security* (2006), pp. 171–186.

[7] W Timothy Holman, J Alvin Connelly, and Ahmad B Dowlatabadi. "An integrated analog/digital random noise source". In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 44.6 (1997), pp. 521–528.

[8] JT Gleeson. "Truly random number generator based on turbulent electroconvection". In: *Applied physics letters* 81.11 (2002), pp. 1949–1951.

[9] Thomas Jennewein et al. "A fast and compact quantum random number generator". In: *Review of Scientific Instruments* 71.4 (2000), pp. 1675–1680.

[10] Feihu Xu et al. "Ultrafast quantum random number generation based on quantum phase fluctuations". In: *Optics express* 20.11 (2012), pp. 12366–12377.

[11] Masatugu Isida and Hiroji Ikeda. "Random number generator". In: *Annals of the Institute of Statistical Mathematics* 8.1 (1956), pp. 119–126.

[12] S Cova et al. "Evolution and prospects for single-photon avalanche diodes and quenching circuits". In: *Journal of modern optics* 51.9-10 (2004), pp. 1267–1288.

[13] Robert GW Brown et al. "Characterization of silicon avalanche photodiodes for photon correlation measurements. 2: Active quenching". In: *Applied Optics* 26.12 (1987), pp. 2383–2389.

[14] F Zappa et al. "Monolithic active-quenching and active-reset circuit for single-photon avalanche detectors". In: *IEEE Journal of solid-state circuits* 38.7 (2003), pp. 1298–1301.

[15] Thomas Frach et al. "The digital silicon photomultiplier—Principle of operation and intrinsic detector performance". In: *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*. IEEE. 2009, pp. 1959–1965.

[16] Samuel Burri et al. “Jailbreak imagers: transforming a single-photon image sensor into a true random number generator”. In: *International Image Sensor Workshop*. EPFL-CONF-191217. 2013.

A Small Integrable QRNG Matlab Simulations

```
clc;
close all;
clear;

colorspec = {[0.4 0 0.8]; [0.4 0.8 0]; [0.4 0.7 0.7]; ...
  [0 0.4 0.8]; [0.8 0.4 0]; [0.7 0.4 0.7]; ...
  [0.8 0 0.4]; [0 0.8 0.4]; [0.7 0.7 0.4]; ...
  [0 0 0.7]; [0 0.7 0]; [0.7 0 0]};

colorspec = {...
[0.0 0 1.0]; ...
[0.2 0 0.8]; ...
[0.4 0 0.6]; ...
[0.6 0 0.4]; ...
[0.8 0 0.2]; ...
[1.0 0 0.0]; ...
};

%graphics_toolkit gnuplot;
%figure ("visible", "off");

PPS = [5e4 5e5 5e6]; % [Hz]

endTime = 100e-6; % [s]
stepSize = 1e-9; % [s]
deadTime = 10000e-9; % [s]
deadTimeSteps = round(deadTime/stepSize);

%endTime = 2e-3; % [s]
%stepSize = 1e-6; % [s]

waitingTime = 0:stepSize:endTime;
iterations = 50e3; % [-]
values = zeros(1,length(waitingTime));
legendString = {};

for pps = PPS
    P_hit = 1-poisspdf(0, pps*stepSize) % chance that one or more
     ↪ photons arrive during t_stepT
    for dTime = deadTimeSteps
        for j=1:10
            fprintf('%d0\n', j);
            for i=1:iterations/10
                new_value = zeros(1,length(waitingTime));
                current = 0;
                dead = 0;
                for t=1:length(waitingTime)
                    if dead>0
                        dead = dead - 1;
                    elseif rand<P_hit
                        current = mod(current+1,2);  % flip the
                         ↪  current value
                        dead = dTime;
                    end
                    new_value(t) = current;
                    %dead = dead - 1;
                end
                values = values + new_value;
            end
        end
    values = values./iterations;
    values2 = -(values).*log2(values) -
     ↪  (1-values).*log2(1-values);
    %plot(waitingTime.*1e6, values);
    filename = sprintf('pps-%d_dead-%dns2.csv', pps,
     ↪  1e9*stepSize*deadTimeSteps);

    fid = fopen(filename, 'w');
    fprintf(fid, 'time values2\n');
    fclose(fid);


    waitingTimePrint = waitingTime(1:100:length(waitingTime))';
    valuesPrint = values2(1:100:length(values2))';

    dlmwrite(filename, [waitingTimePrint valuesPrint],
     ↪  '-append','Delimiter', ' ');
    %legendString{end+1} = sprintf('PPS = %d', pps);


    filename = sprintf('pps-%d_dead-%dns2.csv', pps,
     ↪  1e9*stepSize*deadTimeSteps);

    fid = fopen(filename, 'w');
    fprintf(fid, 'time values\n');
    fclose(fid);

    waitingTimePrint = waitingTime(1:100:length(waitingTime))';
    valuesPrint = values(1:100:length(values))';
```

```
    dlmwrite(filename, [waitingTimePrint valuesPrint],
     ↪  '-append','Delimiter', ' ');
    end
end

%plot(waitingTime*1e6, 0.5*ones(1,length(values)), 'k');
%hold off;

%plot(C1(:,1),C1(:,4), 'Color', colorspec{mod(i,12)+1});
%axis([C1(1,1) C1(end,1) min(min(C1))*1.1 max(max(C1))*1.1]);
%xlim([waitingTime(1)*1e6, waitingTime(end)*1e6]);
%ylim([1e-1, 1e2]);
%xlabel('time [us]', 'fontsize', 14);
%ylabel('probability of measuring ''1''', 'fontsize', 14);
%set(gca, 'FontSize', 12)


%legend(legendString, 'Location', 'northeast');
%title(sprintf('deadtime = %d us', 1e6*stepSize*deadTimeSteps));
%
%print('-dpdf', '-color', fullfile(pwd, 'sneep.pdf'));
%print('-deps', '-color', fullfile(pwd, 'lineplot.eps'));
```

B SCS-Block Matlab Simulations

```
clc;
close all;
clear;

deadTime = [1e-6 100e-9 10e-9]; % [s]
pulseTime = 1e-9;

% Amount of runs, based on different illumination levels
beginPPS = 4;
endPPS = 10;
NumberPoints = 1000;
PPSstepSize = 1;

% Time for which the simluation will run for each PPS
↪  independently
endTime = 2e-3; % [s]
stepSize = 1e-9; % [s]
waitingTime = 0:stepSize:endTime;

%PPS will be looped at the end
PPS = 5e5;
pps =PPS;
P_hit = 1-poisspdf(0, pps*stepSize);

% Kijk, die
new_value = zeros(1,length(waitingTime));
current = 0;
dead = 0;
pulse = 0;
spad1 = 0;
spad2 = 0;

outputString = [0 0];
count = [0 0 0];
overlap = [0 0 0];
overlapoutputString = [0 0 0 0];

fprintf('%d\n', P_hit);

X = logspace(beginPPS,endPPS,NumberPoints);
PPSoutputString = zeros(NumberPoints,4);
overlapoutputString = zeros(NumberPoints,4);

%For loop for the X-axis -> PPS
```

```
for n = 1:NumberPoints
    currentPPS = X(n);
    P_hit = 1-poisspdf(0, currentPPS*stepSize);
    %For loop for the deadtimes

    for i=1:3
        count(i) = 0;
        overlap(i) = 0;
        for t=1:length(waitingTime)
            %A SPAD has fired!
            if pulse>0
                pulse = pulse - stepSize;
                %Pulse is over, check what happened
                if pulse <=0
                    if spad1 == 1 && spad2 == 0
                        %outputString = [outputString;
                         ↪  (waitingTime(t)-1e-12)
                         ↪  outputString(end)];
                        %outputString = [outputString;
                         ↪  waitingTime(t) 0];
                        %fprintf('0');
                        count(i) = count(i) + 1;
                    elseif spad1 == 0 && spad2 == 1
                        %outputString = [outputString;
                         ↪  (waitingTime(t)-1e-12)
                         ↪  outputString(end)];
                        %outputString = [outputString;
                         ↪  waitingTime(t) 1];
                        %fprintf('1');
                        count(i) = count(i) + 1;
                    end
                    if spad1 == 1 && spad2 == 1
                        spad1 = 1;
                        spad2 = 1;
                        overlap(i) = overlap(i) + 1;
                        %fprintf('2');
                    end
                end
            elseif dead>0
                dead = dead - stepSize;
                if dead<=0
                    spad1=0;
                    spad2=0;
                end
            else
                if rand<P_hit
```

```
                    spad1 = 1;  % flip the current value
                    dead  = deadTime(i)-pulseTime;
                    pulse = pulseTime;
                end
                if rand<P_hit
                    spad2 = 1;  % flip the current value
                    dead  = deadTime(i)-pulseTime;
                    pulse = pulseTime;
                end
            end
            new_value(t) = current;
            %dead = dead - 1;
        end
        fprintf('\n');
    end



    count = count/endTime
    PPSoutputString(n,:) = [ currentPPS count(1) count(2)
    ↪  count(3)];
    overlap = overlap/endTime
    overlapoutputString(n,:) = [ currentPPS overlap(1) overlap(2)
    ↪  overlap(3)];

    currentPPS = currentPPS+beginPPS
end
semilogx(PPSoutputString(:,1),PPSoutputString(:,2), 'Linewidth',
↪  2);
hold on;
semilogx(PPSoutputString(:,1),PPSoutputString(:,3), 'Linewidth',
↪  2);
hold on;
semilogx(PPSoutputString(:,1),PPSoutputString(:,4), 'Linewidth',
↪  2);
hold off;

figure;
semilogx(overlapoutputString(:,1),overlapoutputString(:,2),
↪  'Linewidth', 2);
hold on;
semilogx(overlapoutputString(:,1),overlapoutputString(:,3),
↪  'Linewidth', 2);
hold on;
semilogx(overlapoutputString(:,1),overlapoutputString(:,4),
↪  'Linewidth', 2);
```

```
hold off;
%plot(outputString(:,1),outputString(:,2), 'Linewidth', 2);
%title(sprintf('count: %d', count));

[Max1,Loc1] = max(PPSoutputString(:,2))
[Max2,Loc2] = max(PPSoutputString(:,3))
[Max3,Loc3] = max(PPSoutputString(:,4))

X(Loc1)
X(Loc2)
X(Loc3)

    filename = sprintf('SCS.csv');

    fid = fopen(filename, 'w');
    fprintf(fid, 'time values\n');
    fclose(fid);


    %waitingTimePrint = waitingTime(1:100:length(waitingTime))';
    %valuesPrint = values2(1:100:length(values2))';

    dlmwrite(filename, [PPSoutputString(:,1)
    ↪  PPSoutputString(:,2)], '-append','Delimiter', ' ');

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        filename = sprintf('SCS1.csv');

    fid = fopen(filename, 'w');
    fprintf(fid, 'time values\n');
    fclose(fid);


    %waitingTimePrint = waitingTime(1:100:length(waitingTime))';
    %valuesPrint = values2(1:100:length(values2))';

    dlmwrite(filename, [PPSoutputString(:,1)
    ↪  PPSoutputString(:,3)], '-append','Delimiter', ' ');

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        filename = sprintf('SCS2.csv');

    fid = fopen(filename, 'w');
    fprintf(fid, 'time values\n');
```

```
fclose(fid);


%waitingTimePrint = waitingTime(1:100:length(waitingTime))';
%valuesPrint = values2(1:100:length(values2))';

dlmwrite(filename, [PPSoutputString(:,1)
↪  PPSoutputString(:,4)], '-append','Delimiter', ' ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    filename = sprintf('SCSover.csv');

fid = fopen(filename, 'w');
fprintf(fid, 'time values\n');
fclose(fid);


%waitingTimePrint = waitingTime(1:100:length(waitingTime))';
%valuesPrint = values2(1:100:length(values2))';

dlmwrite(filename, [PPSoutputString(:,1)
↪  overlapoutputString(:,2)], '-append','Delimiter', ' ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    filename = sprintf('SCSover1.csv');

fid = fopen(filename, 'w');
fprintf(fid, 'time values\n');
fclose(fid);


%waitingTimePrint = waitingTime(1:100:length(waitingTime))';
%valuesPrint = values2(1:100:length(values2))';

dlmwrite(filename, [PPSoutputString(:,1)
↪  overlapoutputString(:,3)], '-append','Delimiter', ' ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    filename = sprintf('SCSover2.csv');

fid = fopen(filename, 'w');
fprintf(fid, 'time values\n');
fclose(fid);
```

```
%waitingTimePrint = waitingTime(1:100:length(waitingTime))';
%valuesPrint = values2(1:100:length(values2))';

dlmwrite(filename, [PPSoutputString(:,1)
↪  overlapoutputString(:,4)], '-append','Delimiter', ' ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

C DC Compiler Script

```
# Simple DC script for the Simple Scan Register
#
#
# Analyze VHDL sources
#

set ENTITY_NAME Control_System
set ARCH_NAME Structural
set CLK_NAME1 CLK1
set CLK_NAME2 CLK2
set CLK_PERIOD1 1000;#ns
set CLK_PERIOD2 2.5;#ns

set DESIGN_ENTITY "${ENTITY_NAME}_${ARCH_NAME}"
set DESIGN "${ENTITY_NAME}_clk${CLK_PERIOD2}ns"

# Check syntax
analyze -format vhdl { \

  HDL/RTL/JK_FF.vhd \
  HDL/RTL/Trivium_64_Bit.vhd \
  HDL/RTL/5_Bit_Counter.vhd \

  HDL/RTL/9_Bit_Counter.vhd \
  HDL/RTL/Reset_Controller.vhd \
  HDL/RTL/320_Bit_Shift_Register.vhd \
  HDL/RTL/SR_Latch.vhd \
  HDL/RTL/1_Bit_2_to_1_Multiplexer.vhd \

  HDL/RTL/Trivium_Control.vhd \
  HDL/RTL/Key_Control.vhd \

  HDL/RTL/Control_System.vhd\
}


# Elaborate design
# Check if synthesizable, will give a warning
elaborate ${ENTITY_NAME} -library DEFAULT
# Save elaborated design
#
write -hierarchy -format ddc -output DB/${DESIGN}.ddc
```

```
# Link design
#

# Define constraints
#
create_clock -name ${CLK_NAME1} -period ${CLK_PERIOD1} -waveform
    ↪ { 0 1.25 } { CLK1 }
create_clock -name ${CLK_NAME2} -period ${CLK_PERIOD2} -waveform
    ↪ { 0 500 } { CLK2 }
set_max_area 0
#set_load 10 [all_outputs]

current_design ${ENTITY_NAME}
set_fix_multiple_port_nets -all
check_design
# Force the use of scan FF
#set_register_type -exact -flip_flop DFSC3_HV

uniquify
check_design

ungroup -all -flatten
check_design

compile_ultra
check_design

optimize_netlist -area
check_design

link


#check_design
#compile -map_effort high -area_effort high -
    ↪ boundary_optimization
#check_design
# Map and optimize design
#
#compile -map_effort medium -area_effort medium
#ungroup -all -flatten


# Save mapped design
write -hierarchy -format ddc -output DB/${DESIGN}_mapped.ddc
```

```
# Generate reports
report_constraint -nosplit -all_violators > RPT/${DESIGN}
    ↪ _mapped_allviol.rpt
report_area > RPT/${DESIGN}_mapped_area.rpt
report_timing > RPT/${DESIGN}_mapped_timing.rpt
report_resources -nosplit -hierarchy > RPT/${DESIGN}
    ↪ _mapped_resources.rpt

# Generate Verilog netlist
change_names -rule verilog -hierarchy
write -format verilog -hierarchy -output HDL/GATE/${DESIGN}
    ↪ _mapped.v

# Generate SDF timing file
write_sdf -version 2.1 TIM/${DESIGN}_mapped.sdf

# Generate design constraint file
write_sdc -nosplit SDC/${DESIGN}_mapped.sdc

# Generate the VHDL netlist
remove_design -all
read_file -format ddc DB/${DESIGN}_mapped.ddc
change_names -rule vhdl -hierarchy
write -format vhdl -hierarchy -output HDL/GATE/${DESIGN}_mapped.
    ↪ vhd


# prepare for the scan chain insertion
report_dft_signal -view exist

# Connect the scan protocol to existing signals
set_dft_signal -view spec -port ScanEnablexSI -type ScanEnable
set_dft_signal -view spec -port ScanDataInxDI -type ScanDataIn
set_dft_signal -view spec -port ScanDataOutxDO -type ScanDataOut
#set_dft_signal -view existing_dft -type ScanClock -port ClkxCI -
    ↪ timing [list 45 55]
#set_dft_signal -view existing_dft -type Reset -active 0 -port
    ↪ ResetxRBI -timing [list 55 45]
#set_scan_config -exclude_elements i_controlUnit*

report_dft_signal -view spec

create_test_protocol -infer_async -infer_clock
dft_drc
#
#preview_dft -show cells
```

```
preview_dft -show all
#preview_dft
insert_dft

report_constraint -nosplit -all_violators > RPT/${DESIGN}
    ↪ _scan_mapped_allviol.rpt
report_area > RPT/${DESIGN}_scan_mapped_area.rpt
report_timing > RPT/${DESIGN}_scan_mapped_timing.rpt
report_resources -nosplit -hierarchy > RPT/${DESIGN}
    ↪ _scan_mapped_resources.rpt

write -hierarchy -format ddc -output DB/${DESIGN}_scan_mapped.ddc
remove_design -all
read_file -format ddc DB/${DESIGN}_scan_mapped.ddc

# generate the vhdl netlist
change_names -rule vhdl -hierarchy
write -format vhdl -hierarchy -output HDL/GATE/${DESIGN}
    ↪ _scan_mapped.vhd

change_names -rule verilog -hierarchy
write -format verilog -hierarchy -output HDL/GATE/${DESIGN}
    ↪ _scan_mapped.v

# Generate SDF timing file
write_sdf -version 2.1 TIM/${DESIGN}_scan_mapped.sdf
# # Generate design constraint file
write_sdc -nosplit SDC/${DESIGN}_scan_mapped.sdc

# Quit the script
exit
```

D Control System VHDL

```
-----------------------------------------------------------
-------------------- LIBRARY DECLARATIONS ----------------
-----------------------------------------------------------
library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------------------------
-------------------- RESET CONTROLLER ENTITY  ------------
-----------------------------------------------------------
entity Control_System is
    Port
        (
        CLK1         : in    STD_LOGIC;     -- Slow Clock for the
        ↪  Control System
        CLK2         : in    STD_LOGIC;     -- Fast clock for the
        ↪  triviums
        RESET         : in    STD_LOGIC;  -- Reset signal for the
        ↪  circuit, active low input
        RESEED         : in    STD_LOGIC;     -- If this signal is
        ↪  1, it will reseed the trivium as soon a new key is
        ↪  ready
        TENABLE    : in    STD_LOGIC;     -- Pause output signal,
        ↪  by pausing the Clock 2 output when circuit is warmed
        ↪  up
        KEY_STREAM    : in    STD_LOGIC;     -- Connect the
        ↪  external key generator to it, in this case our QRNG
        RNGSELECT     : in    STD_LOGIC;  -- 0 is the QRNG and 1
        ↪  the other input
        KEY_STREAM2    : in    STD_LOGIC;  -- The external RNG
        ↪  that can be connected to the system.

        T1OUT         : out    STD_LOGIC_VECTOR(63 downto 0); --
        ↪  64 bit output of Trivium 1
        T2OUT         : out    STD_LOGIC_VECTOR(63 downto 0); --
        ↪  64 bit output of Trivium 2
        T3OUT         : out    STD_LOGIC_VECTOR(63 downto 0); --
        ↪  64 bit output of Trivium 3
        T4OUT         : out    STD_LOGIC_VECTOR(63 downto 0); --
        ↪  64 bit output of Trivium 4
        TREADY         : out    STD_LOGIC; -- Feedback on
        ↪  triviums, if they are warmed up
        KEYREADY     : out    STD_LOGIC; -- Feedback to the user
        ↪  if the system can be reseeded
        -- Scan map in and outputs
        ScanEnablexSI    : in    STD_LOGIC;
        ScanDataInxDI    : in    STD_LOGIC;
        ScanDataOutxDO    : out    STD_LOGIC
        );
end Control_System;

----------------------------------------------------------
------------------- SYSTEM CONTROLLER ARCHITECTURE -------
----------------------------------------------------------

architecture Structural of Control_System is

    ----------------------------------------------------------
    ------------------    COMPONENTS      --------------------
    ----------------------------------------------------------
    component Key_Control is
    Port
        (
        CLK        : in STD_LOGIC;  -- Slow Clock for the Control
        ↪  System
        RESET        : in STD_LOGIC;  -- Reset signal for the
        ↪  circuit, active low input
        RESEED        : in STD_LOGIC;  -- Signal to reseed the
        ↪  key
        QRNG        : in STD_LOGIC;  -- Input of the QRNG bit
        ↪  stream
        RNGSELECT    : in STD_LOGIC;  -- 0 is the QRNG and 1 the
        ↪  other input
        RNG_2        : in STD_LOGIC;  -- The external RNG that
        ↪  can be connected to the system.

        LOADKEY        : out STD_LOGIC; -- LOADKEY signal, which
        ↪  gives the command to load the key
        KEY            : out STD_LOGIC_VECTOR (319 downto 0); --
        ↪  Contains the 320 bit key connected to the Trivium
        ↪  Control
        KEYREADY    : out STD_LOGIC -- Feedback to the user if
        ↪  the system can be reseeded
        );
    end component;

    component Trivium_Control is
    Port
        (
        CLK        : in STD_LOGIC;  -- Slow Clock for the Control
        ↪  System
        RESET        : in STD_LOGIC;  -- Reset signal for the
        ↪  circuit, active low input
        LOADKEY        : in STD_LOGIC;  -- Signal that gives the
        ↪  ready signal to the Trivium to readout the key and
        ↪  initialize
        KEY        : in STD_LOGIC_VECTOR(319 downto 0);  --
        ↪  Connected to the QRNG control block, which contains
        ↪  the key

        TREADY        : out STD_LOGIC; -- Wether the trivium is
        ↪  warmed up or not, so you know when it can be paused,
        ↪  a 0 is warmed up, 1 is still warming up
        T1OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 1
        T2OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 2
        T3OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 3
        T4OUT        : out STD_LOGIC_VECTOR(63 downto 0)  -- 64
        ↪  bit output of Trivium 4
        );
    end component;

    -----------------------------------------------------------
    -------------------- SIGNALS          ---------------------
    -----------------------------------------------------------
    signal LoadKey_S    : std_logic;
    signal Key_S        : std_logic_vector(319 downto 0);
    signal Clock2_S        : std_logic;
    signal Tready_S        : std_logic;

begin
    -----------------------------------------------------------
    -----------    INSTANTIATE COMPONENTS    -----------------
    -----------------------------------------------------------

    -- Instantiate the Shift Register
    Key_Control_Block : Key_Control
    PORT MAP (
        -- Inputs
        CLK            => CLK1,
        RESET        => RESEED,
        RESEED        => RESEED,
        QRNG        => KEY_STREAM,
        RNGSELECT    => RNGSELECT,
        RNG_2        => KEY_STREAM2,


        -- Outputs
        LOADKEY        => LoadKey_S,
        KEY            => Key_S,
        KEYREADY    => KEYREADY
    );

    -- Instantiate the Reset Controller
    Trivium_Control_Block : Trivium_Control
    PORT MAP (
        -- Inputs
        CLK        => Clock2_S,
        RESET        => RESET,
        LOADKEY        => LoadKey_S,
        KEY        => Key_S,

        -- Outputs
        TREADY        => Tready_S,
        T1OUT        => T1OUT,
        T2OUT        => T2OUT,
        T3OUT        => T3OUT,
        T4OUT        => T4OUT
    );

    -----------------------------------------------------------
    ------------------    CONNECTIONS    ---------------------
    -----------------------------------------------------------

    -- Enables pausing the trivium WHEN warmed up, before it's
     ↪  warmed up it won't have effect
    Clock2_S <= CLK2; --(Tready_S XOR TENABLE) AND CLK2;

    -- Signal to enable feedback to the user if the triviums are
     ↪  warmed up, a 1 means warmed up
    TREADY <= Tready_S;


    -----------------------------------------------------------
    --------------------    PROCESS      --------------------
    -----------------------------------------------------------

    -----------------------------------------------------------
    -------------    COMPARISON STATEMENTS   -----------------
```

```
    -----------------------------------------------------------

end Structural;
```

E Key Control VHDL

```
----------------------------------------------------------
-------------------- LIBRARY DECLARATIONS ----------------
----------------------------------------------------------
library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------------------------
-------------------- RESET CONTROLLER ENTITY  ------------
----------------------------------------------------------
entity Key_Control is
    Port
        (
        CLK        : in STD_LOGIC;  -- Slow Clock for the Control
        ↪  System
        RESET        : in STD_LOGIC;  -- Reset signal for the
        ↪  circuit, active low input
        RESEED        : in STD_LOGIC;  -- Signal to reseed the
        ↪  key
        QRNG        : in STD_LOGIC;  -- Input of the QRNG bit
        ↪  stream
        RNGSELECT    : in STD_LOGIC;  -- 0 is the QRNG and 1 the
        ↪  other input
        RNG_2        : in STD_LOGIC;  -- The external RNG that
        ↪  can be connected to the system.

        LOADKEY        : out STD_LOGIC; -- LOADKEY signal, which
        ↪  gives the command to load the key
        KEY            : out STD_LOGIC_VECTOR (319 downto 0); --
        ↪  Contains the 320 bit key connected to the Trivium
        ↪  Control
        KEYREADY    : out STD_LOGIC -- Feedback to the user if
        ↪  the system can be reseeded
        );
end Key_Control;

----------------------------------------------------------
------------------- SYSTEM CONTROLLER ARCHITECTURE -------
----------------------------------------------------------

architecture Structural of Key_Control is

    ----------------------------------------------------------
    ------------------    COMPONENTS      --------------------
    ----------------------------------------------------------
```

```
component Counter9Bit is
Port (    CLK     : in  STD_LOGIC;
        ENABLE    : in  STD_LOGIC;
        RESET     : in  STD_LOGIC;
        OUTPUT    : out STD_LOGIC_VECTOR (8 downto 0));
end component;

    -- Reset Block NOT USED
component reset_controller is
Port (
    RESET_ASYNC : in  STD_LOGIC;
    CLK         : in  STD_LOGIC;
    RESET_SYNC  : out STD_LOGIC
    );
end component;

component ShiftRegister320Bit is
Port ( DATA    : in  STD_LOGIC; -- Per bit input
       CLK        : in  STD_LOGIC;
       ENABLE    : in  STD_LOGIC;
       OUTPUT    : out STD_LOGIC_VECTOR(319 downto 0)
   );
end component;


component S_R_latch is
Port ( S : in    STD_LOGIC;
       R : in    STD_LOGIC;
       Q : out   STD_LOGIC);
end component;

component Multiplexer_2_to_1 is
Port (    INPUT0    : in STD_LOGIC;
        INPUT1  : in STD_LOGIC;
        SEL        : in STD_LOGIC;
        OUTPUT    : out STD_LOGIC);
end component;

----------------------------------------------------------
-------------------- SIGNALS          --------------------
----------------------------------------------------------

-- Signals for the 9 bit counter
signal Counter9Bit_SO : std_logic_vector(8 downto 0);
signal Counter9BitActivate_S : std_logic;
```

```
    signal Counter9BitReset_S : std_logic;

    signal Comparator_S : std_logic;

    signal ReseedKeyReady_SAsync : std_logic; -- Signal that
     ↪  indicates if the reseed key is ready Async
    signal ReseedKeyReady_SSync : std_logic; -- Signal that
     ↪  indicates if the reseed key is ready after flipflop, so
     ↪  sync
    signal FirstTimeSeeding_S : std_logic;
    signal ProcSeed_S : std_logic;

    signal LoadKey_S : std_logic;
    signal Reseed_S : std_logic;

    signal SRLATCH_R_S : std_logic;

    signal ShiftRegActivate_S : std_logic;

    signal RandomNumber_S : std_logic; -- The signal connecting
     ↪  CounterBlock9Bit and the Multiplexer
begin
    ----------------------------------------------------------
    -----------    INSTANTIATE COMPONENTS    -----------------
    ----------------------------------------------------------

    -- Instantiate the 11 bit counter
    CounterBlock9Bit : Counter9Bit
    PORT MAP (
        CLK => CLK,
        ENABLE    => Counter9BitActivate_S,
        RESET    => Counter9BitReset_S,
        OUTPUT    => Counter9Bit_SO
    );

    ShiftRegisterBlock320Bit : ShiftRegister320Bit
    PORT MAP (
        DATA    => RandomNumber_S,
        CLK    => CLK,
        ENABLE    => ShiftRegActivate_S,
        OUTPUT    => KEY
    );

    SRLATCH : S_R_latch
    PORT MAP(
        S    => Comparator_S,
        R    => SRLATCH_R_S,
        Q    => ReseedKeyReady_SAsync
    );

    Multiplexer : Multiplexer_2_to_1
    PORT MAP(
        INPUT0    => QRNG,
        INPUT1    => RNG_2,
        SEL        => RNGSELECT,
        OUTPUT  => RandomNumber_S
    );


    -----------------------------------------------------------
    ------------------    CONNECTIONS    ----------------------
    -----------------------------------------------------------
    SRLATCH_R_S <= (NOT RESET) OR LoadKey_S;

    -- Feedback to let the user see that the key is ready for
     ↪  reseed
    KEYREADY <= ReseedKeyReady_SSync and (NOT
     ↪  FirstTimeSeeding_S);

    Counter9BitReset_S <= (not Comparator_S) AND RESET;

    Counter9BitActivate_S <= not ReseedKeyReady_SAsync;

    -- Triggers the loadkey signal, dependend on if the key is
        ↪  ready aka ReseedKeyReady and if the key should be sent if
        ↪  ready aka ProcSeed_S
    LoadKey_S <= ReseedKeyReady_SSync AND ProcSeed_S;

    -- ProcSeed_S can be triggered in two ways, the first way is
        ↪  if the circuit is booted for the first time, second way
        ↪  is by the reseed signal
    ProcSeed_S <= Reseed_S OR FirstTimeSeeding_S;

    LOADKEY <= LoadKey_S;

    -- Makes sure the shiftregister is paused while the counter
     ↪  is not counting
    ShiftRegActivate_S <= RESET AND Counter9BitActivate_S;
    -----------------------------------------------------------
    --------------------    PROCESS       ---------------------
    -----------------------------------------------------------
```

```
-- checks if the counter has reached 101000000
Comparator_S <= '1' when Counter9Bit_SO = "101000000" else
↪  '0';

-- Fixes that the first time booting, the system is seeded
↪  automatically
process (CLK)
begin
    if (CLK'Event and CLK = '1') then
        if (RESET='0') then
            FirstTimeSeeding_S <= '1';
        elsif (FirstTimeSeeding_S = '1' and LoadKey_S = '1')
        ↪  then
                FirstTimeSeeding_S <= '0';
        end if;
    end if;
end process;

-- Syncs the resetkey for the output
process (CLK)
begin
    if (CLK'Event and CLK = '1') then
        if (RESET='0') then
            ReseedKeyReady_SSync <= '0';
        else
            ReseedKeyReady_SSync <= ReseedKeyReady_SAsync;
        end if;
    end if;
end process;

-- Sync the reseed signal
process (CLK)
begin
    if (CLK'Event and CLK = '1') then
        if (RESET='0') then
            Reseed_S <= '0';
        else
            Reseed_S <= RESEED;
        end if;
    end if;
end process;

----------------------------------------------------------
-------------    COMPARISON STATEMENTS   -----------------
----------------------------------------------------------
```

```
end Structural;
```

F Multiplexer VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multiplexer_2_to_1 is
    Port (    INPUT0    : in STD_LOGIC;
            INPUT1  : in STD_LOGIC;
            SEL        : in STD_LOGIC;

            OUTPUT    : out STD_LOGIC);
end Multiplexer_2_to_1;

architecture Behavioral of Multiplexer_2_to_1 is
begin
    OUTPUT <= INPUT1 when (SEL = '1') else INPUT0;
end Behavioral;
```

G SR Latch VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity S_R_latch is
    Port ( S : in    STD_LOGIC;
           R : in    STD_LOGIC;
           Q : out   STD_LOGIC);
end S_R_latch;

architecture Behavioral of S_R_latch is
signal Q2   : STD_LOGIC;
signal notQ : STD_LOGIC;
begin

Q    <= Q2;
Q2   <= R nor notQ;
notQ <= S nor Q2;

end Behavioral;
```

H Asynchronous to Synchronous signal converter VHDL

```vhdl
----------------------------------------------------------
-------------------- LIBRARY DECLARATIONS ----------------
----------------------------------------------------------
library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;

----------------------------------------------------------
-------------------- RESET CONTROLLER ENTITY  ------------
----------------------------------------------------------

entity reset_controller is
    Port
        (
        RESET_ASYNC :   in  STD_LOGIC;
        CLK         :   in  STD_LOGIC;
        RESET        :    in  STD_LOGIC;
        RESET_SYNC  :   out STD_LOGIC
        );
end reset_controller;

----------------------------------------------------------
-------------------- RESET CONTROLLER ARCHITECTURE -------
----------------------------------------------------------

architecture Behavioral of reset_controller is

    ----------------------------------------------------------
    -------------------- SIGNALS          --------------------
    ----------------------------------------------------------

    signal memory_FF    :    std_logic;
    signal output_FF    :    std_logic;
    signal clear_FF        :    std_logic;

begin

    ----------------------------------------------------------
    ------------------    CONNECTIONS    ---------------------
    ----------------------------------------------------------

    RESET_SYNC <= output_FF;
    -----------------------------------------------------------
    --------------------    PROCESS      --------------------
    -----------------------------------------------------------
    process(CLK)
    begin
        if (CLK'Event and CLK = '1') then
            if(RESET = '0') then -- Resets the shit
                output_FF <= '0';
                clear_FF <= '0';
                memory_FF <= '0';
            end if;

            if(RESET_ASYNC = '1' and memory_FF = '0') then -- If
            ↪  first rising edge of the reset signal is there
                clear_FF <= '1'; -- Flag to clear next output
                ↪  round
                output_FF <= '1'; -- Ouput to 1
                memory_FF <= '1'; -- Tag the past rising edge
            end if;

            if(clear_FF = '1') then -- Clear the signals one
            ↪  pulse later
                output_FF <= '0';
                clear_FF <= '0';
            end if;

            if(RESET_ASYNC = '0' AND memory_FF = '1') then
                memory_FF <= '0';
            end if;
        end if;
    end process;

end Behavioral;
```

I 320 Bit Shift Register VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ShiftRegister320Bit is
    Port ( DATA        : in STD_LOGIC; -- Per bit input
           CLK        : in STD_LOGIC;
           ENABLE    : in STD_LOGIC;
           OUTPUT    : out STD_LOGIC_VECTOR(319 downto 0)
       );
end ShiftRegister320Bit;

architecture Behavioral of ShiftRegister320Bit is
    signal ShiftReg_D : STD_LOGIC_VECTOR(319 downto 0);
begin

    process (CLK)
    begin
        if (CLK'event and CLK = '1' and ENABLE = '1') then
            ShiftReg_D(318 downto 0) <= ShiftReg_D(319 downto 1);
            ↪  -- Shift to next D-FF
            ShiftReg_D(319) <= DATA; -- Load next value
        end if;
    end process;

    OUTPUT <= ShiftReg_D;
end Behavioral;
```

J Trivium Control VHDL

```
----------------------------------------------------------
-------------------- LIBRARY DECLARATIONS ----------------
----------------------------------------------------------
library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------------------------
-------------------- RESET CONTROLLER ENTITY  ------------
----------------------------------------------------------
entity Trivium_Control is
    Port
        (
        CLK        : in STD_LOGIC;  -- Slow Clock for the Control
        ↪  System
        RESET        : in STD_LOGIC;  -- Reset signal for the
        ↪  circuit, active low input
        LOADKEY        : in STD_LOGIC;  -- Signal that gives the
        ↪  ready signal to the Trivium to readout the key and
        ↪  initialize
        KEY        : in STD_LOGIC_VECTOR(319 downto 0);  --
        ↪  Connected to the QRNG control block, which contains
        ↪  the key

        TREADY        : out STD_LOGIC; -- Wether the trivium is
        ↪  warmed up or not, so you know when it can be paused
        T1OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 1
        T2OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 2
        T3OUT        : out STD_LOGIC_VECTOR(63 downto 0); -- 64
        ↪  bit output of Trivium 3
        T4OUT        : out STD_LOGIC_VECTOR(63 downto 0)  -- 64
        ↪  bit output of Trivium 4
        );
end Trivium_Control;

----------------------------------------------------------
------------------- SYSTEM CONTROLLER ARCHITECTURE -------
----------------------------------------------------------

architecture Behavioural of Trivium_Control is

    ----------------------------------------------------------
    ------------------    COMPONENTS      --------------------
```

```
----------------------------------------------------------

component Counter5Bit is
Port (    CLK    : in    STD_LOGIC;
        ENABLE    : in    STD_LOGIC;
        RESET    : in    STD_LOGIC;

        OUTPUT    : out    STD_LOGIC_VECTOR (4 downto 0));
end component;

    -- Reset Block
component reset_controller is
Port (
        RESET_ASYNC    : in    STD_LOGIC;
        CLK          : in    STD_LOGIC;
        RESET         : in    STD_LOGIC;

        RESET_SYNC   : out    STD_LOGIC
    );
end component;

component Trivium is
Port (    CLK    : in    STD_LOGIC;
        LOADKEY    : in    STD_LOGIC;  -- Signal that gives
        ↪  the ready signal to the Trivium to readout the
        ↪  key and initialize
        KEY    : in    STD_LOGIC_VECTOR(79 downto 0);  --
        ↪  Connected to the QRNG control block, which
        ↪  contains the key
        RESET    : in    STD_LOGIC;

        OUTPUT   : out    STD_LOGIC_VECTOR(63 downto 0)); --
        ↪  64 bit output
end component;

----------------------------------------------------------
-------------------- SIGNALS          --------------------
----------------------------------------------------------

-- Signals for the 11 bit counter
signal Counter5Bit_SO : std_logic_vector(4 downto 0);
signal Counter5BitActivate_S : std_logic;
signal ResetCounter_R : std_logic;

signal WarmedUp_S : std_logic; -- If trivium is warmed up,
↪  shown by the counter
```

```
    -- Signal to enable output
    signal Trivium1_SO : std_logic_vector(63 downto 0);
    signal Trivium2_SO : std_logic_vector(63 downto 0);
    signal Trivium3_SO : std_logic_vector(63 downto 0);
    signal Trivium4_SO : std_logic_vector(63 downto 0);

    signal KeyReadyPulse_S : std_logic;

begin
    ----------------------------------------------------------
    -----------    INSTANTIATE COMPONENTS    -----------------
    ----------------------------------------------------------
    Trivium1 : Trivium
    PORT MAP (
        CLK => CLK,
        LOADKEY => KeyReadyPulse_S, -- Signal that gives the
         ↪  ready signal to the Trivium to readout the key and
         ↪  initialize
        KEY => KEY(319 downto 240),  -- Connected to the QRNG
         ↪  control block, which contains the key FIX THIS WHEN
         ↪  ALL ARE ACTIVE!
        RESET => RESET,

        OUTPUT => Trivium1_SO -- output stream
    );

    Trivium2 : Trivium
    PORT MAP (
        CLK => CLK,
        LOADKEY => KeyReadyPulse_S, -- Signal that gives the
         ↪  ready signal to the Trivium to readout the key and
         ↪  initialize
        KEY => KEY(239 downto 160),  -- Connected to the QRNG
         ↪  control block, which contains the key FIX THIS WHEN
         ↪  ALL ARE ACTIVE!
        RESET => RESET,

        OUTPUT => Trivium2_SO -- output stream
    );

    Trivium3 : Trivium
    PORT MAP (
        CLK => CLK,
        LOADKEY => KeyReadyPulse_S, -- Signal that gives the
         ↪  ready signal to the Trivium to readout the key and
         ↪  initialize
        KEY => KEY(159 downto 80),  -- Connected to the QRNG
         ↪  control block, which contains the key FIX THIS WHEN
         ↪  ALL ARE ACTIVE!
        RESET => RESET,

        OUTPUT => Trivium3_SO -- output stream
    );

    Trivium4 : Trivium
    PORT MAP (
        CLK => CLK,
        LOADKEY => KeyReadyPulse_S, -- Signal that gives the
         ↪  ready signal to the Trivium to readout the key and
         ↪  initialize
        KEY => KEY(79 downto 0),  -- Connected to the QRNG
         ↪  control block, which contains the key FIX THIS WHEN
         ↪  ALL ARE ACTIVE!
        RESET => RESET,

        OUTPUT => Trivium4_SO -- output stream
    );

    -- Instantiate the Reset Controller
    ResetBlock : reset_controller
    PORT MAP (
        RESET_ASYNC => LOADKEY,
        CLK => CLK,
        RESET => RESET,
        RESET_SYNC => KeyReadyPulse_S
    );

    -- Instantiate the 5 bit counter
    CounterBlock5Bit : Counter5Bit
    PORT MAP (
        CLK => CLK,
        ENABLE => Counter5BitActivate_S,
        RESET => ResetCounter_R,
        OUTPUT => Counter5Bit_SO
    );

    ----------------------------------------------------------
    ------------------    CONNECTIONS    ---------------------
    ----------------------------------------------------------
```

```
ResetCounter_R <= RESET AND NOT KeyReadyPulse_S;

-- makes sure the output is disabled when warming up the
↪  trivium --------TODO: FIX THIS SHIT FOR MULTIPLE
↪  CONNECTIONS aka 4x64 bits
T1OUT <= Trivium1_SO;
TREADY <= WarmedUp_S;
T2OUT <= Trivium2_SO;
T3OUT <= Trivium3_SO;
T4OUT <= Trivium4_SO;

-----------------------------------------------------------
--------------------    PROCESS      ---------------------
-----------------------------------------------------------

-- Warmed up when it is 0
process (CLK)
begin
    if (CLK'Event and CLK = '1') then
        if (RESET='0' OR KeyReadyPulse_S = '1') then --
        ↪  Important because if it does not set to 1 at
        ↪  readypulse, the last bit is shown at the output,
        ↪  now it is always a set 0 and ouptut is blocked
        ↪  from start till end of warmup
            WarmedUp_S <= '0';                        --
            ↪  Potential discussion: First number is set,
            ↪  and always the same, can leave
            ↪  KeyReadyPulse_S = '1' out.
        elsif(Counter5Bit_SO = "10010") then -- Change to
        ↪  warmed up when certain number is reached
            WarmedUp_S <= '1';
        end if;

        -- Fix the Counter 5 bit thing

        if(RESET = '0') then
            Counter5BitActivate_S <= '0';
        elsif(KeyReadyPulse_S = '1') then
            Counter5BitActivate_S <= '1';
        elsif(Counter5Bit_SO = "10010") then
            Counter5BitActivate_S <= '0';
        end if;

    end if;
end process;
```

```
    ------------------------------------------------------------
    -------------    COMPARISON STATEMENTS   -----------------
    ------------------------------------------------------------

    -- Fixes when the counter is activated and deactivated,
    ↪  amount of warmup cycles needs to be exact the number of
    ↪  the counter,
    -- as when the counter is at 0, is loading the key for the
    ↪  trivium.
    --Counter5BitActivate_S <= '0' when RESET = '0' else
    --'1' when KeyReadyPulse_S = '1' else
    --'0' when Counter5Bit_SO = "10010";



    -- Activating and deactivating the counter


    -- When Triviums should activate
    --T1Activated_S <= '0' when Reset_S = '0' else
    --'1' when Counter8Bit_SO = "01001111" else --79 (start of
    ↪  warmup phase)
    --'0' when Counter5Bit_SO = "10001111111" else --1151 (end of
    ↪  warmup phase)
    --'1' when ActivateT1andT2_S = '1';


end Behavioural;
```

K 64-Bit Trivium VHDL

```
library IEEE;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------
--------------------- Entity Trivium ---------------------
----------------------------------------------------------

entity Trivium is
    Port (    CLK    : in STD_LOGIC;
            LOADKEY    : in STD_LOGIC;  -- Signal that gives the
             ↪  ready signal to the Trivium to readout the key
             ↪  and initialize
            KEY    : in STD_LOGIC_VECTOR(79 downto 0);  --
             ↪  Connected to the QRNG control block, which
             ↪  contains the key
            RESET    : in STD_LOGIC;

            OUTPUT  : out STD_LOGIC_VECTOR(63 downto 0)); -- 64
             ↪  bit output
end Trivium;

architecture Behavioral of Trivium is

    constant num_bits : integer := 63; -- 0 means 1 bit, so 64
     ↪  bits is 63
    ----------------------------------------------------------
    ------------------------ Signals -------------------------
    ----------------------------------------------------------
    -- Registers
    signal RAxD : STD_LOGIC_VECTOR(92 downto 0);
    signal RBxD : STD_LOGIC_VECTOR(83 downto 0);
    signal RCxD : STD_LOGIC_VECTOR(110 downto 0);

    --signal RAxNew : STD_LOGIC_VECTOR(92 downto 0);
    --signal RBxNew : STD_LOGIC_VECTOR(83 downto 0);
    --signal RCxNew : STD_LOGIC_VECTOR(110 downto 0);

    -- Signal Out of the three registers
    signal RAxSO : STD_LOGIC_VECTOR(num_bits downto 0);
    signal RBxSO : STD_LOGIC_VECTOR(num_bits downto 0);
    signal RCxSO : STD_LOGIC_VECTOR(num_bits downto 0);
    -- Signal in for the three registers (with the added
     ↪  feedback)
    signal RAxSI : STD_LOGIC_VECTOR(num_bits downto 0);
    signal RBxSI : STD_LOGIC_VECTOR(num_bits downto 0);
    signal RCxSI : STD_LOGIC_VECTOR(num_bits downto 0);

begin
    ----------------------------------------------------------
    -------------------- Create feedback ---------------------
    ----------------------------------------------------------
    GEN_FEEDBACK:
    for i in 0 to num_bits generate
        -- feedback concerning registry A
        RAxSO(i) <= RAxD(92-i) XOR RAxD(65-i); -- CHECKED T1
        RAxSI(num_bits - i) <= RCxSO(i) XOR (RCxD(108-i) AND
         ↪  RCxD(109-i)) XOR RAxD(68-i);

        -- feedback concerning registry B
        RBxSO(i) <= RBxD(83-i) XOR RBxD(68-i); -- CHECKED T2
        RBxSI(num_bits - i) <= RAxSO(i) XOR (RAxD(90-i) AND
         ↪  RAxD(91-i)) XOR RBxD(77-i); --

        -- feedback concerning registry C
        RCxSO(i) <= RCxD(110-i) XOR RCxD(65-i); -- CHECKED T3
        RCxSI(num_bits - i) <= RBxSO(i) XOR (RBxD(81-i) AND
         ↪  RBxD(82-i)) XOR RCxD(86-i); -- Make RCxSI(num_bits -
         ↪  i) if feedback switched around

        -- Connecting them together with a 3-XOR gate
        OUTPUT(i) <= RAxSO(i) XOR RBxSO(i) XOR RCxSO(i);
    end generate GEN_FEEDBACK;


    ----------------------------------------------------------
    --------------- Make the Trivium clocked -----------------
    ----------------------------------------------------------

    process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if(RESET = '0') then -- Choose if we want this reset
             ↪  or not, will also work without it but that will
             ↪  consume power, while generating key
                RAxD(92 downto 0) <= (others => '0');
                RBxD(83 downto 0) <= (others => '0');
                RCxD(110 downto 0) <= (others => '0');
            elsif(LOADKEY = '1') then-- INITIALIZE
                RAxD(9 downto 0) <= (others => '0'); -- IV
                RAxD(32 downto 10) <= (others => '0'); -- IV
                RaxD(92 downto 11) <= (others => '0'); --
                 ↪  Everything else 0

                RBxD(79 downto 0) <= KEY; -- Load key, note that
                 ↪  it is 80 bit per trivium!
                RBxD(83 downto 80) <= (others => '0'); --
                 ↪  Everything else 0

                RCxD(110 downto 108) <= (others => '1'); -- The
                 ↪  three rightmost bits set to 1 (given by
                 ↪  theory about trivium)
                RCxD(107 downto 0) <= (others => '0'); --
                 ↪  Everything else 0

            else -- simply run
                RAxD(92 downto 0) <= RAxD(91-num_bits downto 0) &
                 ↪  RAxSI; -- Shift to next D-FF
                -- RAxD(0) <= RAxSI; -- Load next value

                RBxD(83 downto 0) <= RBxD(82-num_bits downto 0) &
                 ↪  RBxSI; -- Shift to next D-FF
                -- RBxD(0) <= RBxSI; -- Load next value

                RCxD(110 downto 0) <= RCxD(109-num_bits downto 0)
                 ↪  & RCxSI; -- Shift to next D-FF
                -- RCxD(0) <= RCxSI; -- Load next value
            end if;
        end if;
    end process;


end Behavioral;
```

L JK-flip-flop VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity JK_FF is
   port( J            : in std_logic;
         K            : in std_logic;
         RESET        : in std_logic;
         ENABLE        : in std_logic;
         CLK        : in std_logic;
         OUTPUT        : out std_logic);
end JK_FF;

architecture Behavioral of JK_FF is
   signal temp: std_logic;
begin

   process (CLK)
   begin
      if(CLK'event and CLK = '1') then
         if RESET='0' then
            temp <= '0';
         elsif ENABLE ='1' then
            if (J='0' and K='0') then
               temp <= temp;
            elsif (J='0' and K='1') then
               temp <= '0';
            elsif (J='1' and K='0') then
               temp <= '1';
            elsif (J='1' and K='1') then
               temp <= not (temp);
            end if;
         end if;
      end if;
   end process;

   OUTPUT <= temp;

end Behavioral;
```

M 5-Bit Counter VHDL

```vhdl
library IEEE;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter5Bit is
    Port (    CLK    : in  STD_LOGIC;
            ENABLE    : in  STD_LOGIC;
            RESET    : in STD_LOGIC;

            OUTPUT    : out  STD_LOGIC_VECTOR (4 downto 0));
end Counter5Bit;


architecture Behavioral of Counter5Bit is

    component JK_FF is
       port( J            : in std_logic;
             K            : in std_logic;
             RESET        : in std_logic;
             ENABLE        : in std_logic;
             CLK        : in std_logic;

             OUTPUT        : out std_logic);
    end component;

    ----------------------------------------------------------
    -------------------- SIGNALS          --------------------
    ----------------------------------------------------------

    signal output_S : STD_LOGIC_VECTOR (4 downto 0);
    signal input_S    : STD_LOGIC_VECTOR (4 downto 0);

begin

    ----------------------------------------------------------
    ------------------    CONNECTIONS    ---------------------
    ----------------------------------------------------------

    GEN_FF:
    for i in 0 to 4 generate
        JK_FF_Block : JK_FF port map(
            input_S(i),
            input_S(i),
                RESET,
                ENABLE,
                CLK,

                output_S(i)
            );
    end generate GEN_FF;

    OUTPUT <= output_S;



    input_S(0) <= '1';
    input_S(1) <= output_S(0);

    GEN_FB:
    for i in 2 to 4 generate
            input_S(i) <= output_S(i-1) AND input_S(i-1);
    end generate GEN_FB;

    ----------------------------------------------------------
    --------------------    PROCESS      --------------------
    ----------------------------------------------------------


end Behavioral;
```

N 9-Bit Counter VHDL

```
----------------------------------------------------------
-------------------- LIBRARY DECLARATIONS ----------------
----------------------------------------------------------

library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;

----------------------------------------------------------
-------------------- COUNTER ENTITY  ---------------------
----------------------------------------------------------

entity Counter9Bit is
    Port (    CLK     : in  STD_LOGIC;
            ENABLE     : in  STD_LOGIC;
            RESET     : in STD_LOGIC;

            OUTPUT     : out  STD_LOGIC_VECTOR (8 downto 0));
end Counter9Bit;



architecture Behavioral of Counter9Bit is

    component JK_FF is
       port( J            : in std_logic;
             K            : in std_logic;
             RESET        : in std_logic;
             ENABLE        : in std_logic;
             CLK        : in std_logic;

             OUTPUT        : out std_logic);
    end component;

    ----------------------------------------------------------
    -------------------- SIGNALS          --------------------
    ----------------------------------------------------------

    signal output_S : STD_LOGIC_VECTOR (8 downto 0);
    signal input_S    : STD_LOGIC_VECTOR (8 downto 0);

begin

    ----------------------------------------------------------
    ------------------    CONNECTIONS    ---------------------
    ----------------------------------------------------------

    GEN_FF:
    for i in 0 to 8 generate
        JK_FF_Block : JK_FF port map(
            input_S(i),
            input_S(i),
            RESET,
            ENABLE,
            CLK,

            output_S(i)
        );
    end generate GEN_FF;

    OUTPUT <= output_S;



    input_S(0) <= '1';
    input_S(1) <= output_S(0);

    GEN_FB:
    for i in 2 to 8 generate
            input_S(i) <= output_S(i-1) AND input_S(i-1);
    end generate GEN_FB;

    ----------------------------------------------------------
    --------------------    PROCESS      --------------------
    ----------------------------------------------------------


end Behavioral;
```

O Control System Testbench VHDL

```
---------------------------------------------------------
------------------  LIBRARY DECLARATIONS  ----------------
---------------------------------------------------------
library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;

---------------------------------------------------------
-----------  CONTROL SYSTEM TESTBENCH ENTITY  ------------
---------------------------------------------------------

entity Control_System_tb is
end Control_System_tb;

architecture Behavior of Control_System_tb is

    ---------------------------------------------------------
    ------------------  COMPONENT OF UUT  --------------------
    ---------------------------------------------------------

    component Control_System is
        Port
            (
                CLK1        : in    STD_LOGIC;    -- Slow Clock
                ↪  for the Control System
                CLK2        : in    STD_LOGIC;    -- Fast clock
                ↪  for the triviums
                RESET        : in    STD_LOGIC;  -- Reset signal
                ↪  for the circuit, active low input
                RESEED        : in    STD_LOGIC;    -- If this
                ↪  signal is 1, it will reseed the trivium as
                ↪  soon a new key is ready
                TENABLE    : in    STD_LOGIC;    -- Pause output
                ↪  signal, by pausing the Clock 2 output when
                ↪  circuit is warmed up
                KEY_STREAM    : in    STD_LOGIC;    -- Connect
                ↪  the external key generator to it, in this
                ↪  case our QRNG
                RNGSELECT    : in    STD_LOGIC;  -- 0 is the QRNG
                ↪  and 1 the other input
                KEY_STREAM2    : in    STD_LOGIC;  -- The
                ↪  external RNG that can be connected to the
                ↪  system.
                T1OUT        : out    STD_LOGIC_VECTOR(63 downto
                ↪  0); -- 64 bit output of Trivium 1
                T2OUT        : out    STD_LOGIC_VECTOR(63 downto
                ↪  0); -- 64 bit output of Trivium 2
                T3OUT        : out    STD_LOGIC_VECTOR(63 downto
                ↪  0); -- 64 bit output of Trivium 3
                T4OUT        : out    STD_LOGIC_VECTOR(63 downto
                ↪  0); -- 64 bit output of Trivium 4
                TREADY        : out    STD_LOGIC; -- Feedback on
                ↪  triviums, if they are warmed up
                KEYREADY    : out    STD_LOGIC; -- Feedback to
                ↪  the user if the system can be reseeded

                -- Scan map in and outputs
                ScanEnablexSI    : in    STD_LOGIC;
                ScanDataInxDI    : in    STD_LOGIC;
                ScanDataOutxDO    : out    STD_LOGIC
            );
    end component;

    ----------------------------------------------------------
    ----------------  TEST INPUT SIGNALS  --------------------
    ----------------------------------------------------------

    signal Clock1_C        : std_logic := '0';
    signal Clock2_C        : std_logic := '1'; -- Reset high
    signal Reset_R            : std_logic := '0';
    signal Reseed_S        : std_logic := '1';
    signal TriviumEnable_S    : std_logic := '1';
    signal QRNGstream_D    : std_logic := '1';
    signal RNGselect_S        : std_logic := '0';
    signal RNG2_D            : std_logic := '0';
    signal ScanEnablexSI_S  : std_logic := '0';
    signal ScanDataInxDI_S    : STD_LOGIC := '0';

    constant clk1_period : time := 10 ns;
    constant clk2_period : time := 2 ns;

begin
    ----------------------------------------------------------
    ------------------  INSTANTIATE UUT  ---------------------
    ----------------------------------------------------------
    UUT : Control_System
    PORT MAP(
            -- Inputs
            CLK1        => Clock1_C,
            CLK2        => Clock2_C,
            RESET        => Reset_R,
            RESEED        => Reseed_S,
            TENABLE    => TriviumEnable_S,
            KEY_STREAM    => QRNGstream_D,
            RNGSELECT    => RNGselect_S,
            KEY_STREAM2    => RNG2_D,

            -- Outputs
            T1OUT        => open,
            T2OUT        => open,
            T3OUT        => open,
            T4OUT        => open,
            TREADY        => open,
            KEYREADY    => open,

            -- Scan map in and outputs
            ScanEnablexSI => ScanEnablexSI_S,
            ScanDataInxDI => ScanDataInxDI_S,
            ScanDataOutxDO => open
    );

    ----------------------------------------------------------
    -----------------------  PROCESS  ------------------------
    ----------------------------------------------------------

    -- Slow clock connected to the Key Control
    clk_process1 :process
    begin
        Clock1_C <= '0';
        wait for clk1_period/2;
        Clock1_C <= '1';
        wait for clk1_period/2;
    end process;

    -- Fast clock connected to the Trivium Control
    clk_process2 :process
    begin
        Clock2_C <= '0';
        wait for clk2_period/2;
        Clock2_C <= '1';
        wait for clk2_period/2;
    end process;

    -- Reset
    reset_process :process
```

```
    begin
        Reset_R <= '0';
        wait for 35 ns;  --signal is '1', on first run a reset
         ↪  will be fired after 50 ns
        Reset_R <= '0';
        wait for 40 ns;  --signal is '0'.
        Reset_R <= '1';
        wait for 10000 ns;  --signal is '1', this will keep going
         ↪  for X + 50ns, then the next reset is fired.
    end process;

    Reseed_S_process : process
    begin
        Reseed_S <= '0';
        wait for 4000 ns;  --signal is '1', on first run a reset
         ↪  will be fired after 50 ns
        Reseed_S <= '1';
        wait for 400 ns;  --signal is '1', this will keep going
         ↪  for X + 50ns, then the next reset is fired.
    end process;

end Behavior;
```