HARDWARE IMPLEMENTATION OF A MONTGOMERY MULTIPLIER BASED LOW-POWER FIPS-COMPLIANT RANDOM PRIME NUMBER GENERATOR

by

Halil İbrahim Kaysici

B.S., Electrical-Electronics Engineering, Middle East Technical University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical-Electronics Engineering
Boğaziçi University

2023

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Assist. Prof. Faik Başkaya for his guidance and support throughout this work. His expertise, patience, and constructive feedback helped to shape my research.

I would like to thank my colleagues from TUTEL for their friendship. Their valuable feedback and assistance helped me improve the necessary skills.

Finally, I would like to express my heartfelt gratitude to my dear wife and family for their lifelong support and patience. Their belief in me has been instrumental in my success, and I am thankful for their endless encouragement and unwavering support. I am blessed to have such a wonderful family, and I dedicate this work to them with love and appreciation.

The images that emerged within the scope of this thesis work and whose copyrights were transferred to the publishing house were used in the thesis book in accordance with the "publishing policy valid for the reuse of the text and graphics produced by the author" on the website of the publisher.

ABSTRACT

HARDWARE IMPLEMENTATION OF A MONTGOMERY MULTIPLIER BASED LOW-POWER FIPS-COMPLIANT RANDOM PRIME NUMBER GENERATOR

The Internet of Things (IoT) connects devices, vehicles, buildings, and other objects to the internet, improving efficiency and automation in various applications. More connected devices require secure communication and data storage to protect data privacy and integrity. Public-key cryptography, which uses two mathematically related keys to encrypt and decrypt data, is better for IoTs and is being used more. Secure hardware and ASIC design create tamper-resistant, energy-efficient devices resistant to physical and logical attacks. Secure IoT communication requires open-source cryptography algorithms and hardware accelerators.

This thesis explains the design of an IoT SoC to generate random prime numbers efficiently. We design secure and compatible hardware using standard specifications and test suites. Parametrized module design allows flexible and scalable hardware design. Pipelining and source-sharing configurations allow us to observe area-latency-bandwidth tradeoffs. Since IoT includes a wide range of hardware, observing different configurations is useful. Designed hardware also interacts with software to benefit from the hardware-software co-design approach.

This thesis proposes two new RNG designs and implements a scalable Montgomery-based modular multiplier. The modular multiplier forms a basis for Miller-Rabin and Lucas probabilistic primality tests. The final hardware design combines the proposed RNGs and primality tests with an open-source Ibex core into an IoT SoC.

ÖZET

MONTGOMERY ÇARPICI TABANLI DÜŞÜK GÜÇLÜ FIPS UYUMLU RASTGELE ASAL SAYI ÜRETECİ DONANIM UYARLAMASI

Nesnelerin İnterneti (IoT), cihazları, araçları, binaları ve diğer nesneleri internete bağlayarak çeşitli uygulamalarda verimliliği ve otomasyonu geliştirir. Daha fazla bağlı cihaz, veri gizliliğini ve bütünlüğünü korumak için güvenli iletişim ve veri depolama gerektirir. Biri gizli biri açık olan matematiksel olarak ilişkili iki anahtarlı "Açık Anahtar Şifreleme" methodu IoT'ler için daha uygundur ve daha yaygın kullanılmaktadır. Kurcalamaya, fiziksel ve mantıksal saldırılara karşı dayanıklı ve enerji açısından verimli cihazlar güvenli ve uygulamaya özel donanım tasarımı ile mümkündür. Dolayısıyla, güvenli IoT iletişimi, açık kaynaklı şifreleme algoritmaları ve donanım hızlandırıcıları gerektirir.

Bu tez, verimli bir şekilde rasgele asal sayılar üretmek için bir IoT SoC tasarımını açıklamaktadır. Standart spesifikasyonları ve test paketlerini kullanarak güvenli ve uyumlu donanımlar tasarlıyoruz. Parametrik modül tasarımı, esnek ve ölçeklenebilir donanıma olanak tanır. Boru hattı ve kaynak paylaşımı yapılandırmaları, alan, gecikme, ve bant genişliği değiş tokuşlarını gözlemlememizi sağlar. IoT geniş bir donanım yelpazesi içerdiğinden, farklı yapılandırmaları gözlemlemek yararlıdır. Tasarım, donanım-yazılım ortak tasarım yaklaşımından yararlanmak için yazılımla da etkileşime girer.

Bu tez, iki yeni RNG tasarımı önerir ve ölçeklenebilir bir Montgomery tabanlı modüler çarpıcı gerçekler. Modüler çarpan, Miller-Rabin ve Lucas olasılıksal asallık testleri için bir temel oluşturur. Nihai donanım tasarımı, önerilen RNG'leri, asallık testlerini ve açık kaynaklı olan Ibex çekirdeği ile bir IoT SoC'de birleştirir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

ApEn	Approximate Entropy
ASIC	Application Spesific Integrated Circuit
BRAM	Block Random Access Memory
CSA	Carry-Save Adder
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECC	Elliptic Curve Cryptography
FF	Flip-Flop
FIFO	First-in First-out
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input-Output
GUI	Graphical User Interface
IC	Integrated Circuit
IoT	Internet-of-Things
IP	Intellectual Property
ISA	Instruction Set Architecture
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
MC	Multiplicand
MD	Modulo
MR	Multiplier
NIST	National Institute of Standards and Technology
PRNG	Pseudo Random Number Generator
RNG	Random Number Generator
RPNG	Random Prime Number Generator
RSA	Rivest–Shamir–Adleman

RTL	Register Transfer Language
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SRAM	Static Random Access Memory
SSL	Secure Socket Layer
TCL	Transaction Control Language
TLS	Transport Layer Security
TRNG	True Random Number Generator
UART	Universal Asynchronous Receiver Transmitter
VLSI	Very Large Scale Integration
XOR	Exclusive-Or

1. INTRODUCTION

The Internet of Things (IoT) is an emerging technology that connects physical devices, vehicles, buildings, and other objects to the internet. These devices can collect and share data, allowing for greater efficiency and automation in a wide range of applications such as smart homes, healthcare, transportation, and industry 4.0. As the number of connected devices increases, so does the need for secure communication and data storage to protect the privacy and integrity of the data. Security is a crucial concern in the IoT, as these devices often collect and transmit sensitive information.

Public-key cryptography is especially well-suited to the requirements of IoT devices, where security and scalability are critical concerns. The use of public-key cryptography in IoT devices allows for secure communication without the need for a pre-shared secret key, making it easy to add new devices to the network. Furthermore, public-key cryptography can be used in IoT devices for digital signatures, which can be very useful for secure firmware updates, secure boot, and secure over-the-air updates of the device.

Public-key cryptography, also known as asymmetric cryptography, is a method of secure communication that uses a pair of mathematically-related keys to encrypt and decrypt data. The two keys, known as the public and private keys, are generated together and are mathematically related, but they are not the same. The public key is used to encrypt data, while the private key is used to decrypt data. This allows for secure communication even if the public key is widely distributed, as only the private key holder can decrypt the data.

One of the most widely used public-key algorithms is the Digital Signature Algorithm (DSA), defined in the Federal Information Processing Standards (FIPS) 186-4. DSA is a digital signature standard that uses the mathematical properties of modular arithmetic and the discrete logarithm problem to provide a digital signature. It is

designed for digital signature applications, where a large file must be "signed" securely without using a secure channel. DSA uses a private key to generate a digital signature, which can be verified using the corresponding public key. DSA is widely used for digital signature applications, such as secure email and software distribution.

Another widely used algorithm is the Rivest-Shamir-Adleman (RSA) algorithm. RSA is a public-key algorithm that can be used for both encryption and digital signature. It is based on the mathematical properties of large prime numbers and is widely used for secure communication, such as in SSL/TLS, which is the standard security protocol used for secure communication on the internet.

Open specification cryptography algorithms, also known as open-source cryptography, are essential for several reasons. One of the main reasons is security. Open-source algorithms are subject to public scrutiny and review, which allows for detecting potential vulnerabilities and weaknesses. This contrasts with proprietary algorithms, which are kept secret, making it challenging to identify and address potential security issues. Additionally, open-source algorithms allow for independent third-party auditing and testing, increasing trust in the algorithm's security. Another important reason for the use of open-source algorithms is interoperability. Proprietary algorithms are often tied to specific products or vendors, making it difficult to communicate and exchange data between different devices or systems. Open-source algorithms, on the other hand, can be implemented on any platform or device, which facilitates communication and data exchange between different devices and systems.

The use of specialized hardware or hardware accelerators can significantly improve the performance of cryptographic operations, allowing for faster and more efficient communication. This is particularly important for IoT devices, which often have limited processing power and memory, making it difficult to perform complex cryptographic operations in software. Furthermore, specialized hardware or hardware accelerators can also provide a higher level of security by providing a dedicated and isolated environment for cryptographic operations. This can reduce the risk of side-channel attacks

and other forms of physical attacks that can compromise the security of cryptographic operations. Additionally, specialized hardware or hardware accelerators can also be designed to be tamper-resistant, which can make it more difficult for an attacker to extract the keys and other sensitive information.

Energy efficiency is also a critical concern for IoT devices as they often operate on battery power, and securing communication and data storage can consume a significant amount of energy.

Secure hardware design in the form of ASIC design plays a vital role in addressing these concerns. Specialized ASIC designs can be used to create tamper-resistant devices that are resistant to physical and logical attacks while also ensuring energy efficiency. One important aspect of secure hardware design is the use of hardware random number generators to generate prime numbers for use in cryptographic operations. Hardware random number generators are less susceptible to attack than software-based generators and can be designed to be tamper-resistant and can be integrated into the secure hardware design of IoT devices.

ASICs are necessary to implement these security measures in a cost-effective and energy-efficient way. Unlike general-purpose processors, ASICs are tailored to specific applications, and their design can be optimized for energy efficiency, which is crucial for battery-powered IoT devices. Additionally, the use of ASICs can also reduce the cost and power consumption of the device, making it more accessible and more widespread.

In this thesis, we design an SoC targeting IoT devices to generate random prime numbers fast and efficiently. To target more secure and compatible hardware, we make use of the standard specifications and test suites. We investigate the area-latency-bandwidth trade-offs for different processing power requirements.

The term IoT hardware spawns from tiny low-power processors whose job is sending low data rate sensor data to a high-speed and high-bandwidth IoT gateway

whose job is managing a large IoT network. As a result, designing a single SoC and using it everywhere is not possible. Therefore, we make our designs as parametric and flexible as possible. Furthermore, we designed our modules so that the resource on the PRNG side can be configured and used by the trusted software allowing more efficient hardware-software co-design.

The contributions of this thesis can be summarized as follows:

- We design ring oscillator-based RNGs by creating a better entropy source and using it in different configurations, namely regular sampling of irregular waveform and chaotic sampling of regular waveform.
- We propose a method that improves the throughput of irregular sampling of regular waveform-based RNGs by avoiding the problem created by an unbalanced duty cycle.
- We analyze a supposably true random number generator to draw attention to possible vulnerabilities of chaos-based RNGs.
- We design a complete and parametrized SoC capable of efficiently generating random prime numbers and using them in software.

The organization of this thesis is as follows. Chapter 2 contains a summary of the related background and literature research. RNG design and implementation, as well as a throughput improvement method and security analysis of an RNG, are located in Chapter 3. Chapter 4 explains the hardware design of primality tests and their required components in detail. SoC integration of designed RPNG, processors, memory and some peripherals is explained in Chapter 5. The implementation results and related discussions are located in Chapter 6. Finally, the thesis is concluded with Chapter 6, containing a brief conclusion and future work.

2. THEORY & BACKGROUND

The FIPS-186 "Digital Signature Standard (DSS)" describes the algorithms for applications that need a digital signature rather than a written signature. In a computer, a digital signature is represented by a string of bits. A digital signature is calculated using a set of guidelines and criteria that make it possible to confirm the signer's identity and the accuracy of the data. Data that is saved or sent can produce a digital signature. Digital signatures are created using a private key, then verified using a public key. The public key is similar to the private key but different. Each signatory holds a pair of private and public keys. While private keys are kept a secret, public keys may be known by anybody. By using the signatory's public key, anybody may validate the signature. Signature creation can only be done by the person who has the private key. The condensed form of the data to be signed, often referred to as a message digest is obtained using a hash function during the signature creation process. In order to create the digital signature, the message digest is sent into the digital signature algorithm, such as DSA, RSA, and ECC. The Secure Hash Standard (SHS), FIPS 180, specifies the hashing operations that must be employed. Digital signature algorithms that have received FIPS approval must be used with a suitable hash function such as SHA-x. The signed data is sent to the intended verifier with the digital signature. Using the claimed signatory's public key and the same hashing algorithm that produced the signature, the verifying entity validates the signature. Signing and verifying signatures for stored and transmitted data may be done using similar processes. [1].

The security of digital signature algorithms and public-key cryptography is based on the secrecy of the private key and, therefore, depends on the difficulty of public key factorization. Due to this fact, large prime numbers are used in the key generation. Random generation of large prime numbers is one of the most challenging parts of public-key cryptography. Therefore, this section investigates the fast and energy-efficient way of generating large prime numbers in hardware.

2.1. Random Number Generators (RNGs)

One of the most critical parts of a cryptographic algorithm is the key since the secrecy of the encryption strictly depends on the secrecy of the key. Therefore, the key should not be predicted even if everything about the key generation is known [2].

An unpredictable key can be generated using random number generators (RNGs). Mainly, there are two types of random number generator (RNG) structures. These are true random number generators (TRNG) and pseudo-random number generators (PRNG). The difference comes from the entropy source. Fundamentally, PRNGs are deterministic complex functions with random initial conditions. On the other hand, the randomness of the TRNGs comes from physical phenomena. Physical phenomena such as thermal noise, shot noise, radiation, and other natural noise sources are assumed random and, therefore, unpredictable [3].
On the other hand, functions are deterministic, and their outcomes can be somehow predicted no matter how complex they are.

Random number generation can be done in both software and hardware. PRNG algorithms can be implemented in software with initial conditions either from hardware or software. Using software-generated random numbers in cryptographic algorithms is riskier than hardware-generated ones since interfering with software is easier, and there is no way of implementing TRNG without including hardware.

Hardware RNGs can be composed of analog, digital, and mixed-signal circuits. Digital RNGs seem to be more popular due to the following reasons:

- The digital RNGs can be used in FPGA designs.
- They can be synthesized and implemented together with other digital circuits.
- They could be embedded into large digital circuits at the RTL level.

On the other hand, generally, a single digital circuit as a entropy source is insufficient. Therefore, repetitive use of the same circuit is preferred to increase the total entropy.

Regular sampling of irregular waveform and irregular sampling (or chaotic sampling) of the regular waveform are the two primary methods used by TRNGs. The chaotic sampling of regular waveforms has several benefits over the regular sampling of irregular waveforms. The chaotic sampling method is less susceptible to interference attacks and more resilient than the regular sampling method. Second, the data rate of the irregular sampling is independent of the source's bandwidth. In addition, the throughput of the irregular sampling method is significantly greater than that of the regular sampling method because it depends on the ratio between the fast regular clock frequency and the irregular sampling clock frequency and because the frequency of the regular clock can be adjusted. In addition, the irregular sampling method permits us to generate random bitstreams without regard to the underlying distribution. In addition, the standard sampling method requires offset compensation and post-processing, which increases its complexity. The irregular sampling method is less complex than the regular one because it does not require offset compensation and post-processing [4]. However, the irregular sampling of the regular waveform method presents some obstacles. Creating a regular, fast waveform with a 50% duty cycle is one of the most difficult aspects. If the duty cycle of the regular clock is not 50%, then there is a bias between the total number of zeros and ones, which degrades randomness. Consequently, it is essential to fine-tune the duty cycle to 50 percent, and therefore, duty cycle correction techniques are required [5].

General digital design concerns such as power dissipation and the area also apply to RNG designs. However, there is another concern which is the most important feature of an RNG, randomness. Hence, The primary objective is to create quality random numbers with high entropy while retaining low power and area consumption.

Testing should be done to evaluate the unpredictability of data produced by an RNG. As a result, one more significant takeaway may be derived from this piece of information, and that is the significance of verifying the RNG's degree of unpredictability. In the testing process, statistical test suites are often utilized. The NIST SP 800-22 [6], FIPS, the DIEHARD, and AIS31 test suites are often used to evaluate the random number generator designs.

An RNG could be called reliable and secure if and only if the following criteria are met:

- No formal approach can be presented to replicate the same bit stream [7].
- No formal technique can be presented to predict future random bits [8]
- Satisfying the statistics tests

If a cryptographic system fails to meet any of these criteria, it cannot be called secure and reliable. And cryptoanalysis reveals the system's vulnerabilities.

A ring oscillator is one of the most common structures used as a building block to construct an RNG. Their popularity in FPGA designs [9] is due to their simple, scalable, and realizable topologies. A ring oscillator's entropy source is phase jitter or timing jitter induced by thermal noise. Numerous RNG designs employ ring oscillators with various configurations, including [10–13].

2.2. Modular Multiplication

Modular multiplication is one of the main components in public-key cryptography. It is used both directly and as the main component of modular exponentiation. This section will briefly investigate some of the well-known modular multiplication algorithms.

2.2.1. Basic Modular Multiplication

Compared to ordinary multiplication, modular multiplication is more complex and needs a mechanism for modular reduction. It is composed of two basic steps:

- Multiplying the operands.
- Finding the remainder using a division technique as the modular reduction.

This method is generally referred to as the classical method [14]. The corresponding algorithm is shown in Figure 2.1.

Basic Modular Multiplication
$Result = (MC \times MR) mod (MD)$ where MC, MR, and MD are integers in binary representation **begin**
Step 1. $Result = MC \times MR$
Step 2. $Result = Remainder(Result, MD)$
end

Figure 2.1. Basic modular multiplier algorithm.

2.2.2. Montgomery Modular Multiplication

One of the most frequently used modular multiplication algorithms is the Montgomery algorithm. The basic Montgomery multiplier algorithm is shown in Figure 2.2. as it can be seen from the algorithm shown in Figure 2.2, the steps are pretty straightforward. They could be implemented without too much area overhead for hardware. Also, they could be implemented with little source usage for software. The Montgomery algorithm's disadvantage is that it computes $MC \times MR \times .2^{-n} mod MD$, introducing

an additional 2^{-n}factor that must be removed. Typically, this component is removed by converting the inputs to MD-residue [15]. The following is how Montgomery multiplication is used to transform the image into the integer set:

- MD-residue transformation: $X' = Montgomery(X, 2^{2n})$
- Integer transformation: $X = Montgomery(X', 1)$.

Basic Montgomery Multiplication

$Result = (MC \times MR) mod(MD)$ where $MC = \{mc_{n-1}, ...mc_1, mc_0\}$,

$MR = \{mr_{n-1}, ...mr_1, mr_0\}$, $MD = \{md_{n-1}, ...md_1, md_0\}$ **begin**

$Sum = 0$

for $i = 0$, $i < NumberOfBits$, $i = i + 1$ **do**

  if $mr_i == 1$ **then**

    $Sum = Sum + MC$

  end if

  if Sum is **Odd then**

    $Sum = Sum + MD$

  end if

  $Sum = Sum/2$

end for

if $Sum > MD$ **then**

  $Sum = Sum - MD$

end if

end

Figure 2.2. Basic Montgomery multiplier algorithm.

2.3. Modular Exponentiation

Modular exponentiation is frequently used in public key cryptography. It is used in primality tests, such as Miller-Rabin, to generate prime keys and in the algorithms, such as RSA. Let us consider the following modular exponentiation

$$Base^{Exp} \ mod \ MD. \tag{2.1}$$

If Exp is replaced with its binary expansion, then we get

$$Base^{(2^{n-1}exp_{n-1}+...+2^1exp_1+2^0exp_0)} \ mod \ MD \ . \tag{2.2}$$

We can rewrite the Equation 2.2 as

$$(Base^{2^{n-1}exp_{n-1}} \times ...Base^{2^1exp_1} \times Base^{2^0exp_0}) \ mod \ MD. \tag{2.3}$$

If we consider each element in Equation 2.3 as an operand, then we have multiplication with n operands as

$$\begin{aligned} Operand_0 &= Base^{2^0 \ exp_0} = Base^{2^0} \ Base^{exp_0} \\ Operand_1 &= Base^{2^1 \ exp_1} = Base^{2^1} \ Base^{exp_1} \\ &... \\ Operand_{n-1} &= Base^{2^{n-1} \ exp_{n-1}} = Base^{2^{n-1}} \ Base^{exp_{n-1}}. \end{aligned} \tag{2.4}$$

We see that the first part of each operand is the square of the first part of the previous operand. Let us assume that we multiply n operands in n iterations, and each operand is multiplied by the previous iteration. Then, we could derive a straightforward yet widely used algorithm as shown in 2.3.

Basic Modular Exponentiation

$Result = Base^{Exp}\ mod\ MD$ **begin**

$R = 1$

$A = Base$

for $i = 0$, $i < NumberOfBits$, $i = i + 1$ **do**

  $A_{i+1} = A_i^2\ mod\ MD$

  if $(Exp[i] == 1)$ **then**

    $R_{i+1} = A_i \times R_i\ mod\ MD$

  else

    $R_{i+1} = R_i$

  end if

end for

return R

end

Figure 2.3. Basic modular exponentiation algorithm.

2.4. Primality Tests

Large prime numbers play a crucial role in public-key cryptography. The security and reliability of the public-key algorithms rely on the difficulty of key factorization. Therefore, they require large random prime keys. Since there is no way to generate large prime keys truly random, true random numbers should be generated and tested afterward for primality.

Primality tests could be organized into two categories, which are deterministic and probabilistic. A deterministic primality test is a mathematical algorithm that gives a definite answer as prime or composite. We can say a number is a prime number only

after showing that it has only two factors: one and itself. The most widely used deterministic primality test is the Agrawal-Kayal-Saxena (AKS) primality test. Therefore, deterministic primality tests are not feasible for large numbers. On the other hand, a probabilistic primality test is a randomized algorithm that uses statistical methods to determine whether a number is prime or composite. The results of a probabilistic test are not definite. However, the probability of reporting a composite number as prime is minimal. The most widely used probabilistic primality test is the Miller-Rabin primality test. In many applications, probabilistic primality tests are preferred over deterministic primality tests because they are much faster and can handle larger numbers [14].

In this work, we decided to use FIPS-compliant primality tests. FIPS-186 DSS document suggests several probabilistic algorithms. One of the several versions of the Miller-Rabin probabilistic primality test, and/or one of the several Lucas probabilistic primality test versions, described in FIPS-186-4, should be used for FIPS compliance. FIPS-186 standard allows two alternatives for testing primality:

- Use the iterated Miller-Rabin test, followed by a single Lucas test.
- Use only the Miller-Rabin test with several iterations (Requires more iterations compared to the first option).

The value of iterations depends on:

- The algorithm being used
- The security strength
- Chosen error probability
- Prime candidate's number of bits
- The type of tests to be performed [1].

2.4.1. Miller-Rabin Primality Test

There are a lot of different implementations of the Miller-Rabin primality test. The FIPS-186 standard document provides two similar Miller-Rabin algorithms. The FIPS compliant algorithm is shown in Figure 2.4. [1]

FIPS Approved Miller-Rabin Primality Test

$A:$ *Odd integer to be tested* **begin**

Step 1: $B = (A-1)/2^x$ Find the largest x such that 2^x divides $(A-1)$.

Step 2: Alen = width(A)

Step 3:

for $i = 1,\ i < NumberOfIterations,\ i = i + 1$ **do**

Step 3.1: Get a Alen bit random number, R

Step 3.2: if(($R \leq 1$) or ($R \geq A-1$)), then go to Step 3.1 and request another random

Step 3.3: $E = R^B\ mod\ A$

Step 3.4: if ((E == 1) or (E == A-1)), then go to step 3.7

Step 3.5:

for $j = 1,\ j < x-1,\ j = j + 1$ **do**

Step 3.5.1: $E = E^2\ mod\ A$

Step 3.5.2: if (E == A-1), then go to step 3.7

Step 3.5.3: if (E == 1), then go to step 3.6

end for

Step 3.6: return **Not Prime**

Step 3.7: continue

end for

Step 4: return **Probable Prime**

Figure 2.4. FIPS approved Miller-Rabin primality test algorithm.

2.4.2. Lucas Primality Test

Different applications of Lucas theorem can be found in the literature. The FIPS-approved Lucas primality test is shown in Figure 2.5 [1].

FIPS Approved Lucas Primality Test

$A:$ *Odd integer to be tested* **begin**

Step 1: Check whether A is perfect square. If it is then return **Not Prime**

Step 2: Find the first element,B, in the Lucas Sequence where Jacobi(B,A)= -1

Step 3: L = A+1

Step 4: Define n as the length of $L = l_n..l_1l_0$

Step 5: $M_n = 1,\ N_n = 1$

Step 6:

for $i = n-1,\ i < 0,\ i = i-1$ **do**

Step 6.1: $M_{temp} = M_{i+1}\ N_{i+1}\ modA$

Step 6.2: $N_{temp} = \frac{N_{i+1}^2 + B\ M_{i+1}^2}{2}\ modA$

Step 6.3:

if $(L_i == 1)$ **then**

Step 6.3.1: $M_i = \frac{M_{temp}+N_{temp}}{2}\ modA$

Step 6.3.2: $N_i = \frac{N_{temp}+BM_{temp}}{2}\ modA$

else

Step 6.3.1: $M_i = M_{temp}$

Step 6.3.2: $N_i = N_{temp}$

end if

end for

Step 7: if $(M_0 = 0)$, then return **Probable Prime**. Else, return **Not Prime**

Figure 2.5. FIPS approved Lucas primality test algorithm.

2.5. NIST SP800-22 Test Suite

2.5.1. Frequency Test

The frequency test assesses the number of ones and zeros in a sequence to check whether they are about the same as anticipated for a genuinely random sequence. The assessment may be done by comparing the sequence in question to a known random sequence. The test determines how close the proportion of ones is to one-half. This also means that the number of ones and zeroes in a sequence should be about equal. The outcome of all following examinations is contingent on how well you do on this test.

2.5.2. Frequency Test within a Block

This assessment aims to establish whether or not the frequency of ones in a block of N-bits is roughly equal to N/2, as would be anticipated on the basis of the assumption that the data is generated at random. When the block size is set to one, it is the same as the frequency test.

2.5.3. Runs Test

A run is defined here as an unbroken series of bits that are similar to one another. X-bit long run is made up of precisely x equal bits and is bookended before and after by bits with the opposite value. The run test aims to assess whether or not the number of runs of ones and zeros of varied lengths is as predicted for a random sequence. In other words, this test aims to assess whether the rate of oscillation between such zeros and ones is too rapid or excessively slow.

2.5.4. Longest Run of Ones in a Block Test

The longest string of consecutive ones that can be found inside N-bit blocks will be the primary focus of the examination. The goal of this test is to discover whether or not the length of the sequence that is being tested has a longest run of ones that is comparable to what would be anticipated in a sequence that is generated randomly. It is crucial to remember that if there is an abnormality in the predicted length of the longest run of ones, then there must also be an abnormality in the expected length of the longest run of zeroes. Consequently, the only test that is required is one for ones.

2.5.5. Binary Matrix Rank Test

The rank of the individual submatrices that make up the whole sequence will be the primary focus of the examination. This test's objective is to determine whether or not the linear dependency of the original sequence's substrings of fixed lengths can be found. It is vital to note that this exam is also included in the battery of tests known as the DIEHARD.

2.5.6. Non-overlapping Template Matching Test

The frequency with which the pre-defined target strings are encountered is the primary focus of this examination. This test's objective is to identify generators that create an excessively high number of instances of a certain aperiodic pattern that is not periodic. An m-bit window is used in both this test and the Overlapping Template Matching test to look for a certain m-bit pattern. In the event that the pattern cannot be located, the window will shift one bit to the left. In the event that the pattern is discovered, the window will be repositioned to the bit that comes after the discovered pattern, and the search will continue.

2.5.7. Overlapping Template Matching Test

The number of times a set of pre-determined target strings appear is the primary emphasis of the Overlapping Template Matching test. An n-bit window is used in both this test and the Non-overlapping Template Matching test to look for a certain n-bit pattern. In the same way, if the pattern is not discovered, the window moves one bit to the left. The test in this section is different from the Non-overlapping Template Matching in that when the pattern is detected, the window glides just one bit before starting the search again.

2.5.8. Discrete Fourier Transform Test

This test is designed to identify any periodic characteristics, also known as recurring patterns, that are located in close proximity to one another in the sequence being evaluated since these features would indicate a departure from the assumption of randomness. This investigation aims to determine whether there is a statistically significant difference between the percentage of peaks that are more than 5% and those greater than 95%.

2.5.9. Approximate Entropy (ApEn) Test

The test aims to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (n and n+1) against the expected result for a random sequence.

2.5.10. Linear Complexity Test

The duration of an LFSR is the primary concern of this examination. This test aims to assess whether or not the sequence is complicated enough to be called random. If it is, then the next step will be to proceed. Longer LFSRs are a distinguishing feature of random sequences. An LFSR that is excessively short may indicate the absence of randomization.

2.5.11. Random Excursions Test

This test counts cycles with precisely K visits in a cumulative sum random walk. After the (0,1) sequence is translated to the proper (-1, +1) sequence, partial sums are used to calculate the cumulative sum random walk. A random walk cycle begins and ends with a random series of unit-length steps. This test determines whether the number of visits to a state throughout a cycle deviates from a random sequence. This exam is a sequence of eight tests and conclusions, one for each state: -4, -3, -2, -1 and +1, +2, +3, +4.

2.5.12. Random Excursions Variant Test

This test aims to identify any significant departures from the predicted number of times the random walk takes participants to each of the different states. This exam comprises a sequence of eighteen tests (and conclusions), one test, and one conclusion for each of the following states: -9, -8,..., -1 and +1, +2,..., +9.

2.5.13. Universal Test

This test will determine whether or not the sequence can be greatly compressed without any information being lost in the process. It is generally accepted that a sequence that can be greatly compressed is not random.

2.5.14. Serial Test

This test aims to establish whether the number of occurrences of the 2^n n-bit overlapping patterns is comparable to what may be anticipated for a random sequence. Uniformity is a property of random sequences; more specifically, every m-bit pattern has the same probability of occurring as every other n-bit pattern. Note that when m is equal to one, the Serial test is identical to the Frequency test.

2.5.15. Cumulative Sums Test

The goal of the test is to assess if the cumulative total of the partial sequences appearing in the sequence being tested is excessively high or excessively low compared to the behavior anticipated for that cumulative sum when applied to random sequences. This cumulative total may be considered a random walk. For a sequence to be considered random, the random walk excursions should be close to zero. Large deviations from zero might be expected for some kinds of non-random sequences when using this random walk algorithm.

3. RANDOM PRIME NUMBER GENERATOR HARDWARE DESIGN AND IMPLEMENTATION

This chapter explains the final hardware designs and the journey to the final hardware of all modules required for the random prime number generation. The literature research and theory of the designed modules are explained in Chapter 2.

FIPS-compliance is taken into account while designing the modules. All the RNG designs are tested with the NIST SP 800-22 test suite. Algorithms for primality tests and the iteration number of the Miller-Rabin primality test comply with the FIPS 186-4 standard document [1]. The basic block diagram of the random prime number generator is shown in Figure 3.1.

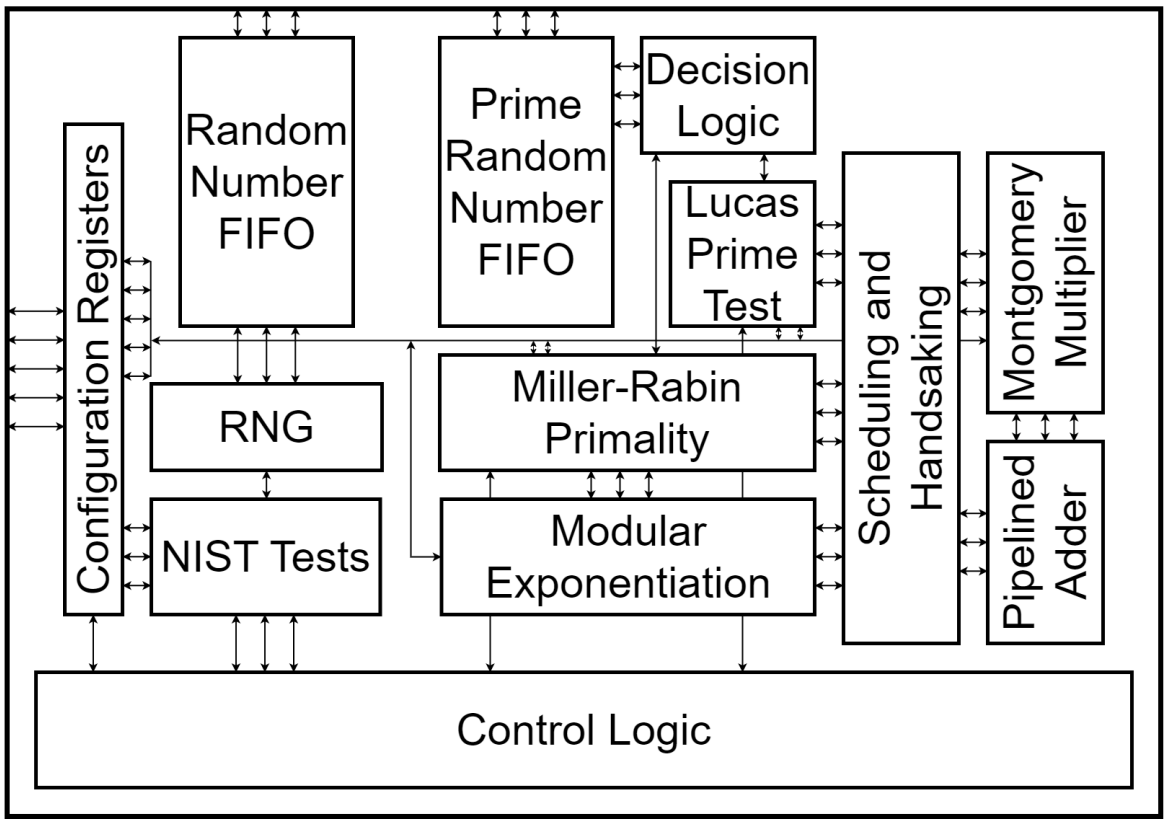

Figure 3.1. Basic block diagram of the proposed random prime number generator.

The RPNG has a read-write interface for the configuration registers and FIFOs. Moreover, the modular multiplier and modular exponentiator are accessible via the same interface by the main processor unit. For this work, the main processor that RPNG connects to is open-source ibex core [16]. The following parameters are set via the configuration registers:

- Parameters of the implemented NIST SP 800-22 randomness tests, such as bit-stream length
- Number of iterations for Miller-Rabin primality test
- Random bit generation enable
- Write to FIFO enable.

Pipelined Montgomery multiplier and pipelined adder are shared with Miller-Rabin, Lucas, and modular exponentiation units for area optimization. Modular exponentiation unit can perform two modular multiplication in parallel. The required additions for the Montgomery multiplier unit are also performed in the shared pipelined adder. Miller-Rabin unit uses both modular exponentiation and modular multiplication. However, not at the same time. Therefore, either Miller-Rabin or modular exponentiation unit needs to use the Montgomery multiplier unit. Lucas unit uses only modular multiplication, requiring at most three modular multiplication in parallel. As a result, at most, five modular multiplication requests are asserted simultaneously, and, therefore, the pipeline stage for the Montgomery multiplier unit is chosen as four.

Both primality tests are started in parallel simultaneously for execution time optimization. If either fails, the candidate random number is marked as non-prime without waiting for the other result. A random number can be marked as prime and stored after both tests are completed and output “Probable Prime.” Furthermore, results of the implemented NIST tests are observed for reliable random number generation. If the tests are failing, for example, due to tempering, then the random number generation stops, and a flag is set to inform the software. Moreover, random number generation and primality testing stop if FIFOs become full.

3.1. Random Number Generator Design

As explained in Section 2.1, random number generation is one of the most critical parts of cryptographic systems. If the generated random numbers are not truly random or somehow susceptible to attacks, then the random numbers can be exposed. If the random numbers are unreliable, the key is unreliable. Therefore, the encrypted information, cipher texts, encrypted with the unreliable key, is not secure. As a result, the key should not be predicted even if everything about the key generation is known.

One of the most popular structures for creating RNGs is the ring oscillator, as mentioned in Section 2.1. As the name suggests, it is a loop that contains an odd number of inverters. The loop is unstable and in an oscillatory state. The period of oscillations is not constant; therefore, it has a period jitter. This jitter is the primary source of entropy in a ring oscillator based RNG. However, the entropy of a single ring oscillator is insufficient to produce a random number that complies with the demands of the statistical test. As a result, to raise the overall entropy, the entropies of numerous ring oscillators are joined by XORing the output of each ring. The resulting waveform is sampled with a D-type flip-flop (DFF) to generate the random bit after enhancing the entropy by mixing numerous rings [17].

3.1.1. Regular Sampling of Irregular Waveform Based RNG Design

Figure 3.2 displays the circuit diagram of a generic RNG based on ring oscillators, and it is suggested by [17]. However, this arrangement has certain issues. The fundamental issue is that the XOR gate may have problems settling before sampling since several transitions may occur at its input. Wold et al. suggested a solution to this issue by adding a D-type flip-flop to each ring. Sampling before the xor operation provides that the inputs of the XOR gate change only at the positive edge of the clock signal. This modification provides enough time for the XOR gate to settle its output [13]. The modified block diagram is shown in Figure 3.3. Further modifications and improvements related to the xor operation are proposed in different studies. How-

ever, the main problem remains unchanged, relying on the timing jitter as their sole entropy source.

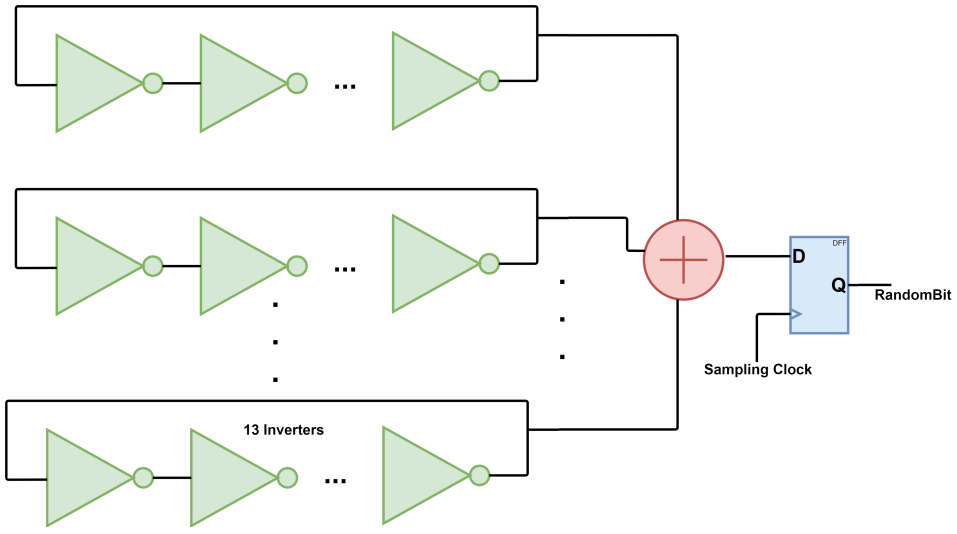

Figure 3.2. Circuit diagram of the basic random number generator based on classical ring oscillators and regular sampling.

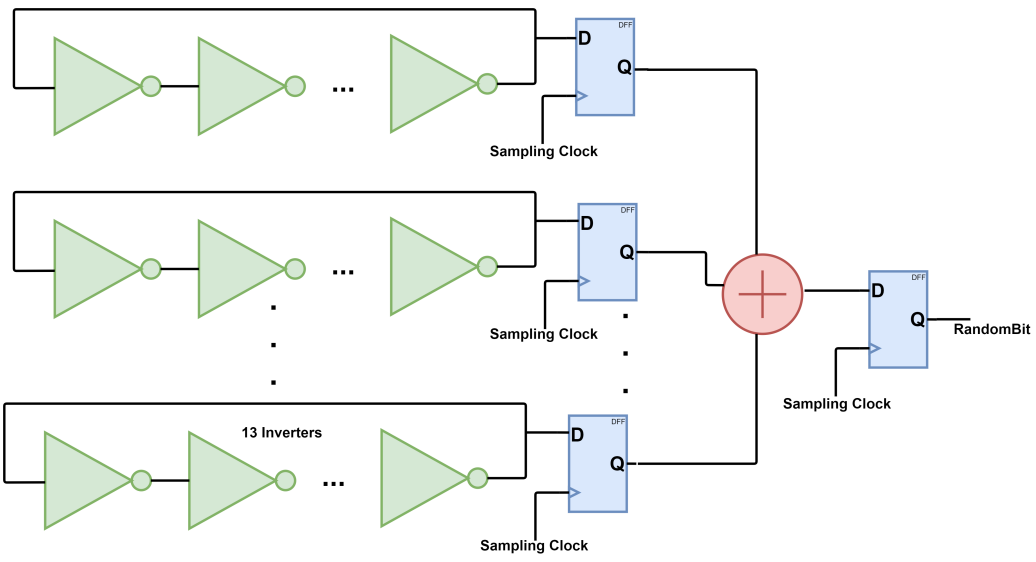

Figure 3.3. Enhanced circuit diagram of the basic random number generator design based on classical ring oscillators and regular sampling.

We need additional metastable events and uncertainty to boost total entropy. According to Nakura et al., metastability is "the state that maintains an unstable equilibrium and minor stimuli such as device noise can push towards one side" [12]. Varchola hypothesized that adding another inverter to the unstable loop creates two extra metastable events and increases the entropy [11]. The first metastability arises from the fact that when the loop becomes stable, the ring oscillator's output may converge to either logic 0 or 1. The other type of metastability is that the oscillations do not stop immediately when the loop becomes stable. Instead, it stabilizes while continuing to oscillate. This behavior results in some regions with multiple transitions around the logic threshold, creating the second metastability. As a result, the RNG's total entropy increases. The ring oscillator that uses this approach is called as shut-down ring oscillator in [18]. Figure 3.4 shows an example output to the shut-down ring oscillator. In order to control the stability of the rings, a multiplexer and an additional inverter are added. The extra inverter is selected and connected to the loop through the multiplexer to make the loop stable. A control unit is also required to generate the select signal.

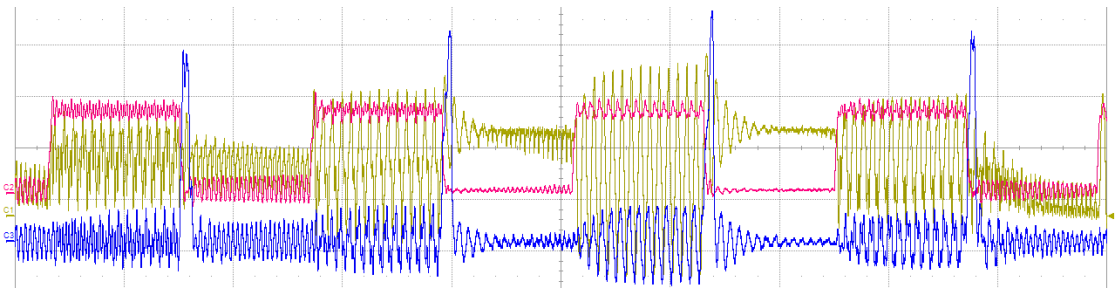

Figure 3.4. An example oscilloscope caption for shut-down ring oscillator. Red: Control signal, Blue: Sampling signal, Yellow: shut-down ring oscillator output.

Nakura et al. suggest a strategy termed "wake-up uncertainty" to boost total entropy [12]. The unpredictability in the wake-up time is used in this strategy. It is the time that has passed since the ring became unstable and began to oscillate. Wake-up time uncertainty combines with timing jitter to increase overall entropy. Figure 3.5

shows an example output to a wake-up ring oscillator. A multiplexer and an extra inverter are added to the ring to implement this circuit, just like in the shut-down ring oscillator.

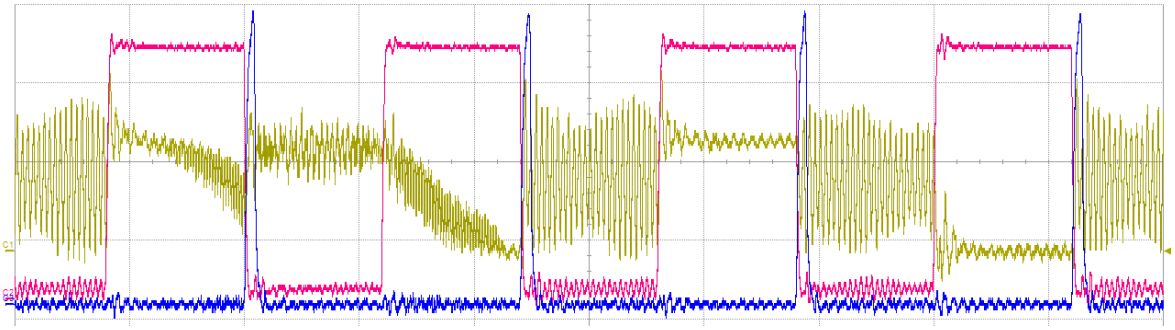

Figure 3.5. An example oscilloscope caption for wake-up ring oscillator. Red: Control signal, Blue: Sampling signal, Yellow: Wake-up ring oscillator output.

We propose [2] using both approaches concurrently to boost the total entropy as well as the throughput of [17], [12], and [11]. That is, while some of the rings are turned off by making the loops stable, the remaining rings are switched on by making the loops unstable. As a result, using the same sampling signal, some rings are sampled when turned off, while others are sampled when turned on. As a result, the total entropy is increased by employing both wake-up uncertainty and shut-down metastabilities. Furthermore, throughput is doubled because the same ring output is sampled at both wake-up and shut-down times.

The circuit depicted in Figure 3.6 is designed to implement the discussed method. The primary ring oscillator is formed of 13 inverters. Each ring features a multiplexer to modify the stability of the rings. When the multiplexer's select input is logic "low," it connects the final output to the first input, resulting in 13 inverters in the loop and the ring being unstable. When the multiplexer input is logic "high," the multiplexer adds one more inverter to the loop, bringing the total to 14, and the ring becomes stable. Each ring oscillator output is coupled to a DFF, as recommended in [13].

Before sampling, all flip-flop outputs are XORed. The original design has 114 rings. The values 13 and 114 for the number of inverters and rings are based on [17]. In addition, a control unit is employed to create "*Select*" and "$\overline{Select}$" signals. "*Select*" is connected to the shut-down ring oscillators' select inputs, while "$\overline{Select}$" is connected to the wake-up ring oscillators' select inputs. In addition, half of the rings are utilized in the Shut-down configuration, while the other half is used in the Wake-up configuration.

The sampling frequency of the proposed RNG is selected as 4.5 MHz since the maximum data frequency that the card that is used to collect data can handle is 10 MHz. As mentioned in the second section, the throughput of the proposed method doubles the shut-down and wake-up methods. Two example ring outputs and the sampling signal are shown in Figure 3.7.

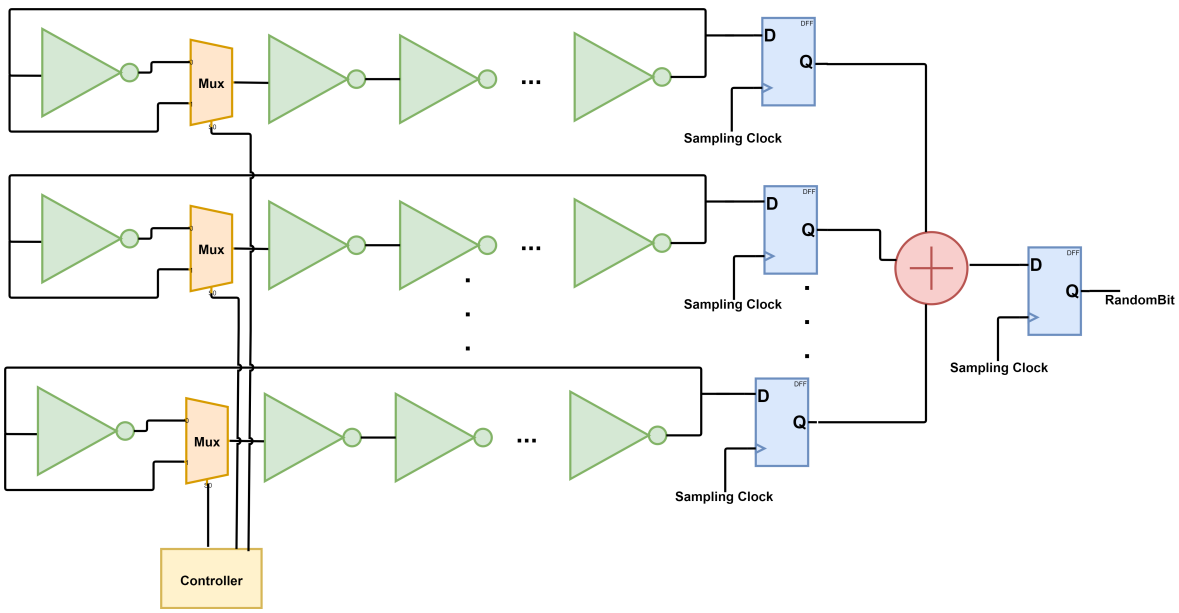

Figure 3.6. The proposed shut-down&wake-up ring oscillator based random number generator circuit diagram.

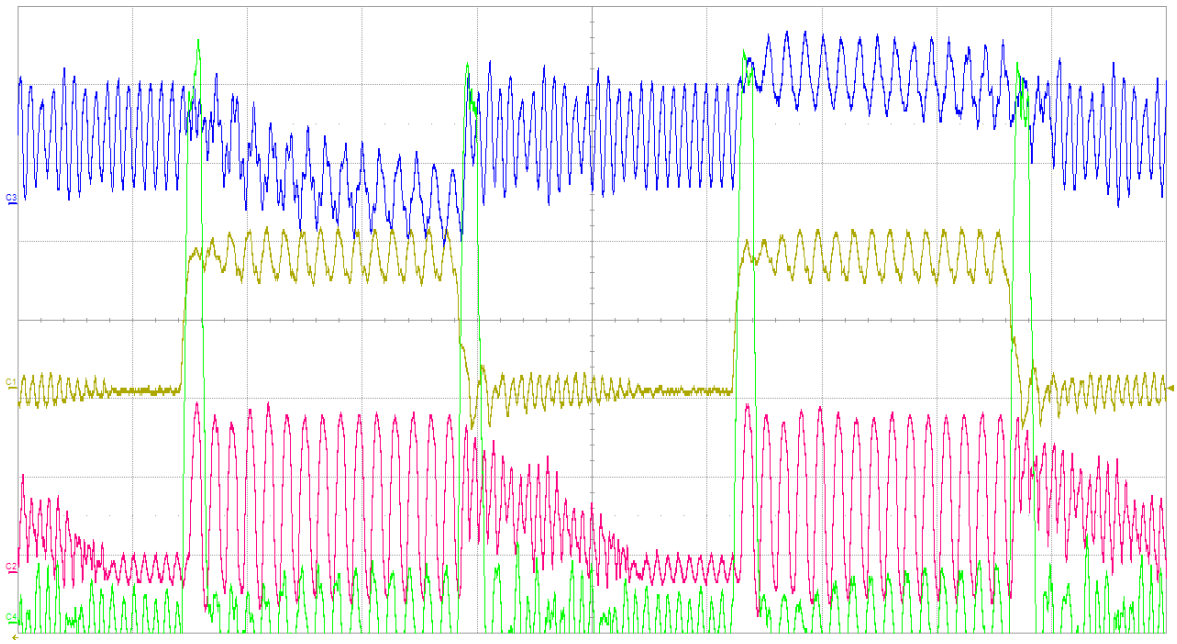

Figure 3.7. An example oscilloscope caption for proposed wake-up&shut-down ring oscillator and regular sampling based random number generator circuit. Blue: Shut-down ring output of the proposed RNG, Red: Wake-up output of the proposed RNG, Yellow: Control signal, Green: Sampling signal.

The rings in the layout for FPGA implementation are manually placed using the Xilinx Plan Ahead Tool. The rings are arranged so that some distance separates each ring, and the XOR operation is performed in the middle. One of the reasons for this placement is to minimize the coupling effect and to make use of process variation from different locations on the FPGA. Furthermore, the XORing occurs in the center, ensuring that each path from the outputs of the rings to the XOR is about equal in length and does not intersect. In addition to XOR, the control signal generating module is in the center, ensuring that each control signal reaches the intended multiplexer with similar routing delays.

3.1.2. Irregular Sampling of Regular Waveform Based RNG Design

As explained in Section 2.1, there are two random number extraction methods, and they have different theories and, therefore, different use cases. As a result, we designed another random number generator with the same entropy source by employing irregular sampling method [19]. In this method, a regular high-frequency clock is sampled with an irregular waveform. An example circuit for this method is shown in Figure 3.8 that is proposed by [20].

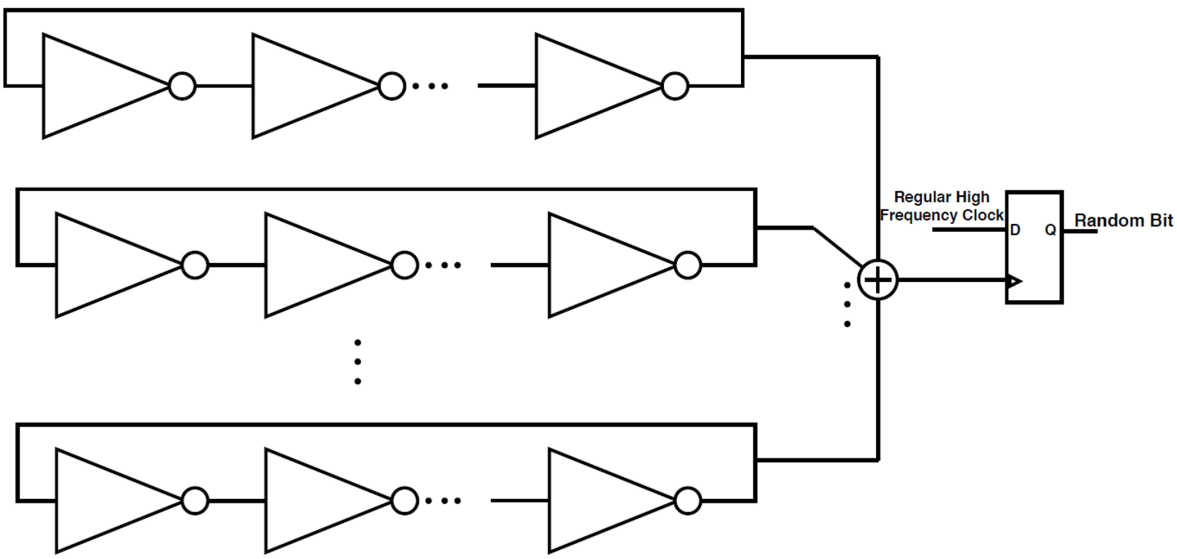

Figure 3.8. An example circuit diagram for irregular sampling of regular waveform based random number generator.

We propose to use both methods at the same time as we did with the previous one and use it as an irregular clock to sample a regular high-frequency clock.

Figure 3.9 shows the circuit used to achieve the suggested technique. As the entropy source, ring oscillators formed of 13 inverters are used. The same method as the previous RNG is used to adjust the stability of the loop. Thirty-four ring oscillators with the stability change are directly combined with an XOR gate to generate

enough entropy. The XOR gate output serves as an irregular clock to sample a regular high-frequency clock. Figure 3.10 depicts the irregular signal. In order to use the metastable portion of the wake-up and shut-down rings, the irregular clock is masked around where the control signal changes state. Figure 3.10 depicts the control and mask signals.

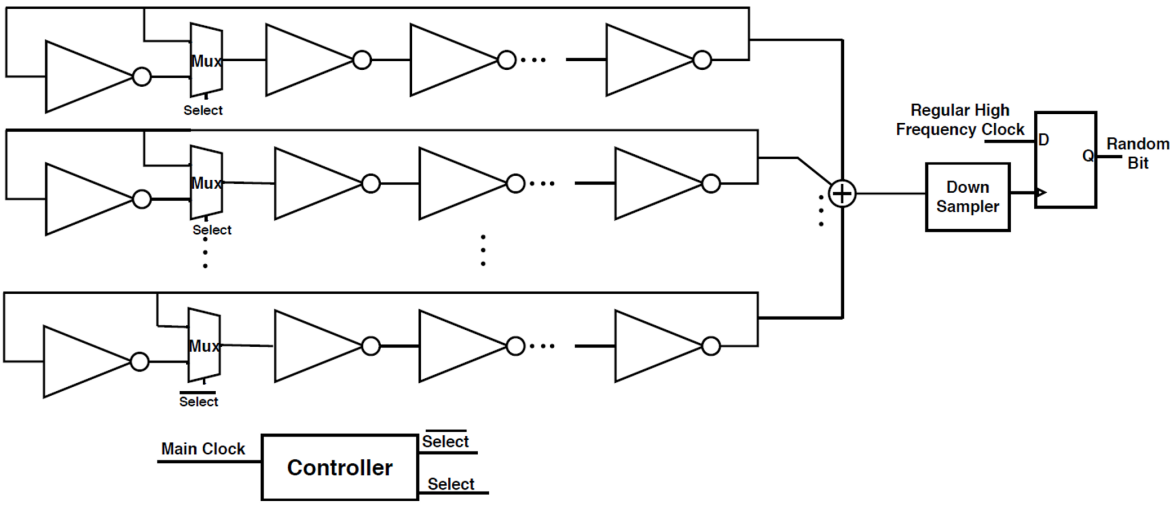

Figure 3.9. Proposed irregular sampling of regular waveform method based random number generator circuit diagram.

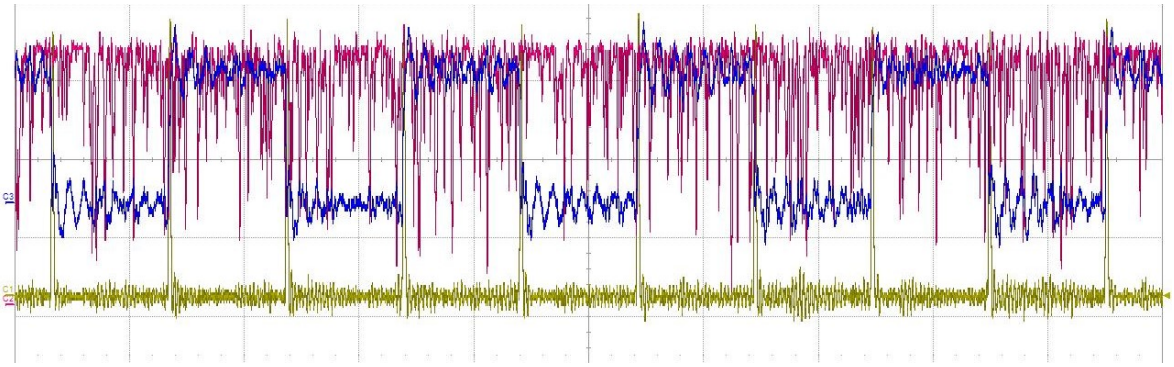

Figure 3.10. An example oscilloscope caption for proposed irregular sampling method random number generator. Red: Irregular clock - XOR output, Blue: Control signal, Yellow: Mask signal.

The frequency of state change is set to 2MHz. It is then followed by generating a 1MHz 50% duty cycle signal as the control signal. To mask the irregular clock, 2MHz pulses with a pulse width of 10ns are created and ANDed. Furthermore, a down-sampler with a rate of 1/85 is utilized to reduce the sampling frequency. It counts the positive edge of the irregular clock and samples the regular clock when the 85th rising edge occurs. The irregular clock is sent through four successive T flip-flops before driving any logic to ensure that the down-sampler and other control logic driven by the irregular clock work properly.

Furthermore, the duty cycle of the sampled 464MHz high-frequency regular clock is not 50%, resulting in a bias between the total number of logic zero and logic one values. In order to balance the duty cycle, the regular high-frequency clock is also routed via a 5 T flip-flop. The resulting regular clock frequency is around 14.5 MHz. We have to downsample the irregular clock with 85 because the frequency difference between the regular clock and irregular clock should be roughly 100 times, as described by [21]. Figure 3.11 depicts the signals utilized to generate random bits.

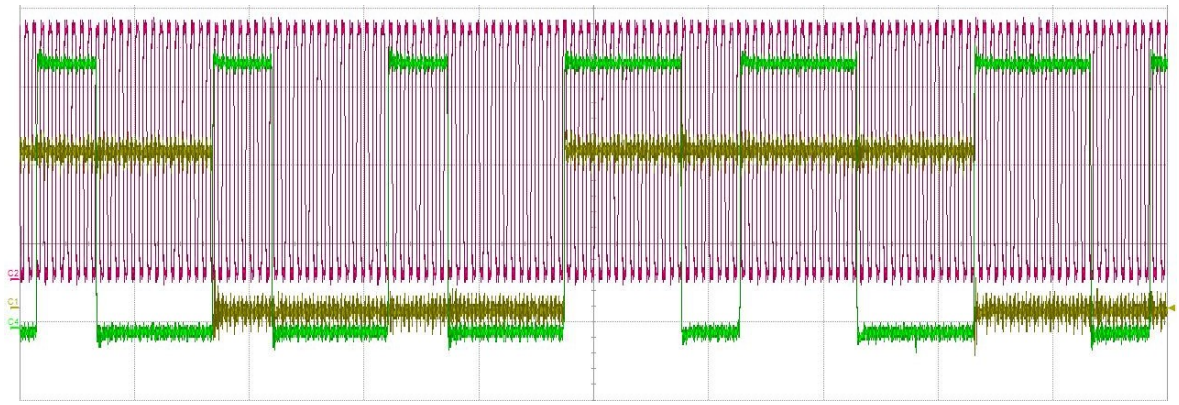

Figure 3.11. An example oscilloscope caption for proposed irregular sampling method random number generator. Red: Regular high-frequency clock, Green: Irregular clock that is masked and passed through TFFs, Yellow: Generated random bits.

If the duty cycle of the regular clock were not biased, then we would not have to downsample the regular and irregular clocks. Therefore, the throughput would be much larger. The next section, Section 3.1.3, discusses possible solutions for the unbalanced duty cycle problem [5].

3.1.3. Duty Cycle Correction

An unbalanced duty cycle problem arises due to two main reasons. The first reason is the different rise-time and fall-time of the logic elements. An example simulation result that shows how a difference in rise-time and fall time can cause an unbalanced duty cycle is given in Figure 3.12. In Figure 3.12, an ideal 50% duty cycle signal is connected to an inverter. As shown in Figure 3.12, the rise-time and fall time of the inverter are not balanced. Therefore, the duty cycle of the inverter is biased. This bias can be seen in the middle of Figure 3.12.

Figure 3.12. Example simulation result to illustrate rise-time, and fall-time effect.

The other reason is the change in period or period jitter. Considering a T-type flip-flop whose T input is logic one, the output toggles with each rising edge of the clock. If the time between each consecutive rising edge is not equal, then the duty cycle differs from 50%.

The duty cycle problem in the irregular sampling of regular waveform method is being tried to solve by dividing the fast regular clock with T-type flip flops by connecting a constant input logic "1" to T input and driving the clock input with the fast regular clock. Since the output changes only when a rising edge hits, the output is expected to have a 50% duty cycle assuming that the period of the clock is stable. However, it is not the case since it is quite hard to obtain low jitter. Moreover, the rise time and fall time may change due to environmental conditions such as temperature.

Trying to make the duty cycle perfect requires much effort and comes with large prices, such as area and power. On the other hand, 50% duty cycle waveforms are critical for RNGs based on the irregular sampling method since the throughput is directly related to the regular clock frequency according to the formula driven by [21]. The formula is as follows:

$$S_{(CSR)i} = (\frac{[(\sum^{i}_{j=1} T_{slowj}) - \Delta T] mod 2}{(2d_{fast})}). \tag{3.1}$$

For the specific problem of the irregular sampling of regular waveform method, we propose an application-specific duty cycle correction method. In the irregular sampling of regular waveform method, the duty cycle is important only when the sampling occurs around the middle of the period. Therefore, if we can somehow determine the sampling occurrence at those points, we can discard them, so that sampling at the problematic point can be prevented. So, we propose to sample both the fast regular clock and its complement to determine the sampling at problematic points. Normally, the sampled bits should be the complement of each other since the sampling occurs at the same time and the sampled signals are complement of each other. If the sampled bits are the same, then this means that the sampling occurs around a transition point, meaning that the sampled bit is problematic. Therefore, the sampled bit should be discarded.

The proposed circuit can be used with any RNG based on irregular sampling method. The proposed duty cycle correction circuit that is shown in Figure 3.13 is added to the wake-up & shut-down RNG described in Section 3.1.2. The regular clock

is inverted by realizing one of the LUT (Look-up Table) as an inverter. After that, both the fast regular clock and the inverted fast regular clock are sampled with the irregular clock created by the wake-up & shut-down RNG with two DFFs (D-type flip-flops). There is some delay due to the LUT inverter between the fast regular clock and its inverted version that results in a phase difference different from the desired value, which is 180 degrees. A LUT buffer is added before the fast regular clock is sampled to be able to compensate for the delay caused by the LUT inverter. The number of buffer and inverter stages that results in the best phase match is found experimentally by employing slide switches and changing the number of stages.

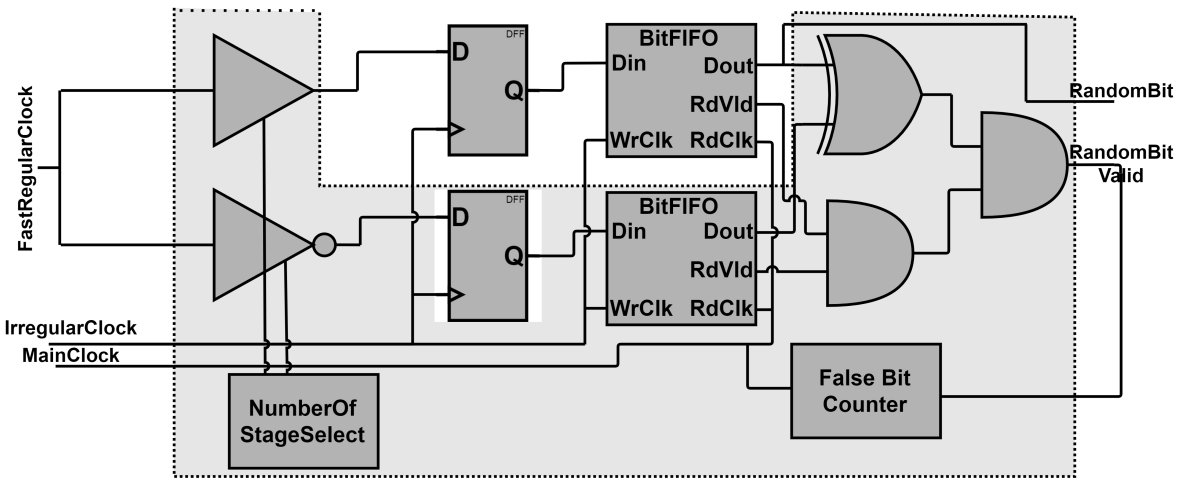

Figure 3.13. The proposed duty cycle correction circuit.

After sampling is done, the sampled bits are appended to an asynchronous first-in-first-out (FIFO) module whose width is one bit and depth is eight with two synchronization stages. This FIFO that is created by using the XILINX IP core generator is used to synchronize the sampled data with the main clock. The main operations are performed in the main clock domain due to difficulties in the asynchronous clock domain.

After the synchronization process is done, the bits are compared with an XOR gate to determine whether the sampled bits are the same or not. The sampled signals should be different from each other, as explained previously. The XOR output is used as a valid signal to the "Random Bit" signal. If the sampled bits are the same, then the XOR output, the "Random Bit Valid" signal, is low. This means that the sampled data is unsuitable for use as a random bit. If the sampled bits are not the same as expected, then the XOR output, the "Random Bit Valid" signal is high. This means that the sampling is done at a proper location and, therefore, it is suitable to be used as a random bit.

An AND gate whose output is ANDed with the XOR output is used to ensure that both of the FIFOs output valid data. In our case, the random bits are appended to another FIFO to send them later via UART. The "Random Bit Valid" signal is used as the write enable signal for the FIFO.

3.2. Modular Multiplication Design

After the background research, the Montgomery Modular Multiplication algorithm [22] seemed the most suitable algorithm for the task since it can be designed as parametrizable and scalable, and it requires non-complex logic. The hardware design of the multiplier is derived from [15], [23], and [24].

The first version of the modular multiplier is designed as a sequential state machine. The algorithm shown in Figure 2.2 is converted to the state machine shown in Figure 3.14.

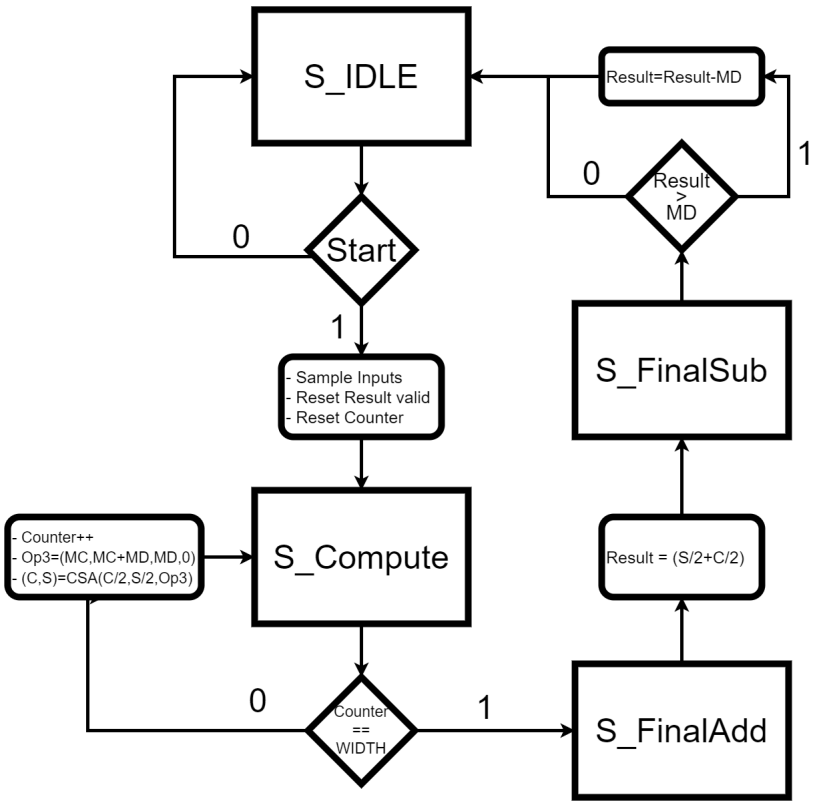

Figure 3.14. Simple flow chart representation of the state machine belonging to the first version of modular multiplier.

The multiplier is composed of four states, and the state machine starts from the S_IDLE state after the reset. The multiplication starts when the "Start" signal is asserted. Therefore, the Start signal should only be asserted after all the inputs (operands) are ready. With the "Start" signal, the inputs are sampled and stored in registers, the counter is flushed, and the state transition from S_IDLE to S_COMPUTE happens. The S_COMPUTE state corresponds to the for the loop at the algorithm shown in Figure 2.2. With the rising edge of each clock cycle, Sum and Carry from the previous output of the carry-save adder and either *multiplicand*, (*multiplicand*+*modulo*), *modulo* or *zero* are summed with the carry-save adder. Then, the shifted, which means divided by two, carry, and sum results are written to corresponding registers. When the counter reaches the "WIDTH" of the operands, the state is switched to S_FINALADD. In this state, the result is converted back to nor-

mal representation from carry-save representation by adding the final carry and sum. Then, the state is switched to S_FINALSUB. In this state, the state transition occurs from S_FINALSUB to S_IDLE unconditionally, and the result valid flag is set. While transition occurs, the modulo is subtracted from the result if the result is greater than the modulo. Otherwise, the result remains unchanged.The block diagram of the design is shown in Figure 3.15, which corresponds to the flow chart shown in Figure 3.14 and the algorithm shown in Figure 2.2.

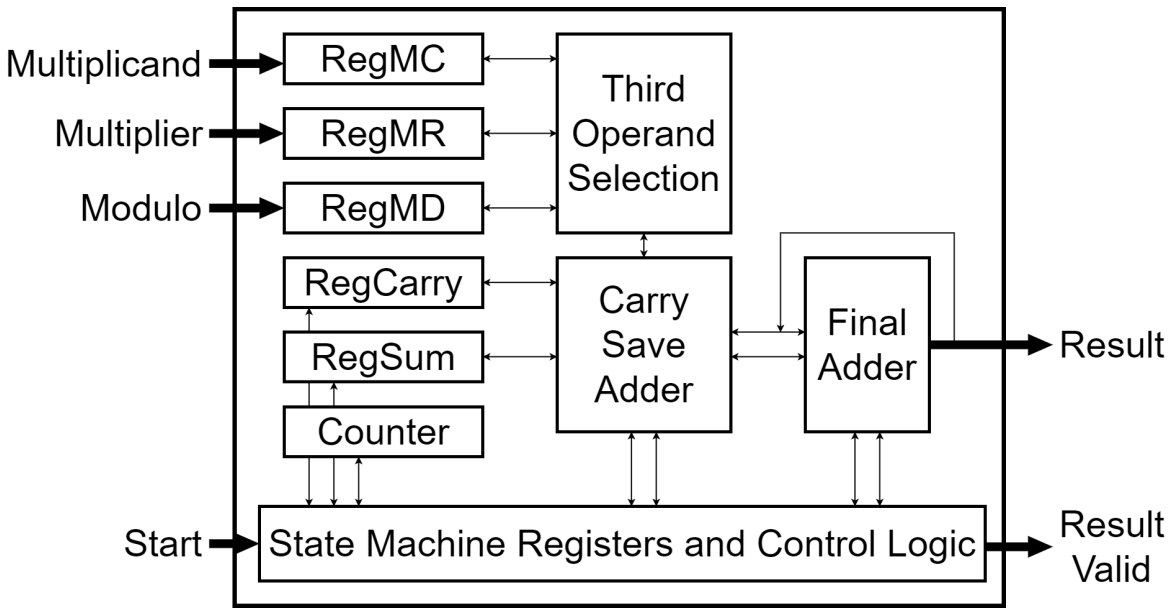

Figure 3.15. Basic block diagram of the first version of modular multiplier.

The design has only one parameter, the "WIDTH" of the operands. The execution time is $WIDTH + 3$ clock cycles. Carry-save adder is used as the main adder since its critical path does not change much with the "WIDTH" change. The CSA can be replaced with a normal adder for small operand sizes. Even though the main adder is a CSA, another adder is required for final addition and subtraction. Furthermore, this is the bottleneck of this configuration. The critical path is directly related to the adder, and the operating frequency is too low for large operand sizes.

The final adder is replaced with a multi-cycle adder in the second revision of the design in order to fix the critical path problem. Therefore, unconditional state transition from S_FINALADD to S_FINALSUB is changed in a way that transition occurs when the addition is completed. Moreover, a new state, S_CHECK, between S_FINALADD and S_FINALSUB is added since the final subtraction also uses the same adder, and its execution is conditional. The size of the adder and, therefore, the number of cycles needed for the addition is determined by the size of the adder. So, a new parameter, ADDERWIDTH, is added. The execution time is increased by $2 \times (WIDTH/ADDERWIDTH) + 1$ at the worst case scenario which is $Result > Modulo$ after the final addition.

The second version improved the operating frequency compared to the first version. However, the critical path for large operands is still long due to the combinational, single-cycle comparison at the S_CHECK state. The comparison is also performed in a multi-cycle fashion in the third version to improve the operating frequency further. A new parameter, COMPWIDTH, is added to the module, which determines how many bits are compared to each cycle. After this modification, the critical path is significantly reduced, and the operating frequency is increased. The cost of this modification to the execution time is $WIDTH/CMPWIDTH$ cycle in worst case scenario.

If we look at the for loop shown in Figure 2.2, which corresponds to the S_COMPUTE state, we can see that the loop can be pipelined. Each iteration in the loop needs the sum and carry results from the previous CSA execution and the corresponding multiplier bits. Some of the iterations can be cascaded combinationally using cascaded carry-save adders. Since the critical path of a CSA is just a full adder for any operand size, cascading CSA does not increase the critical path too much. An example of the modified for loop with cascaded CSAs and two stages multi-cycle pipeline is shown in Figure 3.16.

Pipelined For Loop

First Stage $Sum = 0$

for $i = 0$, $i < WIDTH/2$, $i = i + ((WIDTH/2)/MCYCLE)$ **do**

  for $j = 0$, $j < ((WIDTH/2)/MCYCLE)$, $j = j + 1$ **do**

    if $mr_{i+j} == 1$ **then**

      $Sum = Sum + MC$

    end if

    if Sum is **Odd then**

      $Sum = Sum + MD$

    end if

    $Sum = Sum/2$

  end for

end for

Second Stage

for $k = WIDTH/2$, $k < WIDTH$, $k = k + ((WIDTH/2)/MCYCLE)$ **do**

  for $l = 0$, $l < ((WIDTH/2)/MCYCLE)$, $l = l + 1$ **do**

    if $mr_{k+l} == 1$ **then**

      $Sum = Sum + MC$

    end if

    if Sum is **Odd then**

      $Sum = Sum + MD$

    end if

    $Sum = Sum/2$

  end for

end for

Figure 3.16. Example modified for loop to enable pipelining with two pipeline stages and "REPLICATION" times combinational round.

Two more parameters are added to the design for the next version of the multiplier design. One of the parameters, REPETITION, is for the number of pipeline stages. The other one is MCYCLE, and it determines the number of cycles a pipeline stage runs. The primitive building block of the multiplier is shown in Figure 3.17. The circuit corresponds to one iteration of the For loop. It adds the three operands of the CSA. The first two operands are "Sum" and "Carry" from the previous stage. The third operand is selected according to the corresponding multiplier bit and whether the final sum will be odd or even. The WIDTH parameter determines the CSA width.

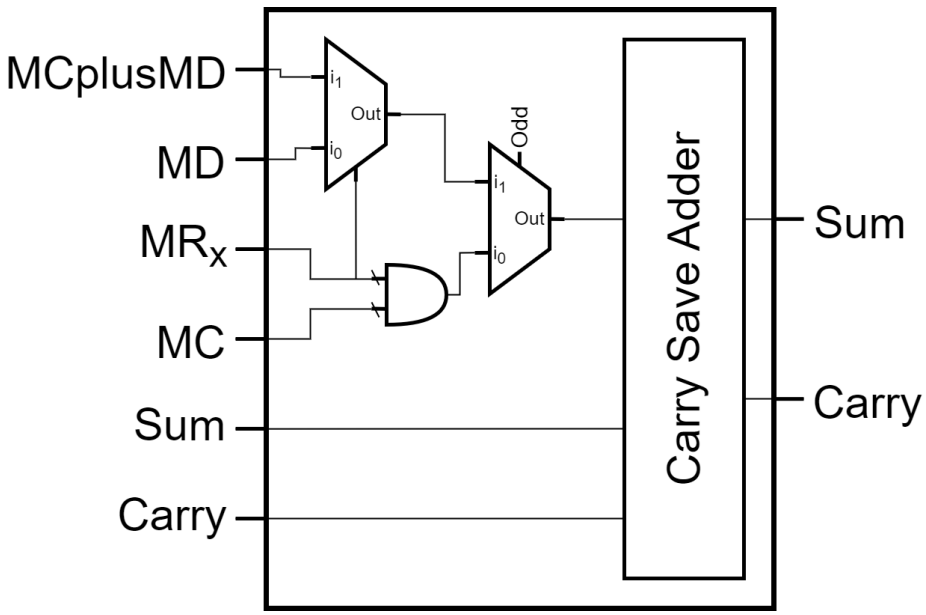

Figure 3.17. Block diagram of the Montgomery primitive that is used as the building block for pipelined montgomery modular multiplier.

More than one Montgomery primitive can be combinationally cascaded. The block diagram of the cascaded circuit is shown in Figure 3.18. The multiplicand, modulo, and the sum of multiplicand and modulo is the same for all primitives. Only the multiplier bit is different from each other. Moreover, the Sum and Carry output of each primitive is shifted and given to the next primitive as input. Therefore, the "Multi-bit Montgomery Primitive" performs n-iterations of the for loop combinationally. In

addition to the WIDTH parameter, the module has a REPETITION parameter that determines how many Montgomery primitives will be cascaded. This parameter is given as ((WIDTH/REPETITION)/MCYCLE) from a higher hierarchy.

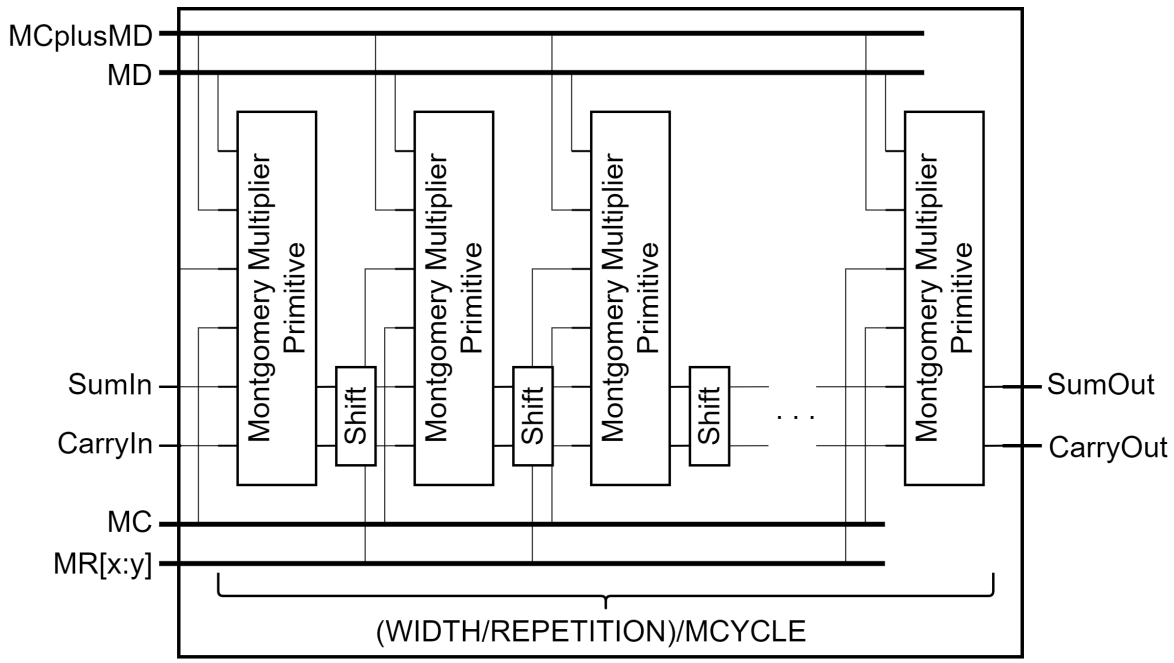

Figure 3.18. Block diagram of the multi-bit Montgomery primitive that provides multiple combinational stages to the pipelined Montgomery modular multiplier.

The logic depth and the area increase with the depth of the "Multibit Montgomery Primitive." Therefore, resource reuse is required. The outputs of the multi-bit Montgomery primitive are shifted and stored in registers. Next cycle, these outputs are used as the input. The multi-cycle operation of this module constructs a pipeline stage. In addition to WIDTH, this module has two other parameters, BWIDTH and MCYCLE. BWIDTH corresponds to the total number of iterations combinational and sequential, and MCYCLE corresponds to two numbers of sequential iterations or multi-cycle operations. The block diagram of the "Montgomery Stage" modular multiplication is shown in Figure 3.19.

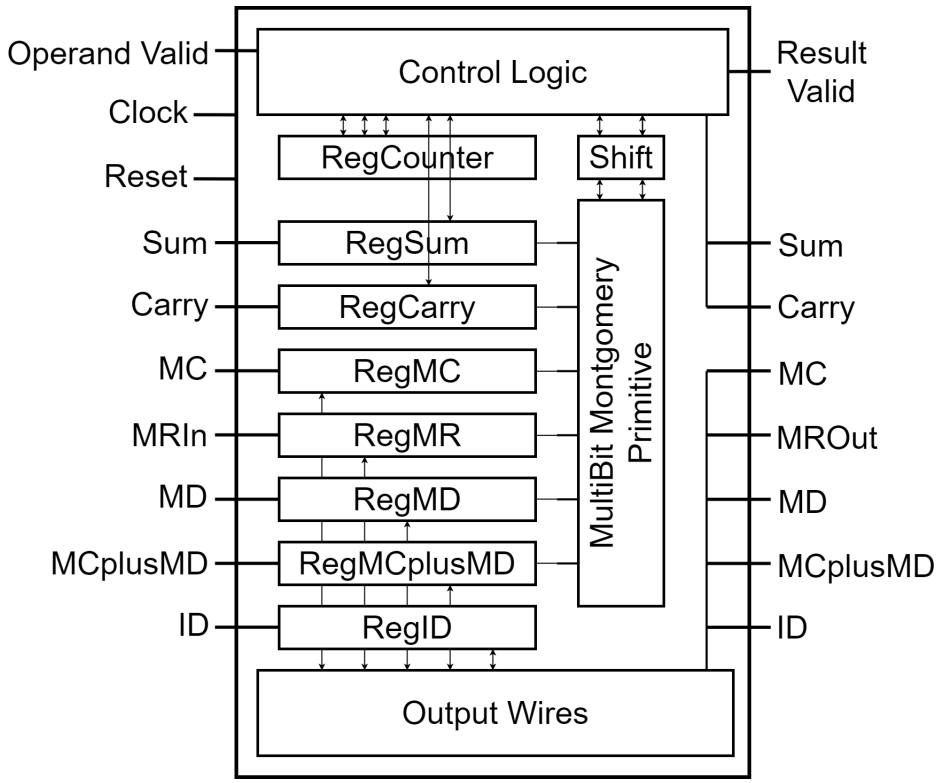

Figure 3.19. Block diagram of a multi-cycle Montgomery stage that forms the pipelined Montgomery modular multiplier.

The block diagram of the pipelined Montgomery modular multiplication is shown in Figure 3.20. This module cascades the Montgomery stages. An output of a stage becomes an input for the next stages. The module has a pipelined adder for standalone use. Moreover, if the multiplier will be used standalone, the same adder can calculate the $Multiplicand + Modulo$ and the final subtraction if required. Otherwise, the pipelined adder can be taken outside, as shown in Figure 3.1, so that area is optimized via resource sharing. The module has four parameters, WIDTH, WORDWIDTH, REPETITION, and MCYCLE. WIDTH determines the size of the multiplication. WORDWIDTH is used to set the size of the base adder in the pipelined adder. REPETITION determines the pipeline stage of the Montgomery Multiplier. MCYCLE determines the sequential operation cycle of each stage.

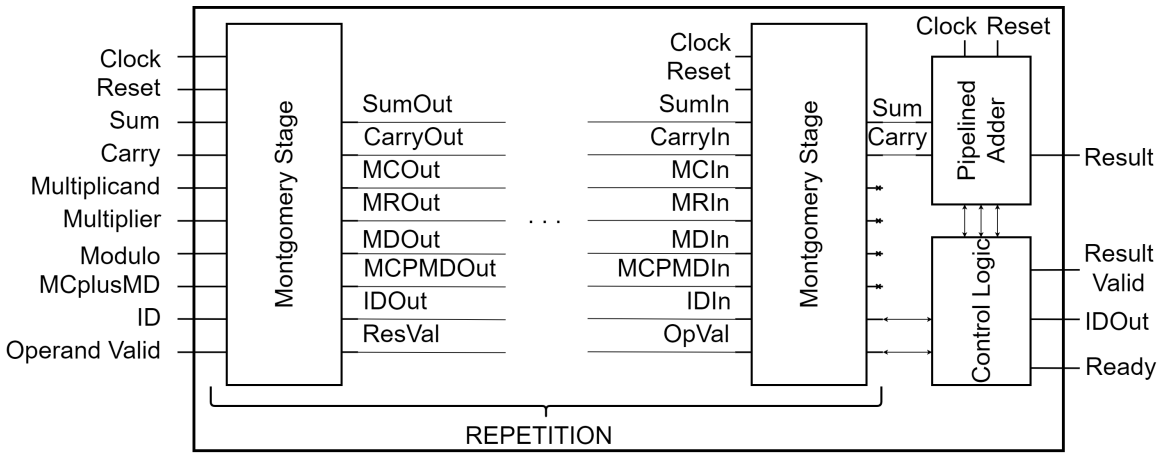

Figure 3.20. Block diagram of the pipelined Montgomery modular multiplier.

3.3. Modular Exponentiation Design

We used the basic modular exponentiation algorithm shown in Figure 2.3 and explained in Section 2.3. The block diagram of the designed module is shown in Figure 3.21.

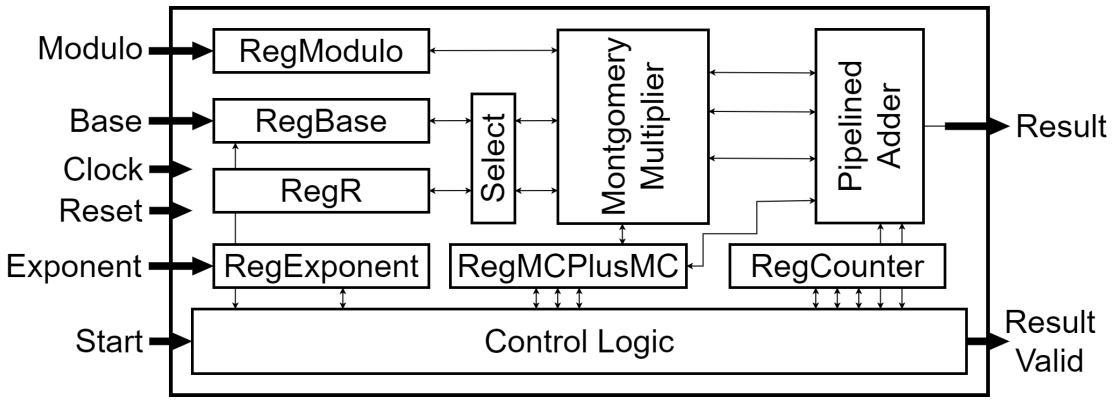

Figure 3.21. Block diagram of the modular exponentiation.

The inputs are sampled with the "Start" signal. The modular square calculation is required unconditionally. If the corresponding exponent bit is one, then the conditional

modular multiplication is also started without waiting for the square modulo result when the pipeline becomes available. Both modular multiplications require the same $Multiplicand + Modulo$ since the multiplicand inputs are RegBase for both operations. Therefore, it is computed only once with the pipelined adder. As mentioned in Section 3.2, the Montgomery multiplier uses the same pipelined adder for the final addition and subtraction. The "RegBase" register samples the "Base" input at the beginning but is updated at each iteration with the result of the square modulo operation. The "RegR" register is updated at each iteration with the conditional modulo result and evolves to the final result. Two pipeline stages are used in the Montgomery Multiplier since, at most, two multiplication is required simultaneously. The Montgomery multiplier and pipelined adder are included in the module for standalone use. Suppose the modular exponentiator will be used with other modules that also require a modular multiplier. In that case, modules are taken outside and shared, as mentioned at the beginning of Chapter 3, and the exponentiator sends requests outside.

3.4. Primality Tests

Generated random numbers are subjected to primality tests. We implemented the probabilistic primality tests for the reasons mentioned in Section 2.4. We chose to implement the first alternative, using the iterated Miller-Rabin test, followed by a single Lucas test. Using FIPS-approved algorithms [1], the Miller-Rabin and Lucas primality tests are designed with FIPS compliance.

The candidate number is started to test with both algorithms simultaneously to reduce execution time. If both tests output "Probable Prime," the candidate is marked as prime. If either of the tests fails, the candidate is marked as "Not Prime" immediately without waiting for the other test to finish, and the next candidate number is taken.

The Modular multiplier unit is shared among the Miller-Rabin, Lucas, and modular exponentiation units. Miller-Rabin and exponentiator modules require, at most,

two modular multiplication simultaneously. Lucas unit also uses two modular multiplication simultaneously. Therefore, the modular multiplication unit has four pipeline stages. The pipelined adder is also shared among all the modules.

The block diagram of the FIPS-compliant primality tester is shown in Figure 3.22.

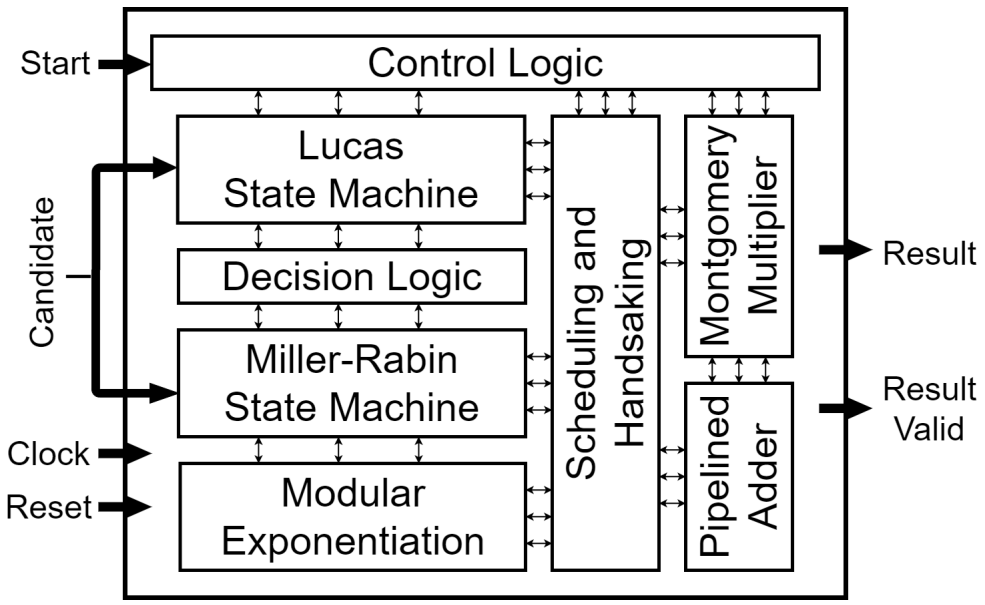

Figure 3.22. Basic block diagram of the primality test unit.

3.4.1. Miller-Rabin Primality Test Design

The algorithm shown in Figure 2.4 is designed for the Miller-Rabin primality test unit. The basic block diagram of the FIPS-compliant Miller-Rabin tester module is shown in Figure 3.23. The candidate number is sampled and stored in the RegA register. RegB is an even number that is the decremented version of RegA. Then, the RegB is factorized to find the maximum element, which is the power of two, by right shifting and checking the least significant bit. After calculating the required data, two for loop is iterated sequentially. Requests to the modular exponentiator and multiplier are inserted inside the for loops.

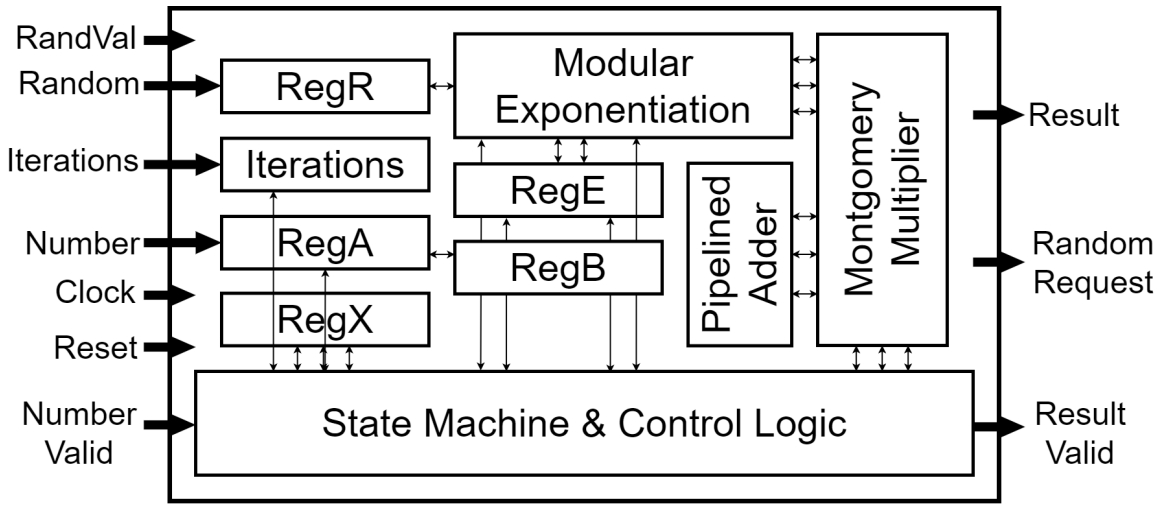

Figure 3.23. Basic block diagram of the FIPS-compliant Miller-Rabin primality test.

3.4.2. Lucas Primality Test Design

The algorithm shown in Figure 2.5 is designed for the Lucas probabilistic primality test unit. The basic block diagram of the FIPS-compliant Lucas tester module is shown in Figure 3.24.

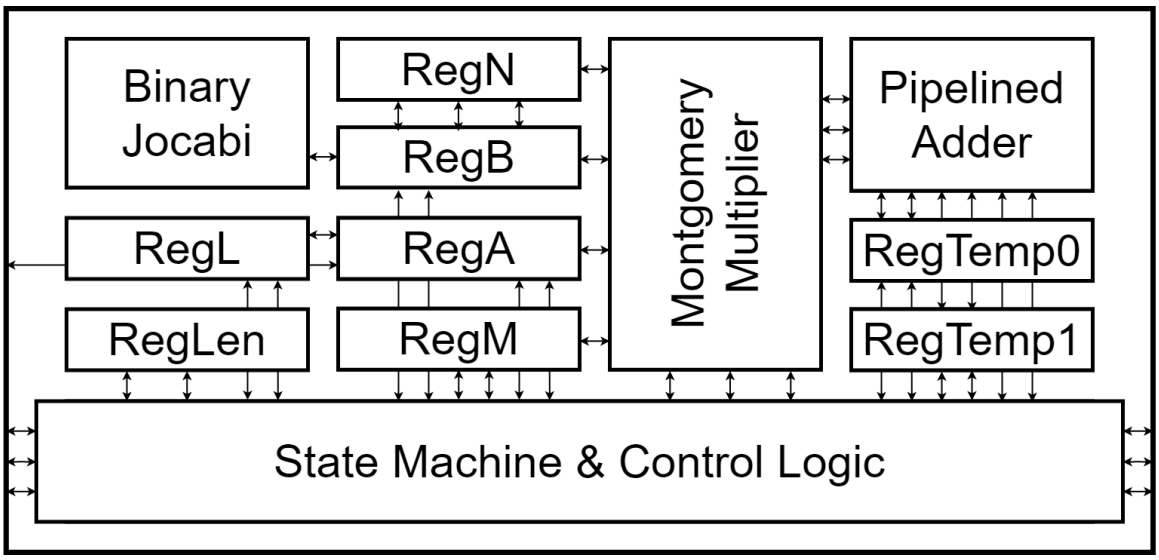

Figure 3.24. Basic block diagram of the FIPS-compliant Lucas primality test.

The candidate number is sampled and stored in the RegA register. RegL is an even number that is the decremented version of RegA. Step four of the algorithm is not required for our purposes since we always use a WIDTH-bit operand to test. If the module will be used for general purposes, then an indice calculator which determines the bit location of the most significant non-zero bit is required. Then, step 6, the for loop, is executed. In the for loop, there are four unconditional modular multiplication, one unconditional modular addition, and two conditional modular addition. Three of the modular multiplications can be done in parallel. Therefore, the number of pipeline stages is set to two so that the pipeline stays busy. To minimize the $MC + MD$ operation, RegM is always used as the multiplicand. Modular additions and the final addition/subtraction of the Montgomery multiplier are performed in the pipelined adder. Intermediate results are stored in two temporary registers, RegTemp0 and RegTemp1.

4. SoC HARDWARE DESIGN AND IMPLEMENTATION

This Chapter integrates all of the modules explained in previous Chapters and a Risc-V core to create an example SoC for IoT applications. The principle block diagram of the SoC is shown in Figure 4.1.

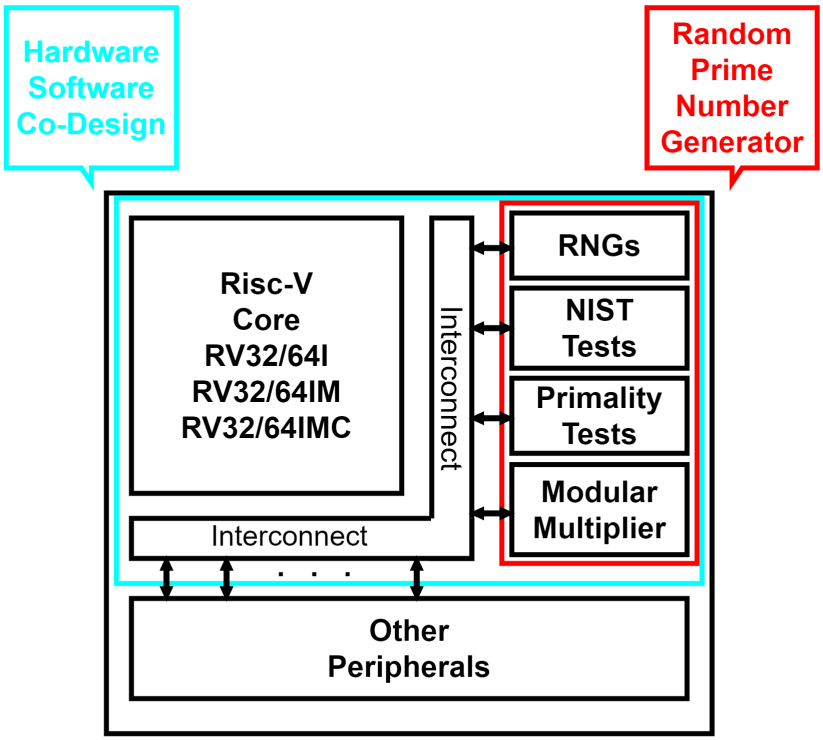

Figure 4.1. The proposed system-on-chip with the proposed random prime number generator.

The small rectangle on the right-hand side shown in Figure 4.1 is the Random Prime Number Generator hardware explained previously. The RPNG can run autonomously without any help from the core side.

The big rectangle shown in Figure 4.1, represents the hardware-software co-design which is the real advantage of the SoC. The co-design approach enables us to create efficient IoT solutions. For example, in addition to the standalone run of the RPNG, the core can enable or disable it via the configuration registers. Moreover, some of the resources, such as the modular multiplier, can be used by the core. Furthermore, A simple load instruction can meet the software's random and prime numbers.

The SoC can also have other standard peripherals, such as GPIO, Timer, UART, SPI, and I2C, depending on the application.

4.1. System-on-Chip (SoC Design)

This section shows the top module design of the SoC. The block diagram of the designed SoC for our thesis is shown in Figure 4.2.

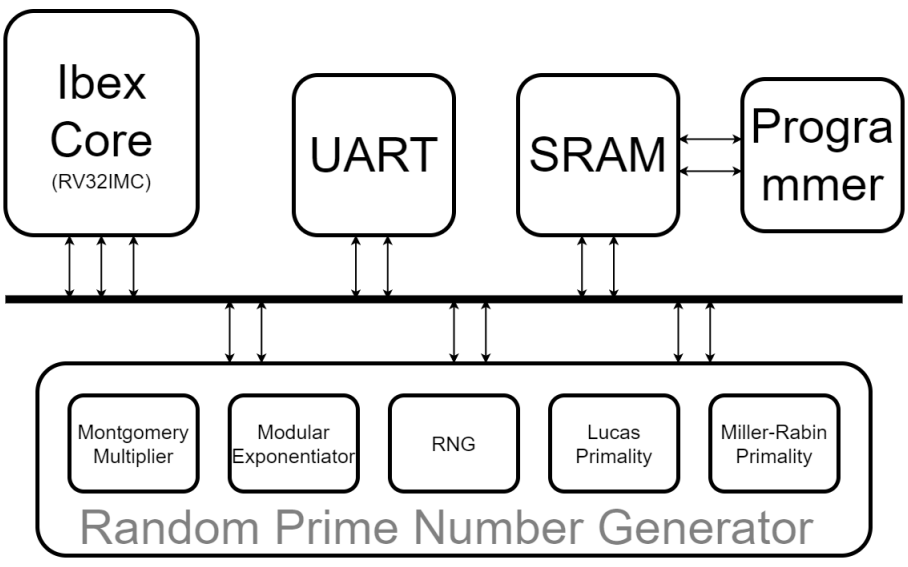

Figure 4.2. Block diagram of the system-on-chip.

The main general-purpose processing element of the SoC is the 32-bit RV32IMC Ibex core. The core is explained in Section 4.2. We only need a UART peripheral to

communicate with the SoC for our proof-of-concept use-case design. The core parameters we use with the Ibex do not include memory instantiation. Therefore, we designed a block-ram based, which corresponds to SRAM in ASIC, memory for instruction and data storage. Moreover, we include a UART-based programmer to easily upload new software while the core is on the run. Finally, SoC includes the designed RPNG.

All parts are connected with a simple and custom bus interface. The peripherals' configuration, control, and data registers are memory-mapped and can be read or written with load-store instructions. The memory map of the device is shown in Table 4.1.

4.2. Core Design

The RISC-V IBEX core is a 32-bit, open-source embedded CPU core intended for use in a variety of applications, including IoT, embedded systems, and other low-power devices. It is based on the RISC-V instruction set architecture (ISA), which is a free and open ISA that can be used for any purpose, allowing processors to be designed with greater flexibility.

The IBEX core is intended to be tiny and low-power by having a minimal number of gates and employing low-power methods. It supports a subset of the RISC-V standard ISA, including the RV32IMC instruction set, which consists of instructions for integer arithmetic, logical operations, and control flow. It is designed in a parametric fashion. Therefore, many parameters, such as enabling the I$ and branch prediction, give design flexibility to the designers.

The IBEX core is open source, and the source code, documentation, and tools are accessible via the Apache 2.0 open-source license. Being open-source makes it possible for developers to use, alter, and distribute the core without incurring any legal or monetary costs.

Table 4.1. Memory map of the system-on-chip.

Module	Address	Width	Attribute	Description
Empty Region	0x0000_0000	NA	NA	Reserved
	...			
	0x1FFF_FFFF			
UART	0x2000_0000	4B	RW	TxFIFOFull/WriteToTxFIFO
	0x2000_0004	4B	R	RxFIFOEmpty
	0x2000_0008	4B	R	RxFIFOData
	0x2000_000C	4B	RW	UARTControl
Empty Region	0x2000_0010	NA	NA	Reserved
	...			
	0x20FF_FFFF			
Random Prime Number Generator	0x2100_0000	4B	R	Random Number FIFO Status
	0x2100_0004	4B	R	Random Number FIFO Data
	0x2100_0008	4B	R	Prime Number FIFO Status
	0x2100_000C	4B	R	Prime Number FIFO Data
	0x2100_0010	4B	RW	Miller-Rabin Number of Iterations
	0x2100_0014	4B	R	Miller-Rabin Performance Counter(Prime)
	0x2100_0018	4B	R	Lucas Performance Counter(Prime)
	0x2100_001C	4B	W	Montgomery
	0x2100_0020	4B	W	Trigger Primality
	0x2100_0024	4B	R	Miller-Rabin Performance Counter(Not Prime)
	0x2100_0028	4B	R	Lucas Performance Counter(Not Prime)
Empty Region	0x2100_002C	NA	NA	Reserved
	...			
	0x21FF_FFFF			
NIST Tests	0x2200_0000	4B		Status
	0x2200_0004	4B		Control
	0x2200_0008	4B		Frequency Result
	0x2200_000C	4B		Runs Result
	0x2200_0010			Reserved
	...			
	0x2200_FFFF			
Empty Region	0x2201_0000	NA	NA	Reserved
	...			
	0x3FFF_FFFF			
Main Memory	0x4000_0000	B 2B 4B	RWX	SRAM(64KB)
	...			
	0x4000_FFFF			
	0x4001_0000	NA	NA	Reserved
	...			
	0xFFFF_FFFF			

The IBEX core is designed to be readily incorporated into a variety of systems. It is compatible with many development tools and platforms, including the Open-Source RISC-V Toolchain, the Open-Source RISC-V Debugger, and the Open-Source RISC-V Testbench.

Ibex was initially developed in 2015 as a component of the PULP platform for energy-efficient computing under the title "Zero-riscy." A significant portion of the code was created by simplifying the RV32 CPU "RI5CY" [25] core to show how compact a RISC-V CPU core may realistically be. Under the code name "Micro-riscy," support for the "E" extension was added to make it even more compact. The core serves as the PULP ecosystem's control core for PULP, PULPino, and PULPissimo. Zero-riscy's development was taken up by lowRISC in December 2018, and it was given the new name Ibex. [16]

4.3. Peripherals and Memory

Memory region between 0x2000_0000 to 0x3FFF_FFFF is reserved for the peripherals.

We add a UART module whose baud rate is tested up to 2Mb/s as the primary communication channel. Its base address is 0x2000_000, and It has four memory-map registers. A byte is written to the 0x2000_0000 address with a store byte command to send data through the UART. The UART has a built-in transmit FIFO. The same address is read in order to check the status of the transmit FIFO. Similarly, the received data is read with a simple load byte command to address 0x2000_0008, which corresponds to Receive FIFO. The check whether any data is received or not, the receive FIFO Empty flag can be read from 0x2000_0004.

The functionality of the RPNG modules is explained in their respective sections. Each memory-mapped RPNG register shown in Figure 4.1 can be used with load-store instructions.

The main memory starts at 0x4000_0000. A parametrized SRAM and SRAM controller with dual-port capability is designed. For this SoC, the SRAM size is 64KB. The programmer uses the write pin of the first port, and the fetch unit uses the read pin. The other port is dedicated to data read and write.

A UART-base programmer is added for on-the-run programming functionality. The programmer consists of a UART receiver with a fixed baud rate, 1MBaud/s, and a state machine. The state machine checks for a specific programming sequence, "SECURE SOC" for our design. After receiving the sequence, the first upcoming word is the amount of expected data. Then, the upcoming data bytes are packed into SRAM data line words and written to the SRAM.

4.4. Hardware-Software Co-design

The aim of the co-design for this work is to offload the operations that are costly for software to the dedicated hardware.

Instead of running the primality tests on hardware-generated random numbers with software, the costly primality tests are run on the dedicated hardware. The software can use the result.

Moreover, the software can use dedicated modular multiplier hardware if the application requires modular multiplication.

Furthermore, some of the NIST randomness tests can run on hardware while the random number is being generated. The software can set the parameters and use the results to take a measure.

The functionality of the co-design can be extended further depending on the applications.

5. RESULTS AND DISCUSSIONS

This chapter presents the implementation results of the designed modules. With the help of parametrized modules, each module is synthesized and implemented with different configurations. This allows us to compare different configurations. Moreover, we can determine the optimum design for various IoT devices. For the FPGA implementations, we used Vivado Synthesis 2020 tool with Vivado synthesis default settings and Vivado Implementation 2020 tool with Vivado implementation default settings. Tools are used as a part of Vivado Design Suite 2020.2 with GUI, TCL, and Batch mode.

The submodules are implemented in out-of-context flow so that the tool does not try to map a huge number of IOs to external IOs. This allows us to see more realistic results when the module is used as a part of a bigger design.

5.1. Random Number Generators

5.1.1. Regular Sampling Method Based Random Number Generator Results

The proposed regular sampling method based shut-down & wake-up circuits are implemented in the Xilinx Virtex5 ML506 evaluation board that contains xc5vlx50t FPGA. Moreover, all the designed circuits are tested in NIST SP800-22 [6] test suite.

All designs are started to test with 114 rings. If the designed RNG passes the test, then the number of rings is decreased to find the minimum number of rings that satisfies the NIST tests. As mentioned in the third section, the sampling rate of the proposed circuit is 4.5 MHz. The sampling rate of the shut-down [11] and wake-up [12] ring oscillators is 2.25 MHz. Sampling points are selected as the same for all designs so that the comparison is fair while showing that our proposal provides a significant

improvement. The sampling point is chosen at which the stability of the loop changes since metastable events occur most at that point.

The data sets contain 320 million bits as 320 bitstreams. All collected data are tested using the NIST SP800-22 test suit. The resultant minimum number of rings required for each design is shown in Table 5.1. The proposed method satisfies the statistical test using only 14 ring oscillators without any post-processing methods. The test results of the proposed method are presented in Table 5.2.

Table 5.1. Design summary of the proposed, shut-down-only, and wake-up-only random number generators.

	Proposed Method	**Shut-Down Method**	**Wake-Up Method**
Number of Ring Oscillators	14	25	24
Number of Slice LUTs	211	368	354
Number of Slice Registers	26	37	36
Sampling Frequency	4.5MHz	2.25MHz	2.25MHz

The minimum number of rings that the designed RNGs satisfy the NIST tests is chosen as the figure of merit since a lower amount of rings means lower silicon area, power, and cost. Therefore, our method lowers the number of rings by 44% compared to the Shut-down method and 41.67% compared to the Wake-up method. Furthermore, our method uses 41.5% fewer resources compared to the Shut-down method and 39% fewer resources compared to the Wake-up method. In addition to these improvements, the throughput of our proposed method doubles the other two methods.

Table 5.2. NIST SP800-22 statistical test suite results of the proposed regular sampling based random number generator.

Statistical Test	**P Values**	**Proportion**	**P/F**
Frequency	0.350485	0.98125	Pass
Block Frequency	0.000533	0.97813	Pass
Cumulative Sums	0.598138	0.98438	Pass
Runs	0.828826	0.99375	Pass
Longest Run	0.966721	0.98125	Pass
Rank	0.371101	1.00000	Pass
FFT	0.894201	0.98750	Pass
Nonoverlapping Template	0.953553	0.99063	Pass
Overlapping Template	0.585209	0.99375	Pass
Universal	0.267238	0.98125	Pass
Approximate Entropy	0.617605	0.97813	Pass
Random Excursions	0.915031	0.98941	Pass
Random Excursions Variant	0.799176	1.00000	Pass
Serial	0.572333	0.99063	Pass
Linear Complexity	0.701879	0.99063	Pass

5.1.2. Irregular Sampling Method Based Random Number Generator Results

The proposed Shut-down&Wake-up ring oscillator and irregular sampling method based RNG is implemented for Xilinx Zynq7000 FPGA. For the zynq7000 implementation, shut-down-only and wake-up-only methods require forty rings to fulfill the statistical tests while the regular sampling method is employed. On the other hand, when we use both methods simultaneously, thirty-four rings are enough to fulfill the statistical tests. Furthermore, using both methods simultaneously with thirty-four rings provides a larger ApEn value than individual methods. ApEn values for simultaneous, shut-down-only, and wake-up-only methods are 0.618, 0.420, and 0.479, respectively.

A UART module with a 2000000 baud rate is used to collect the random data for testing purposes. We used one asynchronous FIFO and one synchronous FIFO before sending the collected data. The asynchronous FIFO is used to synchronize the random data which is generated with the irregular clock. The synchronized random data is packed into bytes and stored in the synchronous FIFO. The UART module reads data from the synchronous FIFO whenever it is ready.

After determining the number of rings required to satisfy the statistical tests in the regular sampling method, thirty-four rings are used for irregular sampling methods. We collect and test many bitstreams of one million bits to perform the NIST tests. Table 5.3 shows a summary of the NIST results of the collected data.

Table 5.3. Summary of NIST SP800-22 statistical test suite results of the proposed irregular sampling based random number generator.

Statistical Test	Proportion			P-Value			Minimum Pass Rate	Pass/Fail		
	Shut-Down &Wake-Up	Wake-Up	Shut-Down	Shut-Down &Wake-Up	Wake-Up	Shut-Down		Shut-Down &Wake-Up	Wake-Up	Shut-Down
Frequency	268/270	268/270	266/270	0,15122	0,76194	0,20509	262/270	Pass	Pass	Pass
Block Frequency	268/270	270/270	203/270	0,11715	0,00000	0,00000		Pass	Fail	Fail
Cumulative Sums	266/270	268/270	265/270	0,66409	0,33280	0,73248		Pass	Pass	Pass
Runs	264/270	1/270	0/270	0,08992	0,00000	0,00000		Pass	Fail	Fail
Longest Run	265/270	262/270	255/270	0,70994	0,00002	0,00000		Pass	Fail	Fail
Rank	262/270	267/270	268/270	0,92542	0,46860	0,17469		Pass	Pass	Pass
FFT	267/270	263/270	265/270	0,31572	0,35652	0,51192		Pass	Pass	Pass
Non-Overlapping Template	269/270	178/270	132/270	0,93406	0,00000	0,00000		Pass	Fail	Fail
Overlapping Template	268/270	234/270	163/270	0,61785	0,00000	0,00000		Pass	Fail	Fail
Universal	269/270	260/270	263/270	0,54914	0,10051	0,04700		Pass	Fail	Pass
Approximate Entropy	267/270	0/270	38/270	0,00013	0,00000	0,00000		Pass	Fail	Fail
Random Excursions	166/168	158/160	180/183	0,77678	0,42582	0,05139	162/168, 154/160, 177/183	Pass	Pass	Pass
Random Excursions Variant	168/168	157/160	183/183	0,76466	0,35049	0,90772		Pass	Pass	Pass
Serial	267/270	268/270	254/270	0,74731	0,70994	0,00000	262/270	Pass	Pass	Fail
Linear Complexity	267/270	265/270	270/270	0,42036	0,43386	0,46860		Pass	Pass	Pass

The results that are presented in Table 5.3 shows that using both methods at the same time fulfills all the NIST requirements. In contrast, individual methods failed from some of the requirements when the same conditions were applied. These results show that the proposed method significantly improves the overall entropy. Therefore, the proposed method requires lower area and power consumption than individual methods since it fulfills the NIST requirements with less number of rings.

5.1.3. Duty Cycle Correction Results

The proposed duty cycle correction is applied to the proposed irregular sampling based RNG, and it is implemented for Xilinx Zynq7000 FPGA. Random data is collected with the same UART module as described in Section 5.1.2.

During the place and route process of the FPGA implementation, the tool tries to remove the buffer or inverter stages which are used to adjust the phase. Special constraints such as "KEEP" and "S" are written to avoid such optimization. Furthermore, the routing of the RNG is performed by hand so that the rings can be placed separately.

Firstly, the designed RNG is tested without applying any duty cycle correction method. The point at which the data starts to pass the frequency test is found, and then the downsampling rate is increased until the collected data passes all the NIST requirements. The final throughput of the RNG is found as 13 kbps. Then the proposed and described duty cycle correction method is applied to the RNG. The first observation is that all collected data yield to better zero-one balance. Table 5.4 shows some example frequency test results. These results prove that the proposed method provides better frequency results in all cases. All the P values of the RNG without duty cycle correction are zero and, therefore, not shown in the table. The 'F' means the test is failed, and 'P' means the test is passed. The same procedure has been applied to the RNG with the proposed duty cycle correction method. The RNG passes all the NIST requirements with a final throughput of 160.2 kbps. The NIST test results are shown in Table 5.5.

Table 5.4. Frequency test results of the irregular sampling of regular waveform method based random number generator with and without duty cycle correction circuit.

Without Duty Cycle Correction		With Duty Cycle Correction		
Data Rates	**Proportion**	**Data Rates**	**P Values**	**Proportion**
1647Kbps	0.125 (F)	1440Kbps	0.6438	0.984 (P)
1098Kbps	0.161 (F)	960Kbps	0.0256	0.971 (P)
549Kbps	0.115 (F)	480Kbps	0.1056	0.980 (P)
470Kbps	0.181 (F)	411Kbps	0.1626	0.948 (P)
411Kbps	0.179 (F)	360Kbps	0.0002	0.962 (P)
366kbps	0.088 (F)	320Kbps	0.2856	0.957 (P)
329Kbps	0.161 (F)	288Kbps	0.7231	0.984 (P)
299Kbps	0.071 (F)	261Kbps	0.1626	0.947 (P)

Considering these results, the proposed duty cycle correction circuit increased the throughput 12.3 times even though it discards approximately 23000 bits every hundred million bits. Moreover, the area overhead of the proposed circuit is quite small. Furthermore, the proposed circuit can be used with any irregular sampling based RNG.

5.2. Modular Multiplier

This section discusses the results of the designed Montgomery modular multiplier. As mentioned in Section 3.2, the module is parametrized, and we can control the number of pipeline stages and combinational stages.

Table 5.5. Statistical test results of the irregular sampling of regular waveform method based random number generator with duty cycle correction circuit.

Statistical Test	**P Values**	**Proportion**	**P/F**
Frequency	0.407091	0.941	Pass
Block Frequency	0.534146	1.000	Pass
Cumulative Sums	0.022503	0.941	Pass
Runs	0.178278	1.000	Pass
Longest Run	0.534146	0.971	Pass
Rank	0.014216	1.000	Pass
FFT	0.100508	0.941	Pass
Nonoverlapping Template	0.671779	1.000	Pass
Overlapping Template	0.804337	1.000	Pass
Universal	0.739918	1.000	Pass
Approximate Entropy	0.739918	1.000	Pass
Random Excursions	0.122325	1.000	Pass
Random Excursions Variant	0.213309	1.000	Pass
Serial	0.911413	1.000	Pass
Linear Complexity	0.602458	1.000	Pass

Table 5.6 shows the synthesis result for different configurations of the 1024 Montgomery Modular Multiplier design, which targets the xc7a200tsbg484-1 FPGA. The synthesis flow is run with rough estimate timing constraints. Furthermore, Table 5.7 shows the implementation result that is run following the synthesis results, targeting the same FPGA. The PS and CC in Table 5.6, and 5.7 correspond to "Pipeline Stages" and "Combinational Cascading" respectively.

The implementation results for some configurations are not included due to the overutilization of the target FPGA.

Table 5.6. Synthesis results of 1024-bit the Montgomery modular multiplier targeting xc7a200tsbg484-1 FPGA.

Configuration		Clock Frequency (MHz)	LUT	FF	Power (mW/MHz)	Latency (uS)	Data Rate
PS	CC						
1	1	265,393	10 447	13 426	1,315	4,220	242,645
1	2	265,393	13 511	13 435	1,785	2,291	446,977
1	4	149,031	19 190	13 436	4,069	2,362	433,546
1	8	84,083	29 557	13 553	9,308	2,664	384,380
1	16	45,806	50 300	13 812	18,160	2,445	418,802
2	1	265,393	17 723	19 613	2,603	4,220	485,290
2	2	221,729	26 913	19 614	4,965	2,742	746,878
2	4	148,920	33 026	19 616	9,505	2,364	866,446
2	8	80,730	51 293	19 913	20,590	2,180	939,401
2	16	45,750	95 250	20 330	40,410	2,448	836,569
4	1	265,393	30 036	31 937	5,833	4,220	970,579
4	2	221,779	45 365	31 938	10,882	2,741	1494,088
4	4	148,544	56 179	31 981	20,132	2,047	2001,439
4	8	80,613	97 801	32 597	42,275	2,183	1876,076
4	16	45,750	185 226	33 416	84535,000	2,448	1673,137
8	1	265,393	43 458	56 578	11,868	4,220	1941,159
8	2	220,264	63 981	56 579	21,300	2,542	3222,152
8	4	148,544	107 516	56 689	41,860	2,047	4002,877
8	8	81,867	189 276	57 921	85,987	2,150	3810,516
8	16	45,750	365 130	59 557	172,985	2,448	3346,274

Table 5.7. Implementation (post-route) results of the 1024-bit Montgomery modular multiplier targeting xc7a200tsbg484-1 FPGA.

Configuration		Clock Frequency (MHz)	LUT	Power (mW/MHz)	Latency (uS)	Data Rate (Mb/s)
PS	CC					
1	1	214,592	10 449	1,470	5,219	196,199
1	2	178,285	13 533	1,798	3,410	300,269
1	4	117,000	20 206	4,148	3,009	340,364
1	8	72,327	30 456	9,410	3,097	330,640
1	16	33,246	50 951	22,904	3,369	303,961
2	1	209,996	17 549	2,960	5,333	383,992
2	2	151,699	25 871	5,804	4,008	510,986
2	4	101,719	33 903	11,715	3,461	591,820
2	8	62,984	54 529	22,624	2,794	732,907
2	16	34,785	95 687	45,520	3,220	636,069
4	1	180,018	29 981	5,945	6,222	658,352
4	2	141,723	46 290	12,628	4,290	954,768
4	4	85,434	57 133	25,060	3,558	1151,105
4	8	58,275	102 784	48,688	3,020	1356,220
8	1	169,779	46 893	12,904	6,597	1241,814
8	2	122,699	65 110	26,520	4,564	1794,917
8	4	82,298	107 940	51,832	3,694	2217,708

Investigating the results shown in Table 5.6 and 5.7, the graphs shown in Figure 5.1, and 5.2 are plotted.

If we look at the frequency graphs, we see that frequency decreases with increasing pipeline and combinational stages. In theory, we do not expect too much frequency decrease in the pipeline stage increase since the logic depths remain the same. This expectation can be observed in the synthesis result. On the other hand, the implementation results show more significant changes. The reason for this is due to the routing source of the FPGAs. With the increasing resource usage, net routing becomes more difficult, and the net delay becomes much more dominant.

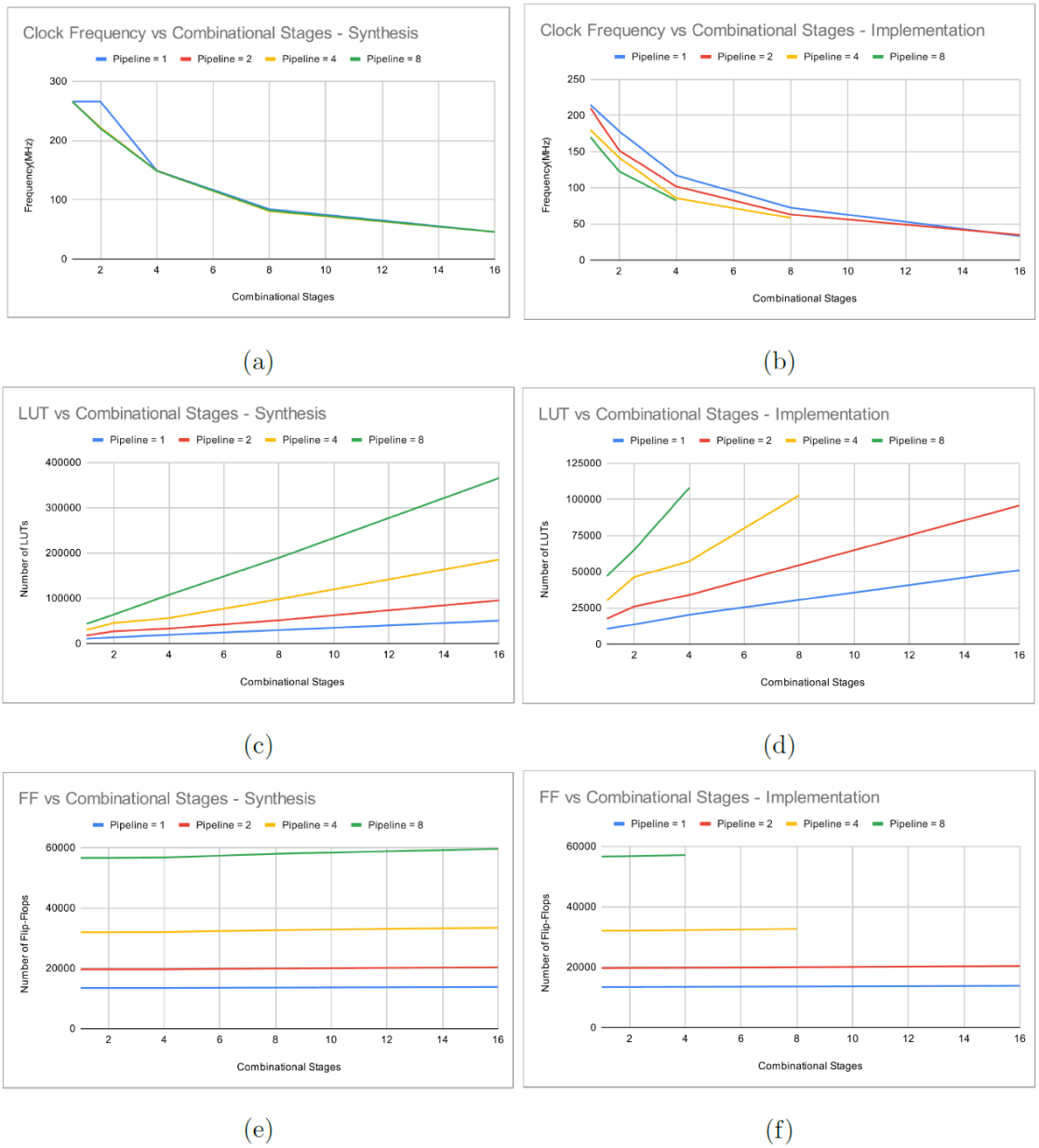

Figure 5.1. Maximum operating frequency (MHz), LUT usage, and FF usage for different configurations after synthesis and implementations.

As expected, the LUT usage increases with the increasing pipeline and combinational stages. The FF usage, on the other hand, increases only when pipeline stages increase since, with each pipeline stage, the results and operands from the previous stage must be stored.

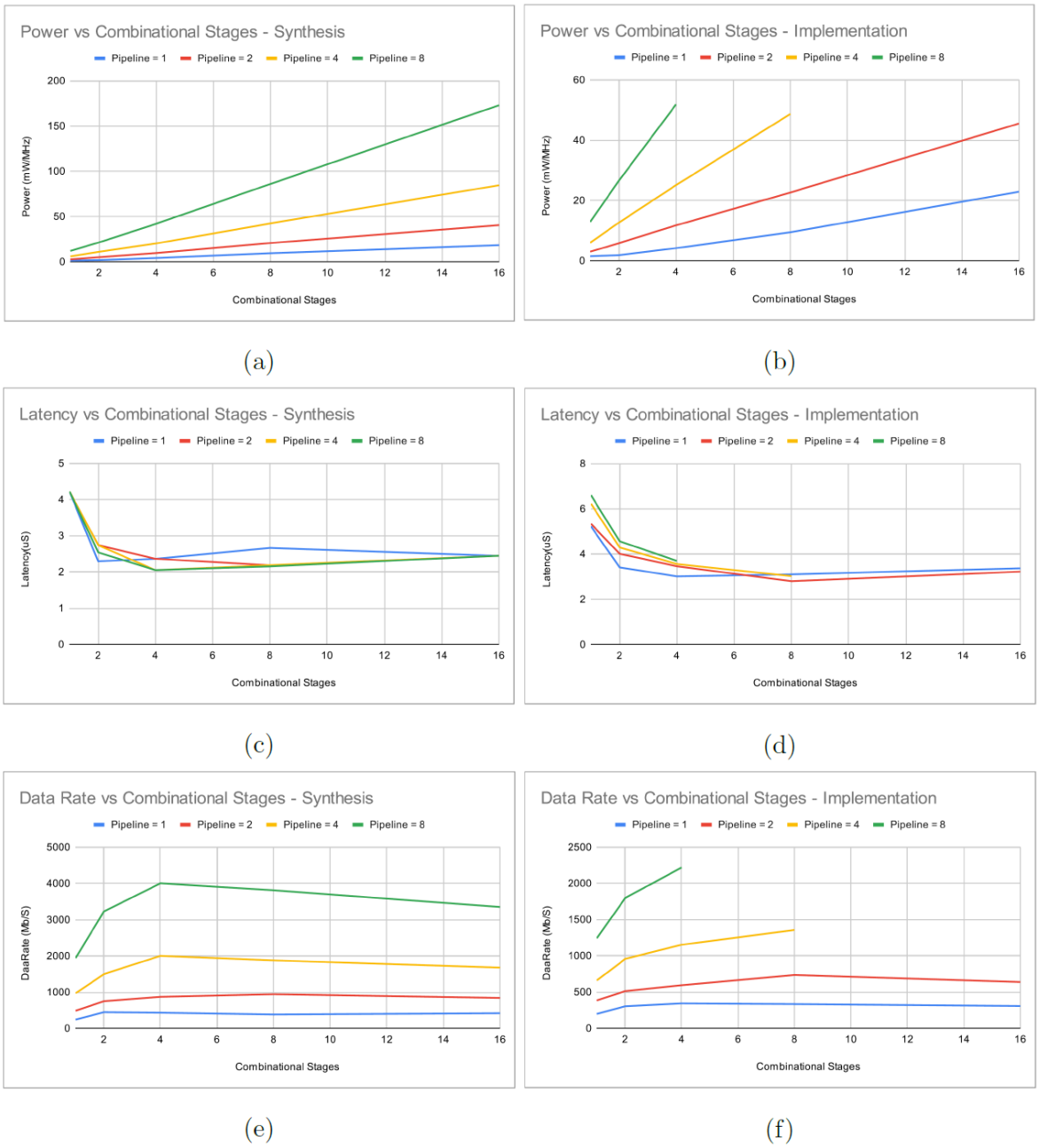

Figure 5.2. Power consumption (mW/MHz), latency(uS), and data rate(Mb/S) for different configurations after synthesis and implementations.

If we look into power consumption graphs, power per MHz increases with increasing pipeline and combinational stages. This is expected since more stage means more utilization, and more utilization means higher power consumption.

Latency graphs show interesting and useful results. The latency for different pipeline stages is similar as expected. On the other hand, the latency seems to saturate after a point with increasing combinational stages. This means the latency gain from the combinational stages is lost to the decrease in operating frequency.

Data rates are increasing with the pipeline stages even though latency for an individual operation remains the same.

5.3. Modular Multiplier Design Comparison

There are a lot of studies on modular multiplication design and implementation. The papers [24], [23], and [26] investigate, propose and compare different modular multiplier designs. The paper [26] compares the modular multiplication designs presented in [27], [28], [29], [30], and [31]. Furthermore, [26] claims that their design is better than most of the configurations compared to others. Therefore, we compare some of our designs from Table 5.7 with those proposed and investigated in [26]. Amanor, Daly, MaslePHd, McIvor, Oksuzoglu, and Tang keywords in the "Paper" column of Table 5.8 corresponds to [27], [28], [26], [29], [30], and [31], respectively.

Table 5.8. 1024-bit modular multiplier comparison.

Paper	Configuration		Frequency (MHz)	LUT	Latency (µs)	Data Rate (Mb/S)	Data Rate/ LUT (Normalized)	Target
	PS	CS						
MaslePHd	1	1	195,31	11415	5,51	194,93	0,6429	XC5VLX110T-3
MaslePHd	1	8	90,19	32937	2,00	710,42	0,7857	XC5VLX110T-3
MaslePHd	2	1	170,56	17585	6,32	339,79	0,7143	XC5VLX110T-3
MaslePHd	2	4	82,47	36038	3,76	649,60	0,6786	XC5VLX110T-3
MaslePHd	2	8	70,83	60629	2,57	1098,94	0,6786	XC5VLX110T-3
MaslePHd	8	7	51,48	204923	4,23	2510,26	0,4643	XC5VLX330T-3
This Work	1	1	214,59	10449	5,22	196,20	0,6786	xc7a200tsbg484-1
This Work	1	2	178,28	13533	3,41	300,27	0,8214	xc7a200tsbg484-1
This Work	2	1	210,00	17549	5,33	383,99	0,7857	xc7a200tsbg484-1
This Work	2	2	151,70	25871	4,01	510,99	0,7143	xc7a200tsbg484-1
This Work	4	1	180,02	29981	6,22	658,35	0,7857	xc7a200tsbg484-1
This Work	4	2	141,72	46290	4,29	954,77	0,7500	xc7a200tsbg484-1
This Work	8	1	169,78	46893	6,60	1241,81	0,9643	xc7a200tsbg484-1
This Work	8	2	122,70	65110	4,56	1794,92	**1,0000**	xc7a200tsbg484-1
This Work	8	4	82,30	107940	3,69	2217,71	0,7500	xc7a200tsbg484-1
Tang	N/A	N/A	90,11	N/A	1,49	688,6	N/A	XC2V3000-6
McIvor	N/A	N/A	75,23	23234	13,46	76,08	0,1429	XC2V3000
Daly	N/A	N/A	54,61	10116	19,03	54,45	0,2143	XCV1000
Amanor	N/A	N/A	49	8064	21	46,86	0,2143	XVC2000E-6
Oksuzoglu	N/A	N/A	119	6906	7,62	134,35	0,7143	XC3S500E-4FG320C

Most of our configurations are better than [27], [28], [29], [30], and [31], and [31]. As mentioned in [26], these papers are outdated due to the limitation of the used old FPGA devices. Therefore, a comparison with [26] is more valuable. When we compare our results, most of the configuration performs similarly.

Unfortunately, we cannot compare the power consumption values since they are not provided in some of the papers. When we use the DataRate/LUT as the figure of merit, our design with eight pipeline stages and two combinational stages performs best in Table 5.8.

5.4. Modular Exponentiator

This section discusses the results of the designed modular exponentiator. The modular exponentiator uses the parametrized modular multiplier as mentioned in Section 3.3. However, the same section discusses that in order to keep the pipeline busy, the number of pipeline stages should be at most two. Moreover, as we see from the mod-

ular multiplier graphs, the latency starts to saturate after four combinational stages. Therefore, the modular exponentiator design is implemented for one or two pipeline stages with one to four combinational stages.

Table 5.9 shows the synthesis and implementation results for different configurations of the 1024-bit modular multiplier, which targets the xc7a200tsbg484-1 FPGA. The synthesis flow is run with rough estimate timing constraints. The PS and CC in Table 5.9 correspond to "Pipeline Stages" and "Combinational Cascading" respectively.

Table 5.9. Synthesis and implementation (post-route) of 1024-bit modular exponentiator for xc7a200tsbg484-1 FPGA.

Synthesis Results							
Configuration		Clock Frequency (MHz)	LUT	FF	Slice	Power (mW/MHz)	Latency (uS)
PS	CC						
1	1	177,968	11828	16545	x	2,31	12704,5
1	2	177,968	13328	16542	x	3,19	6812,6
1	4	150,173	17478	16585	x	6,1525	4582,2
2	1	177,968	19089	21695	x	4,26	9758,5
2	2	177,968	22138	21710	x	6,61	5339,6
2	4	150,173	28286	21759	x	10,395	3709,4
Implementation Results							
Configuration		Clock Frequency (MHz)	LUT	FF	Slice	Power (mW/MHz)	Latency (uS)
PS	CC						
1	1	142,735	11832	16562	4292	2,605	15840,5
1	2	149,031	13327	16554	4885	3,355	8135,3
1	4	109,111	17484	16593	5766	6,59	6306,7
2	1	149,611	18950	21713	6543	4,2725	11608,1
2	2	143,823	21970	21728	7391	6,9625	6607,2
2	4	113,804	28346	21793	9375	11,1375	4894,9

5.5. Modular Exponentiator Design Comparison

In addition to modular multiplier designs, modular exponentiation designs are also investigated in [26]. The paper [26] investigates, proposes, and compares different modular exponentiations. We again compare our design in Table 5.9 with the designs

proposed and investigated in [26]. MaslePHd in the paper column of Table 5.9 corresponds to [26].

Table 5.10. 1024-bit modular exponentiaton design comparison.

Paper	Configuration		Clock Frequency (MHz)	LUT	Latency (mS)	Latency x Area
	PS	CC				
MaslePHd	8	5	95,37	157674	0,750	1,0907
This Work	2	4	113,804	28346	4,895	1,2797
This Work	2	2	143,823	21970	6,607	1,3389
MaslePHd	2	2	88,95	29940	6,570	1,8143
MaslePHd	2	1	138,68	23791	8,000	1,7555
This Work	1	2	149,031	13327	8,135	**1,0000**
MaslePHd	1	2	102,7	20695	10,290	1,9641

When we compare our results, most of the configurations perform similarly. We use $Latency \times Area$ as a figure of merit, as proposed by [26]. As can be seen from Table 5.10, our one pipeline two combinational stages design has the best value.

5.6. Primality Tests

Results belonging to the Miller-Rabin and Lucas primality test designs are discussed in this section. The pipeline stages are chosen as at most two so that the pipeline stays busy, as explained in Section 3.4.

The synthesis and implementation results for the primality tests are presented in Table 5.11. The operand widths are 1024 bits, and the target FPGA is xc7a200tsbg484-1. The PS and CC in Table 5.11 correspond to “Pipeline Stages” and “Combinational Cascading” respectively.

When the results given in Table 5.11 are investigated, we can see that the operating frequencies do not change much even though the pipeline and combinational stages change, unlike the modular multiplier and exponentiator modules. Unchanged

operating frequencies show that the critical path of the primality test modules does not come from the modular multiplier. Instead, the post-implementation timing reports confirm that the critical path comes from the Jacobi calculator module for the Lucas test and a decrementer for the Miller-Rabin test. Therefore, we marked these critical paths as future work.

Table 5.11. Synthesis and implementation (post-route) of 1024-bit Miller-Rabin and Lucas primality tests for xc7a200tsbg484-1 FPGA.

Synthesis							
Module	Configuration		Clock Frequency (MHz)(Imp)	LUT	FF	Slice	Power (mW/MHz)
	PS	CC					
MillerRabin	1	1	108,684	22391	21734	x	2,430
MillerRabin	1	2	108,684	23910	21718	x	3,693
MillerRabin	1	4	108,684	28066	21781	x	6,083
MillerRabin	2	1	108,684	29650	26885	x	4,385
MillerRabin	2	2	108,684	32191	26885	x	2,588
MillerRabin	2	4	109,878	38851	26897	x	10,995
Lucas	1	1	53,342	30542	25831	x	4,190
Lucas	1	2	53,342	32571	25814	x	5,253
Lucas	1	4	53,342	36711	25860	x	7,110
Lucas	2	1	53,342	37771	30980	x	5,923
Lucas	2	2	53,342	40855	30982	x	7,758
Lucas	2	4	53,342	47516	31034	x	11,300
Implementation (Post Route)							
Module	Configuration		Clock Frequency (MHz)	LUT	FF	Slice	Power (mW/MHz)
	PS	CC					
MillerRabin	1	1	89,799	22501	21838	7211	2,630
MillerRabin	1	2	94,233	24618	21833	7774	3,758
MillerRabin	1	4	100,898	28744	21864	9560	6,078
MillerRabin	2	1	95,456	29736	26989	10062	4,655
MillerRabin	2	2	100,786	32821	27002	11060	6,850
MillerRabin	2	4	101,616	39461	27004	12094	11,485
Lucas	1	1	46,999	31119	25899	10611	4,530
Lucas	1	2	47,293	33158	25905	10860	5,770
Lucas	1	4	47,708	37283	25938	11474	7,828
Lucas	2	1	47,646	38145	31052	12606	6,195
Lucas	2	2	47,037	41331	31061	12977	8,560
Lucas	2	4	46,970	48095	31116	15075	12,355

5.7. Prime Number Generator

Implementation results of the prime number generator (PNG), which takes random numbers and subjects to primality tests, are shown in Table 5.12. The table shows both synthesis and implementation (post route) results for 1024 bits numbers. Moreover, the target FPGA isxc7a200tsbg484-1, and the PS and CC in Table 5.12 correspond to "Pipeline Stages" and "Combinational Cascading" respectively.

The PNG design consists of a Lucas primality state machine, a Miller-Rabin primality state machine, a modular exponentiation state machine, a Montgomery multiplier, and a pipelined adder as described in Chapter 3.

Table 5.12. Synthesis and implementation (post-route) of 1024-bit prime number generator for xc7a200tsbg484-1 FPGA

Synthesis						
Configuration		Clock Frequency (MHz)	LUT	FF	Slice	Power (mW/MHz)
PS	CC					
4	1	53,342	74042	70969	x	10,61
4	2	53,342	80222	71001	x	14,73
4	4	53,342	93509	71087	x	23,10
4	8	53,342	127494	71330	x	41,81
Implementation (Post Route)						
Configuration		Clock Frequency (MHz)	LUT	FF	Slice	Power (mW/MHz)
PS	CC					
4	1	44,547	74250	71190	22246	11,43
4	2	47,452	80840	71243	27960	16,41
4	4	47,831	94260	71351	28993	26,22
4	8	46,953	127712	71615	33274	44,76

When we investigate the results in Table 5.12 closely, we can see that we have similar behavior to the results shown in Table 5.11. The reason is simple. The PNG uses the same Lucas module as the one in Section 5.6. Therefore, the critical path is the same. The optimized PNG design is also marked as future work.

6. CONCLUSION AND FUTURE WORK

The Internet of Things (IoT) connects physical devices, vehicles, buildings, and other objects to the internet, allowing for greater efficiency and automation in various applications. However, as the number of connected devices increases, so does the need for secure communication and data storage to protect the privacy and integrity of the data. Public-key cryptography, which uses a pair of mathematically-related keys to encrypt and decrypt data, is considered more suitable for IoTs and is increasingly used in IoT applications. Secure hardware design and ASIC design play a vital role in addressing these concerns by creating tamper-resistant devices resistant to physical and logical attacks while also ensuring energy efficiency. Additionally, open-source cryptography algorithms and specialized hardware or hardware accelerators are essential for secure communication in IoT devices.

In this thesis, we designed an SoC for IoT devices to efficiently generate random prime numbers. Our design methodology utilizes standard specifications and test suites, allowing us to create more secure and compatible hardware. Moreover, parametrized module design allows flexible and scalable hardware design. We investigate different pipelining and source-sharing configurations allowing us to observe the tradeoffs between area, latency, and bandwidth. Analyzing different configurations is especially useful since IoT is a broad term, including both small and low-power hardware and performance-intensive hardware. In addition, we follow a design methodology that allows us to interact with software which is essential for efficient HW-SW co-design.

This thesis' contributions can be summarized as follows:

- We present a new true random number generator by combining and improving wake-up and shut-down methods. The proposed RNG offers 100% more throughput, up to 44% lower area, and up to 41% fewer resources compared to the aforementioned methods.

- We present a new fully digital duty cycle correction method for high-speed irregular sampling-based TRNG designs. The application of the proposed circuit is not limited to the presented RNG. Applying the proposed duty cycle correction method to the presented wake-up & shut-down RNG improves the throughput by 12.3 times. Moreover, the application of the proposed circuit is not limited to the presented RNG. It can be integrated into any RNG design based on irregular sampling method.
- We present a unique clone system to experimentally uncover the security vulnerabilities of an FPGA-based hyperchaotic "true" random number generator. Auto-synchronization is accomplished by only monitoring one of the state variables. Therefore, the next output of the target RNG is precisely predicted, and the clone system identically obtains the entire bit stream.
- A parametrized Montgomery modular multiplier is designed. Twenty different configurations of the Montgomery multiplier are implemented and analyzed.
- The Montgomery multiplier is used as the baseline for modular exponentiation, Miller-Rabin primality, and Lucas primality designs. The designs consequently turn into a prime number generator. We obtain a random prime number generator by combining the RNG designs with the prime number generator.
- We design a comprehensive and parametrized SoC capable of generating random prime numbers and utilizing them in software in an efficient manner.
- Ultimately, we propose and implement an SoC containing the designed RPNG and open-source Risc-V Ibex core.

As a future work, the Miller-Rabin and Lucas algorithms will be investigated further for optimum frequency, power, and area. Moreover, since there is a lot of pipelining and source sharing, there is a significant amount of redundant registers. Therefore, the redundant storage will be optimized. Moreover, the number of NIST SP 800-22 tests implemented in hardware will be increased. Furthermore, word-based Montgomery multipliers will be investigated thoroughly. The word-based approach allows us to design more flexible and scalable structures. On top of that, instead of using a large number of flip-flops, SRAM-based (or BRAM for FPGAs) storages will

be investigated. Since FFs are large, they consume routing resources more, resulting in higher net delays. Finally, countermeasures for side-channel analysis will be investigated. Finally, all modules will be implemented for ASIC, and the results will be analyzed.

REFERENCES

1. Kerry, C. F. and P. D. Gallagher, *Digital Signature Standard (DSS)*, National Institute of Standards and Technology (NIST), http://dx.doi.org/10.6028/NIST.FIPS.186-4, 2013.

2. Kaysici, H. İ. and S. Ergün, "A Low Area Random Number Generator Based on Stability Changes of Ring Oscillators", *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Lansing, MI, USA, 2021.

3. Sunar, B. and D. Schellekens, "Random Number Generators for Integrated Circuits and FPGAs", *Secure Integrated Circuits and Systems*, pp. 107–124, 2010.

4. Ergün, S., "Compensated True Random Number Generator Based on a Double-Scroll Attractor", *Proceedings International Symposium on Nonlinear Theory and its Applications, Sept.*, pp. 391–394, Bologna, Italy, 2006.

5. Kaysici, H. İ. and S. Ergün, "Duty Cycle Correction Circuit and Its Application for High Speed Random Number Generation", *IEEE International Symposium on Circuits and Systems (ISCAS)*, Daegu, Korea, 2021.

6. Banks, D., E. Barker, L. Bassham, J. Dray, N. Heckert, S. Leigh, M. Levenson, J. Nechvatal, A. Rukhin, M. Smid, J. Soto, M. Vangel and S. Vo, *Sp 800-22 Revision 1a. A statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, National Institute of Standards & Technology, Gaithersburg, MD, 2010.

7. Martin, K. M., *Everyday Cryptography: Fundamental Principles and Applications*, Oxford University Press Inc., New York, USA, 2012.

8. Göv, N. C., M. K. Mıhçak and S. Ergün, "True Random Number Generation Via Sampling From Flat Band-Limited Gaussian Processes", *IEEE Transactions on*

Circuits and Systems I: Regular Papers, Vol. 58, No. 5, pp. 1044–1051, 2010.

9. Xu, X. and Y. Wang, "High Speed True Random Number Generator Based on FPGA", *International Conference on Information Systems Engineering (ICISE)*, Los Angeles, CA, USA, 2016.

10. Demir, K. and S. Ergun, "Random Number Generators Based on Irregular Sampling and Fibonacci–Galois Ring Oscillators", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 66, No. 10, pp. 1718–1722, 2019.

11. Varchola, M., "New Method of Randomness Extraction Based on a Modified Ring Oscillator for Cryptographic Trngs Embedded in Fpgas", 2009, `http://www.varchola.com/embedded/src/pdf/Var_FPL_PHD_forum.pdf`, accessed on January 1, 2023.

12. Nakura, T., M. Ikeda and K. Asada, "Ring Oscillator Based Random Number Generator Utilizing Wake-up Time Uncertainty", *IEEE Asian Solid-State Circuits Conference*, Taipei, Taiwan, 2009.

13. Wold, K. and C. H. Tan, "Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings", *International Journal of Reconfigurable Computing*, Vol. 2009, No. 4, pp. 1–8, 2009.

14. Menezes, A. J., P. C. V. Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL, USA, 1996.

15. Tenca, A. F. and Çetin K. Koç, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm", *IEEE Transactions on Computers*, Vol. 52, No. 9, pp. 1215–1221, 2003.

16. Low-Risc, "Ibex RISC-V Core", `https://github.com/lowRISC/ibex`, accessed on January 12, 2023.

17. Sunar, B., W. J. Martin and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks", *IEEE Transactions on Computers*, Vol. 56, No. 1, pp. 109–119, 2006.

18. Şarkışla, M. A. and S. Ergün, "An Area Efficient True Random Number Generator Based on Modified Ring Oscillators", *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Chengdu, China, 2018.

19. Kaysici, H. İ. and S. Ergün, "Random Number Generator Based on Metastabilities of Ring Oscillators and Irregular Sampling", *27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Glasgow, UK, 2020.

20. Acar, B. and S. Ergün, "A Digital Random Number Generator Based on Irregular Sampling of Regular Waveform", *IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS)*, Armenia, Colombia, 2019.

21. Ergün, S., U. Guler and K. Asada, "A High Speed IC Truly Random Number Generator Based on Chaotic Sampling of Regular Waveform", *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E94-A, No. 1, pp. 180–190, 2011.

22. Montgomery, P. L., "Modular Multiplication Without Trial Division", *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521, 1985.

23. Masle, A. L., W. Luk, J. Eldredge and K. Carver, "Parametric Encryption Hardware Design", *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium on Applied Reconfigurable Computing (ARC)*, Bangkok, Thailand, 2010.

24. Masle, A. L., *Parametric Encryption Hardware Design*, M.S. Thesis, Imperial College London, 2009.

25. Davide Schiavone, P., F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand and

L. Benini, "Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications", *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Thessaloniki, Greece, 2017.

26. Masle, A. L., *Reconfigurable Architectures for Cryptographic Systems*, Ph.D. Thesis, Imperial College London, 2013.

27. Amanor, D. N., C. Paar, J. Pelzl, V. Bunimov and M. Schimmler, "Efficient Hardware Architectures for Modular Multiplication on FPGAs", *International Conference on Field Programmable Logic and Applications*, Tampere, Finland, 2005.

28. Daly, A. and W. Marnane, "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *Proceedings of the 10th International Symposium on Field-Programmable Gate Arrays (SIGDA)*, Monterey, CA, USA, 2002.

29. Mclvor, C., M. McLoone and J. McCanny, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures", *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, Pacific Grove, CA, USA, 2003.

30. Oksuzoglu, E. and E. Savas, "Parametric, Secure and Compact Implementation of RSA on FPGA", *International Conference on Reconfigurable Computing and FPGAs*, Cancun, Mexico, 2008.

31. Tang, S., K. Tsui and P. H. W. Leong, "Modular Exponentiation Using Parallel Multipliers", *Proceedings IEEE International Conference on Field-Programmable Technology (FPT)*, Tokyo, Japan, 2003.