5-2019

Design and Verification of an RSA Encryption Core

Gowtham Ramakrishnan
gr3249@rit.edu

DESIGN AND VERIFICATION OF AN RSA ENCRYPTION CORE

by
Gowtham Ramakrishnan

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 2019

I would like to thank all my family and friends for their support throughout my career.

Abstract

Cryptoprocessors are becoming a standard to make the data-usage more discrete. A well-known elector-mechanical cipher machine called the “enigma machine” was used in early 20th century to encrypt all confidential military and diplomatic information. With the advent of microprocessors in late 20th century the world of cryptography revolutionized. A cryptosystem is system on chip which contains cryptography algorithms used for encryption and decryption of data. These cryptoprocessors are used in ATM’s and highly portable communication systems. Encryption and decryption are the fundamental processes behind any cryptosystem. There are many encryption and decryption algorithms available; one such algorithm is known as the RSA (Rivest-Shamir-Adlean) algorithm. This project focuses on development of an encryption cryptoprocessor which will deal with key generation, key distribution, and encryption parts of the RSA algorithm and also discusses the verification environment required to verify this core.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Gowtham Ramakrishnan

May 2019

Acknowledgements

I would like to thank my advisor Prof.Mark A Indovina for his support, guidance, feedback and encouragement which helped in the successful completion of my graduate research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The information has been and will be one of the treasured resources since the dawn of civilization. In the last 20 years, many economies have changed from manufacturing and heavy industry to knowledge and data. This has emphasized the importance of protecting the information through cryptography. Cryptography is an art of secret writing, the first documented application of cryptography is traced back to 1900 BC when Egyptians used hieroglyphs. The widespread of cryptography came into existence after the development of computers. Everything surrounding us from our smartphone to internet banking heavily uses cryptography. The main functions of modern-day cryptography systems are

1. **Privacy**: To make sure no third party can access or read the information.

2. **Authentication**: When we use a correct cryptographic system, we establish a secured communication between remote user and host. The best example for this is the SSL certificate feature of web server. Once we connect to that server, it will give the user a proof that they are connected to the correct server. Here the identity of the user is not the question but the cryptographic key used by that user. Meaning, assuming the keys are secured to the user, it is okay to assume that the user is the correct sender. In the case

of highly valuable data, the data are encrypted with a private key and then with a public key.

3. **Integrity**: The cryptosystems should make sure that the receiver has received the message in the same form of the original. For example, some cryptography systems ensure that rivaling companies cannot tamper with their internal data. The standard way to achieve this data integrity is through "hashing."

4. **Key exchange**: The process by which encryption keys are distributed between senders and receivers.

One crucial process of cryptography is encryption and decryption. The process of encoding a standard data also referred to as "plain-text" in a way such that only designated end users can gain access to the information contained in it. The encoded data is also referred to as "cipher-text." Decryption is the process of converting back a cipher text to a plain-text. The main ingredient of this process is the "key."

A key is a variable or a small piece of information which determines the output of a cryptographic algorithm. Key can be created in various ways, but the underlining concept is, it should be a random number, and it should not be easily predicted. Most commonly known key generators are pseudo-random number generator and random-number generator. Random number generators create random number automatically while the pseudo-random number generator needs a seed value. These seed values can be changed to create a new set of random number. One disadvantage of this pseudo-random generator is the ability to recreate the random number if the seed value is known. Research is going on to develop an utterly random number based on autonomously occurring events like electron spin.

One important thing to note about the key concept is the "key length." It determines actually how many combinations of random numbers are possible. The total possible combinations can be determined using this formula $2^{(keylength)}$. For example, if a key is 20 bit wide, then 2^{20} combinations are possible which is around a million combinations. Even though this is a large number, the key is still insecure given the ability of the modern day computers to crack it in minutes.

Development of processor chips and need to improve the data security has given rise to cryptoprocessor. A cryptoprocessor is a dedicated system on chip which is used for carrying out cryptographic algorithms. Some other functions of cryptoprocessors are encryption accelerators, intrusion detection, and tamper resistance enhanced key and data protection and enhanced secured memory access. The first application of cryptoprocessors was during the 1980s when IBM's 3848 model was released. This cryptoprocessor was used in the ATM, and it was the first financially successful ATM for safe banking. Some cryptoprocessors are used as hardware between the bus interface and cache or memory. The data is decrypted on the fly while accessing from the RAM or cache and encrypted while writing back into the RAM or cache

Following phases are carried out for designing a cryptoprocessor.

- Hardware design
- Application design
- Security design
- Software design

The following section will be discussing more regarding the hardware design phase carried out for this project.

1.1 Research Goals

There are many encryption and decryption algorithms available in cryptography, and one such algorithm is called the RSA (Rivest-Shamir-Adleman) algorithm. This is an asymmetric algorithm were the public-key is different from the private key. It contains the following stages:

- Key generation
- Key distribution
- Encryption
- Decryption

This project will be focusing on the development of an RSA cryptoprocessor which will only deal with key generation, key distribution and encryption parst of the algorithm. This is can be achieved through the following stages:

- A software model of the same must be designed for comparing the hardware results with the software results. This software model will be running concurrently with the hardware model.
- The verification environment must be provided, which will act as a platform to compare the results of hardware model with that of the software model.

1.2 Organization

- Chapter 1: Brief introduction to cryptography and research goals.
- Chapter 2: Will discuss basic concepts behind cryptographic processors.

- Chapter 3: Will discuss the RSA algorithm and basics of modular exponentiation techniques.
- Chapter 4: Will discuss the software and hardware implementation of RSA encryption core.
- Chapter 5: Will discuss the verification strategy and test bench components and software model required for verifying the hardware model.
- Chapter 6: Will discuss the results of this work
- Chapter 7: Conclusion

Chapter 2

Bibliographical Research

The basics of Cryptography as discussed in [2] are briefly explained below

One of the key aspects behind hardware design is their computational power and ability to reconfigure the design during the latter part of the design stage. A trade-off must be achieved between strength, performance, security, and flexibility to get the right product. The cryptographic hardware is classified into three types:

1. **Custom Cryptoprocessor**: A custom cryptoprocessor includes are a standalone functional unit which performs the cryptographic algorithms. This functional unit should be able to perform either Symmetric operations or Asymmetric operations. These processors have dedicated arithmetic logic units (ALUs) for this purpose which run concurrently with the program execution. Most of these processors have two channels: one for secure I/O, and one clear (plain text) I/O. There must be precise isolation between these two channels to defend the hardware against non-invasive attacks like power analysis and electromagnetic analysis. The data is decrypted inside the functional unit and is stored into the RAM or cache using a secure I/O channel. These processors have a crucial dedicated management unit which is used for key generation and securing the key.

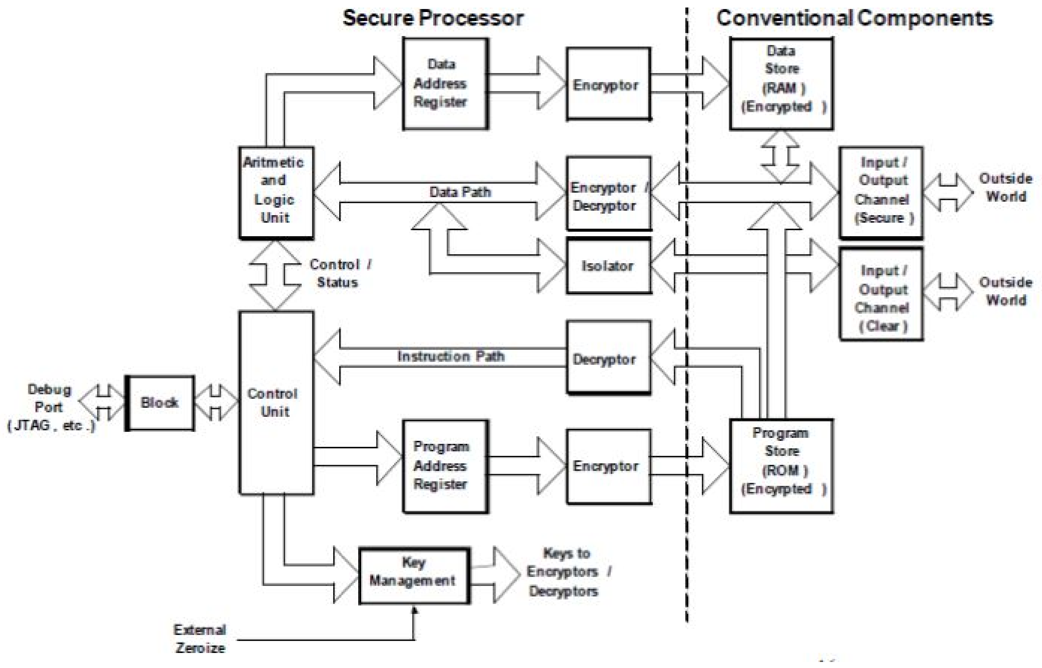


Figure 2.1: Custom Cryptoprocessor [1]

The keys are sometimes generated externally and fed to the hardware or hardwired in the system. Some cryptoprocessors store keys in their memory which is a significant disadvantage because this makes the processor more vulnerable to software attacks. Since these processors are flexible, it is easy to configure the instruction set of the hardware when such attacks happen. Some processors are extremely fast because they use internal data path to the internal memory and do not have to use external memory. Therefore a custom cryptoprocessors increases security, performance, flexibility at the expense of computational power. Figure 2.1 shows an example of custom cryptoprocessor.

2. **Crypto-Coprocessor**: These are hardware modules which are present in concurrent to the main processor. Many of these co-processors also perform the same task done by the custom cryptoprocessor, but they do not have the same efficiency as their counterparts. Since they are integrated into an SoC, they do not have high flexibility and

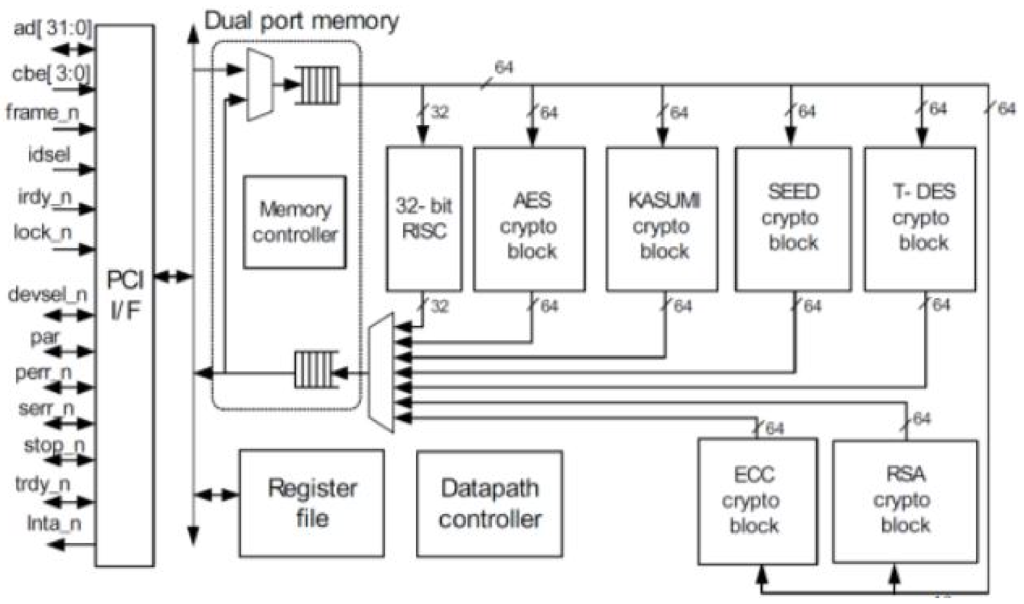


Figure 2.2: Cyrpto-Coprocessor [1]

cannot be reconfigured. But some of the modern day processors can be configured by using a host processor. Another advantage of such systems is multiple dedicated cryptoprocessors can be attached to the same host processor. This allows the use of various cryptographic applications. Parallel computing and pipeline architecture have made it possible to perform multiple cryptographic applications simultaneously. In [3] the proposed co-processor utilizes different characteristics of cryptographic algorithms and optimizes the size of the core and performance. The co-processor implemented in this system is a High-speed functional unit which includes two hash functions SHA-1 (Secure Hash Algorithm 1), SHA-2 (Secure Hash Algorithm 2), and AES (Advanced Encryption Standard). The main idea behind the use of the proposed design is to share the available resources. This coprocessor is configurable by host core and architecture allows parallel computation of three cryptographic primitives. Figure 2.2 shows an example of Crypto-Coprocessors.

3. **Crypto-arrays**: These are developed recently, and they have many advantages in terms of both performance and flexibility.

The common item in the above-listed cryptosystems is encryption and decryption. Note that encryption algorithms are classified into two families based on the type of key they use to encrypt and decrypt the data: symmetric or secret key, and asymmetric or public key.

2.1 Symmetric key

This is the oldest type of encryption algorithm and used for data protection for a long time. The systems employing this algorithm need a secure way to transfer the private key to the concerned party. If there is an interception in the transfer, the algorithm can be easily cracked, and the entire system will be easily compromised. Thus while designing the symmetric encryption algorithm, it is vital to evaluate a secure way to transfer the key. Fortunately, this necessity has given rise to many ingenious methods to transfer the key. One way is to use a key to encrypt a data and send the data to the receiver; the receiver decrypts the data and destroys the key. This way the algorithm can never be broken, and it has been proved mathematically. Such systems are known as a one-time pad. So, Naturally, a lot number of keys have to generate to transfer data through one-off pad systems. Still there exists a question of securely transferring the key and such method is not commercially viable.

Symmetric encryption is used for encrypting the bulk of data together. Two main techniques of symmetric encryption are substitution and transposition. Substitution ciphers are basic encryption technique where another aspect replaces a character. For example, the letter “G” will be replaced by “A” wherever it occurs. These ciphers are not used much often today due to the obvious fact that it can be easily broken with regressive cryptanalysis. In contrast, transposition ciphers cannot be easily broken. In this, the text or part of the data is moved to a different location in the cipher-text rather than replacing each letter. A simple example of transposition is “word jumble” which comes in daily newspapers. But the word jumble is an example of random transposition that is the word is moved to a random location, but in the transposition based encryption, the words are moved in definite pattern, which can be easily reversed and decrypted. Even pure transposition based encryption is rarely used today because of the ease of breaking it using computer-based cryptanalysis.

So researchers came up with a way to encrypt data using both of the above methods. All modern day symmetric key encryption works on the diffusion of these two principles. Diffusion algorithms that are being used today substitute different letter for the same letter based on its location and data which is preceding it.

Most secret keys today have a property called Avalanche effect, in which a single bit change will result in a change of one-half of cipher-text bits. Symmetric encryption is fast and compact concerning code and memory utilization, which is an essential property since most of the modern encryptors should be compatible with smartphones which have a smaller, low power processor.

Symmetric algorithms can be further classified as stream ciphers and block ciphers. Block cipher algorithms encrypt and decrypt a block size of data, whereas stream ciphers continuously encrypt data of any sized that is available at its input.

The following are secret vital algorithms that are commonly used today.

- **Data Encryption Standard (DES)**: This is a standardized encryption algorithm which is approved by the US government in 1977 after years of analysis. The origin of this algorithm can be traced back to an encryption method called “Lucifer” invented by IBM. It uses a 64-bit key which includes additional parity bits. DES uses a block cipher technique and encrypts 64-bit data. Although the algorithm was difficult to crack when it was first introduced, now due to improvements in cryptanalysis and brute force methods it has become insecure now. So the government agencies have considered using longer keys to prevent penetration. Despite all these, it is still known as most crypts analyzed algorithm in the world withstanding all attacks.

- **RC4 (Rivest Cipher 4)**: This is a stream cipher published in 1987. The disadvantage of these techniques is the generation of the key-stream, which is a potentially long sequence of values consisting of 40 bit to 128-bit key and a 24-bit initialization vector. But the encryption is straightforward, the actual plain-text is xor'd with the key-stream. The key-stream is replicable if the IV is known. In practice, both the sender and receiver will be generating the same keystream, and this technique is ten times faster than DES.

- **RC5 (Rivest Cipher 4)**: This is a block cipher algorithm, published in 1994, which uses variable block size, key size, encryption steps.

- **Advanced Encryption Standard (AES)**: AES is a block cipher created by two Belgian cryptographers Vincent Rijmen and Joan Daemen. The National Institute of Standards and Technology (NIST) replaced DES with AES after any analysis. The US government updated the cipher standards from DES to AES in 2002, and since then it has become a commercially accessible standard for encryption purpose. AES algorithm is available to the public with unrestricted access.

2.2 Asymmetric key or Public key

As per the above discussion, a noteworthy drawback of the secret key encryption is the need for sending the cipher key to the concerned parties. This need has given rise to a key exchange system where the parties can exchange a known variable and which would be then used to derive a secret key. In 1976 Hellman created simple arithmetic which allowed both the parties to be able to obtain the same secret key. This distribution method was known as Diffie-Hellman (DH) key exchange. One inconvenience of DH key exchange is its iterative process for key generation. It requires the participation of both parties to generate a new secret key.

MIT researcher Ronald Rivest, Ari Shamir and Leonard Adleman created a new public key algorithm called the RSA algorithm [4]. For this algorithm, there isn't a need to share the secret keys. When the sender wants to encrypt the data, it can be done by using the receiver's public key. As the name suggests, the public key can be sent freely without any concern. But the private key must be kept safe. To decrypt the data, the receiver will use its private key.

The fundamental difference between symmetric and asymmetric encryption is that while using a secret key there exists many private keys for all the parties that are communicating with the host. Whereas in public key cipher there exists only one key pair between sender and receiver. Any host can send encrypted data to receiver using the receivers public key, but the receiver decides which data should decrypted.

Other advantages of public key encryption is that it supports authentication, i.e the sender needs to be identified first.

2.2.1 RSA public key algorithm

The logic behind the RSA algorithm is simple, factoring the product of any two prime numbers. Here the product of the two prime numbers are known as the public key, and each prime number is the private key. If anyone can factor the prime number, the private key is compromised.

A question might arise why this algorithm isn't used for all encryption given its benefits: the reason is speed. The RSA algorithm is not viable for bulk encryption. The only way to crack the RSA algorithm is by factoring the public key or through brute-force, which is quite hopeless because of the fact that the most public keys are 1024 - 2048 bit long.

Modular multiplication is the most important process of the RSA algorithm, and many papers discuss using different techniques to implement modular multiplication.

[5] discusses the implementation of RSA algorithm using an FPGA. In this system, a 1024-bit public key is used. This system employs multiple and square algorithms for modular operation. The design is described using VHDL and implemented in Xilinx Spartan3 (XC3S50) field-programmable gate array (FPGA) board. This is a single FPGA board and the design consumed only 29% of chip resources with an operating frequency of 68.573 MHz. This confirms that RSA designs can consume less hardware resources than other crypto algorithms.

In any data communication network, security is an important characteristic. As discussed in [6], most public key cryptography, including the RSA cryptography, this characteristic is dependent on modular exponentiation, i.e security lies in the inability to factor out the two large prime numbers. This modular exponentiation is achieved through modular multiplication of two large prime numbers. The main challenge here is to achieve this process in less amount of time and hardware. There are many types of modular multiplication but they generally fall into these two categories. 1) Division-after-multiplication 2) Division during multiplication. In this paper, the authors take advantage of both methods and come up with a new algorithm. The basic idea is to split the operand into many equal-sized segments and perform multiplication and residue operation in each stage of the pipeline.

In [7], the authors have implemented the modular exponentiation by introducing “Vedic Multipliers” into the pipelined system. The authors have concluded by comparing three versions of the same system using different multipliers (Vedic Multiplier, Array Multiplier, Pipelined Multiplier) and have presented a comparative analysis.

When the key size is increased the dedicated hardware accelerators are required to perform heavy computations at a high throughput rate. Two popular methods are available for modular multiplication of large number. One is Montgomery’s multiplication algorithm [8] and the other is multiplication succeeded by modular reduction. [9] discusses implementing Montgomery multiplication for carrying out modular multiplication. Carry-save adders are used to stop carry propagation at each stage. Also, buffers are introduced to synchronize the worst case delay with the clock. A comparative analysis of power consumption and area of this proposed design with other designs are made and significant results are presented.

The authors of [10] use a modified radix-4 multiplier based on Booth's multiplication technique. The difficulty of implementing multiplication and Modular exponentiation depends upon the algorithm which uses these techniques. The Karatsuba implementation and FFT implementation are the fastest multiplication algorithms used in software, But these algorithms are rarely used in hardware application due to their recursive nature. In [11] authors have performed an in-depth analysis of above-mentioned implementation and have proposed an optimized version of these algorithms. These algorithms show that recursive deployment increases efficient implementation for small bit operands s. But the modified algorithms showed improved efficient implementation for systems having 346-bit operands

In the previous section we discussed using re-configurable Crypto-Coprocessors which can be configured by the host CPU. [12] proposes using scalable modules which can be configured by the host system. Here the RSA algorithm is implemented as a co-processor and system contain 6 levels 1) random generation of two prime numbers, 2) multiplication of two prime numbers, 3) decrementing their value, 4) creating encryption key (public key), 5) creating decryption key (secret key), and 6) the encryption and decryption process. And interleaved multiplication algorithm has been included into the modular multiplication to make forward correction more robust. The entire design is scalable to fit into an ALTERA Cyclone IV EP4CE115F29C7 FPGA.

A much more efficient hardware implementation of RSA cryptography is discussed in [13] where the authors have implemented the Miller Rabin algorithm to check primality of the operands and then performing interleaved multiplication. Since RSA is an computationally intensive algorithm, a cryptographic accelerator can be embedded as a co-processor and share the computation process with the main host. The final design is an 8-bit RSA circuit but it is a fully parameterized design which can be extended to 1024 / 2048 bit implementation.

There is research on-going to improve the throughput and minimize the power for RSA cryptoprocessors. [14] introduce the following implementation as an improvement:

1. A fast half-carry-save Montgomery modular multiplication algorithm.

2. A high-speed dual-core multiplier accumulator to optimize critical path.

3. And several other low power optimizations for RSA co-processor.

The previous papers focused more on performance and fast processing of the system. This paper focuses on improving power relation as well.

As discussed earlier, modular multiplication forms the heart of RSA encryption. Its security and reliability depend on whether the modulus can be factored out or not. Nowadays the modulus is of 2048 bits size which provides good security to a certain level. As more powerful computers emerge, the key size of 2048 bits will be soon insufficient. So it is necessary to increase the key size to 4096 or to even 8192 bits to provide security. [15] presents the design of 8192-bit RSA cryptoprocessor. This design uses radix-2 Montgomery multiplier which is designed as a systolic architecture. This hardware was verified by carrying out two exhaustive tests. One performs encryption and decryption of a 2048-bit message using standard test vectors recommended by RSA laboratories

The performance of the RSA system entirely depends upon the throughput of the modular multiplication model. When the key size becomes larger, high throughput hardware accelerators are needed. The Fast Fourier transform based Strassen algorithm is generally sufficient for large number multiplication. In [16] the authors propose a novel approach by combining

FFT-base multiplication and Montgomery reduction. The authors have implemented 8k bit, 12k bit, and 48k bit RSA cryptosystems and their performance is compared with their previous work. Their results show that their 48k-bit RSA design outperforms its counterpart with respect to throughput and efficiency.

Implementation of RSA algorithm or any public key cryptosystems in a general purpose processor (GPP) is very flexible because of the ability to drive many cryptosystems at run-time. One disadvantage of GPP is they result in low throughput and consume more power. For most real-time purposes dedicated hardware is required to increase the computation power of cryptosystems. So to fill the gap between dedicated hardware and GPP, a novel approach has been discussed in [17] where domain-specific reconfiguration is utilized to provide a highly flexible system. The resulting system is capable of performing cryptographic primitives, with a fully re-programmable modulus. In [18], a domain-specific reconfiguration is further explored to improve operating speed and hardware utilization. In particular, algorithms sharing the same resource for different arithmetic operations are explored to reduce hardware utilization. Experimental results reveal that users are able to program crypt-processor in microcode sequences for cryptographic algorithms like RSA and elliptic curve cryptosystems.

In 1999, "Tenac" and "Koc" proposed a new algorithm for matrix multiplication called Multiple-word Radix-2 Montgomery multiplication [19]. This system optimized for minimum latency performed Montgomery multiplication in $2n$ clock cycles where n is the size of the operand. In [20] the authors propose a two hardware architecture which was able to perform the same operation in n clock cycles. They achieved this by precomputing partial results of the operation and arriving at two best possible solutions.

The previous section discussed 4 important steps in any cryptographic processors: 1) key generation, 2) key distribution, 3) Encryption, and 4) Decryption. The generation of a prime number is the basic task of public-key schemes and essentially needed for the creation of key pairs. Despite years of investigation of primality testing, prime number generation algorithms are still scarcely researched upon. In [21] authors have proposed new algorithms for generating pseudo-random numbers. This chapter is the basis for all prime number generation algorithms. This paper shows a way to reduce the value of hidden constants and provide an efficient way to generate prime numbers. The problem of validating prime number has posed a great challenge. Differentiating between prime and composite numbers is the most important problem in arithmetic. It is easy to multiply two prime numbers but factoring them is difficult. FPGA can be used as a platform for validating any digital systems. In [22] authors propose a scalable architecture for validating the prime numbers by using re-configurable FPGAs. In this paper, the authors use the Rabin-Miller primality test for validating the prime number. They also use the Montgomery algorithm for modular multiplication. The system was designed and implemented on Spartan XC3S2000-4-FG900 chip. Since the design is scalable it can also be implemented on smaller and bigger FPGAs.

In [23] authors have proposed an efficient solution to accelerate key pair generation process by using a residue number system. Transport Triggered Architecture (TTA) is the type of a processor design where external programs control the internal buses of a processor. For example writing data into a data bus will actually trigger the functional unit of the processor to initiate the computational process. TTA hardware is proposed where the functional unit is a residue number system which will generate a pair of keys. Whereas in [24], the key generation process is based on selecting a random public key from the predefined list of public keys then using Euclid's extended algorithm to derive at the private key. The Euclidean method is the best way to find the Greatest common divisor of a two natural number a and b.

Modular multiplication on large operands makes RSA computation more challenging. The software designs of such systems give more flexibility to the designer but lacked performance. On the other hand, the hardware model gave high performance but lacked flexibility. In [25] the authors have presented a hardware/software co-design of the RSA encryption core. A Zynq-7000 SoC platform is adopted for this purpose, which has a Dual ARM core as the main processing system. A comparative analysis of hardware/software design with that of a complete software model was done, which proved that a speedup of 57 times is achieved for a 2048-bit operand system. To achieve this task a proper partitioning between hardware and software process of the algorithm must be made.

The main motivation behind [26] is to design an efficient hardware algorithm for the RSA encryption/decryption algorithm using Montgomery multiplication. FPGA nowadays have embedded DSP blocks and block RAMs (BRAMs). This paper presents the implementation of RSA encryption with just one DSP block and one BRAM thereby reducing logic units. The multiplier inside the DSP clocks for more 97% of the clock cycles for all clock cycles, and key size ranges from 64-bit to 2048-bit can be applied to the same architectural design. The design runs at 447.027 Mhz. This hardware implementation is close to optimal, in other words, the DSP works on almost all processing clock and can be executed in parallel to obtain high throughput.

Chapter 3

RSA (Rivest–Shamir–Adleman) Algorithm

This chapter will discuss about the architecture of the RSA_CORE. Section 3.1 will start with basics of RSA encryption with a simple example.

3.1 RSA Encryption and Decryption

Securing Information has become a huge factor with development of modern communication networks. Having a small loophole will lead to devastating effects. There are two types of encryption methods used by modern computers for encryption and decryption of messages: symmetric and asymmetric encryption. The former encryption method uses same cryptographic keys for both encryption and decryption of plain text. A cryptographic key is a string of random data used by cryptographic algorithms to convert a plain text to cipher text. Here the involved parties share the key, passphrase. Whereas the latter method uses a public key for encrypting the plain text and private key for decryption of the cipher text. RSA is a type of asymmetric

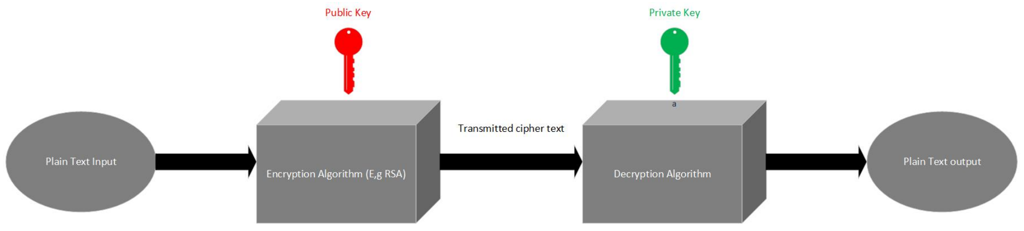


Figure 3.1: Asymmetric Cipher block

encryption algorithm developed in the year 1978 and is still used today. The security of this encryption method depends upon the difficulty to factor large prime numbers. Recent times computers can factor large prime numbers with size over 200 digits. Nevertheless these large prime numbers require reasonable time duration to crack it. For example, a test conducted in 2005 took nearly 1.5 years to factor a 200 digit number. The figure3.1 represent a Asymmetric cipher model.

3.2 RSA algorithm for Public and Private Key pair.

3.2.1 Key generation

The main components required for the RSA algorithm are: Message, Public key, and Private Key.

- A public key is made up of two components, a modulus and exponent value. Let's call these n and e respectively.
- A private key is also made up of two components, a modulus and exponent value. Let's call that n and d.

The first step of the RSA Algorithm is two generate two prime numbers p and q. The product of these numbers is the modulus n. The modulus value will be exposed in the encryption and decryption key, but the value p and q will not be revealed explicitly. p and q should be large enough so that it is difficult to derive from n. Next a Euler's totient function of n is calculated using the formula $\Phi[n] = (p-1)*(q-1)$. The value of e is randomly chosen such that $1 < e < \Phi[n]$ where e and $\Phi[n]$ are relatively prime. Then the d value of the private key component is calculated using the relation $d*e = 1(mod(\Phi[n]))$. In other words, the value d is calculated such that $d*e-1$ can be evenly divided by $\Phi[n]$.

Once complete, we have the Public key: (e,n), and Private Key: (d,n).

The following remarks can be made as far as operands size are concerned.

- The prime numbers p and q should be selected in such a way to make sure that it is computationally infeasible to factor them. A rule of thumb is to have these primes approximately of same bit length. For example, p and q should be around 512 bits long each for a modulus n, of size 1024bits.

- The value of exponent is usually a small number , in order to increase the efficiency of the exponentiation.

3.2.2 Encryption

Let "m" be the message that is to be encrypted using public key. The following relation shows how an encrypted cipher text is calculated.

$C = m^e(mod(n))$

3.2.3 Decryption

The decrypted message from the cipher text can be obtained from following relation.

$m = C^d(mod(n))$

3.3 Modular Exponentiation

The equations 3.2.2 and 3.2.3 are known as modular exponentiation. There are many methods using which the above two relations can be realized. The performance of any asymmetric encryption and decryption algorithm primarily depends upon the efficiency of the modular exponentiation. The following section will cover in brief about the available methods.

3.3.1 Direct Method

This is the most direct way of calculating the modular exponent value. It calculates the resultant power value by recursive multiplication and taking the modulus of this value at the end. For example, given m = 4 , e = 13 , and n = 497.

$C = 4^{13}(mod(497))$

computing 4^{13} would result in to $67,108,854$. Taking modulo 497 with this value will result in to 445 (C = 445).

The size of the modulus is as same as that of the resultant 8 digits. A strong cryptography method requires modulus value at least 1024 bit long. These calculations are possible on modern computers , but the it would cause the speed of the calculation considerably slow.

3.3.2 Memory efficient method

This method is similar to Direct method 3.3.1 but require more operations compared to the direct method, because the memory operation is less. This method makes use of following relativity conditions.

- $c(mod(n)) = (x*y)(mod(n))$
- $c(mod(n)) = [x(mod(n))*y(mod(n))]mod(n)$

The algorithm for this method is as follows,

1. Initialize $e^{'} = 0$ and $c = 1$;
2. Increment $e^{'}$ by 1.
3. Calculate $c = (m * c) mod(n)$
4. if $e^{'} < e$, go to step 2 and repeat the process , else C is the final output.

The example for above algorithm is as follows, let us take the same values used in the previous method so as to compare the final result, i.e $m = 4$, $e = 13$, and $n = 497$.

1. Initialize $e^{'} = 0$, $c = 1$
2. Increment $e^{'} = e^{'} + 1 = 1; c = (4 * 1) * mod(497) = 4$
3. Increment $e^{'} = e^{'} + 1 = 2; c = (4 * 4) * mod(497) = 16$
4. Increment $e^{'} = e^{'} + 1 = 3; c = (4 * 16) * mod(497) = 64$
5. Increment $e^{'} = e^{'} + 1 = 4; c = (4 * 64) * mod(497) = 256$
6. Increment $e^{'} = e^{'} + 1 = 5; c = (4 * 256) * mod(497) = 30$
7. Increment $e^{'} = e^{'} + 1 = 6; c = (4 * 30) * mod(497) = 120$
8. Increment $e^{'} = e^{'} + 1 = 7; c = (4 * 120) * mod(497) = 480$
9. Increment $e^{'} = e^{'} + 1 = 8; c = (4 * 480) * mod(497) = 429$
10. Increment $e^{'} = e^{'} + 1 = 9; c = (4 * 429) * mod(497) = 225$
11. Increment $e^{'} = e^{'} + 1 = 10; c = (4 * 425) * mod(497) = 403$

12. Increment $e^{'} = e^{'} + 1 = 11; c = (4 * 403) * mod(497) = 121$

13. Increment $e^{'} = e^{'} + 1 = 12; c = (4 * 121) * mod(497) = 484$

14. Increment $e^{'} = e^{'} + 1 = 13; c = (4 * 484) * mod(497) = 445$

15. Increment $e^{'} = e^{'} + 1 = 13; e^{'} > e$ so, 445 is the final answer.

Both direct method and this method requires e multiplications to arrive at the final result.

3.3.3 Square and Multiply Algorithm or Binary exponentiation Algorithm

This algorithm is used to find results of a large integers power. i,e when powers are in the range of thousands.This algorithm is done by scanning each bit of the exponent. On each step a exponent bit is scanned and a square operation is done and if and only if the scanned exponent bit value is 1 then a multiply operation is performed. There are two types of this algorithm

- left to right binary method.
- right to left binary method.

The left to right binary methods start from the MSB of the exponent value and traverse to the LSB of the exponent value. In this method after each multiplication the result is reduced (mod n) before proceeding to next step. The following example shows steps to compute b^{13}. The binary representation of 13 is 1101 of size 4 bits. Starting from the MSB of 1101.

1. Initialize the resultant register r with b^0value, i.e 1 in this case

2. Square the resultant , $r = r^2$ ($= b^0$); bit 1 = 1 , so calculate $r = r * b$ ($= b^1$)

3. Square the resultant , $r = r^2$ ($= b^2$); bit 2 = 1 , so calculate $r = r * b$($= b^3$)

4. Square the resultant , $r = r^2$ ($= b^6$); bit 3 = 0 , so no subsequent multiplication

5. Square the resultant , $r = r^2$ ($= b^{12}$); bit 4 = 1 , so calculate $r = r * b(= b^{13})$

The right to left binary method follow the below algorithm, Starting from the least significant bit (LSB) of 1101.

- Step 1) Initialize $R = 1$ and $x = m$
- Step 2) bit $1 = 1$; So compute $R = R * x$; Set $x = x^2$
- Step 3) bit 2 = 0 ; So do not compute anything ; Set $x = x^2$
- Step 4) bit 3 = 1; So compute $R = R * x$; Set $x = x^2$
- Step 5) bit 3 = 1; So compute $R = R * x$

The run time of this method is log e. This algorithm gives better speed benefit compared to the other methods.

3.3.4 Montgomery Multiplication Algorithm

This algorithm is especially used when there is a need to perform recursive modular multiplication by transforming the input variables to other form. This algorithm is inferior to other methods for single multiplication.This algorithm has fast execution time compared to the others. This algorithm is to be used where value of m and e is less than modulus n. Also there is an another integer introduced r and $gcd(r,m) = 1$, where gcd represents calculating the greatest common divisor. Following steps summarize Montgomery multiplication to compute $c = (x * y * mod(n))$.

1. Choose the value r such that $r > n$ and $gcd(r,n) = 1$.

2. $k = \frac{r(r^{-1} modn)-1}{n}$

3. $\bar{x} = (x * r * mod(n))$, $\bar{y} = (y * r * mod(n))$ (Montgomery Input form)

4. $x = \bar{x} * \bar{y}$

5. $s = (x * k * mod(r))$

6. $t = x + sn$

7. $u = \frac{t}{r}$

8. $\bar{c} = u$ if $u < n$ else $u - n$ (Output Montgomery form)

9. $c = (\bar{c}r^{-1} mod(n))$ (Standard Output form)

Explanation :

- This algorithm is used when we have to compute $c = (x * y * mod(n))$. with many values of x and y but with same modulus n.
- The value of r has to be greater n and coprime with n.
- The r^{-1}mod n value exists because r is co -prime with n. This is calculated using extended euclidean algorithm.
- $\bar{x} = (x * r * mod(n))$ and $\bar{y} = (y * r * mod(n))$ converts input number $x$and y to Montgomery form.
- Convert the output Montgomery form $\bar{c}$ to standard form using $c = (\bar{c}r^{-1} mod(n))$.

Chapter 4

Software and Hardware Implementation of RSA Encryption Core

This chapter will be discussing the hardware and software implementation of the RSA encryption core.

4.1 Software Model

As discussed in previous chapters the RSA algorithm consists of three steps,

- Key generation
- Encryption
- Decryption

Usually, a linear feedback shift register (LFSR) of size n bit is used to generate a random number of size $2n-1$ bit long. The random number generated is then tested for primality using Miller-Rabin primality tester. The software model performs all the steps and creates public and private key pairs and stores it in a file to be used by the Design under Test (DUT). All functions in the software model are modular and can be called as per user requirements. A library is built using this model and will be run in concurrent with DUT for verification purposes.

4.1.1 rsa_gen_keys()

This function is used in the software model to generate the keys. Once the prime numbers are generated, the modulus and totient function is calculated as per the formula described in the previous chapter. In this model, the Extended Euclidean Algorithm is used to find the greatest common divisor (gcd) of the selected exponent value and Euler's totient function. This is done to verify if the exponent value selected is indeed correct. Once the entire process is finished, each key pair is stored inside a text file to be used by the DUT. This function is looped multiple time to generate a large variety of keys to be used by the DUT.

4.1.2 rsa_encrypt()

This function performs the RSA encryption algorithm using the direct approach. The inputs to this function are public key pair , message. The function takes in the message and stores each character into a character array based on message length. The key pair encrypts each character and outputs cipher data of all the characters. The modular exponentiation process is performed by recursive multiplication method where the message data is multiplied that many times based on exponent value. And then the final output is modulo with n to obtain the encrypted the data. This process is performed until all the characters are encrypted. Once the message is encrypted, the cipher-text is stored into a text file for further analysis.

4.1.3 rsa_decrypt()

This is the decryption function which performs the RSA decryption algorithm using a direct approach. This function takes the encrypted data, private key pairs as input. The modular exponentiation process is as same as encryption algorithm. The final output is the original input message.

4.2 Hardware Architecture

The main objective of this hardware is to perform RSA encryption using the direct approach as discussed in the last chapter. For this purpose, the three input necessary data is needed the exponent, modulus, and message. The verification environment will provide these inputs. The wr-data port is used to get Exponent and Modulus value whereas the msg port is used to get message data. Since the same port is used to both exponent and modulus data, memory is used inside the DUT to store the incoming data. Table 4.1 contains all the top-level Input and Outputs of the Hardware design.

4.2.1 Hardware Components

The RSA hardware consist of following components,

4.2.1.1 Write Cycle

A single write cycle is utilized in this design. The process begins from MASTER in this instance the verification environment representing the valid address on the address line, valid data on wr_data, valid assert on wr_stb, valid bank select on chip_select. The DUT once it receives wr_stb and chip_select, the corresponding block memory is selected and wr_data is sent

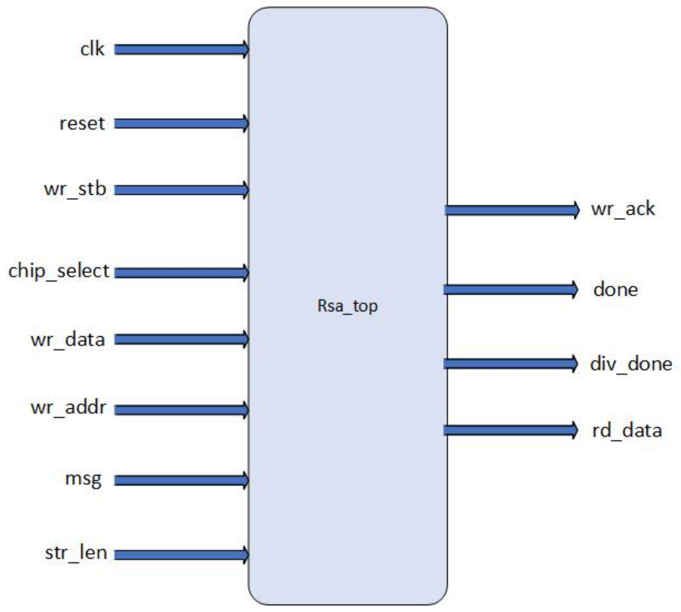


Figure 4.1: RSA encryption top

Table 4.1: Input/Output Ports

Port Name	Direction	Size	Description
clk	Input	1	Clock
reset	Input	1	Reset
chip_select	Input	2	Chip select to select the Block memory
wr_stb	Input	1	Write enable signal
wr_data	Input	64/32	Write Data bus
msg	Input	800	Message bus
str_len	Input	32	Message Length bus
done	Output	1	Message encryption done signal
div_done	Output	1	Character encryption done signal
o_data	Output	64/32	Output data

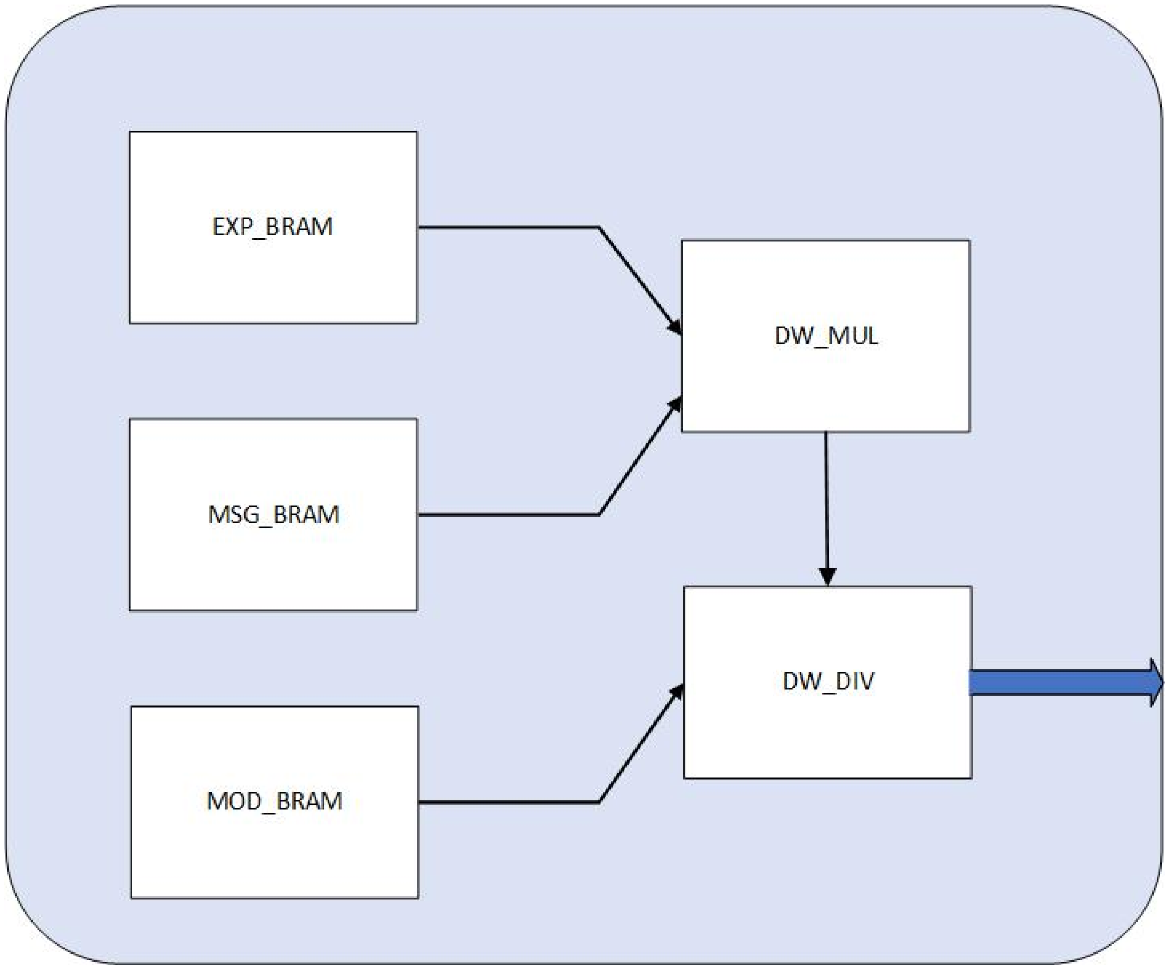


Figure 4.2: Hardware Components

Table 4.2: DW_MUL pin description

Port name	Direction	Size	Description
A	Input	A_width	Input Multiplier operand. The A_width size ≥ 1
B	Input	B_width	Input Multiplicand operand.The B_width size ≥ 1
TC	Input	1	Two's complement control signal.
Product	Output	A_width + B_width	Output product

through it. The block memory latches that data and sends a acknowledge signal back to Master to indicate locked data via the wr_ack port. The Master on receiving this acknowledgment negates the wr_stb and chip_select to terminate the cycle

4.2.1.2 DW_MUL

This is a Synopsys DesignWare IP used for multiplication purposes. This multiplies input A and B. Both unsigned and signed multiplication could be done using this IP, and the input word length is parameterized. A control signal is used to indicate if signed or unsigned multiplication to be done.

4.2.1.3 DW_DIV

This is also a Synopsys DesignWare IP which is used for division purposes. This IP divided the input dividend A by the input divisor B. Both quotient and remainder are produced as output. The remainder output calculates the modulus, which is required in the current design.

4.2.1.4 Block Memory

A block memory is used to store all the incoming data to the DUT. In this design separate block memories are used to store the exponent and modulus and message data. This is a simple block

Table 4.3: DW_DIV pin description

Port name	Direction	Size	Description
A	Input	A_width	Input Dividend operand. The A_width size ≥ 1
B	Input	B_width	Input Divisor operand.The B_width size ≥ 1
quotient	Input	A_width	quotient
remainder	Output	B_width	remainder/modulus

memory where it gets data and puts into the memory based on address value. Once the data has been stored, an acknowledge signal is sent to top. The RSA finite state machine receives these signals and instructs the next data to be addressed.

4.3 Design Flow

This is the central part of the RSA core whose purpose is properly to control data flow between the components. Two state machines are present in the top model. One is to get the data and store it in the memory, and other is to fetch all these data and control modular exponentiation and reduction process. Both the state machine is responsible for all the control signals for a required operation.Figure 4.3 details the working of the first state machine. This state machine consists of six states namely IDLE, EXP, MOD, MSG, ACK, GO. The state will be in the idle state until wr_stb is asserted. Once this signal is asserted the based on the incoming chip select value, the next state is moved to either of EXP or MOD or MSG state. The state machine will be in these state until wr_ack from the block memory is asserted. Once the acknowledge is received from block memory, the control is moved to ACK state. A control signal is maintained in this state which will be asserted indicating once all the acknowledgment from all the block memory is received and a “go” signal is issued, which shows that start of modular exponentiation process.

Figure 4.4 details the working of the second finite state machine which controls all the modular exponentiation and reduction process. There is a total of six states namely S0, S1, S2, S3, S4, S5.

- State S0: This is the ideal state of the state machine which waits for the "go" signal to be asserted. Once the signal is asserted control is moved to the next state.

- State S1: This state sets up all the registers required for modular exponentiation. A read signal is asserted to read the data from the block memory to the local registers. Also, The length of the string is stored into a local register which will be decremented every time a character has been encrypted.

- State S2: The modular exponentiation process starts at this state. Register named “counter” has the exponent value which will be reduced every time a multiplication operation is done. The message byte is sent as operands to the multiplier.

- State S3: This state waits for the multiplication process to be finished until the exponent is zero. Once the process is finished the control is moved to the next state for the modular reduction process.

- State S4: The product result and the modulus value is sent to the divider for modular reduction operation. DW_div has two outputs, a quotient and remainder. The remainder port has the reduced output. A division done signal is asserted to indicate completion of encryption of one character. If all the characters are encrypted the control is moved to the next state, or it is moved back to state S2 for encryption of next character.

- State S5: A done signal is asserted which indicate the encryption of all the characters and output is available in o_data port. And control moves back to state S0 where it waits

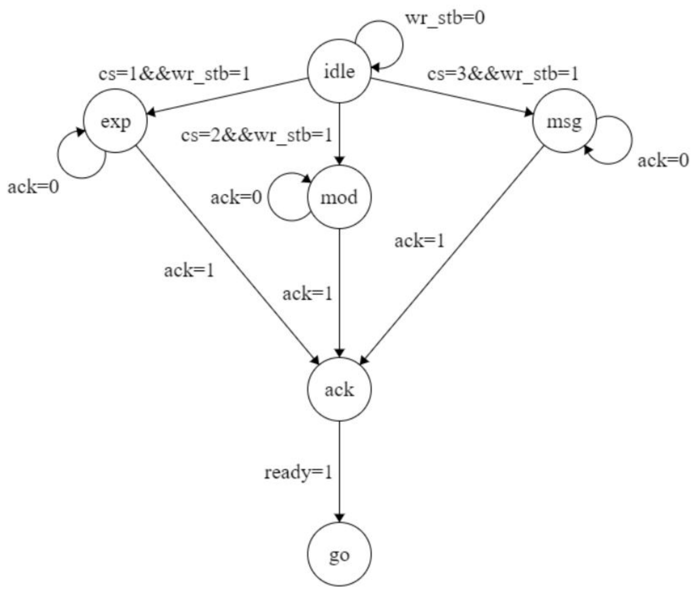


Figure 4.3: Finite State Machine 1

for the next data to be fed.

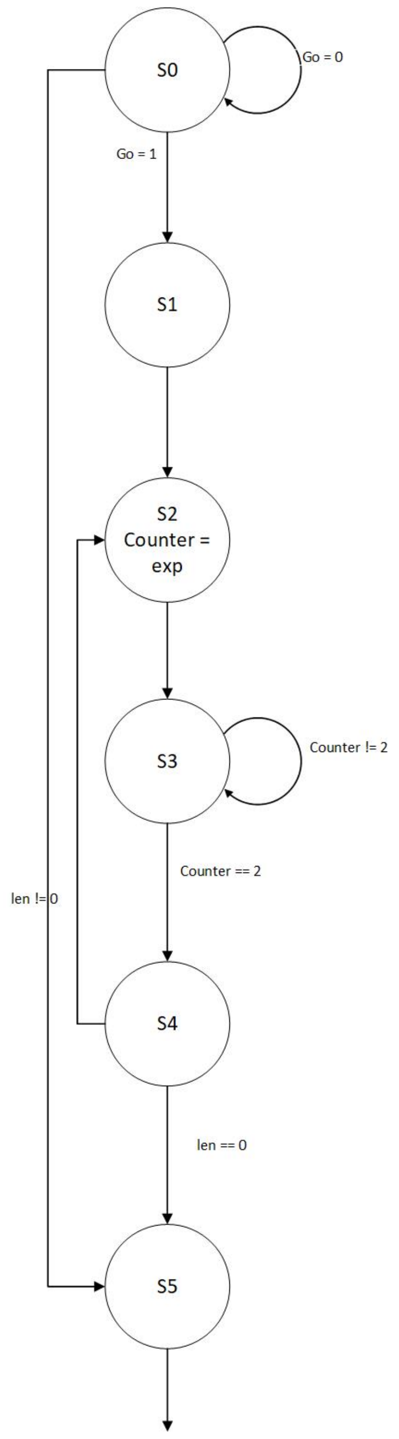


Figure 4.4: Finite State Machine 2

Chapter 5

Verification Concepts and Methodology

This chapter discusses the basic verification concepts and methodology adopted to verify the DUT.

5.1 SystemVerilog And Universal Verification Methodology (UVM)

Over the years, constrained random testing has been adopted as a methodology for functional verification of application-specific integrated circuit (ASIC) projects. SystemVerilog is a widely accepted hardware verification language which is supported by all the three major electronic design automation (EDA) vendors. Certain features of SystemVerilog is well-defined and implemented in most of the simulators. These features include:

- ANSI style port declarations.
- C-style constructs for statements like for, for each, break, continue, etc.
- Classes based programming

- Data types, dynamic arrays, associative arrays, constrained random generation, and queues.
- Communication between modules and classes via the virtual interface.

One of the reasons SystemVerilog is implemented consistently apart from the above-mentioned features is because they are used widely to create libraries of base classes which forms the basis of UVM (Universal Verification Methodology), OVM (Open Verification Methodology), etc. Although SystemVerilog was adopted for verification for block-level modules, a problem arose when it was difficult to use the same verification environment for all other modules. All these needs led to the creation of verification methodologies. UVM was introduced in 2011 by Accellera based on OVM v2.1.1.

5.1.1 Object oriented programming concepts

Object-oriented programming concepts (OOP) are important to constraint random testing because of the ability to reuse. OOP techniques allow the testbench components to be reused without modifying their source code. Also, there is structured communication between the components using function calls. Unlike structure oriented programming, in OOP the programs are organized using objects and data rather than logic and actions. Because of this, there is great flexibility to manipulate the objects and their relations. Following are some of the important concepts of OOP.

5.1.1.1 Class

Class is the main part of a program which represents the behavior, properties and initial values of the state. Classes are important for creating objects. In UVM everything happens inside the class and it accounts a group of objects with common behavior.

5.1.1.2 Object

Objects are basic units of code which are eventually obtained from the process contained inside the class. While programming, classes are made more generic so that objects can reuse the definitions in their code. Every object is an instance of a class or derived class with their own methods and data variables.

5.1.1.3 Methods

The methods are used to represent the behaviors of a state, whereas the attributes are represented by a variable.

5.1.1.4 Inheritance

Inheritance is a property of OOP where a class can inherit some behavior of other class. This allows classes to have the same properties of some other general class along with its own special properties. Better data analysis and reduced development time is achieved due to this property.

5.1.1.5 Abstraction

Abstraction is a property of OOP which is used to hide certain details of an object. In other words, selecting the only particular pool of relevant details from the object.

5.1.1.6 Encapsulation

Encapsulation is one of the fundamental properties of OOP where data and methods are bundled together within one unit, for example a class. There are three types of hiding data constructs : public, private and protected.

5.1.1.7 Polymorphism

This property details the ability to process objects differently based on their data type. Using this property a single object can be treated as another and can be used to define multiple levels of interface.

5.2 UVM classes

UVM is a library of many classes for development and reuse of the verification environment authored in SystemVerilog. The test environment can be designed by extending these classes. The UVM library consist of three main classes.

5.2.1 uvm_objects

This is the top level class for uvm_transaction and uvm_component classes. The common methods required for operations like copy, create, compare, print, record are defined in this class.

5.2.2 uvm_transaction

This is the base class for all the transaction properties. This class inherits all the methods defined in the uvm_object class. Timing, recording interface and notification events functions are added to this class.

5.2.3 uvm_components

This is the base class for all the uvm verification environment components. Section 5.3 details all the components used for verifying this DUT. This class extends the uvm_object and

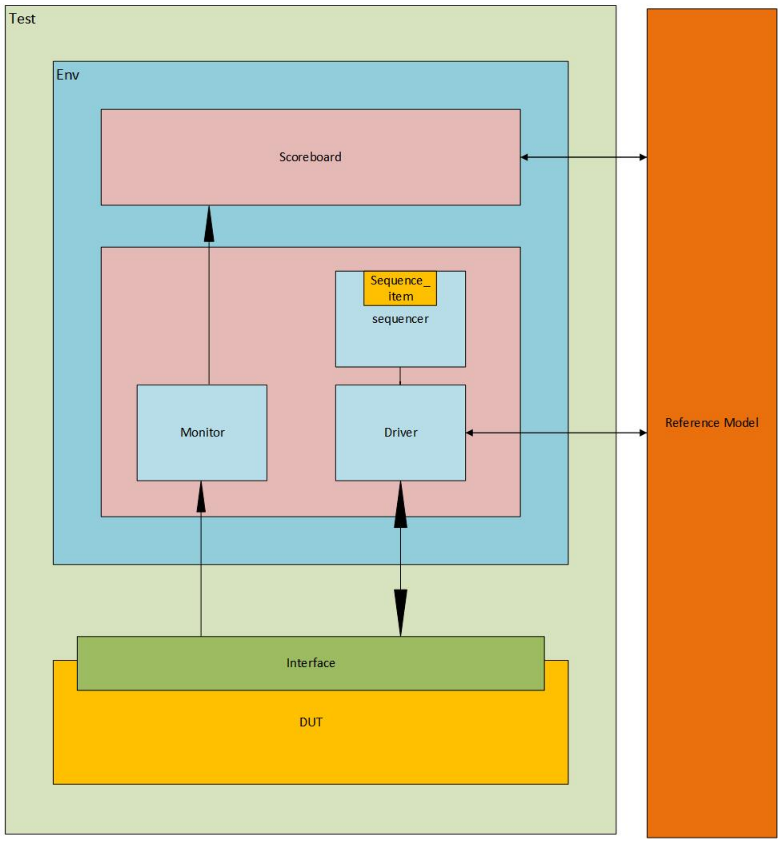


Figure 5.1: Basic Testbench Environment

uvm_transaction class. All the uvm_phases methods are defined in this class.

5.3 UVM components

Figure 5.1 details the structure of the verification environment.

5.3.1 Sequence_item and Sequence

Sequence_item is the basic unit of the testbench environment which is used to model a piece of information to be transmitted between the two component of the environment. For example, a write protocol is a sequence item. The sequence is the object which is used by its body()

method to send the data to the driver. In other words, the sequence is a collection of sequence items.

5.3.2 Driver

The driver is the main component of this environment as it communicates with the DUT via the virtual interface. A state machine is used to drive the sequence data to the DUT. The driver fetches the data from sequencer using get_next_item() method, and the data which is available in the FIFO is sent to the DUT. Once the data is sent, the driver waits for the done signal from the DUT such that the next batch of data can be sent to the DUT.

5.3.3 Monitor

The monitor is used to used detect the output signals coming from the DUT and perform necessary logic to determine the validity of the output signal. It communicates with DUT via the virtual interface.

5.3.4 Agent

The agent is a wrapper component which comprises of the driver, monitor, and the sequencer. Agents are specific to the DUT and verification environments can consist of many agents. The agent provides sequence_item data to the DUT and scoreboard.

5.3.5 Scoreboard

The scoreboard is the component where the actual analysis of the data coming from the DUT and a reference model is done.

5.3.6 Environment

The environment is a wrapper component which holds all the above-mentioned components. A complex verification environment has multiple agents and a scoreboard, and it is the duty of the environment to provide a connection between these components accurately.

5.3.7 Test

This is the top level component which is used to configure the base class components as per the task-specific work. It also performs an important task of invoking the sequence to be passed to the DUT via the environment.

5.3.8 Top

This is the top level testbench component which is used to connect the testbench with the DUT. An interface instance is created and appropriate signals are connected with each other in this component. The interface is passed to the base class components via config_db() factory method.

5.4 UVM phases

All of these UVM components has to go through a set of pre-defined phases and will not proceed to next phase until the current phase is completed. UVM phases can be grouped into three phases, ran in order:

1. Build phase
2. Run phase

3. Clean-up phase

5.4.1 Build phase

This phase is executed before the uvm testbench simulation is started and its main objective is to build, connect, and configure all the testbench components.

5.4.1.1 build

This phase constructs all the uvm components in top-down hierarchy. UVM factory is used to register all the component objects.

5.4.1.2 connect

All the TLM ports of the uvm components are connected using this phase.

5.4.1.3 end_of_elaboration

This phase is used to make any last adjustments to the body , configuration , connectivity of the verification environment.

5.4.2 Run phase

5.4.2.1 start_of_simulation

This phase is used to set the run-time configurations. This phase is mainly used to communicate with other environments using DPI.

5.4.2.2 run_phase

This main phase where all the stimulus generation and testbench activities are scheduled. This is a time consuming phase and all the test-case activities is executed. All the uvm_components run phase's s are executed in parallel. The below listed phases are the scheduled actions that are performed in run_phase.

5.4.2.3 pre_reset and post_reset

The main purpose of this method is to sample signals that have to be occur before and after the reset signal should occur; for example: waiting for the power signal to be asserted.

5.4.2.4 reset

This phase is used to generate the reset and drive the reset to the DUT.

5.4.2.5 pre_configure

This phase is to prepare the DUT configurations once it has been out of the reset process.

5.4.2.6 configure

This phase is to put the DUT into an ideal state or a defined state before the stimulus could be driven to the DUT.

5.4.2.7 post_configure

This phase is used to wait for the configurations to be distributed through the DUT.

5.4.2.8 pre_main

This phase ensures that all components are generated which are needed for the stimulus to be applied to the DUT.

5.4.2.9 main

This is the main phase where the generated stimulus of a test-case is applied to the DUT. Sequences are executed in this phase. The phase ends on successful completion of all the stimulus and / or when a timeout occurs.

5.4.2.10 post_main

This phase performs all the jobs necessary after main phase.

5.4.2.11 pre_shutdown

This phase is used as buffer to apply any remaining stimulus to the DUT before the shutdown occurs.

5.4.2.12 shutdown

This phase make certain that all the stimulus data are driven to the DUT.

5.4.2.13 post_shutdown

This phase is used as an final act before the beginning of clean-up phase.

5.4.3 Clean up phase

All the below methods are used by the analysis side of a uvm component.

5.4.3.1 extract

This method is used to collect all the functional coverage information from the scoreboard and monitors.

5.4.3.2 check

This method is used analyze the collected data from the monitor and scoreboard if the behavior of the DUT is correct or not.

5.4.3.3 report

This method reports the analysis of the output data to a standard output or to a file.

5.5 Test methodology

The basic idea behind the verification method adopted in this project is to verify the functional correctness of the design by sending the same stimulus data to both the DUT and a software model, and compare the output results from the DUT and a software model. The flow chart in figure5.2 details the flow of both hardware and software flow. There are two types of testing , constrained random and direct testing. Constrained random testing is adopted in this test plan to verify all the test corners of the design.

5.5.1 Sequence_item

As discussed in 5.3.1, a sequence is a basic unit of data, in this case, the data for e,n,d, message are already generated via rsa_gen_keys() and stored in a text file. The top once it raises the sequence, each of these keys from the file is read stored into a separate local memory starting

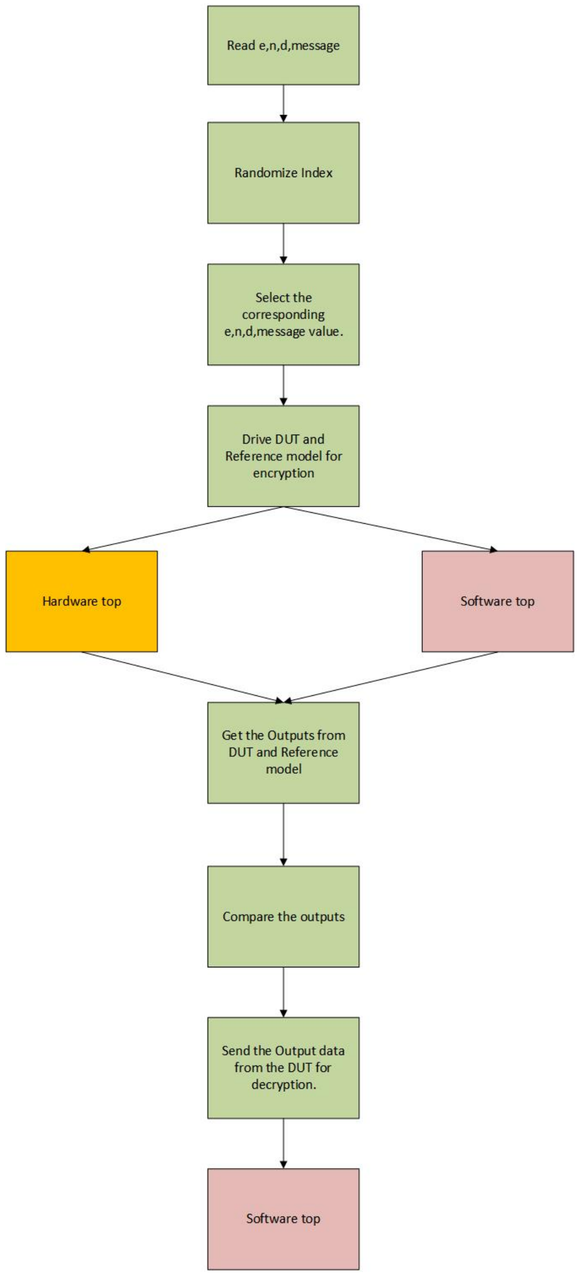

Read e,n,d,message
Randomize Index
Select the corresponding e,n,d,message value.
Drive DUT and Reference model for encryption
Hardware top
Software top
Get the Outputs from DUT and Reference model
Compare the outputs
Send the Output data from the DUT for decryption.
Software top

Figure 5.2: The Verification flow

from index 0. A variable “index” is used which gets randomized for every sequence and based on the randomized value the corresponding index data is selected from the local memory of each data. The following subsection provides all the variables used in the sequence_item.

5.5.1.1 Sequence_item variables

- *e*: A 64-bit width variable with 3000 locations to store the exponent value of the public key.
- *n*: A 64-bit width variable with 3000 locations to store the modulus value of the public key.
- *d*: A 64-bit width variable with 3000 locations to store the exponent value of the private key.
- *m*: A 100-bit width variable with 3000 locations to store the message.
- *Index*: a 32-bit variable that gives the index location when randomized.

Constraints are used to restrict the random value generated of the index variable between 0 to 3000.

5.5.2 Sequencer

The sequencer is used to generate the sequence_item and queue it up in order for transferring it to the driver.

5.5.3 Driver

Once the Data is ready in sequencer it signals the driver to start driving the data to the DUT. A state machine is implemented in the driver which constantly gets this data and send it to the

DUT via the virtual interface. Meanwhile the same data is pushed to the software model via DPI. Once the data is sent it waits for the “done” signal from DUT to send the next batch of data.

5.5.4 Monitor

The monitor performs the same task as mentioned in 5.3.3 . The Output from the DUT and reference model is compared in this model and objection is raised if the data mismatches. The method compare() performs this task.

5.5.5 Coverage

A thorough understanding of the DUT is required to set up the verification role. Following cover groups is created for this purpose.

- Modulus data of the public is key covered to check if all the combinations of 64-bit values are driven to the DUT.
- A cover group for the message port is created to check if all the ASCII values are being covered,
- The output from the DUT is covered.
- Assertions are created for the write cycle and check its functionality.
- Cross coverage between Output from DUT and Reference model.

Chapter 6

Results and Discussion

This Chapter will be dealing with result and analysis of the results of the RSA_CORE.

6.1 RSA _CORE Synthesis results

The table 6.1 details the area, power, timing details obtained after logic synthesis of the RSA_CORE using three different technology node libraries. Synopsys Design Compiler is used for logic synthesis. The latency of the process depends upon the length of the message to be encrypted. Figure 6.1 and 6.2 plots the area and power distribution of the RSA_CORE for three different technology nodes.

6.2 RSA_CORE Coverage results

Figure 6.4 shows the simulation outputs obtained from the RSA_CORE. Cadence SimVision was used to run simulations. The coverage reports are analyzed and explored using Cadence Integrated Metrics (IMC) tool. As observed from the figure the latency of the core depends upon the string length of the message. The *go* signal denotes the start of the encryption process

Parameter		180 nm	65 nm	32 nm
Area (um^2)	Combinational Area	1639073.63	154920.60	194733.77
	Non combi- national Area	167534.13	23638.68	17460.70
	Total area	1806607.76	178559.28	269028.01
Power	Internal Power (mW)	32.1622	1.5125	2.376
	Switching Power (mW)	32.4103	0.3566	0.70139808
	Leakage Power (nW)	$7.8655e+03$	$8.5201e+03$	16.705
	Total Power (mW)	64.5807	1.8776	19.795
Timing (ns)	Worst case Delay	51.0993	21.2107	77.5613
DFT coverage	%	86.44	86.30	85.79
Gate count		181204	123999	176992

Table 6.1: RSA _CORE Logic synthesis report

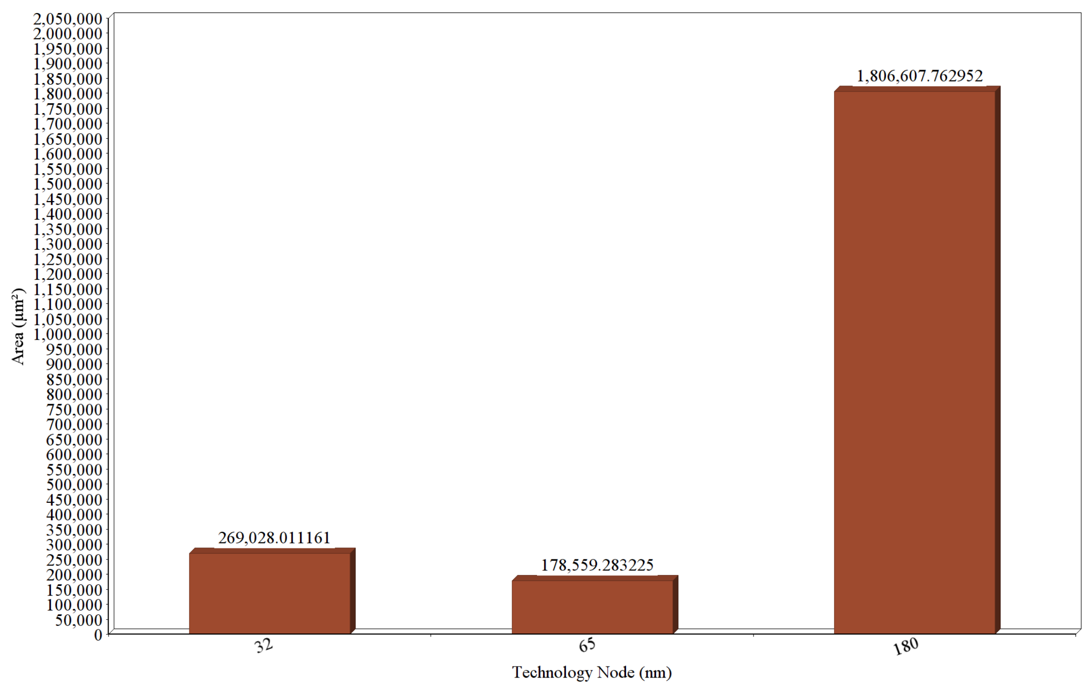


Figure 6.1: RSA _CORE Area

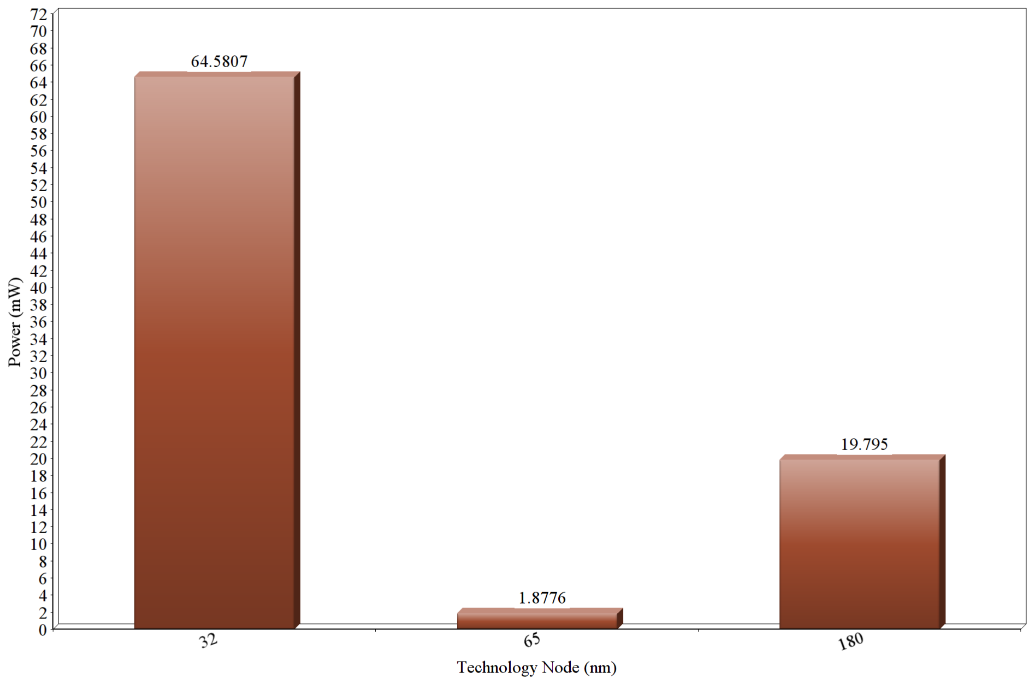


Figure 6.2: RSA _CORE Power

Table 6.2: Coverage results

Runs	Code Coverage %	Functional Coverage %		Overall Functional Coverage %
		Assertions Coverage	Cover Groups	
10	82.95	100	41.15	85.29
100	85.2	100	78.12	94.53
1000	83.46	100	97.42	98.7
10000	83.48	100	99.48	99.74
100000	83.48	100	99.48	99.74
1000000	83.48	100	99.48	99.74

and *done* denotes the end of the encryption process. Figure 6.5 details the coverage run for $1,000,000$ vectors using the IMC tool . As discussed in the last chapter cover-ports are created for input and outputs. It can be seen that the code coverage is 83.48%, this low percentage is due to because of unsynthesizable expressions in the DesignWare IP. These Expressions are never attained and therefore can be neglected. But the functional coverage run was 99.74% as show in the figure. Figure 6.3 plots functional coverage percentage against number of runs. Until 10000 runs , a steep increase in functional coverage percentage is observed in correspondence with the number of runs. After 10000 runs the percentage seems to flattens out. Table6.2 details the code and functional coverage report of the DUT for different runs.

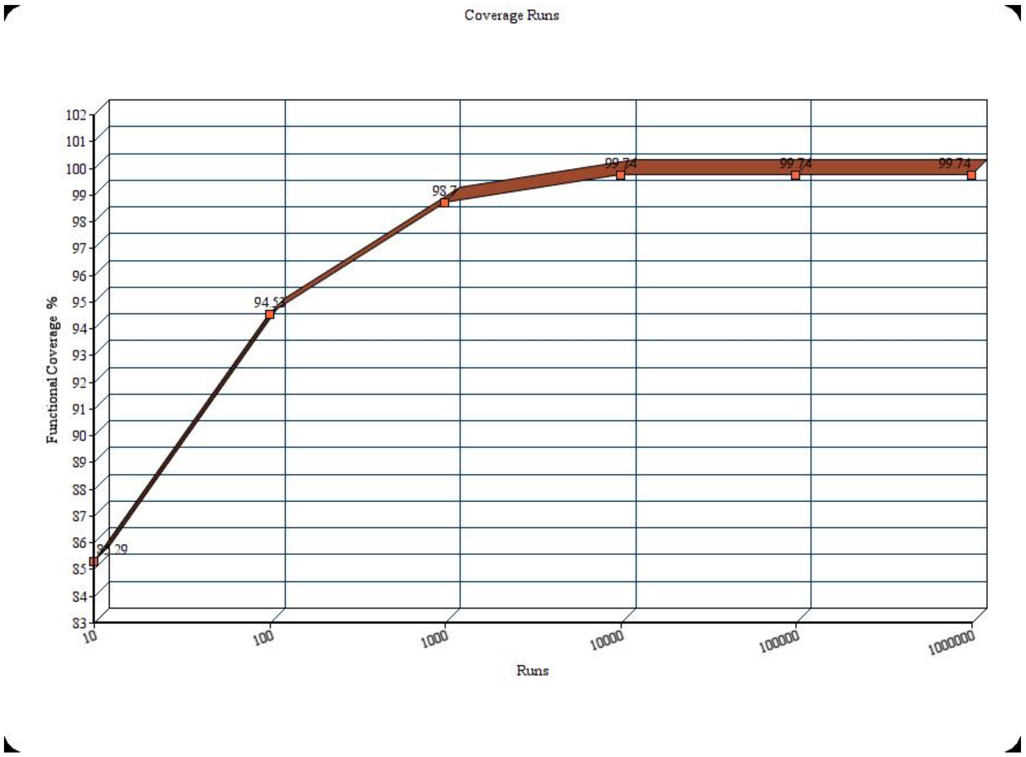


Figure 6.3: Functional Coverage vs Runs

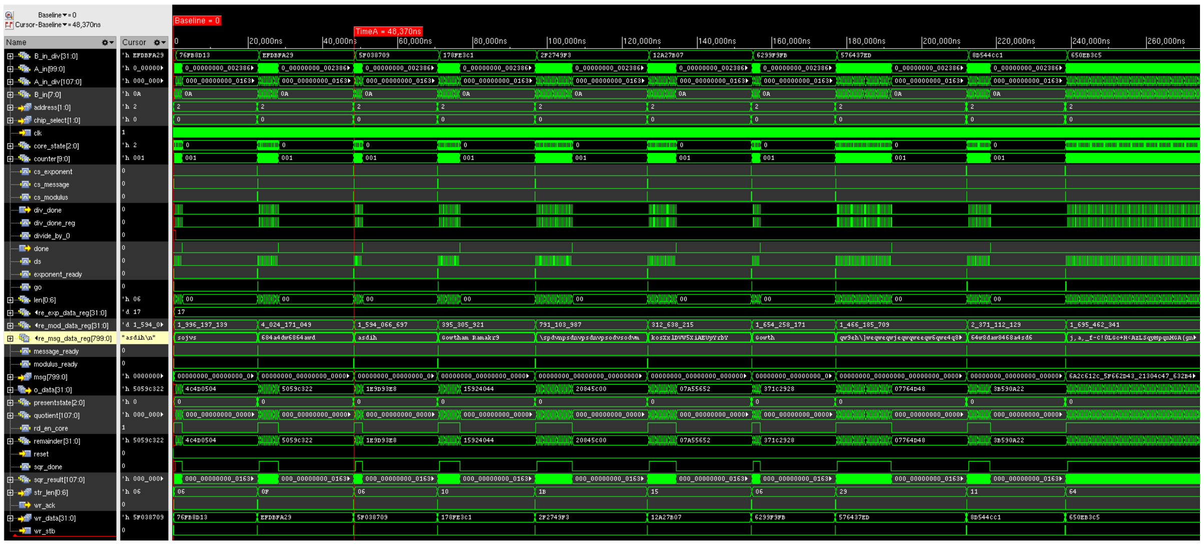
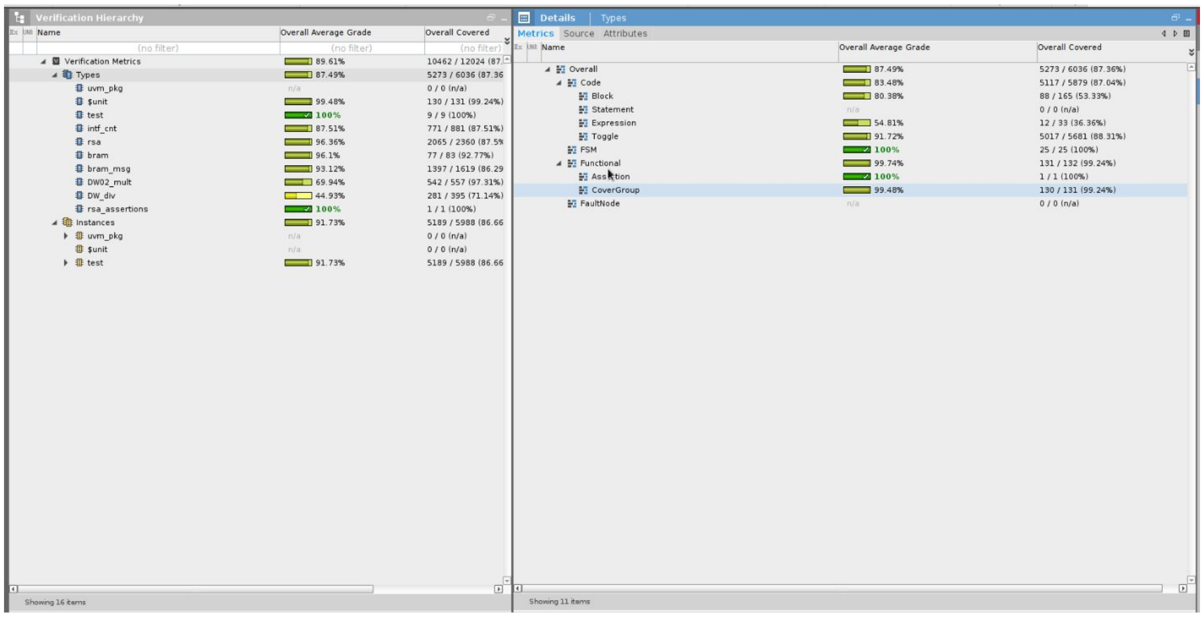

```
Baseline = 0
Cursor-Baseline = 48,370ns
Name                        Cursor
B_in_div[31:0]              'h EFDBFA29
A_in[99:0]                  'h 0_00000
A_in_div[107:0]             'h 000_000
B_in[7:0]                   'h 0A
address[1:0]                'h 2
chip_select[1:0]            'h 0
clk                         1
core_state[2:0]             'h 2
counter[9:0]                'h 001
cs_exponent                 0
cs_message                  0
cs_modulus                  0
div_done                    0
div_done_reg                0
divide_by_0                 0
done                        0
ds                          0
exponent_ready              0
go                          0
len[0:6]                    'h 06
re_exp_data_reg[31:0]       'd 17
re_mod_data_reg[31:0]       'd 1_594_0
re_msg_data_reg[799:0]      "asdih\n"
message_ready               0
modulus_ready               0
msg[799:0]                  'h 0000000
o_data[31:0]                'h 5059c322
presentstate[2:0]           'h 0
quotient[107:0]             'h 000_000
rd_en_core                  1
remainder[31:0]             'h 5059c322
reset                       0
sqr_done                    0
sqr_result[107:0]           'h 000_000
str_len[0:6]                'h 06
wr_ack                      0
wr_data[31:0]               'h 5F038709
wr_stb                      0
```

Figure 6.4: RSA_CORE simulation



Verification Hierarchy

Name	Overall Average Grade	Overall Covered
Verification Metrics	89.61%	10462 / 12024 (87.
Types	87.49%	5273 / 6036 (87.36
uvm_pkg	n/a	0 / 0 (n/a)
$unit	99.48%	130 / 131 (99.24%)
test	100%	9 / 9 (100%)
intf_cnt	87.51%	771 / 881 (87.51%)
rsa	96.36%	2065 / 2360 (87.5%
bram	96.1%	77 / 83 (92.77%)
bram_msg	93.12%	1397 / 1619 (86.29
DW02_mult	69.94%	542 / 557 (97.31%)
DW_div	44.93%	281 / 395 (71.14%)
rsa_assertions	100%	1 / 1 (100%)
Instances	91.73%	5189 / 5988 (86.66
uvm_pkg	n/a	0 / 0 (n/a)
$unit	n/a	0 / 0 (n/a)
test	91.73%	5189 / 5988 (86.66

Showing 16 items

Details | Types

Metrics | Source | Attributes

Name	Overall Average Grade	Overall Covered
Overall	87.49%	5273 / 6036 (87.36%)
Code	83.48%	5117 / 5879 (87.04%)
Block	80.38%	88 / 165 (53.33%)
Statement	n/a	0 / 0 (n/a)
Expression	54.81%	12 / 33 (36.36%)
Toggle	91.72%	5017 / 5681 (88.31%)
FSM	100%	25 / 25 (100%)
Functional	99.74%	131 / 132 (99.24%)
Assertion	100%	1 / 1 (100%)
CoverGroup	99.48%	130 / 131 (99.24%)
FaultNode	n/a	0 / 0 (n/a)

Showing 11 items

Figure 6.5: RSA _CORE Coverage report

Chapter 7

Conclusion

In this project a RSA encryption core is designed using the direct method approach of modular exponentiation. For this purpose a DesignWare multiplier and divider was used. The code is parameterized hence the word size can be modified using the configuration file for higher key size. To verify the functionality of this core a UVM verification environment was designed. A software model was integrated into this environment to compare the DUT and model results. The testbench is capable of providing the randomized data to the DUT. This ensured a wide range of data was passed to the DUT.

The RTL design was started using top down design methodology, and initially the wrapper code and control flow of the design was coded and then DesignWare IP's were added to the code and it was completely synthesizable. The DUT design and verification environment design was done in parallel to minimize the time required to complete the project. Constraint random testing test plan was adopted for verifying the core. A 99.74 % of functional coverage was achieved by running 1,000,000 random vectors.

7.1 Future work

- Chapter 3 discussed different types of the modular exponentiation techniques. Other techniques can be adopted to do this process and merits can be compared with this model.
- A decryption core can be designed and integrated with this encryption core to create a complete encryption and decryption unit.
- More research can be done on modular exponentiation process for higher key sizes.

References

[1] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic Processors-A Survey. In *Proceedings of IEEE*, 2006.

[2] An Introduction to Cryptography.

[3] C.E. Goutis. A.P. Kakarountas, H. Michail. Implementation of HSSec: a High-Speed Cryptographic Co-processor. In *IEEE Conference on Emerging Technologies and Factory Automation.*, 2007.

[4] L. Adleman. R. Rivest, A. Shamir. A method for obtaining digital signatures and public-key cryptosystems. In *Magazine Communications of the ACM CACM Homepage archive Volume 21 Issue 2, Feb. 1978 ,Pages 120-126*, 1978.

[5] Ari Shawkat Tahir. Design and Implementation of RSA Algorithm using FPGA. In *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY*, 2015.

[6] Jia-Lin Sheu, Ming-Der Shieh, Chien-Hsing Wu, and Ming-Hwa Sheu. A pipelined architecture of fast modular multiplication for RSA cryptography. In *ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No.98CH36187)*, volume 2, pages 121–124, 1998. doi:10.1109/ISCAS.1998.706856.

[7] Parvathy R and Prof. G. K. Sadanandan. An Efficient RSA Algorithm using Pipelined Vedic Multiplier. In *International Journal of Science Technology & Engineering*, 2016.

[8] Peter L. Montgomery. Modular multiplication without trial division. In *Mathematics of Computation, Vol. 44, No. 170. (Apr., 1985), pp. 519-521.*, 1985.

[9] N. Vinodhini and C. Suganya. Pipelined VLSI Architecture for RSA Based on Montgomery Modular Multiplication. In *International Conference on Innovations in Engineering and Technology (ICIET) - 2016*, 2016.

[10] H. Bozorgi. S. S. Ghoreishi, M.A. Pourmina. High Speed RSA Implementation Based on Modified Boothś Technique and Montgomery Multiplication for FPGA Platform. In *Second International Conference on Advances in Circuits, Electronics and Micro-Electronics*, 2009.

[11] Jean Pierre David, Kassem Kalach, and Nicolas Tittley. Hardware Complexity of Modular Multiplication and Exponentiation. In *IEEE Transactions on Computers*, 2007.

[12] Monther Al-Ja'fari Qasem Abu Al-Haija, Mahmoud Smadi and Abdullah Al-Shua'ibi. Efficient FPGA Implementation of RSA Coprocessor Using Scalable Modules. In *International Symposium on Emerging Inter-networks, Communication and Mobility*, 2013.

[13] Iqbalur Rahman Rokon Mostafizur Rahman. Efficient Hardware Implementation of RSA Cryptography. In *IEEE Conferences*, 2009.

[14] Y. Ding, J. Hu, D. Wang, and H. Tan. A High-Performance RSA Coprocessor Based on Half-Carry-Save and Dual-Core MAC Architecture. In *Chinese Journal of Electronics (Volume: 27, Issue: 1, 1 2018)*, 2018.

[15] Jaime Velasco-Medina Claudia P. Renteria-Mejia, Vladimir Trujillo-Olaya. Design of an 8192-bit RSA Cryptoprocessor based on Systolic Architecture. In *2012 VIII Southern Conference on Programmable Logic*, 2012.

[16] Wei Wang. Xinming Huang. A Novel and Efficient Design for an RSA Cryptosystem With a Very Large Key Size. In *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS, VOL. 62, NO. 10, OCTOBER 2015*, 2015.

[17] Anantha P. Chandrakasan James Goodman. An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. In *IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 36, NO. 11, NOVEMBER 2001*, 2001.

[18] Wen-Ching Lin. Jun-Hong Chen, Ming-Der Shieh. A High-Performance Unified-Field Reconfigurable Cryptographic Processor. In *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 18, NO. 8, AUGUST 2010*, 2010.

[19] Cetin K. Koc. Alexandre F. Tenca. A Scalable Architecture for Modular Multiplication Based on Montgomery Algorithms. In *IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 9, SEPTEMBER 2003*, 2003.

[20] Tarek El-Ghazawi. Miaoqing Huang, Kris Gaj. New Hardware Architectures for Montgomery Modular Multiplication Algorithm. In *IEEE TRANSACTIONS ON COMPUTERS, VOL. 60, NO. 7, JULY 2011 .*, 2011.

[21] S. Vaudenay M. Joye, P. Paillier. Efficient generation of prime number. In *Cryptographic hardware and embedded systems : CHES 2000. 2nd international workshop, Worcester, MA, USA, 2000. Proceedings*, 2000.

[22] Wayne Luk Ray Cheung, Ashley Brown. A Scalable Hardware Architecture for Prime

Number Validation. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, 2004.

[23] Jizeng Wei Jingwei Hu, Wei Guo. A Novel Architecture for Fast RSA Key Generation Based on RNS. In *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, 2011.

[24] Omid Sarbishei . Milad Bahadori1, Mohammad Reza Mali. A Novel Approach for Secure and Fast Generation of RSA Public and Private Keys on SmartCard. In *Proceedings of the 8th IEEE International NEWCAS Conference 2010*, 2010.

[25] M. U. Sharif and M. Rogawski. Hardware-software codesign of RSA for optimal performance vs. flexibility trade-off. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[26] Koji Nakano Bo Song, Kensuke Kawakami. An RSA Encryption Hardware Algorithm Using a Single DSP Block and a Single Block RAM on the FPGA. In *First International Conference on Networking and Computing*, 2010.

Appendix A

Source Code

A.1 RSA top

```
module rsa (
            reset ,
            clk ,
            scan_in0 ,
            scan_en ,
            test_mode ,
            scan_out0 ,
                    address ,
                    chip_select ,
            wr_stb ,
                    wr_data ,
                    wr_ack ,
                    msg ,
                        done ,
                        o_data ,
                        str_len ,
                        div_done
        );
`include "./include/config.h"

input reset;                            // system reset
input clk;                               // system clock

input
    scan_in0,                           // test scan mode data input
    scan_en,                            // test scan mode enable
    test_mode;                          // test mode select

output
    scan_out0;                          // test scan mode data output

input [`STATE_SIZE_2-1:0]address;
input wr_stb;
input [`DATA_WIDTH-1:0] wr_data; ///2048
input [`MSG_WIDTH-1:0] msg;
input [`STATE_SIZE_2-1:0] chip_select;
input [0:6] str_len;
output  wr_ack;
output [`DATA_WIDTH-1:0] o_data;
output reg done ;
output div_done;

//Register
//reg [`STATE_SIZE_2-1:0] address_out;
reg cs_exponent;
reg cs_modulus;
reg cs_message;
reg wr_en_exponent;
reg wr_en_modulus;
reg wr_en_message;
reg [`STATE_SIZE_1-1:0] presentstate;
reg [`STATE_SIZE_1-1:0] core_state;
reg go;
reg sqr_done;

reg exponent_ready;
reg modulus_ready;
reg message_ready;
reg rd_en_core;

reg div_done_reg;

//wires
```

```
wire ack_exponent;
wire ack_modulus;
wire ack_message;



reg [0:6] len;
wire [`DATA_WIDTH-1:0] mem2core_exp_data_reg;
wire [`MODULUS_WIDTH-1:0] mem2core_mod_data_reg;
wire [`MSG_WIDTH-1:0] mem2core_msg_data_reg;
reg [`A_DATA_WIDTH-1:0] A_in;
reg [`B_DATA_WIDTH-1:0] B_in;
wire [`A_DATA_WIDTH+`B_DATA_WIDTH-1:0] sqr_result;
reg [`A_DATA_WIDTH+`B_DATA_WIDTH-1:0] A_in_div;
reg [`DATA_WIDTH-1:0] B_in_div;
wire [`A_DATA_WIDTH+`B_DATA_WIDTH-1:0] quotient;
reg [`MODULUS_WIDTH-1:0] remainder;
reg TC;
reg [`EXP_WIDTH-1:0] counter;
reg ds;
assign o_data = remainder;
assign div_done = div_done_reg;

//Instantiations
        ///Exponent memory
                bram  exp_bram  (
                                        .reset(reset),
                                        .clk(clk),
                                        .scan_in0(scan_in0),
                                        .scan_en(scan_en),
                                        .test_mode(test_mode),
                                        .scan_out0(scan_out0),
                                        .wr_en(wr_en_exponent),
                                        .cs(cs_exponent),
                                        .ack(ack_exponent),
                                        .addr(address),
                                        .wr_data(wr_data),
                                        .rd_data(
                                          mem2core_exp_data_reg),
                                        .rd_en(rd_en_core)
      );

       ///Modulus  memory
                bram    mod_bram    (
                                        .reset(reset),
                                        .clk(clk),
                                        .scan_in0(scan_in0),
                                        .scan_en(scan_en),
                                        .test_mode(test_mode),
                                        .scan_out0(scan_out0),
                                        .wr_en(wr_en_modulus),
                                        .cs(cs_modulus),
                                        .ack(ack_modulus),
                                        .addr(address),
                                        .wr_data(wr_data),
                                        .rd_data(
                                           mem2core_mod_data_reg),
                                        .rd_en(rd_en_core)
        );
         ///Message  memory
                bram_msg   msg_bram          (
                                        .reset(reset),
                                        .clk(clk),
                                        .scan_in0(scan_in0),
                                        .scan_en(scan_en),
                                        .test_mode(test_mode),
                                        .scan_out0(scan_out0),
                                        .wr_en(wr_en_message),
                                        .cs(cs_message),
                                        .ack(ack_message),
                                        .addr(address),
                                        .wr_data(msg),
                                        .rd_data(
                                           mem2core_msg_data_reg),
                                        .rd_en(rd_en_core)
        );
```

```
    //Multiplier for squaring operation
            DW02_mult           #(
                                .A_width(`A_DATA_WIDTH),
                                .B_width(`B_DATA_WIDTH)
                                )
                                mul_sqr  (
                                .A(A_in),
                                .B(B_in),
                                .TC(TC),
                                .PRODUCT(sqr_result)
                                );
    //Modulus Calculation
            DW_div              #(
                                .a_width((`A_DATA_WIDTH+
                                   `B_DATA_WIDTH)),
                                .b_width(`DATA_WIDTH)
                                )
                        div     (
                                .a(A_in_div),
                                .b(B_in_div),
                                .quotient(quotient),
                                .remainder(remainder),
                                .divide_by_0(divide_by_0)
                                );
```

```
assign wr_ack = ack_exponent ? 1'b1 :
                ack_modulus  ? 1'b1 :
                ack_message  ? 1'b1 : 1'b0;

////Main FSM to receive data and put into the memory
always @(posedge clk or posedge reset)
begin
        if (reset)
                begin
                presentstate <= `IDLE;
                cs_exponent <= 1'b0;
                cs_modulus <= 1'b0;
                cs_message <= 1'b0;
                wr_en_exponent <= 1'b0;
                wr_en_modulus <= 1'b0;
                wr_en_message <= 1'b0;
                exponent_ready <= 1'b0;
                modulus_ready <= 1'b0;
                message_ready <= 1'b0;
                go <= 1'b0;
                end
        else
        begin
        case(presentstate)
```

```
`IDLE: //0
begin

if(wr_stb)
begin
        case(chip_select)
                `EXPONENT_BASE_ADDRESS:
                        begin
                        presentstate <=
                           `EXPONENT_BRAM_EN_WR;
                        cs_exponent <= 1'b1;
                        wr_en_exponent <= wr_stb ;
                        end
                `MODULUS_BASE_ADDRESS:
                        begin
                        presentstate <=
                           `MODULUS_BRAM_EN_WR;
                        cs_modulus <= 1'b1;
                        wr_en_modulus <= wr_stb;
                        end
                `MESSAGE_BASE_ADDRESS:
                        begin
                        presentstate <=
                           `MESSAGE_BRAM_EN_WR;
                        cs_message <= 1'b1;
```

```
                              wr_en_message <= wr_stb;
                              end
                        default:
                              begin
                              presentstate <= `IDLE;
                              end
                  endcase
            end//wr_stb
            else
                  begin
                  presentstate <= `IDLE;
                  end
            end//idle
            `EXPONENT_BRAM_EN_WR: //1
                  begin
                  presentstate <= `ACK;
                  end
            `MODULUS_BRAM_EN_WR: //2
                  begin
                  presentstate <= `ACK;
                  end
            `MESSAGE_BRAM_EN_WR: //3
                  begin
                  presentstate <= `ACK;
                  end
```

```
      `ACK: //4
              begin
              if(ack_exponent)
                     begin
                     cs_exponent <= 1'b0;
                     wr_en_exponent <= wr_stb ;
                     exponent_ready <= 1'b1;
                     presentstate <= `IDLE;
                     end
              else if(ack_modulus)
                     begin
                     cs_modulus <= 1'b0;
                     wr_en_modulus <= wr_stb;
                     modulus_ready <= 1'b1;
                     presentstate <= `IDLE;
                     end
              else if(ack_message)
                     begin
                     cs_message <= 1'b0;
                     wr_en_message <= wr_stb;
                     message_ready <= 1'b1;
                     presentstate <= `IDLE;
                     end


              end
```

```
        default:
                begin
                presentstate <= `IDLE;
                end
        endcase//presentstate

        if(exponent_ready && modulus_ready && message_ready)

                begin
                presentstate <= `IDLE;
                go <= 1'b1;
                exponent_ready <= 1'b0;
                modulus_ready <= 1'b0;
                message_ready <= 1'b0;
                end
        else
                begin
                go <= 1'b0;
                end
        end//else
end//always
always @(posedge clk or posedge reset)
        begin
                if(reset)
                        begin
                        core_state <= `S0;
                        rd_en_core <= 1'b0;
                        ds <= 1'b0;
                        done <= 1'b0;
                        end
        else
        begin
        case(core_state)
        `S0:
        begin
                if(go)
                        begin
                        core_state <= `S1;
                        rd_en_core <= 1'b1;
                        done <= 1'b0;
                        end
                else
                        begin
                        core_state <= `S0;
                        done <= 1'b0;
                        rd_en_core <= 1'b0;
                        end
        end
        `S1: ///setup the values for squaring operation
        begin
```

```
            core_state <= `S2;
            len <= str_len;

        end
        `S2://LUT for converting the hex to preffered
           conversion.before deleting check the software model
        begin
            core_state <= `S3;
            counter <= mem2core_exp_data_reg;
            ds <= 1'b1;
            TC <= 1'b0;
            div_done_reg <= 1'b0;
        end
        `S3:///wait for the sqr_done and then move on to next
           state for setting the values for the product
           operation
        begin
            if(ds == 1)
                begin
                A_in <= mem2core_msg_data_reg[(len-1)
                   *8 +:7] ;
                B_in <= mem2core_msg_data_reg[(len-1)
                   *8 +:7] ;
                counter <= counter - 1'b1;
                ds <= 1'b0;
                end
        else
                begin
                A_in <= sqr_result ;
                B_in <= mem2core_msg_data_reg[(len-1)
                   *8 +:7];
                counter <= counter - 1'b1;
                end

        if(counter == 2)
                begin
                core_state <= `S4;
                sqr_done <= 1'b1;
                len <= len-1'b1;
                end
        else
                core_state <= `S3;
end
`S4:
begin
        if(len == 0)
                begin
                core_state <= `S5;
                sqr_done <= 1'b0;
```

```
                        end
                else
                        core_state <= `S2;
                        div_done_reg <= 1'b1;
                        A_in_div <= sqr_result ;
                        B_in_div <= mem2core_mod_data_reg;
        end
        `S5:
        begin
                A_in_div <= sqr_result ;
                B_in_div <= mem2core_mod_data_reg;
                done <= 1'b1;
                div_done_reg <= 1'b0;
                core_state <= `S0;
        end
        default:
        begin
                core_state <= `S0;
        end

        endcase
end
end

endmodule // rsa
```

A.2 BRAM

```
module bram (
           reset,
           clk,
           scan_in0,
           scan_en,
           test_mode,
           scan_out0,
           wr_en,
           cs,
           ack,
           addr,
           wr_data,
           rd_data,
           rd_en
       );

//Include Files
`include "./include/config.h"

///Clock Signals
input
    reset,                      // system reset
    clk;                        // system clock
```

```
//Test Insertion Input Signals
input
    scan_in0,                       // test scan mode data input
    scan_en,                        // test scan mode enable
    test_mode;                      // test mode select


//Test Insertion Output Signals
output
    scan_out0;                      // test scan mode data output



//Input Signals
input                   wr_en;
input                   rd_en;
input          [`BRAM_ADDR_SIZE-1:0] addr;
input                   [0:`DATA_WIDTH-1] wr_data;
input                 cs;

//Output Signals
output   reg        [0:`DATA_WIDTH-1] rd_data;
output                ack;
```

```
//memory
reg [`DATA_WIDTH-1:0] memory [0:`MEMORY_SIZE-1];


//Reg
reg ack_reg;




assign ack = ack_reg;


always @(posedge clk)
        begin
                if(wr_en)
                        begin
                                memory[0] <= wr_data;
                                ack_reg <= 1'b1;
                        end
                else
                        begin
                                ack_reg <= 1'b0;
                        end
        end


always @(posedge clk)
        begin
```

```
                    if(rd_en)
                              begin
                                        rd_data <= memory[0] ;
                              end
                    else
                              begin
                                        //ack_reg <= 1'b0;
                              end
          end


endmodule // bram
```

A.3 BRAM_MSG

```
module bram_msg (
           reset,
           clk,
           scan_in0,
           scan_en,
           test_mode,
           scan_out0,
           wr_en,
           cs,
           ack,
           addr,
           wr_data,
           rd_data,
           rd_en
       );

//Include Files
`include "./include/config.h"

///Clock Signals
input
    reset,                      // system reset
    clk;                        // system clock
```

```
//Test Insertion Input Signals
input
    scan_in0,                    // test scan mode data input
    scan_en,                     // test scan mode enable
    test_mode;                   // test mode select


//Test Insertion Output Signals
output
    scan_out0;                   // test scan mode data output


//Input Signals
input                 wr_en;
input                 rd_en;
input        [`BRAM_ADDR_SIZE-1:0]  addr;
input             [`MSG_WIDTH-1:0]  wr_data;
input               cs;

//Output Signals
output  reg       [`MSG_WIDTH-1:0]  rd_data;
output              ack;
```

```
//memory
reg [`MSG_WIDTH-1:0] memory [0:`MEMORY_SIZE-1];


//Reg
reg ack_reg;




assign ack = ack_reg;


always @(posedge clk)
        begin
                if(wr_en)
                        begin
                                memory[0] <= wr_data;
                                ack_reg <= 1'b1;
                        end
                else
                        begin
                                ack_reg <= 1'b0;
                        end
        end

always @(posedge clk)
        begin
```

```
                if(rd_en)
                        begin
                                rd_data <= memory[0] ;
                                //ack_reg <= 1'b1;
                        end
                else
                        begin
                                //ack_reg <= 1'b0;
                        end
        end


endmodule // bram
```

A.4 Interface

```
`include "./include/config.h"
interface intf_cnt (input bit clk , input bit reset);

        bit [`STATE_SIZE_2-1:0] address;
        bit wr_stb;
        bit [`STATE_SIZE_2-1:0] chip_select;
        bit [`DATA_WIDTH-1:0] wr_data;
        bit [`MSG_WIDTH-1:0] msg;
        bit [`DATA_WIDTH-1:0] o_data;
        bit wr_ack;
        bit done;
        bit div_done;
        bit [0:6] str_len;





//clocking cb @(posedge clk);

//      output address;
//      output wr_stb;
//      output rd_stb;
//          output  chip_select;
//          output  wr_data;
//          output  rd_ack;
//          input   rd_data;
//          input   wr_ack;
//endclocking

endinterface
```

A.5 Sequence

```
class simple_sequence_item extends uvm_sequence_item;
        // integer i = 1'b1;
         reg [0:63] e [3200];
         bit [0:63] n [3200];
         bit [0:63] d [3200];
         bit [0:63] o_data[3200];
//      reg [8-1:0] m [1024];
         bit [0:6] str_len[$];
        string arr [3200];
         string m[$];
         rand integer index;



        `uvm_object_utils_begin(simple_sequence_item)
                //`uvm_field_array_int(e, UVM_ALL_ON)
                //`uvm_field_int(d, UVM_ALL_ON)
                //`uvm_field_int(m, UVM_ALL_ON)
                `uvm_field_int(index ,UVM_ALL_ON)
                //`uvm_field_int(index ,UVM_ALL_ON)
        `uvm_object_utils_end



        function new (string name = "simple_sequence_item");
```

```
                super.new(name);
        endfunction

        constraint cons { index inside {[0:3165]};} ///change
           the constraints value as per the depth of the
           memory

endclass



//——————————————————————————————————————//
//———— Sequence 1 ———//
//——————————————————————————————————————//
class rsa_sequence1 extends uvm_sequence #(
   simple_sequence_item);
        `uvm_object_utils(rsa_sequence1)

        function new (string name = "rsa_sequence1");
                super.new(name);
        endfunction

virtual task body();
        string w;
        integer file;
        int i = 0;
        simple_sequence_item seq_item;
        seq_item = simple_sequence_item::type_id::create("
           seq_item");

        $readmemh("e.txt", seq_item.e);
        $readmemh("n.txt", seq_item.n);
        $readmemh("d.txt", seq_item.d);

        file = $fopen("m.txt","r");

        while (!$feof(file))
                begin
                $fgets(w, file);
        //      $display("len : %d", w.len());
                seq_item.str_len.push_back(w.len());
                seq_item.arr[i] = w;
                i = i+1;
                //seq_item.m.push_back(w);
                //$display("len : %d", seq_item.str_len[0]);
                //$display("%p",seq_item.arr);
                end

$fclose(file);

        repeat(1000)
```

```
                    begin
                              seq_item.randomize(index);
                              repeat(3) ///one sequence is made up
                                 of sending modulus , exponent ,
                                 message data (Have another repeat
                                 statement to make this run 100000
                                 times with different index value
                              //forever

                                        begin
                                        start_item(seq_item);
                              //        seq_item.print();
                                        finish_item(seq_item);
                                        get_response(seq_item);
                                        //seq_item =
                                           simple_sequence_item::
                                           type_id::create("seq_item")
                                           ;
                                        end
                    end
                    #200;
endtask


endclass : rsa_sequence1
```

```
//———————————————————————————————————————//
//——— Sequence 2 ———//
//———————————————————————————————————————//
class rsa_sequence2 extends uvm_sequence #(
   simple_sequence_item);
        `uvm_object_utils(rsa_sequence2)

        function new (string name = "rsa_sequence2");
                super.new(name);
        endfunction

virtual task body();
        string w;
        integer file;
        int i = 0;
        simple_sequence_item seq_item;
        seq_item = simple_sequence_item::type_id::create("
           seq_item");

        $readmemh("e.txt", seq_item.e);
        $readmemh("n.txt", seq_item.n);
        $readmemh("d.txt", seq_item.d);
```

```
file = $fopen("m.txt","r");

while (!$feof(file))
        begin
        $fgets(w,file);
//      $display("len : %d", w.len());
        seq_item.str_len.push_back(w.len());
        seq_item.arr[i] = w;
        i = i+1;
        //seq_item.m.push_back(w);
        //$display("len : %d", seq_item.str_len[0]);
        //$display("%p",seq_item.arr);
        end

$fclose(file);

repeat(1000)
        begin
                seq_item.randomize(index);
                repeat(3) ///one sequence is made up
                   of sending modulus , exponent ,
                   message data (Have another repeat
                   statement to make this run 100000
                   times with different index value
                //forever
```

```
                                        begin
                                        start_item(seq_item);
                              //        seq_item.print();
                                        finish_item(seq_item);
                                        get_response(seq_item);
                                        //seq_item =
                                           simple_sequence_item::
                                           type_id::create("seq_item")
                                           ;
                                        end
                    end
                    #200;
endtask



endclass : rsa_sequence2



//--------------------------------------------------------------------//
//------ Sequence 3 ---//
//--------------------------------------------------------------------//
class rsa_sequence3 extends uvm_sequence #(
   simple_sequence_item);
         `uvm_object_utils(rsa_sequence3)
```

```
        function new (string name = "rsa_sequence3");
                super.new(name);
        endfunction

virtual task body();
        string w;
        integer file;
        int i = 0;
        simple_sequence_item seq_item;
        seq_item = simple_sequence_item::type_id::create("
           seq_item");

        $readmemh("e.txt", seq_item.e);
        $readmemh("n.txt", seq_item.n);
        $readmemh("d.txt", seq_item.d);

        file = $fopen("m.txt","r");

        while (!$feof(file))
                begin
                $fgets(w, file);
        //      $display("len : %d", w.len());
                seq_item.str_len.push_back(w.len());
                seq_item.arr[i] = w;
```

```
                    i = i+1;
                    //seq_item.m.push_back(w);
                    //$display("len : %d", seq_item.str_len[0]);
                    //$display("%p",seq_item.arr);
                    end

$fclose(file);

          repeat(1000)
                    begin
                              seq_item.randomize(index);
                              repeat(3) ///one sequence is made up
                                 of sending modulus , exponent ,
                                 message data (Have another repeat
                                 statement to make this run 100000
                                 times with different index value
                              //forever

                                        begin
                                        start_item(seq_item);
                              //        seq_item.print();
                                        finish_item(seq_item);
                                        get_response(seq_item);
                                        //seq_item =
                                           simple_sequence_item::
```

```
                                                         type_id::create("seq_item")
                                                         ;
                                                     end
                                end
                                #200;
endtask



endclass : rsa_sequence3

//——————————————————————————————————————————————————————//
//—————— Sequence 4 ———//
//——————————————————————————————————————————————————————//
class rsa_sequence4 extends uvm_sequence #(
   simple_sequence_item);
         `uvm_object_utils(rsa_sequence4)

         function new (string name = "rsa_sequence4");
                  super.new(name);
         endfunction

virtual task body();
         string w;
         integer file;
         int i = 0;
```

```
        simple_sequence_item seq_item;
        seq_item = simple_sequence_item::type_id::create("
           seq_item");

        $readmemh("e.txt", seq_item.e);
        $readmemh("n.txt", seq_item.n);
        $readmemh("d.txt", seq_item.d);

        file = $fopen("m.txt","r");

        while (!$feof(file))
                begin
                $fgets(w, file);
        //      $display("len : %d", w.len());
                seq_item.str_len.push_back(w.len());
                seq_item.arr[i] = w;
                i = i+1;
                //seq_item.m.push_back(w);
                //$display("len : %d", seq_item.str_len[0]);
                //$display("%p",seq_item.arr);
                end

$fclose(file);

        repeat(1000)
```

```
                begin
                        seq_item.randomize(index);
                        repeat(3) ///one sequence is made up
                           of sending modulus , exponent ,
                           message data (Have another repeat
                           statement to make this run 100000
                           times with different index value
                        //forever

                                begin
                                start_item(seq_item);
                        //      seq_item.print();
                                finish_item(seq_item);
                                get_response(seq_item);
                                //seq_item =
                                   simple_sequence_item::
                                   type_id::create("seq_item")
                                   ;
                                end
                end
                #200;
endtask


endclass : rsa_sequence4
```

```
//--------------------------------------------------------------//
//------ Sequence 5 ---//
//--------------------------------------------------------------//
class rsa_sequence5 extends uvm_sequence #(
   simple_sequence_item);
        `uvm_object_utils(rsa_sequence5)

        function new (string name = "rsa_sequence5");
                super.new(name);
        endfunction

virtual task body();
        string w;
        integer file;
        int i = 0;
        simple_sequence_item seq_item;
        seq_item = simple_sequence_item::type_id::create("
           seq_item");

        $readmemh("e.txt", seq_item.e);
        $readmemh("n.txt", seq_item.n);
        $readmemh("d.txt", seq_item.d);
```

```
file = $fopen("m.txt","r");

while (!$feof(file))
        begin
        $fgets(w, file);
//      $display("len : %d", w.len());
        seq_item.str_len.push_back(w.len());
        seq_item.arr[i] = w;
        i = i+1;
        //seq_item.m.push_back(w);
        //$display("len : %d", seq_item.str_len[0]);
        //$display("%p",seq_item.arr);
        end

$fclose(file);

repeat(1000)
        begin
                seq_item.randomize(index);
                repeat(3) ///one sequence is made up
                   of sending modulus , exponent ,
                   message data (Have another repeat
                   statement to make this run 100000
                   times with different index value
                //forever
```

```
                                        begin
                                        start_item(seq_item);
                              //        seq_item.print();
                                        finish_item(seq_item);
                                        get_response(seq_item);
                                        //seq_item =
                                           simple_sequence_item::
                                           type_id::create("seq_item")
                                           ;
                                        end
                    end
                    #200;
endtask
endclass : rsa_sequence5



//------------------------------------------------------------------//
//------ Top Level Sequence---//
//------------------------------------------------------------------//
class rsa_sequence extends uvm_sequence #(simple_sequence_item
   );
         `uvm_object_utils(rsa_sequence)
```

```
        function new (string name = "rsa_sequence");
                super.new(name);
        endfunction

rsa_sequence1 seq1 ;
rsa_sequence2 seq2 ;
rsa_sequence3 seq3 ;
rsa_sequence4 seq4 ;
rsa_sequence5 seq5 ;

virtual task body();
        seq1 = rsa_sequence1::type_id::create("seq1");
        seq2 = rsa_sequence2::type_id::create("seq2");
        seq3 = rsa_sequence3::type_id::create("seq3");
        seq4 = rsa_sequence4::type_id::create("seq4");
        seq5 = rsa_sequence5::type_id::create("seq5");

        m_sequencer.set_arbitration(UVM_SEQ_ARB_USER);

        fork

        begin
                seq1.start(m_sequencer, this, 100 );
        end
        begin
                seq2.start(m_sequencer, this, 200 );
        end

        begin
                seq3.start(m_sequencer, this, 300 );
        end

        begin
                seq4.start(m_sequencer, this, 400 );
        end

        begin
                seq5.start(m_sequencer, this, 500 );
        end

        join

endtask : body

endclass : rsa_sequence
```

A.6 Sequencer

```
class rsa_sequencer extends uvm_sequencer #(
   simple_sequence_item);
        `uvm_component_utils(rsa_sequencer)

        function new (string name = "rsa_sequencer" ,
           uvm_component parent=null);
                super.new(name, parent);
        endfunction

endclass
```

A.7 Driver

```
import "DPI" function void encrypt(input string msg , input
   longint e_1 , n_1 , d_1);
class driver extends uvm_driver #(simple_sequence_item);
`include "./include/config.h"
`uvm_component_utils(driver)
virtual intf_cnt vif;
string msg ;
longint e_1 , n_1 , d_1;



// ——————————————Covergroup 1————————————//

        covergroup cov_1 ;
        chip_select:    coverpoint vif.chip_select{
        bins b1 = { 1 }; //exp_mem
        bins b2 = { 2 }; //mod_mem
        bins b3 = { 3 }; //msg_mem
        }
        endgroup: cov_1

// ————————————————————————————————————//
// ——————————————Covergroup 2————————————————//

        covergroup cov_2  ;
        input_data:     coverpoint vif.wr_data{
        }
        endgroup: cov_2

// ——————————————————————————————————————————//




        function new (string name = "driver" , uvm_component
           parent=null);
        super.new(name,parent);
        cov_1 = new();
        cov_2 = new();

        endfunction



        function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        void'(uvm_config_db #(virtual intf_cnt)::get(this,"*",
           "intf_vi", vif));
```

```
$display("Build phase driver");
endfunction

task run_phase (uvm_phase phase);
super.run_phase (phase);
$display("Run phase driver start");
start();
$display("Run phase driver end");
endtask

task start();
integer file;
integer r;
int state = `EXPONENT;
int counter;
int f_flag = 1'b1;
int flag = 1'b0;
int c_flag = 1'b0;
simple_sequence_item seq_item;
        forever
        begin
        seq_item_port.get_next_item(seq_item);
        if(f_flag == 1'b1)
                begin
                e_1 = seq_item.e[seq_item.index] ;
```

```
                        n_1 = seq_item.n[seq_item.index];
                        d_1 = seq_item.d[seq_item.index];
                        uvm_config_db #(longint) :: set (null,
                            "", "n", n_1);///send the n and d
                           value to config_db to be used by
                           monitor later
                        uvm_config_db #(longint) :: set (null,
                            "", "d", d_1);
                        encrypt(seq_item.arr[seq_item.index],
                           e_1,n_1,d_1);
                        f_flag = 1'b0;
                        end
        @(posedge vif.clk)
                begin
                case(state)
                `EXPONENT:
                 begin
                        vif.chip_select = 3'b001;
                        #10
                        vif.wr_stb = 1'b1;
                        vif.address = 3'b000;
                        vif.wr_data = seq_item.e[seq_item.
                           index];
                        state=`MODULUS;
                        flag = 1'b1;
                        cov_1.sample();
                        //        uvm_report_info(get_full_name
                           (),"Sending exponent data",UVM_LOW)
                           ;
                end
                `MODULUS:
                 begin
                        vif.chip_select = 3'b010;
                        #10
                        vif.wr_stb = 1'b1;
                        vif.address = 3'b001;
                        vif.wr_data = seq_item.n[seq_item.
                           index];
                        state=`MESSAGE;
                        cov_1.sample();
                        cov_2.sample();
                        flag = 1'b1;
                        //        uvm_report_info(get_full_name
                           (),"Sending Modulus data ",UVM_LOW)
                           ;
                end
                `MESSAGE:
                 begin
                        vif.chip_select = 3'b011;
                        #10
```

```
                    vif.wr_stb = 1'b1;
                    vif.address = 3'b010;
                    vif.msg = seq_item.arr[seq_item.index
                      ];
                    vif.str_len = seq_item.str_len[
                      seq_item.index];
                    cov_1.sample();
                    //      uvm_report_info(get_full_name
                      (),"Sending Message data ",UVM_LOW)
                      ;
                    c_flag = 1'b1;
                    //state = `WAIT_FOR_DONE;
                    counter = 1'b1;
            end

    endcase
    //end
    end
    @(posedge vif.wr_ack)
    begin
            vif.chip_select = 3'b000;
            #10
            vif.wr_stb = 1'b0;
            seq_item_port.item_done();
            //          uvm_report_info(get_full_name(),"Done
                Sending data",UVM_LOW);
    end

    if(flag)
    begin
            flag = 1'b0;
            seq_item_port.put(seq_item);
    end
    if(c_flag)
    begin
            c_flag = 1'b0;
            @(posedge vif.done)
            begin
                    //uvm_report_info(get_full_name(),"
                        Start Next sequence ",UVM_LOW);
                    f_flag = 1'b1;
                    seq_item_port.put(seq_item);
            end
    end
    else
    begin
    //          uvm_report_info(get_full_name(),"Outside C-
        flag",UVM_LOW);
    end
```

```
            if(counter==1'b1)
            begin
                    repeat(1000)
                    @(posedge vif.clk);
                    counter = 1'b0;
                    state = `EXPONENT;
            end
            end
endtask
endclass
```

A.8 Monitor

```
import "DPI" function void decrypt(input longint n_1 , d_1 );
class monitor extends uvm_monitor;
        int bar;
        string name;
       `uvm_component_utils(monitor)
       virtual intf_cnt vif;
       longint d_1 , n_1;
       simple_sequence_item seq_item_mon;
       uvm_analysis_port #(simple_sequence_item)
          collect_mon_port;

// ——————————————————Covergroup 3——————————————————//

       covergroup cov_3;
               output_data:    coverpoint vif.o_data{
                                                          }
               // cross vif.wr_data, vif.o_data;
   endgroup: cov_3

// ——————————————————————————————————————————————————————//

       function new (string name = "monitor" , uvm_component
          parent=null);
```

```
                super.new(name,parent);
                cov_3 = new();
        endfunction

        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                void'(uvm_config_db #(virtual intf_cnt)::get(
                   this,"*","intf_vi", vif));
                collect_mon_port = new(.name("collect_mon_port
                   "), .parent(this));
        endfunction

task run_phase (uvm_phase phase);
integer str_len [1023];
integer counter = 0;
integer exponent , modulus  ;
super.run_phase (phase);
seq_item_mon = simple_sequence_item::type_id::create( .name("
   seq_item_mon"), .contxt(get_full_name()));
forever
begin
@(posedge vif.clk)
        begin
                if(vif.div_done == 1'b1)
                        begin
```

```
                                        seq_item_mon.o_data[counter] =
                                            vif.o_data;
                                        //cov_2.sample();
                                        //$display("DUT value is : %d
                                           ",seq_item_mon.o_data[
                                           counter]);
                                        //$display("Counter value is :
                                            %d",counter);
                                        counter = counter + 1'b1;
                                end
                        if(vif.done == 1'b1)
                                begin
                                        start_compare();
                                        cov_3.sample();
                                        counter = 1'b0;
                                end
        end
end
endtask
function void start_compare();
bit [0:2047] w;
bit [0:2047] mem [1024];
integer file;
integer fp;
integer counter = 0;
integer flag;
// ////////////////////////////////// get private modulus and
   exponent
if (uvm_config_db #(longint) :: get (null, "", "n", n_1))
        begin
           //$display("n is %d",n_1);
          //`uvm_info ("ENV", $sformatf ("Found %d", n_1),
             UVM_MEDIUM)
        end
if (uvm_config_db #(longint) :: get (null, "", "d", d_1))
        begin
           //$display("d is %d",d_1);
          //`uvm_info ("ENV", $sformatf ("Found %d", d_1),
             UVM_MEDIUM)
        end

// /////////////////////////////////////////////// write encrypt
   key
file = $fopen("encrypt_key.txt","r");
while (!$feof(file))
        begin
                fp=$fscanf(file ,"%d\n",w);
                //$display("Model data is %",w);
                mem[counter] = w;
                counter = counter + 1'b1;
                //$display("Model data is %d",mem[counter]);
        end
$fclose(file);

///////////////////////////////////////////////////////comparing dut
    and model value
for(int i = 0; i<counter ;i=i+1)
        begin
                if(mem[i] == seq_item_mon.o_data[i])
                        begin
                                flag = 1'b1;



                        //      $display("Model data is %d",
                          mem[i]);
                        //      $display("Dut data is %d",
                          seq_item_mon.o_data[0]);
                        end
                else
                        begin
                        //      $display("Model data and dut
                          not matching");
                        //      $display("Model data is %d",
                          mem[0]);
```

```
                    //        $display("Dut data is %d",
                       seq_item_mon.o_data[0]);
                    end


        end
        if(flag == 1'b1)
                    begin
                    $display("Model data and dut matching"
                       );
                    decrypt(n_1,d_1);
                    flag = 1'b0;
                    end
        else
                    begin
                    $display("Model data and dut not
                       matching");
                    end

endfunction
endclass
```

A.9 Agent

```
class agent_1 extends uvm_agent;
        rsa_sequencer seq;
        driver drv;
        monitor mon;

        `uvm_component_utils(agent_1)

        uvm_analysis_port #(simple_sequence_item)
           collect_agnt_port;

        function new (string name = "agent_1" , uvm_component
           parent=null);
                super.new(name,parent);
        endfunction

        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                drv = driver::type_id::create(.name("drv"), .
                   parent(this));
                seq = rsa_sequencer::type_id::create(.name("
                   seq"), .parent(this));
                mon = monitor::type_id::create(.name("mon"), .
                   parent(this));
```

```
            collect_agnt_port = new(.name("
               collect_agnt_port"), .parent(this));
    endfunction

    function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            drv.seq_item_port.connect(seq.seq_item_export)
               ;//connecting sequencer and driver
            mon.collect_mon_port.connect(collect_agnt_port
               );
    endfunction



endclass
```

A.10 Scoreboard

```
class scoreboard extends uvm_scoreboard;
        `uvm_component_utils(scoreboard)

        uvm_analysis_export #(simple_sequence_item)
           collect_sb_port;
        uvm_tlm_analysis_fifo #(simple_sequence_item) fifo;



        function new (string name = "scoreboard" ,
           uvm_component parent=null);
                super.new(name,parent);
        endfunction

        function void build_phase (uvm_phase phase);
                super.build_phase(phase);
                collect_sb_port = new(.name("collect_sb_port")
                   , .parent(this));
                fifo = new(.name("fifo"), .parent(this));
        endfunction

        function void connect_phase (uvm_phase phase);
                super.connect_phase(phase);
                collect_sb_port.connect(fifo.analysis_export);
```

```
          endfunction



endclass
```

A.11 Environment

```
class rsa_env extends uvm_env;

        `uvm_component_utils(rsa_env)
        agent_1 agnt;
        scoreboard sb;

        function new (string name = "rsa_env" , uvm_component
           parent=null);
                super.new(name,parent);
        endfunction

        function void build_phase (uvm_phase phase);
                super.build_phase (phase);
                agnt = agent_1::type_id::create(.name("agnt"),
                    .parent(this));
                sb = scoreboard::type_id::create(.name("sb"),
                   .parent(this));
        endfunction

        function void connect_phase (uvm_phase phase);
                super.connect_phase (phase);
                agnt.mon.collect_mon_port.connect(sb.
                   collect_sb_port);
        endfunction

endclass
```

A.12 Test

```
class rsa_test extends uvm_test;

        //Registration
        `uvm_component_utils(rsa_test)
         rsa_env env;
        //Construction
        function new (string name = "rsa_test" , uvm_component
            parent=null);
                super.new(name,parent);
        endfunction

        function void build_phase (uvm_phase phase);
                super.build_phase (phase);
                env = rsa_env::type_id::create(.name("env"), .
                   parent(this));
        endfunction
        //execution
        task run_phase(uvm_phase phase);
                //super.run_phase;
                rsa_sequence seq;
                seq = rsa_sequence::type_id::create(.name("seq
                   "), .parent(this));
                phase.raise_objection(this);
                seq.start(env.agnt.seq);///start the sequencer
                phase.drop_objection(this);
        endtask

  endclass
```

A.13 Top

```
import uvm_pkg::*;
`include "uvm_macros.svh"
`include "./interface.sv"
`include "./sequence.sv"
`include "./sequencer.sv"
`include "./driver.sv"
`include "./monitor.sv"
`include "./scoreboard.sv"
`include "./agent.sv"
`include "./env.sv"
`include "./assertions.sv"
`include "./test.sv"



module test;


bit  clk;
bit  reset;


initial


        begin
        clk = 1'b0;
        reset = 1'b0;
        #10
        reset = 1'b1;
        #10
        reset = 1'b0;
        end

// 50 MHz clock
always
    #10 clk = ~clk ;

//Declare Interface
        intf_cnt intf(clk,reset);




//Declare DUT and connect interface
        rsa top(
                .reset(reset),
                .clk(clk),
                .address(intf.address),
                .wr_stb(intf.wr_stb),
                .chip_select(intf.chip_select),
                .wr_data(intf.wr_data),
                .msg(intf.msg),
`ifdef NETLIST
                .scan_in0(intf.scan_in0),
                .scan_en(intf.scan_en),
                .test_mode(intf.test_mode),
                .scan_out0(intf.scan_out0),
`else

                .wr_ack(intf.wr_ack),
                .done(intf.done),
                .o_data(intf.o_data),
                .str_len(intf.str_len),
                .div_done(intf.div_done)
`endif
            );

        rsa_assertions ass(intf);

initial
        begin
        uvm_config_db #(virtual intf_cnt)::set (uvm_root::get
           (),"*","intf_vi",intf);
        run_test("rsa_test");
        //$display("Data is %h", intf.wr_data);
        end
```

```
initial
begin

            $timeformat(-9,2,"ns", 16);
            $set_coverage_db_name("rsa");
`ifdef SDFSCAN
            $sdf_annotate("sdf/rsa_scan.sdf", test.top);
`endif
end


endmodule
```

A.14 Defines

```


`define ENC_32

`ifdef ENC_64
`define DATA_WIDTH 64
`define MODULUS_WIDTH 64
`endif
```

```
`ifdef ENC_32
`define DATA_WIDTH 32
`define MODULUS_WIDTH 32
`endif

`define EXP_WIDTH 10

`define STR_LEN 8
`define BRAM_ADDR_SIZE 2
`define A_DATA_WIDTH 100
`define B_DATA_WIDTH 8
`define MSG_WIDTH 800

`define MEMORY_SIZE 8
`define STATE_SIZE_1 3
`define STATE_SIZE_2 2
`define IDLE 3'b000
`define EXPONENT_BRAM_EN_WR 3'b001
`define MODULUS_BRAM_EN_WR 3'b010
`define MESSAGE_BRAM_EN_WR 3'b011
`define ACK 3'b100
`define MUL_DONE 3'b101
`define DIV_DONE 3'b110
`define S0 3'b000
```

```
`define S1 3'b001
`define S2 3'b010
`define S3 3'b011
`define S4 3'b100
`define S5 3'b101
`define EXPONENT_BASE_ADDRESS 3'b001
`define MODULUS_BASE_ADDRESS 3'b010
`define MESSAGE_BASE_ADDRESS 3'b011
`define MUL 3'b001
`define ADD 3'b010
`define DONE 3'b011
`define RESET 3'b000
`define EXPONENT 3'b001
`define MODULUS 3'b010
`define MESSAGE 3'b011
`define A 8'h41
`define B 8'h42
`define C 8'h43
`define D 8'h44
`define E 8'h45
`define F 8'h46
`define G 8'h47
`define H 8'h48
`define I 8'h49
`define J 8'h4A
```

```
`define K 8'h4B
`define L 8'h4C
`define M 8'h4D
`define N 8'h4E
`define O 8'h4F
`define P 8'h50
`define Q 8'h51
`define R 8'h52
`define S 8'h53
`define T 8'h54
`define U 8'h55
`define V 8'h56
`define W 8'h57
`define X 8'h58
`define Y 8'h59
`define Z 8'h5A
`define a 8'h61
`define b 8'h62
`define c 8'h63
`define d 8'h64
`define e 8'h65
`define f 8'h66
`define g 8'h67
`define h 8'h68
`define i 8'h69
```

```
`define j 8'h6A
`define k 8'h6B
`define l 8'h6C
`define m 8'h6D
`define n 8'h6E
`define o 8'h6F
`define p 8'h70
`define q 8'h71
`define r 8'h72
`define s 8'h73
`define t 8'h74
`define u 8'h75
`define v 8'h76
`define w 8'h77
`define x 8'h78
`define y 8'h79
`define z 8'h7A
```