Assessment of VLSI resources requirement for a sliced trusted platform module

Hala Mohammad Hamadeh

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd

Part of the Computer Engineering Commons

Recommended Citation

Hamadeh, Hala Mohammad, "Assessment of VLSI resources requirement for a sliced trusted platform module" (2015). Graduate Theses and Dissertations. 14569.
http://lib.dr.iastate.edu/etd/14569
Assessment of VLSI resources requirement for a sliced trusted platform module

by

Hala M Hamadeh

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Electrical Engineering

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Ahmed Kamal
Soma Chaudhuri

Iowa State University
Ames, Iowa
2015

Copyright © Hala M Hamadeh, 2015. All rights reserved.
## TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. iv

LIST OF TABLES ...................................................................................................................... vi

NOMENCLATURE ................................................................................................................... vii

ACKNOWLEDGEMENTS ......................................................................................................... viii

ABSTRACT ............................................................................................................................... ix

CHAPTER 1. INTRODUCTION ................................................................................................. 1

1.1 Overview .......................................................................................................................... 1

1.2 Solution Approaches ....................................................................................................... 3

CHAPTER 2. TRUSTED COMPUTING PLATFORMS ................................................................. 4

2.1 Trusted Computing Group .............................................................................................. 4

2.2 Trusted Platform Modules (TPM) .................................................................................. 4

2.2.1 Introduction .................................................................................................................. 4

2.2.2 Platform Configuration Register (PCR) ...................................................................... 6

2.2.3 TPM Key Types ........................................................................................................... 6

2.2.4 TPM Commands ........................................................................................................ 9

CHAPTER 3. BASIC TPM COMPONENTS ............................................................................ 11

3.1 RSA Cryptography Engine ............................................................................................. 11

3.1.1 Introduction .................................................................................................................. 11

3.1.2 RSA algorithm ............................................................................................................. 11

3.1.3 Primality Tests ............................................................................................................. 15

3.1.4 Extended Euclidean Algorithm ............................................................................... 16

3.1.5 Montgomery multiplication ..................................................................................... 16

3.2 Random Numbers Generation Engine ......................................................................... 19

3.2.1 Introduction .................................................................................................................. 19

3.2.2 Physical Unclonable Functions (PUFs) .................................................................... 19

3.2.3 Linear Feedback Shift Registers (LFSRs) ................................................................. 20

3.3 Hash Engine ................................................................................................................... 22

3.3.1 Introduction .................................................................................................................. 22

3.3.2 The Hash Function SHA1 ........................................................................................ 22

3.4 Symmetric Crypto Engine .............................................................................................. 26

3.4.1 Introduction .................................................................................................................. 26

3.4.2 AES algorithm ............................................................................................................ 26
CHAPTER 4. RNS PRELIMINARIES ................................................................. 28

4.1 Introduction .......................................................................................... 28
4.2 RNS Representation .............................................................................. 28
4.3 Selection of the Moduli ........................................................................ 30
4.4 Chinese Remainder Theorem ............................................................... 30

CHAPTER 5. APPROACH ............................................................................. 31

5.1 Problem Definition .............................................................................. 31
5.2 TPM-sliced Architecture ..................................................................... 31

CHAPTER 6. HARDWARE IMPLEMENTATION AND RESULTS .................... 33

6.1 Hardware Implementation .................................................................... 33
   6.1.1 Single TPM Implementation ......................................................... 33
   6.1.2 TPM-sliced implementation ......................................................... 44
6.2 Simulation and Synthesis Result ........................................................... 45
   6.2.1 Simulation results ......................................................................... 45
   6.2.2 Summary ...................................................................................... 52
   6.2.3 Synthesis results .......................................................................... 53

CHAPTER 7. CONCLUSIONS ....................................................................... 55

REFERENCES .............................................................................................. 56

APPENDIX: TPM SOURCE FILES ................................................................ 59
LIST OF FIGURES

Figure 2.1: *TPM major components.* ................................................................. 5

Figure 2.2: *Key hierarchy which is maintained by the TPM.* ........................................ 8

Figure 3.1: *RSA Architecture.* ................................................................................ 14

Figure 3.2: *RO-PUF basic structure.* ......................................................................... 20

Figure 3.3: *512-bit LFSR.* ......................................................................................... 21

Figure 3.4: Consecutive SHA-1 operation blocks for one iteration................................. 25

Figure 3.5: *AES algorithm.* ....................................................................................... 27

Figure 5.1: *General structure of the proposed architecture.* ........................................ 32

Figure 6.1: *Block diagram of the TPM_TOP module.* ................................................. 33

Figure 6.2: *Example flow for TPM Extend command.* ............................................... 34

Figure 6.3: *Input message block for the TPM_Extend command.* .............................. 35

Figure 6.4: *Output message block for the TPM_Extend command.* ........................... 35

Figure 6.5: *Example flow for TPM GetRandom command.* ....................................... 36

Figure 6.6: *Input message block for the TPM_GetRandom command.* ....................... 36

Figure 6.7: *Output message block for the TPM_GetRandom command.* .................... 37

Figure 6.8: *TPM_CreateWrapKey command flow.* .................................................... 37

Figure 6.9: *Input message block for the TPM_CreateWrapKey command.* ................. 38

Figure 6.10: *Output message block for the TPM_CreateWrapKey command.* .......... 39

Figure 6.11: *TPM_Sign command flow.* .................................................................... 40

Figure 6.12: *Input message block for the TPM_Sign command.* ............................... 40

Figure 6.13: *Output message block for the TPM_Sign command.* ............................ 41

Figure 6.14: *TPM_Sea1 command flow.* ................................................................... 42
Figure 6.15: Input message block for the TPM_Seal command. ........................................ 42
Figure 6.16: Input message block for the TPM_Unseal command. ........................................ 43
Figure 6.17: Output message block for the TPM_Seal command. ........................................ 44
Figure 6.18: Output message block for the TPM_Unseal command. ........................................ 44
Figure 6.19: Waveform for executing TPM_Extend command. ........................................ 45
Figure 6.20: Waveform for executing TPM_GetRandom command. ........................................ 46
Figure 6.21: Waveform for executing TPM_CreateWrapKey command. ................................ 47
Figure 6.22: Waveform for executing TPM_Sign command. .................................................. 48
Figure 6.23: Waveform for executing TPM_Seal command. .................................................. 49
Figure 6.24: Waveform for Input/Output signals of the TPM_Seal command. ......................... 49
Figure 6.25: Waveforms for Input “Data” and Output “SealedData” signals. ............................. 49
Figure 6.26: Waveform for executing TPM_Seal command. .................................................. 50
Figure 6.27: Waveform for Input/Output signals of the TPM_Seal command. ......................... 51
Figure 6.28: Waveforms for Input “Data” and Output “Secret” signals. ................................ 51
LIST OF TABLES

Table 3.1: *Montgomery Step.* ................................................................. 18
Table 3.2: *Initial Hash Values [27].* ...................................................... 23
Table 3.3: *Constant K Values [27].* ......................................................... 23
Table 4.1: *RNS representation for moduli set [2, 3, 5].* .............................. 29
Table 6.1: *Input output signals for TPM_TOP module.* .................................. 34
Table 6.2: *Execution time for TPM Commands.* ........................................... 52
Table 6.3: *Synthesis area results.* ............................................................. 54
Table 6.4: *Synthesis energy results* ............................................................ 54
## NOMENCLATURE

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>EK</td>
<td>Endorsement Key</td>
</tr>
<tr>
<td>PCR</td>
<td>Platform configuration registers</td>
</tr>
<tr>
<td>PUF</td>
<td>Physical Unclonable Function</td>
</tr>
<tr>
<td>RNG</td>
<td>Random Number Generator</td>
</tr>
<tr>
<td>RNS</td>
<td>Residue number system</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest Shamir Adleman</td>
</tr>
<tr>
<td>RTM</td>
<td>Root of Trust for Measurement</td>
</tr>
<tr>
<td>RTR</td>
<td>Root of Trust for Reporting</td>
</tr>
<tr>
<td>RTS</td>
<td>Root of Trust for Storage</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Secure Hash Algorithm 1</td>
</tr>
<tr>
<td>SRK</td>
<td>Storage Root Key</td>
</tr>
<tr>
<td>TCG</td>
<td>Trusted Computing Group</td>
</tr>
<tr>
<td>TCPA</td>
<td>Trusted Computing Platform Alliance</td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to express my deep sense of gratitude to my thesis supervisor, Akhilesh Tyagi, for giving me an opportunity to work with him in this research and providing me a tremendous level of support and cooperation throughout my research work and graduate studies. I would also like to thank the thesis committee member Ahmed Kamal, and Soma Chaudhuri for their valuable time in reviewing this thesis.

Finally, thanks to my family for their encouragement and to my husband for his hours of patience, respect and love.
Recent increases in cybercrime suggest questions such as: How can one trust a secure system? How can one protect private information from being stolen and maintain security? Trust in any system requires a foundation or root of trust. A root of trust is necessary to establish confidence that a machine is clean and that a software execution environment is secure. A root of trust can be implemented using the Trusted Platform Module (TPM), which is promising for enhancing security of general-purpose computing systems.

In cloud computing, one of the proposed approaches is to use homomorphic encryption to create $k$ program slices to be executed on $k$ different cloud nodes. The TPM at the cloud node can then also be distributed or sliced along the lines presented in this thesis.

In this work, we propose to increase TPM efficiency by distributing the TPM into multiple shares using Residue Number Systems (RNS). We then perform an evaluation of the silicon area, and execution time required for a sliced-TPM implementation and compares it to a single TPM. We characterize the execution time required by each TPM command using measurements obtained on ModelSim simulator.

Finally, we show that the proposed scheme improves TPM efficiency and that execution time of TPM commands was noticeably improved. In the case of 4 shares the required execution time of the TPM commands that involving RSA operation in each slice was decreased by 93%, and the area of each slice was decreased by 2.93% while the total area was increased by 74%. In the case of 10 shares the required execution time of the TPM commands that involving RSA operations in each slice was decreased by 99%, and the area of each slice was decreased by 3.3% while the total area was increased by 85%.
CHAPTER 1. INTRODUCTION

1.1 Overview

With increasing demand for Cloud Computing, maintaining security has never been more important. The common e-services must be accompanied by numerous security mechanisms to protect our personal information. For instance, if a user submits private data to an online site or to another user’s machine, there may be no mechanism to verify that that machine is trustworthy and has a Cyber-Secure environment. In other words, it is critical that a user be able to judge whether a system is trusted and that personal information cannot be stolen or misused. Trust in any system requires a foundation or a root of trust. A root of trust should establish that a machine is clean and that the execution environment is secure. A static root of trust is not a complete solution for confirming trust; it is only one component of the solution [1].

To establish and ensure an appropriate level of trust, an industry consortium of more than 160 companies, including HP, IBM, Microsoft, Intel, and many others, has taken responsibility for establishing trust through establishment of a not-for-profit organization, the Trusted Computing Group (TCG) [2], to propose Trusted Platform Module (TPM) specifications that have become ISO/IEC standards.

In its trusted-platform architecture, the TCG defines three distinctive roots of trust: a root of trust for measurement (RTM) to generate reliable integrity measurements, a root of trust for storage (RTS) to protect data and keys entrusted to the TPM, and a root of trust for reporting (RTR) responsible for reliably reporting integrity information to a verifier. Each root is a
computing engine entrusted with performing one or more of the main functions a trusted platform must provide [3].

A Trusted Platform Module (TPM) is a hardware module attached to a computer's motherboard. Using this hardware along with supporting software, a TPM’s goal is to ensure trust in end-user systems and to provide reliable and secure platforms for mission-sensitive individual or business operations. The TPM provides a root of trust, including a Root of Trust for Reporting, a Root of Trust for Storage, and cryptographic key generation, along with other functions. A Root of Trust for measurement (RTM) measures a digest of a sequence of events and reports to the TPM; using a boost platform.

To avoid a wide range of hardware-based attacks, a TPM chip should implement tamper-resistance techniques. Supplementing computers with this hardware module adds powerful functionality for their security. Because of the critical functionality provided by the TPM, there are many vendors fabricating TPM chips. Most new desktop and laptop PCs, for example, come with built-in TPMs [4].

The major cryptographic features supported in TPMs are random-number generation (RNG), SHA-1 (Hash algorithm), RSA, and asymmetric encryption/decryption. The RSA algorithm is used for asymmetric encryption/decryption and for digital signatures. To provide protected storage, the minimum recommended RSA key size is 2048 bits, but using such large keys tends to make the TPM relatively slow. For instance, Private Key operations can require more than 0.5 million cycles for completion [5]. In addition, a TPM chip may be inefficient not only with respect to time; it also tends to have high energy consumption and requires a large silicon area.
1.2 Solution Approaches

In cloud computing, a residue number system can be used to design a homomorphic encryption function representing a form of encryption directly performing a ciphertext operation, with the operation’s results automatically encrypted even if the secret key is unknown. Using a homomorphic encryption scheme in the cloud allows it to perform meaningful computations on the data even though it is encrypted [6].

We also can distribute the TPM to the cloud node using homomorphic encryption. Our approach is to divide the TPM into $k$ shares using a residue number system, then distribute the resulting slices to multiple nodes; we envision that these nodes can be hosted on $k$ separate clouds.

Our hypothesis is that such a sliced implementation would result in a lower execution time for the TPM operations. This will also increase the security of the TPM since the data is distributed.
CHAPTER 2. TRUSTED COMPUTING PLATFORMS

2.1 Trusted Computing Group

The Trusted Computing Group (TCG) is a not-for-profit international industry standards group. TCG was formed in 1999 when Compaq, HP, IBM, Intel and Microsoft founded the Trusted Computing Platform Alliance (TCPA) whose main goal was to promote trust and security in personal computing platforms. By 2003 over 200 companies had joined this organization, resulting in a Trusted Computing Group with a variety of structures but with the same goals as the TCPA [7]. The TCG is responsible for developing guidelines and specifications for adding trusted hardware to various computing platforms. These specifications help protect data, hardware, and other resources from theft, damage, or compromise without adversely impacting the rights of participating individuals or businesses. A Trusted Platform Module (TPM) is a major building block in achieving the goals of a TCG [2].

2.2 Trusted Platform Modules (TPM)

2.2.1 Introduction

In general, trust in computer infrastructure is defined as confidence that an entity will act as expected [8]. In the context of trusted platforms, trusted authority certifies that a platform with a particular identity can be trusted to operate as expected. The TCG has released a set of specifications specifying how a Trusted Platform must be constructed; it should contain a Core Root of Trust for Measurement, a Trusted Platform Module, and support software. A Trusted Platform Module (TPM) is a hardware chip attached to the main platform CPU, and it can securely store cryptographic keys used to authenticate the platform and protect its information.
Furthermore, to ensure that a platform remains trustworthy, a TPM can be employed to store platform measurements. A TPM also serve as a key generator and as a key management device [9].

![Figure 2.1: TPM major components.](image)

Figure 2.1 shows the major components of a TPM, including the following:

- **I/O**: This component manages information flow over the communication bus.
- **Key generation**: This component responsible for creating RSA key pairs and symmetric keys.
- **RSA engine**: The RSA algorithm is used for encryption and digital signatures.
- SHA-1 engine: hash capability, a trusted implementation of a hash algorithm. It is thus a primary component of the TPM.
- Random number generator (RNG): This is the source of random values required for key generation and nonce values.
- Platform configuration registers (PCRs): These are 160 bit values that are SHA-1 digests. The integrity metrics, obtained from measuring state, are stored in PCRs.
- Non-Volatile memory: This memory contains persistent state information, e.g., TPM identity information.
- Symmetric Crypto Engine: Used for encryption / decryption of command packets and to safely store internal TPM data on an external device.

2.2.2 Platform Configuration Register (PCR)

Platform Configuration Registers (PCRs) are sets of 20-byte storage locations used to store SHA-1 digests that represent platform integrity measurements. It is shielded inside the TPM where data is protected against interference and exposure. Each TPM contains 16 PCRs, 8 for hardware and 8 for software; These PCRs allow a secure representation of the host system's configuration metric that can be used to monitor any changes in the software and hardware configurations. The hash value of any PCR is updated by concatenating a new digest value with the original digest value, followed by an SHA-1 operation whose result is stored back to the PCR [10].

2.2.3 TPM Key Types

TPM keys are classified into three categories: migratable keys, non-migratable keys and certifiable migratable keys. Such classification determines whether a key can transfer from one
TPM to another. When a key is generated, the category value is established and cannot be changed [10].

- Non-migratable keys are bound and unique to a single TPM and can’t be exported from the TPM. Non-migratable keys are stored at a TPM-shielded location and are only usable within the TPM that generated them.
- Migratable keys are not bound to a single TPM; they are designed to protect data that can be used on more than one platform and can be generated either outside or inside the TPM. One advantage to keys of this type is that they can be restored from a defective platform onto a new platform.
- Certifiable Migratable Keys are midway between migratable keys and non-migratable keys.

2.2.3.1 TPM Key Hierarchy

The Trusted Computing Group (TCG) defines a key hierarchy with respect to to the Endorsement Key (EK) and the Storage Root Key (SRK). Figure 2.2 shows a typical TPM key hierarchy. The important TPM keys are defined as follows [10]:

- Endorsement Key (EK)

  At the end of TPM chip fabrication, the manufacturer creates a unique RSA private/public Endorsement key pair in a secure environment. This key is stored in the TPM and it is a private key that can only be used inside the TPM. It can’t leave the TPM. An Endorsement Key is used to decrypt messages associated with AIK generation and to decrypt owner authorization data.

- Storage Root Key (SRK)
The SRK is a key pair created inside the TPM during execution of its ownership protocol. The SRK is a non-migratable key; it therefore never leaves the TPM. A new SRK key pair can be created if the TPM is reset.

![Key hierarchy maintained by the TPM](image)

**Figure 2.2: Key hierarchy which is maintained by the TPM.**

- **Storage Key**
  An RSA private/public key. Storage Keys are used to encrypt others keys and data outside a TPM.

- **Signing Key**
  An asymmetric key that can be used to sign external data and messages or internal TPM state.

- **Binding Key**
  An asymmetric key that can be used to decrypt (unbind) data encrypted on another platform.
2.2.4 TPM Commands

The following are some useful TPM Commands [11]:

- **TPM_LoadKey2**
  
  This function is used to Load a key into the TPM to allow the TPM to use the key in other commands such as seal, unseal, sign, and unbind.

- **TPM_Sign**
  
  This command is used to sign data presented to the TPM and return the resulting digital signature.

- **TPM_UnBind**
  
  This command used to decrypt data that has been encrypted with a public key outside the TPM.

- **TPM_Seal**
  
  Encrypts data so that it can be decrypted (Unseal) only in the same TPM by sealing the data to specific PCR values.

- **TPM_Unseal**
  
  This command is used to decrypt sealed blobs inside the TPM.

- **TPM_Extend**
  
  This command is used to add new data to a PCR. Specifically, the PCR is extended according to the following formula:

  \[
  \text{PCR}_i^{\text{NEW}} \leftarrow \text{SHA}-1(\text{PCR}_i^{\text{Old value}} \parallel \text{data to add})
  \]

- **TPM_GetRandom**
  
  This command is used to obtain a random number from the TPM random-number generator. TPM uses the random number to generate keys and nonce.
- TPM_CreateWrapKey

This command is used to create a secure RSA key pair that can be locked to a specific PCR value.
CHAPTER 3. BASIC TPM COMPONENTS

3.1 RSA Cryptography Engine

3.1.1 Introduction

In 1977, R. Rivest, A. Shamir, and L. Adleman introduced the RSA cryptosystem [12], and it became the most widely-used public-key cryptosystem in the world. It is used to secure network traffic systems, e-mail, e-business, and e-commerce, and it can be used to provide data encryption and digital signature capability. Success of RSA cryptography ultimately rests on the idea that there is no efficient way to factor very large integers [13]. The RSA algorithm is based on exponentiation in a finite (Galois) field over integers modulo a prime.

RSA encryption/decryption is a slow process since it is based on arithmetic modulo large numbers. While many studies seeking a fast implementation of RSA have been carried out, as far as we know the Montgomery reduction algorithm has been the most efficient way to speed up the modular multiplications and squaring required during the exponentiation process [14].

3.1.2 RSA algorithm

The RSA algorithm defines a protocol to secure message exchanges in communication systems. An RSA cryptosystem includes three algorithms: Key Generation, an encryption algorithm, and a decryption algorithm.

3.1.2.1 Key Generation

The key generation algorithm is the most complex part of RSA whose purpose is to generate private and public keys used separately for RSA encryption and decryption operations. The steps for generating a key pair are given in algorithm 1 [15].
Algorithm 1: key generation

**Outputs:** a private key \((n, d)\), a public key \((n, e)\).

**Function:** keyGeneration

- Select two random prime numbers \(p\) and \(q\).
- Calculate \(n = p \times q\).
- Calculate \(\Phi(n) = (p - 1) \times (q - 1)\).
- Select integer \(e\) such that \(\gcd \left( \Phi(n), e \right) = 1\) \((1 < e < \Phi(n));\) where \(e\) & \(\Phi(n)\) are relatively prime.
- Calculate \(d = e^{-1} \mod \Phi(n)\) (by using the Extended Euclidian algorithm).

To generate secure RSA keys, the two prime numbers \(p\) and \(q\) should be very large and have at least 512 digits, and they should have approximately the same bit length. Finding two large random prime numbers \(p\) and \(q\) determines the approximate total key generation time; an efficient algorithm for generating such large prime numbers is given in detail in Chapter 4 [16]. The public key \((n, e)\) is a prime number chosen in the range \([3, \Phi(n)]\); it is usually chosen small to increase the efficiency of the exponentiation algorithm required by the encryption step. The TCG specifies that the RSA public exponent must be \(e = 2^{16}+1\). The private key \((n, d)\) and public key \((n, e)\) are multiplicative inverses modulo \(\Phi(n)\). Since the public key has a GCD of 1 with \(\Phi(n)\), Euclid’s extended algorithm can be used to calculate \(d\), as described in Section 3.1.4 [17].

**Encryption and Decryption algorithm.**

This algorithm is required for transforming a plaintext message into ciphertext and vice versa using public and private keys, as illustrated in Algorithm 2.
Algorithm 2: RSA encryption-decryption protocol

Alice encrypts a message \( m \) and sends it to Bob; Bob decrypts the message

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Alice received public key ((n, e)) from Bob.</td>
<td>1. Bob used its private key ((d)) to recover the message ( m ) from the ciphertext ( m = c^d \mod n ).</td>
</tr>
<tr>
<td>2. Alice represents the message ( m ) as an integer.</td>
<td></td>
</tr>
<tr>
<td>3. Alice computes ( c = m^e \mod n ).</td>
<td></td>
</tr>
<tr>
<td>4. Alice sends the ciphertext to Bob.</td>
<td></td>
</tr>
</tbody>
</table>

Modular exponentiation for a large number is performed as a series of modular multiplication and squaring operations. Many methods have been proposed to accelerate modular multiplication, and the most efficient method has been found to be a Montgomery reduction algorithm [17] introduced by Peter Montgomery. It replaces trial divisions by a modulus with a series of additions and divisions. The Montgomery algorithm is introduced in Section 3.1.5.

Example 3.1: RSA cryptosystem.

Assume that Bob wants to create a public-key cryptosystem, and Alice wants to use the public key to encrypt plaintext string 5423 and send it to Bob. Figure 3.1 shows a common procedure used by the RSA Architecture.

First, Bob should generate public and private keys via the following steps:

- Choose the primes \( p = 137 \) and \( q = 151 \)
- Compute \( n = p \cdot q = 20687 \), and \( \Phi(n) = (p - 1) \cdot (q - 1) = 20400 \).
- Select the public exponent \( e = 3503 \) such that it satisfies the constraint \( \gcd(e, \Phi(n)) = 1 \), and is small compared with \( \Phi(n) \).
  
  Compute \( d = e^{-1} \mod \Phi(n) = 6767 \).
- Publish the pair \((20687, 3503)\) and keep \( p, q, \) and \( d \) private.
When Alice receives Bob’s public key, she should:

- Compute the cyphertext as \(5423^e \mod n = 5423^{3503} \mod 20687 = 16631\)
- Send 16631 to Bob

When Bob receives this cyphertext, he should use his private key to recover the message.

- Compute \(16631^d \mod n = 16631^{6767} \mod 20687 = 5423\).

![RSA Architecture](image)

**Figure 3.1:** RSA Architecture.
3.1.3 Primality Tests

Primality tests are methods for determining whether or not an input is a prime number. These methods are classified into two categories. In the first category, for example, one can prove that a number is prime by using a trial division test and the Wilson's theorem test. In the second category, one can prove that a number is composite using, for example, the Miller-Rabin tests, the Fermat primality test, or the Solovay-Strassen primality test. The Miller-Rabin test is the most widely-used primality test because it yields an accurate result in a relatively short time [18].

In the present study, the Miller-Rabin test is implemented to specify the primality of a number. It relies on Fermat's theorem that states that, if $n$ is a prime number, then $(a^{n-1} \mod n) = 1$ for any integer $a$ in the range $[2, n - 1]$. A full algorithm implementing Miller and Rabin's primality test is shown below [19].

<table>
<thead>
<tr>
<th>Algorithm 4: Primarily Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: $n$, is an integer to be tested for primality</td>
</tr>
<tr>
<td><strong>Output</strong>: probably prime, or composite</td>
</tr>
<tr>
<td><strong>Function</strong>: PrimarilyTests</td>
</tr>
<tr>
<td>write $n - 1$ as $2^d \cdot d$ with $d$ odd by factoring powers of 2 from $n - 1$</td>
</tr>
<tr>
<td>LOOP 1: repeat $k$ times: $k$, determines the accuracy of the test</td>
</tr>
<tr>
<td>select a randomly in the range $[2, n - 1]$</td>
</tr>
<tr>
<td>$x \leftarrow a^d \mod n$</td>
</tr>
<tr>
<td>if ($x = 1$ or $x = n - 1$) then next LOOP 1</td>
</tr>
<tr>
<td>LOOP 2: repeat $s - 1$ times</td>
</tr>
<tr>
<td>$x \leftarrow x^2 \mod n$</td>
</tr>
<tr>
<td>if ($x = 1$) then return composite</td>
</tr>
<tr>
<td>if ($x = n - 1$) then next LOOP 1</td>
</tr>
<tr>
<td>return composite</td>
</tr>
<tr>
<td>return probably prime</td>
</tr>
</tbody>
</table>
3.1.4 Extended Euclidean Algorithm.

The extended Euclidean algorithm is an extension to the Euclidean algorithm for finding the greatest common divisor of two positive integers a and b; it can also be used to find integers x and y such that ax + by = gcd(a, b). The extended Euclidean algorithm is especially useful when a and b are coprime, since x is the modular multiplicative inverse of a mod b; similarly, y is the modular multiplicative inverse of b mod a [20]. Implementation of the extended Euclidean algorithm is given by Algorithm 5.

<table>
<thead>
<tr>
<th>Algorithm 5: Extended Euclidean Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> integers $\Phi(n)$ and $e$.</td>
</tr>
<tr>
<td><strong>Output:</strong> $d = e^{-1} \mod \Phi(n)$.</td>
</tr>
<tr>
<td><strong>Function:</strong> ExtendedEuclideanAlgorithm</td>
</tr>
<tr>
<td>Set $d = 0$, and newd = 1.</td>
</tr>
<tr>
<td>Set $r = \Phi(n)$, and newr = $e$.</td>
</tr>
<tr>
<td>while newr $\neq 0$</td>
</tr>
<tr>
<td>quotient = ($r / newr$).</td>
</tr>
<tr>
<td>$(d, \text{newd}) := (\text{newd}, d - \text{quotient} \times \text{newd})$</td>
</tr>
<tr>
<td>$(r, \text{newr}) := (\text{newr}, r - \text{quotient} \times \text{newr})$</td>
</tr>
<tr>
<td>if ($r &gt; 1$) return &quot;$e$ is not invertible&quot;.</td>
</tr>
<tr>
<td>if $d &lt; 0$ then $d := d + \Phi(n)$.</td>
</tr>
<tr>
<td>return $d$</td>
</tr>
</tbody>
</table>

3.1.5 Montgomery multiplication

The Montgomery modular multiplication algorithm was introduced in 1985 by Peter Montgomery. It computes modular multiplication $z = (a \times b) \mod (n)$ using simple operations such as additions, multiplications, and shifts. The idea is that, by using an n-residue representation of the operands, division by modulus n is replaced by division by a power of 2, accomplished by a simple shift operation. However, to avoid performing division-by-n operations, the costly operations come at the expense of the need to convert operands into
Montgomery’s domain; this time requirement is negligible compared with that of multi-modular multiplication [21], [22]. The Montgomery multiplication algorithm for calculating \( z = a \times b \mod (n) \) is outlined in Algorithm 6.

\[
\text{Algorithm 6: Montgomery multiplication algorithm}
\]

- Convert operands into Montgomery’s domain
  \[
  A = a \times r \mod (n) \quad \text{where } r = 2^k, \text{ and } n \text{ is the } k\text{-bit modulus.}
  \]
  \[
  B = b \times r \mod (n) \quad \text{where } r = 2^k, \text{ and } n \text{ is the } k\text{-bit modulus.}
  \]
- Calculates the Montgomery product of A and B
  \[
  \text{Input: } n \text{ where } n \text{ is the } k\text{-bit modulus, } A \text{ in Montgomery’s domain, }
  \]
  \[
  B \text{ in Montgomery’s domain}
  \]
  \[
  \text{Output: } Z = (A \times B \times r^\text{-1}) \mod (n)
  \]
  \[
  \text{Function: MontgomeryProduct (A,B)}
  \]
  \[
  \text{Set } Z[0] = 0
  \]
  \[
  \text{LOOP : repeat } k \text{ times: for } i=0 \text{ to } i= k-1.
  \]
  \[
  q_i = (Z[i]0 + A_i \times B0) \mod 2.
  \]
  \[
  Z[i+1] = (Z[i] + A_i \times B + q_i \times n) \div 2.
  \]
  \[
  \text{end LOOP}
  \]
  \[
  \text{return } Z[k]
  \]
- Backward conversions from Montgomery’s domain
  \[
  z = Z \times r^\text{-1} \mod (n) \text{ where } r = 2^k, \text{ and } n \text{ is the } k\text{-bit modulus}
  \]

It is important to note that these forward and backward conversions can be carried out using the MontgomeryProduct algorithm as follows:

\[
A = \text{MontgomeryProduct (a, r2 )} = (a \times r^2 \times r^\text{-1} ) \mod (n) = (a \times r ) \mod (n)
\]

\[
a = \text{MontgomeryProduct (A, 1)} = (a \times 1 \times r^\text{-1} ) \mod (n)= (a \times r^\text{-1} ) \mod (n).
\]

Example 3.2: using the Montgomery method to compute \( a \times b \mod (n) \), where \( a = 21, b = 7, \) and \( n=13; \)

- Convert operands into Montgomery’s domain
\[ r = 2^{\log_{2}13} = 2^{4} = 16 \]

\[ A = a \cdot r \pmod{n} = 21 \cdot 16 \pmod{13} = 11 = 1011_2 \]

\[ B = b \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8 = 1000_2 \]

- Calculate the Montgomery product of \(A\) and \(B\).

**Table 3.1: Montgomery Step.**

<table>
<thead>
<tr>
<th>(i)</th>
<th>(q_i = (Z[i]_0 + A_i \cdot B_0) \mod 2)</th>
<th>(Z[i+1] = (Z[i] + A_i \cdot B + q_i \cdot n) \div 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(q_0 = (Z[0]_0 + A_0 \cdot B_0) \mod 2)</td>
<td>(Z[1] = (Z[0] + A_0 \cdot B + q_0 \cdot n) \div 2)</td>
</tr>
<tr>
<td></td>
<td>(q_0 = (0+1*0) \mod 2) = 0)</td>
<td>(Z[1] = (0+1<em>8+0</em>13) \div 2 = 4)</td>
</tr>
<tr>
<td>1</td>
<td>(q_1 = (Z[1]_0 + A_1 \cdot B_0) \mod 2)</td>
<td>(Z[2] = (Z[1] + A_1 \cdot B + q_1 \cdot n) \div 2)</td>
</tr>
<tr>
<td></td>
<td>(q_1 = (0+1*0) \mod 2) = 0)</td>
<td>(Z[2] = (4+1<em>8+0</em>13) \div 2 = 6)</td>
</tr>
<tr>
<td>2</td>
<td>(q_2 = (Z[2]_0 + A_2 \cdot B_0) \mod 2)</td>
<td>(Z[3] = (Z[2] + A_2 \cdot B + q_2 \cdot n) \div 2)</td>
</tr>
<tr>
<td></td>
<td>(q_2 = (0+0*0) \mod 2) = 0)</td>
<td>(Z[3] = (6+0<em>8+0</em>13) \div 2 = 3)</td>
</tr>
<tr>
<td>3</td>
<td>(q_3 = (Z[3]_0 + A_3 \cdot B_0) \mod 2)</td>
<td>(Z[4] = (Z[3] + A_3 \cdot B + q_3 \cdot n) \div 2)</td>
</tr>
<tr>
<td></td>
<td>(q_3 = (1+1*0) \mod 2) = 1)</td>
<td>(Z[4] = (3+1<em>8+1</em>13) \div 2 = 12)</td>
</tr>
</tbody>
</table>

- Backward conversions from Montgomery’s domain

\[ z = Z \cdot r^{-1} \pmod{n} = 12 \cdot 16^{-1} \pmod{13} = 4 \] as it should.
3.2 Random Numbers Generation Engine

3.2.1 Introduction

High-quality random numbers are required for generating unpredictable cryptographic keys. Random-number generation approaches are classified into two methods: pseudo-random number generators (PRNGs) and true random number generators (TRNGs). The pseudo-random numbers are generated by a combination of binary flip-flops initially loaded with a small value; PRNG sequences are thus relatively predictable and not suitable for use in cryptographic keys. True random number generators (TRNGs) generate truly random number sequences by measuring random natural physical phenomena such as electronic and thermal noise; TRNGs are therefore unpredictable [23] [24]. In this work, random numbers were generated using Ring Oscillator Physical Unclonable Functions (PUFs) that produce random values from hidden delay information then we used this random numbers as seeds to Linear Feedback Shift Registers (LFSRs).

3.2.2 Physical Unclonable Functions (PUFs)

The concept of ring-oscillator-derived Physical Unclonable Functions (RO-PUFs) is based on the delay contrast between two identically-structured ring oscillators that occurs because each ring oscillator has a unique frequency resulting from variations in the hardware manufacturing process.

Figure 3.2 illustrates an RO-PUF circuit. The basic structure of a conventional RO-PUF is comprised of: n identically ring oscillators, an n-to-2 multiplexer, two counters, and one comparator. Each of the identical ROs oscillates with a different frequency, to generate an
unpredictable sequence of bits; one pair of ring oscillators is selected. The counters then count the oscillation cycles for a certain time period.

![RO-PUF basic structure](image)

**Figure 3.2: RO-PUF basic structure.**

Finally, the outputs of the counters are compared based on which oscillator has a higher frequency. If the upper counter shows a higher value than the lower counter, the corresponding output is set to ‘1’; otherwise, the output is set to ‘0’ [25] [26].

### 3.2.3 Linear Feedback Shift Registers (LFSRs)

The Linear Feedback Shift Register is a shift register whose input bit is a linear function of its previous state; the initial value of the LFSR is called the seed. LFSR is the most useful techniques used for generating pseudorandom numbers.

An example of a 512-bit LFSR is shown in Figure 3.3. The feedback is created by performing the exclusive-OR (XOR) of the outputs of selected flip-flops and then feeding this back to the input of least significant bit flip-flop.
In this work 512-bit pseudorandom numbers are generated using LFSR. The polynomial used for generating this sequence is as below [27].

\[ P(X) = X^{512} + X^{510} + X^{507} \]

**Figure 3.3:** 512-bit LFSR
3.3 Hash Engine

3.3.1 Introduction

A cryptographic hash function is a procedure that takes an arbitrary block of input data and returns a output shorter than the input with a fixed number of bits. The input data to be encoded is called the "message," and the output or “hash” values are called the message digest or simply the digest. While it is infeasible to recover the original text from a hash alone, it is also impossible that different inputs will produce an identical digest. These properties make hashes ideally suited to many aspects of cryptography, and there are many different types of hash functions with various security properties [27].

3.3.2 The Hash Function SHA1

The hash function known as SHA1” Secure Hashing Algorithm” was finalized in 1995, when a FIPS (Federal Information Processing Standard) was advanced by the US National Institute of Standards that specified SHA1. The SHA-1 algorithm is valid for messages with a size less than $2^{64}$ bits; if a message is longer than $2^{64}$ bits, it must be separated into several groups with each containing $2^{64}$ bits. Each message group will then consist of a block of size 512 bits, a word size of 32 bits, and a resultant message digest of 160 bits. SHA-1 can be described as operating in two stages: preprocessing and hash computation.

3.3.2.1 Preprocessing

Preprocessing consists of three steps:

- Pad the message with a single one followed by zeroes until the resulting block has 448 bits. Then the size of the original message is appended as an unsigned 64-bit integer.
- Parse the Padded Message into N m-bit blocks.
- Initialize the hash values $H_i^{(0)}$ to the specific constants defined in the SHA1 standard (shown in table 3.2).

### Table 3.2: Initial Hash Values [27].

<table>
<thead>
<tr>
<th>Intermediate Hash Variable</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>0x67452301</td>
</tr>
<tr>
<td>H1</td>
<td>0xEFCDAB89</td>
</tr>
<tr>
<td>H2</td>
<td>0x98BADCFE</td>
</tr>
<tr>
<td>H3</td>
<td>0x10325476</td>
</tr>
<tr>
<td>H4</td>
<td>0xC3D2E1F0</td>
</tr>
</tbody>
</table>

#### 3.3.2.2 SHA-1 hash Computation

SHA-1 uses a sequence of eighty constant 32-bit words, $K_0$, $K_1$,…, $K_{79}$ defined in the SHA1 standard (shown in table 3.3). In addition, it uses a sequence of logical functions, $f_0$, $f_1$,…, $f_{79}$, each operating on three 32-bit words, $x$, $y$, and $z$, to produce a 32-bit word as output. The function $f_t(x, y, z)$ is defined as follows [27]:

$$f_t(x, y, z) = \begin{cases} 
    Ch(x, y, z) = (x \land y) \oplus (\neg x \land z) & 0 \leq t \leq 19 \\
    Parity(x, y, z) = x \oplus y \oplus z & 20 \leq t \leq 39 \\
    Maj(x, y, z) = (x \land y) \oplus (x \land z) \oplus (y \land z) & 40 \leq t \leq 59 \\
    Parity(x, y, z) = x \oplus y \oplus z & 60 \leq t \leq 79.
\end{cases}$$

### Table 3.3: Constant $K$ Values [27].

<table>
<thead>
<tr>
<th>Range of $t$</th>
<th>$K$ Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 &lt; t &lt; 19$</td>
<td>0x5A827999</td>
</tr>
<tr>
<td>$20 &lt; t &lt; 39$</td>
<td>0x6ED9EBA1</td>
</tr>
<tr>
<td>$40 &lt; t &lt; 49$</td>
<td>0x8F1BBCDC</td>
</tr>
<tr>
<td>$60 &lt; t &lt; 79$</td>
<td>0xCA62C1D6</td>
</tr>
</tbody>
</table>
The process requires 4 rounds of 20 operations each, resulting in a total of 80 operations, to generate the final digest.

After preprocessing is completed, each message block, M(1), M(2), …, M(N) requires 4 rounds of 20 operations each, resulting in a total of 80 operations, to produce the final 160-bit message digest.

The message schedule is first prepared as follows:

\[
W_t = \begin{cases} 
M_t^{(i)} & 0 \leq t \leq 15 \\
\text{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leq t \leq 79 
\end{cases}
\]

Then 32-bit variables a, b, c, d, and e are initialized as follows:

\[
\begin{align*}
    a &= H_0^{(i-1)} \\
    b &= H_1^{(i-1)} \\
    c &= H_2^{(i-1)} \\
    d &= H_3^{(i-1)} \\
    e &= H_4^{(i-1)}
\end{align*}
\]

Sequential operations are next repeated 80 times. One such operation is illustrated in Figure 3.3.

Kt and Wt are constant values for iteration t and the t\textsuperscript{th} w-bit word of the message schedule, correspondingly. ROTL\(_{(i)}\) stands for circular shift by i positions to the left, and the function \(f_t(x,y,z)\) described above represents the non-linear function of the SHA-1.
Finally, the $i^{th}$ intermediate hash value $H^{(i)}$ is computed as follows:

$$
\begin{align*}
H^{(i)}_0 &= a + H^{(i-1)}_0 \\
H^{(i)}_1 &= b + H^{(i-1)}_1 \\
H^{(i)}_2 &= c + H^{(i-1)}_2 \\
H^{(i)}_3 &= d + H^{(i-1)}_3 \\
H^{(i)}_4 &= e + H^{(i-1)}_4 
\end{align*}
$$

After repeating these steps for each massage $M$ ($N$), a 160-bit message digest is produced by combining the hash value $H^{(N)}$. 

Figure 3.4: Consecutive SHA-1 operation blocks for one iteration
3.4 Symmetric Crypto Engine

3.4.1 Introduction

The symmetric cryptographic engine acts as a 128-bit block cipher engine. The term “symmetric cryptographic” refers to applying one cryptographic key for both decryption and encryption. The algorithm, the Advanced Encryption Standard (AES), is also known as Rijndael. It operates on blocks of 128 bits by using keys with lengths of 128 bits for the decryption and encryption processes [28].

3.4.2 AES algorithm

The AES algorithm operates on a 4×4 matrix of bytes called a State. At the beginning of the encryption process, the plaintext message is expanded into a State. Then, an initial Round Key operation is conducted on the State. This is followed by 10 rounds of transformation, each round consisting of all the following transformations, except that the final round also includes the SubBytes, ShiftRows and AddRoundKey transformations [29].

- **SubBytes**: a non-linear substitution step in which each byte is replaced with another from a lookup table.
- **ShiftRows**: a transposition step in which each row of the state is shifted cyclically a certain number of steps.
- **MixColumns**: a mixing operation operating on the columns of the state, combining the four bytes in each column
- **AddRoundKey**: each byte of the state is combined with the round key; each round key is derived from the cipher key using a key schedule.
After the cipher operations are performed on the State, the final value of the state is copied to the ciphertext. A complete algorithm is shown in Figure 3.4.
CHAPTER 4. RNS PRELIMINARIES

4.1 Introduction

A residue number system RNS is a non-weighted number system based on the Chinese remainder theorem of modular arithmetic. RNS represents large integer numbers as a set of relative smaller integers called residues. The main advantage of the RNS is absence of carry propagation between the columns in addition steps, resulting in improved high-speed computational performance. However, since RNS is a non-weighted system, magnitude-related operations such as division, magnitude comparison, and sign detection are more difficult to perform than are arithmetic operations such as additions, subtraction, and multiplication [30].

4.2 RNS Representation

A residue number system is completely defined by a set of relatively prime integers called moduli \( M = (m_1, m_2, \ldots, m_k) \), where \( m_i \) is the \( i^{th} \) modulus. The moduli should be pairwise relatively prime, meaning that, for every pair of moduli, the greatest common divisor is 1.

An integer \( X \) can be represented as a list of smaller integers called residues \([ x_1, x_2, \ldots, x_k ]\) where \( x_i \) is the \( i^{th} \) residue. The relationship between the modulus and the residues can be written as:

\[
X \mod (m_i) = x_i
\]

\[
X_M = [ x_1, x_2, \ldots, x_k ] = [X \mod (m_1), X \mod (m_2), \ldots, X \mod (m_k) ]
\]

The RNS system provides unique representations for integers in the range from 0 to the product of all the moduli. The moduli \((m_1, m_2, \ldots, m_k)\) has a dynamic range \( m_1 \times m_2 \times \cdots \times m_k \) [31].

Example 4.1: RNS numerical example.

Consider a residue number system defined by the moduli set \([2, 3, 5]\). The representation of the numbers is shown in Table 4.1.
Table 4.1: RNS representation for moduli set [2, 3, 5].

<table>
<thead>
<tr>
<th>X</th>
<th>[2, 3, 5]</th>
<th>X</th>
<th>[2, 3, 5]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The residues in table 4.1 uniquely identify a number for all numbers in the range from 0 to 29. For instance, the decimal number 10 is represented by the configuration [0, 1, 0] just as uniquely as binary 1010. However, outside that range, the RNS representation repeats itself. For example, the RNS representation of 31 is the same as that of 1. In addition, the moduli must be relatively prime to ensure unique representation within the dynamic range. For example, consider the moduli set is [2, 3, 9] not relatively prime since 3 and 9 have a common divisor of 3. We observe that the RNS representation for 6 = [0, 0, 6] is the same as the RNS representation for 24 = [0, 0, 6].
4.3  Selection of the Moduli

There are certain moduli more appealing than others; one example would be Mersenne numbers because they are efficient in their binary representation. All numbers of the form $2^k-1$ (where k is a prime), called Mersenne numbers, were first suggested by Mersenne [30].

The modulus values should be selected as close to one another as possible to limit the size of the largest modulus. Merrill suggests in [32] that the largest modulus should be of the form $2^k$ and the second largest modulus should be of the form $2^k-1$. The moduli should be relatively prime.

For example, a 17-bit number could be represented by the moduli set [32, 31, 15, 7]. We have

$$2^5 \times (2^5-1) \times (2^4-1) \times (2^3-1) = 2^{17} - \mathcal{O}(2^{14})$$

where $\mathcal{O}(2^{14})$ indicates a term on order of $2^{14}$. As a result, we have lost less than 1 bit of dynamic range representation.

4.4  Chinese Remainder Theorem

The uniqueness property is the consequence of a most important theory, the Chinese Remainder Theorem, that allows the computation of the equivalent number of an RNS representation [33].

Theorem 1: (Chinese Remainder Theorem)

Given a set of pairwise relatively prime moduli ($m_1,m_2, ..., m_k$), there exists exactly one integer that satisfies the conditions:

$$n \leq X < n + M$$

$$x_i = X \mod(m_i)$$

Where $i = 1, ..., k$, $M = m_1 \times m_2 \times \cdots \times m_k$ and $n$ is an integer.
CHAPTER 5. APPROACH

5.1 Problem Definition

As described in Chapter 2, the RSA Engine is one of the major components of the TPM. The primary advantage of using the RSA cryptography in the TPM is its increased security and convenience. However, the downside of this approach is that the TPM becomes a relatively slow device; the RSA operations are time-consuming and very complex.

In this thesis, we propose a new architecture to advance the speed of the TPM by breaking it into multiple slices based on a residue number system (RNS) and then distributing the resulting slices to multiple platforms.

5.2 TPM-sliced Architecture

In this work, we propose a fast TPM based on Residue Number Systems. In other words, we divided the TPM into multiple slices by reducing the Key size for each share based on the RNS. The TPM consists of three major components: the RSA Engine, the HASH Engine, and the AES Engine. To divide the TPM we should divide each component by itself. However, since the RSA is the most expensive computation of all there is no reason to distribute or slice the Hash function or the AES function.

RNS exhibits several advantages that facilitate fast, parallel, implementation of long-integer arithmetic [34]. This makes RNS a good candidate for splitting the TPM. We know from [35] that slicing the RSA system into multiple shares using a residue number system is possible. For example, if we have a TPM that uses a 1024-bit RSA, we can divide it into 4 slices; each slide uses a 256-bit RSA Cryptosystem rather than a 1024-bit Cryptosystem. Figure 5.1 shows the structure of the proposed architecture.
The following steps explain the underlying process of the proposed architecture:

- Select a moduli-set \([m_1, m_2, m_3, m_4]\) where the moduli are relatively prime.
- Convert RSA keys \((N, d, e)\) to RNS representation.

  The residue sets are \([N_{m_1}, N_{m_2}, N_{m_3}, N_{m_4}], [d_{m_1}, d_{m_2}, d_{m_3}, d_{m_4}]\) and \([e_{m_1}, e_{m_2}, e_{m_3}, e_{m_4}]\), respectively.

If a message \(X\) requires encryption or decryption using the RSA cryptosystem,

- Convert the message to RNS representation. The residue set is \([X_{m_1}, X_{m_2}, X_{m_3}, X_{m_4}]\).
- Divide each residue into the corresponding sliced-TPM.
- Compute the cyber-text for each message residue inside the sliced-TPM.
- Backward convert the encrypted/decrypted message from RNS representation into ordinary weighted numbers.

![Figure 5.1: General structure of the proposed architecture.](image)
CHAPTER 6. HARDWARE IMPLEMENTATION AND RESULTS

6.1 Hardware Implementation

For hardware design and implementation, we first implemented a small version of a single TPM that can perform a few commands. We then divided this TPM into k shares. The functionality of the proposed algorithm was coded in Verilog, synthesized using 0.05 µm CMOS technology, and simulated in Modelsim.

6.1.1 Single TPM Implementation

TPM\_TOP is the main module of the TPM. It controls the functions of 6 sub-modules: TPM\_Extend, TPM\_GetRandom, TPM\_CreateWrapKey, TPM\_Sign, TPM\_Seal, and TPM\_Unseal. The following sections describe the operation of each individual sub-module. A block diagram of the TPM\_TOP module is given in Figure 6.1.

![Block diagram of the TPM\_TOP module.](image)

Figure 6.1: Block diagram of the TPM\_TOP module.
This module has 7 input signals and 3 output signals. The input signals are \textit{clk}, \textit{reset}, \textit{start}, \textit{addr}, \textit{we}, \textit{oe}, and \textit{IN\_Bytes}, while the output signals are \textit{ready}, \textit{valid}, and \textit{Out\_Bytes}. The functions of these signals are described in Table 6.1.

\textbf{Table 6.1: Input output signals for TPM\_TOP module.}

<table>
<thead>
<tr>
<th>Pin name</th>
<th>Size (bit)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Clk}</td>
<td>1</td>
<td>Standard clock</td>
</tr>
<tr>
<td>\textit{Reset}</td>
<td>1</td>
<td>Reset all output to zero when reset is high</td>
</tr>
<tr>
<td>\textit{Start}</td>
<td>1</td>
<td>Start the functions</td>
</tr>
<tr>
<td>\textit{Addr}</td>
<td>8</td>
<td>Address Input</td>
</tr>
<tr>
<td>\textit{We}</td>
<td>1</td>
<td>Write Enable</td>
</tr>
<tr>
<td>\textit{Oe}</td>
<td>1</td>
<td>Output Enable</td>
</tr>
<tr>
<td>\textit{IN_Bytes}</td>
<td>32</td>
<td>Input message block</td>
</tr>
<tr>
<td>\textit{Valid}</td>
<td>1</td>
<td>Indicates that the source has put valid data on the Out Bytes</td>
</tr>
<tr>
<td>\textit{Ready}</td>
<td>1</td>
<td>Indicates that the module is ready to receive new data</td>
</tr>
<tr>
<td>\textit{Out_Bytes}</td>
<td>32</td>
<td>Output message block</td>
</tr>
</tbody>
</table>

6.1.1.1 \textit{TPM\_Extend} Module

This module implements the basic functions of the \textit{TPM\_Extend} command. This command modifies the current PCR digest relative to the 20-byte input digest. Figure 6.2 shows an example of flow between the operator and TPM.

\textbf{Figure 6.2: Example flow for TPM Extend command.}

Figure 6.3 Example for an input message block of the \textit{TPM\_Extend} command.
There are 5 input signals in this module. The Authorization Tag “defines the level of authorization”, the Parameter Size “the size of the incoming block”, the Ordinal “the command code of the TPM function”, the PCR index “points to shielded memory location”, and the Input Digest is a “20-byte digest”.

The resulting output command message block associated with this input is given in Figure 6.4.

![Figure 6.3: Input message block for the TPM_Extend command.](image)

![Figure 6.4: Output message block for the TPM_Extend command.](image)
6.1.1.2 **TPM\_GetRandom** Module

This module implements the basic functions of the **TPM\_GetRandom** command. This is a simple command that generates a random number; a 512-bit Linear Feedback Shift Register and 512-bit PUF are used to produce a wide sequence of random bits. Figure 6.5 shows an example of **TPM\_GetRandom** command flow between the TPM and a user application.

![Flow Diagram](image]

**Figure 6.5:** Example flow for **TPM GetRandom** command.

Figure 6.6 gives an example of an input message block for the **TPM\_GetRandom** command.

<table>
<thead>
<tr>
<th>Authorization Tag</th>
<th>0x00</th>
<th>0xC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Size</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>Ordinal</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>Bytes Requested</td>
<td>NB1</td>
<td>NB2</td>
</tr>
</tbody>
</table>

**Figure 6.6:** Input message block for the **TPM\_GetRandom** command.

There are 4 input signals in this module. The **Authorization Tag**, the **Parameter Size**, the **Ordinal**, and the **Bytes Requested** “specify the size of the random Data”. The resulting output command message block associated with this input is given in Figure 6.7.
Figure 6.7: Output message block for the TPM\_GetRandom command.

6.1.1.3 **TPM\_CreateWrapKey Module**

The **TPM\_CreateWrapKey** command generates an RSA key, then encrypts the private key using a parent key provided in the input message. Figure 6.8 shows an example of **TPM\_CreateWrapKey** command flow between the TPM and a user application.

Figure 6.8: **TPM\_CreateWrapKey command flow.**

Figure 6.9 gives an example of an input message block for the **TPM\_CreateWrapKey** command.
There are 8 input signals in this module: Authorization Tag, Parameter Size, Ordinal, key Handle” to handle a loaded key that can perform key wrapping”, nonceOdd” Nonce generated by the system associated with Key Handle”, TCG-Key “Information about key to be created, pubkey.keyLength and keyInfo” that consists of: Key to create, Digest at Creation, and Digest at Release. The resulting output command message block associated with this input is given in Figure 6.10.
Figure 6.10: Output message block for the TPM\_CreateWrapKey command.

6.1.1.4 **TPM\_Sign Module**

The *TPM\_Sign* command generates an RSA signature of up to 512-bit data.

Figure 6.11 shows an example of *TPM\_Sign* command flow between the TPM and a user application. Example of an input message block for the *TPM\_Sign* command is given in Figure 6.12.
There are 7 input signals in this module: Authorization Tag, Parameter Size, Ordinal, Key Handle” identifier of a loaded key that can perform digital signatures”, Area to Sign” The value to sign” and nonceOdd ”Nonce generated by the system associated with Key Handle”. The resulting output command message block associated with this input is given in Figure 6.13.
6.1.1.5 $TPM\_Seal \ \& \ \TPM\_Unseal$ modules

$TPM\_Seal$ command encrypts data using 1024-bit public key and then associates it with specific PCR values. While $TPM\_Unseal$ command decrypts the data if the PCR values match those at the time of sealing. Figure 6.14 shows an example of $TPM\_Seal$ command flow between the TPM and a user application.
Figure 6.14: *TPM Seal command flow.*

Figure 6.15 and Figure 6.16 show examples of input message blocks of the *TPM Seal* and *TPM Unseal* commands.

Figure 6.15: *Input message block for the TPM Seal command.*
There are 10 input signals in the \textit{TPM\_Seal} module: \textit{Authorization Tag}, \textit{Parameter Size}, \textit{Ordinal}, \textit{Key Handle}, \textit{encAuth} “The encrypted AuthData for the sealed data”, \textit{PCR\_Info\_Size}, \textit{PCR\_Info} “The PCR selection information”, \textit{Data Size}, \textit{Data}, and \textit{nonceOdd}. The resulting output command message block associated with this input is given in Figure 6.17.

![Figure 6.16: Input message block for the \textit{TPM\_Unseal} command.](image)

There are 7 input signals in the \textit{TPM\_Unseal} module: \textit{Authorization Tag}, \textit{Parameter Size}, \textit{Ordinal}, \textit{Key Handle}, \textit{TCG\_STORED\_DATA}” The encrypted data generated by \textit{TPM\_Seal}”, \textit{nonceEven} and \textit{nonceOdd}. The resulting output command message block associated with this input is shown in Figure 6.18.
6.1.2 TPM-sliced implementation

After implementing the single-block TPM, we divide it into $k$ shares as described in Chapter 5. We then used MATLAB to divide the TPM input messages and the RSA Keys. To evaluate the effect of selecting different numbers of shares, we divided the TPM into 4 and 10 shares.
6.2 Simulation and Synthesis Result

The results shown below are the waveforms generated from the **TPM_TOP** testbench module described in Appendix A. The **TPM_TOP** test bench simulation fully validated the design by constructing random TPM messages, passing them to the model, and comparing the output messages to the expected results. The output simulation was performed using the ModelSim Simulator.

6.2.1 Simulation results

This section gives the simulation results for various TPM Commands.

6.2.1.1 **TPM_Extend** Command

To test the **TPM_Extend** command, a sample input message block with the same structure as that shown in Figure 6.3 was passed to the **TPM_TOP** test bench. Figure 6.19 shows the simulation Waveform of **TPM_Extend** command. The clock period was 100 ns, and the output message was generated in about 9 μs.

![Figure 6.19: Waveform for executing TPM_Extend command.](image-url)
6.2.1.2 *TPM_GetRandom* command

structure as that shown in Figure 6.6 was passed to the TPM_TOP test bench. Figure 6.20 shows the simulation Waveform of TPM_GetRandom command. The clock period was 100 ns, and the output message was generated in about 33 us.

![Figure 6.20: Waveform for executing TPM_GetRandom command.](image)

6.2.1.3 *TPM_CreateWrapKey* command

To test the *TPM_CreateWrapKey* command, a sample input message block with the same structure as that shown in Figure 6.9 was passed to the TPM_TOP test bench. Figure 6.21 shows the simulation Waveform of *TPM_CreateWrapKey* command. The clock period was 100 ns and the output message was generated in about 0.16 sec.
To test the `TPM_Sign` command, a sample input message block with the same structure as that shown in Figure 6.12 was passed to the `TPM_TOP` test bench. Figure 6.22 shows the simulation Waveform of `TPM_Sign` command. The clock period was 100 ns, and the output message is generated in about 0.16 sec.
6.2.1.5 **TPM_Sea**l command

To test the **TPM_Sea**l command, a sample input message block with the structure of Figure 6.15 was applied to the **TPM_TOP** test bench. The clock period was 100 ns, and the output message was generated in about 0.47 sec. The output is displayed in three waveforms to provide a clear view of the values of the various signals. Figure 6.23 shows the simulation waveforms of the **TPM_Sea**l command.
Figure 6.23: Waveform for executing TPM_Seal command.

Figure 6.24: Waveform for Input/Output signals of the TPM_Seal command.

Figure 6.25: Waveforms for Input “Data” and Output “SealedData” signals.
6.2.1.6  **TPM\_Unseal** command

To test the **TPM\_Unseal** command, an input message block with the same structure as in Figure 6.16 was passed to the **TPM\_TOP** test bench. The clock period was 100 ns, and the output message was generated in about 0.16 sec. The output is displayed in three waveforms to provide a clear view of the values of the various signals. Figure 6.26 shows the simulation waveforms of the **TPM\_Unseal** command.

![Figure 6.26: Waveform for executing TPM\_Seal command.](image-url)
To validate the functionality of the **TPM\_Unseal** command, we chose the input data value to be the same as the output *sealedData* value from the previous command. Figure 6.25 shows the input data block and the output *sealedData* block. In Figure 6.28, the *sealedData* value is used as an input and the output secret returns the same value of the data block used in the **TPM\_Seal command**.

**Figure 6.27**: Waveform for Input/Output signals of the **TPM\_Seal** command.

**Figure 6.28**: Waveforms for Input “Data” and Output “Secret” signals.
6.2.2 Summary

The execution times for various TPM commands are presented in Table 6.2. Results are presented for the single TPM, as well as the proposed sliced TPM.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Number of</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time for each shares</td>
</tr>
<tr>
<td>TPM_extend</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
<tr>
<td>TPM_GetRandom</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
<tr>
<td>TPM_CreateWrapKey</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
<tr>
<td>TPM_sign</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
<tr>
<td>TPM_Seed</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
<tr>
<td>TPM_Unseal</td>
<td>single TPM</td>
</tr>
<tr>
<td></td>
<td>4 Shares</td>
</tr>
<tr>
<td></td>
<td>10 Shares</td>
</tr>
</tbody>
</table>

The results indicate that commands involving RSA operations impose high execution-time overhead. For example, the **TPM_CreateWrapKey** command, involving 1,024-bit RSA key generation and validation as well as encryption of the private key and some other cryptographic functions, requires 158 ms. Similarly, large execution times are required for **TPM_sign**, **TPM_Seed**, and **TPM_Unseal** operations. However, this overhead can be reduced by splitting the
TPM into \( k \) shares. For example, the execution time for the \textit{TPM\_CreateWrapKey} command is reduced to about 1.2 ms.

6.2.3 Synthesis results

Synthesis is the process of converting the register transfer level (RTL) representation of a design into an optimized gate-level netlist. The TPM code was synthesized for 0.05 \( \mu \)m CMOS technology at a 33MHz clock signal.

6.2.3.1 Synthesis area results

The synthesis silicon area results are summarized in Table 6.3. They reflect a significant area increase when the TPM is divided into \( k \) shares. In the case of 4 shares the area for each share was decreased by 2.93\% while the total area was increased by 74\%. In the case of 10 shares the area for each share was decreased by 3.3\% while the total area was increased by 85\%.

The total area of the TPM consists of the following: The AES engine utilizes approximately 82,600 cells area (3\%), the SHA-1 engine utilizes approximately 20,700 cells area (2\%), the RNG engine utilizes approximately 78,500 cells area (3\%), the RSA engine utilizes approximately 120,000 cells area (7\%), the SHA-1 engine utilizes approximately 20,700 cells area (2\%), and the six TPM commands engines utilizes approximately 4,780,000 cells (85\%).

The six TPM commands engines dominate the total area. Since our implementation was especially specific to these six TPM commands, we did not try to share the resources for the executions; we created a separate engine for each of these TPM commands. Overall, in a real TPM, the TPM commands should implement with share resources as an ALU or as a common basic hardware engine.
Table 6.3: Synthesis area results.

<table>
<thead>
<tr>
<th>Number of Shares</th>
<th>RSA key size</th>
<th>RSA area</th>
<th>Each TPM share area</th>
<th>Ratio</th>
<th>Total area</th>
</tr>
</thead>
<tbody>
<tr>
<td>single TPM</td>
<td>1024-bit</td>
<td>117285.12</td>
<td>5978405.87</td>
<td>1</td>
<td>5978405.87</td>
</tr>
<tr>
<td>4 Shares</td>
<td>256-bit</td>
<td>29298.44</td>
<td>5802143.44</td>
<td>2.93%</td>
<td>23209653.76</td>
</tr>
<tr>
<td>10 Shares</td>
<td>160-bit</td>
<td>18162.36</td>
<td>5780160.16</td>
<td>3.30%</td>
<td>57801601.6</td>
</tr>
</tbody>
</table>

6.2.3.2 Synthesis energy results

The synthesis energy results are summarized in Table 6.4. They reflect a significant energy increase when the TPM is divided into $k$ shares. In the case of 4 shares the total energy was increased by 67%. While in the case of 10 shares the total energy was increased by 85%.

Table 6.4: Synthesis energy results

<table>
<thead>
<tr>
<th>Number of Shares</th>
<th>RSA key size</th>
<th>RSA energy</th>
<th>Each TPM share energy</th>
<th>Ratio</th>
<th>Total energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>single TPM</td>
<td>1024-bit</td>
<td>.0034 J</td>
<td>.159 J</td>
<td>1</td>
<td>.159 J</td>
</tr>
<tr>
<td>4 Shares</td>
<td>256-bit</td>
<td>.00039 J</td>
<td>.0077 J</td>
<td>81.14%</td>
<td>.03 J</td>
</tr>
<tr>
<td>10 Shares</td>
<td>160-bit</td>
<td>.0000056 J</td>
<td>.0009 J</td>
<td>94.5%</td>
<td>.0089 J</td>
</tr>
</tbody>
</table>
CHAPTER 7. CONCLUSIONS

In this thesis, we implemented and simulated a small version of a TPM that can perform a subset of TPM commands, using both a single TPM implementation and a sliced TPM implementation. We then analyzed the execution time, silicon area requirement, and energy usage under different conditions. We also varied the number of shares. The functionality of the proposed algorithm was coded in Verilog, synthesized using 0.05 µm CMOS technology, and simulated in Modelsim.

The results indicate that the proposed algorithm improves TPM efficiency; the sliced TPM achieved lower execution time for the commands involving RSA operations, while the silicon area consumed for the sliced TPM increased compared to that of the single TPM. In the case of 4 shares the required execution time of the TPM commands that involving RSA operation in each slice was decreased by 93%, and the area of each slice was decreased by 2.93% while the total area was increased by 74%. In the case of 10 shares the required execution time of the TPM commands that involving RSA operations in each slice was decreased by 99%, and the area of each slice was decreased by 3.3% while the total area was increased by 85%.
REFERENCES


APPENDIX: TPM SOURCE FILES

`timescale 100ns/10ns
`define CYCLE 1.0
`define End_CYCLE 1000000
`define TOTAL_DATA 6
`define TEST_DATA 6

Module TPM_TOP_tb;

//==============================================================
= //==== signal declaration
============
//--------------------------------------------------------- -
//-------- signals in top module --------------------------
-
reg clk;
reg reset;
wire ready,valid;
reg we;
reg oe;
reg start;
reg [7:0] addr;
reg [31:0] IN_Bytes;
wire [31:0] Out_Bytes;

// -------- variables &indices -----------------------------
-
integer i,j;
reg [3199:0] command_buffer [0:5];
initial $readmemh("./tpmtest/tpm.dat", command_buffer);

//==== module connection
========================================
TPM_TOP dut(
 .clk(clk),
 .reset(reset),
 .ready(ready),
 .we(we),
 .oe(oe),
 .start(start),
 .addr(addr),
);
always begin
  #(\`CYCLE/2) clk = ~clk;
end

initial begin
  #0; // t = 0
  clk  = 1'b1;
  reset = 1'b0;
  we   = 1'b0;
  oe   = 1'b0;
  start = 1'b0;
  addr  = 12'd0;
  IN_Bytes  = 32'd0;

  #(\`CYCLE) reset = 1'b1; // t = 1
  #(\`CYCLE) reset = 1'b0; // t = 2

  j = 0;
  while(j<\`TEST_DATA) begin
    // command_buffer
    i = 0;
    while(i<100) begin
      #(\`CYCLE);
      if(i==0) begin
        we = 1'b1;
        addr = 8'd0;
      end
      else begin
        addr = addr+8'd1;
      end
      IN_Bytes = command_buffer[j][(addr*32) +: 32];
      i = i+1;
    end
  end

  #(\`CYCLE);
  start = 1'b1;
  we = 1'b0;
addr = 8'd0;
IN_Bytes = 0;

#(`CYCLE) start = 1'b0;
@(posedge ready);
@(posedge clk);
#(`CYCLE*0.01);

i = 0;
while(i<100) begin
  if(i==0) begin
    oe = 1'b1;
    addr = 8'd0;
  end
  else begin
    addr = addr+8'd1;
  end
  #(`CYCLE);
  if(Out_Bytes !== 32'd0) begin
    // $display("-----------------------------------
                  ---------------------
                      \n        Command_CODE= %th   Buffer[%d:%d]:
                  output %h \n", command_buffer[j][49:41] , addr*32+31, addr*32, Out_Bytes);
      #1;
  end
  i = i+1;
end
oe = 1'b0;
addr = 8'd0;
#(`CYCLE*2);
j = j+1;

$display("-----------------------------------
                  ---------------------
                      \n        Congratulations! All data have been generated successfully!
                  ---------------------
                      \n        PASS-------------------
                  
      $finish;
module TPM_TOP(
    clk,
    reset,
    start,
    ready,
    valid,
    IN_Bytes,
    we,
    oe,
    addr,
    Out_Bytes
);

//==== in/out declaration ========================
//========== input =================-------------

input clk;
input reset;
input we;
input oe;
input start;
input [7:0] addr;
input [31:0] IN_Bytes;

//-------- output --------------------------------
output ready, valid;
output [31:0] Out_Bytes;

//==== reg/wire declaration ================================

reg [ 31:0] Out_Bytes;
wire [ 31:0]  next_Out_Bytes;

reg reset_1;
reg ready, valid;
reg [2:0] cnt;
reg [2:0]state;
    reg [2:0] nxstate;
    parameter TPM_Extend_command=20;
    parameter TPM_GetRandom_command=70;
    parameter TPM_CreateWrapKey_command=31;
    parameter TPM_Seal_command=23;
    parameter TPM_Unseal_command=24;
    parameter TPM_sign_command=60;

    parameter idle = 0;
parameter test = 1;
parameter nonce_generator = 2;
parameter command_exc = 3;
parameter command_output = 4;
reg [0:3199] a, next_a, Buffer, next_a_out, a_out;

assign next_Out_Bytes = (oe & ready)? a_out[(addr*32)+:32] :
32'b0; // reading

always @(*) begin // writing
next_a = a;
next_a_out = a_out;
if (we & ready) begin
next_a [(addr*32)+:32] = IN_Bytes;
$display("Writing a[%d+:32] = %h",addr*32,IN_Bytes);
end
else next_a_out = Buffer;
end

always @(posedge clk or posedge reset) begin
if (reset == 1'b1) begin
a <= 1024'b0;
Out_Bytes <= 32'b0;
end else begin
a <= next_a;
next_a_out <= next_a_out;
Out_Bytes <= next_Out_Bytes;
end
end

wire nonce_Gen_ready, nonce_Gen_valid;
reg nonce_Gen_start, test_valid, nonce_valid, End_rng;
wire [159:0] RN_out;
nonce_Gen nonce_Gen_dut(
  .clk(clk),
  .reset(reset),
  .start(nonce_Gen_start),
  .ready(nonce_Gen_ready),
  .valid(nonce_Gen_valid),
  .RN_out(RN_out),
  .cnt(cnt),
  .End_rng(End_rng)
);

wire TPM_Extend_ready, TPM_Extend_valid;
reg TPM_Extend_start;
reg [7:0] tag_in; // TPM_TAG TPM.TAG_RQU_COMMAND2
reg [31:0] paramSize_in; // paramSize Total number of input bytes including paramSize
reg [7:0] ordinal_in, Command_CODE;
reg [3:0] pcrNum;
reg [159:0] inDigest
wire [7:0] tag_out_e;
wire [31:0] paramSize_out_e;
wire [3:0] returnCode_e;
wire [7:0] ordinal_out_e
wire [159:0] outDigest;
TPM_Extend TPM_Extend_dut (    
    .clk(clk),
    .reset(reset),
    .start(TPM_Extend_start),
    .ready(TPM_Extend_ready),
    .valid(TPM_Extend_valid),
    .tag_in(tag_in), .paramSize_in(paramSize_in),
    .ordinal_in(ordinal_in), .pcrNum(pcrNum),
    .inDigest(inDigest), .tag_out(tag_out_e),
    .paramSize_out(paramSize_out_e), .returnCode(returnCode_e)
);
wire TPM_Seal_ready,TPM_Seal_valid;
reg TPM_Seal_start;
reg [7:0] keyhandle;
reg [31:0] inDataSize;
reg [1023:0] inData;
reg [3071:0] inData2;
reg [319:0] pcrInfo;
reg [319:0] encAuth;
reg [159:0] nonceEven_in,nonceOdd_in;
wire [159:0] nonceEven_out_s,nonceOdd_out_s;
wire [31:0] resAuth_s,resAuth_i,resAuth_c;
wire [3071:0]sealedData;
wire [7:0] tag_out_s;
    wire [31:0] paramSize_out_s;
    wire [3:0] returnCode_s;
    wire [7:0] ordinal_out_s;
TPM_Seal TPM_Seal_dut(
    .clk(clk),
    .reset(reset),
    .start(TPM_Seal_start),
    .ready(TPM_Seal_ready),
    .valid(TPM_Seal_valid),
    .tag_in(tag_in), .paramSize_in(paramSize_in),
    .ordinal_in(ordinal_in),
    .keyhandle(keyhandle),
    .encAuth(encAuth),
    .inDataSize(inDataSize),
    .nonceEven_in(nonceEven_in),
.nonceOdd_in(nonceOdd_in),
.inData(inData),
.pcrInfo(pcrInfo),
.tag_out(tag_out_s),
.paramSize_out(paramSize_out_s), .returnCode(returnCode_s),
.ordinal_out(ordinal_out_s),
.nonceEven_out(nonceEven_out_s),
.nonceOdd_out(nonceOdd_out_s),
.sealedData(sealedData),
.resAuth(resAuth_s),
);

wire TPM_Unseal_ready,TPM_Unseal_valid;
reg TPM_Unseal_start;
wire [1023:0] secret;
wire [31:0] secret_size;
wire [7:0] tag_out_u; //TPM_TAG_RSP_COMMAND2
wire [31:0] paramSize_out_u;
wire [3:0] returnCode_u;
wire [7:0] ordinal_out_u;
wire [159:0] nonceEven_out_u,nonceOdd_out_u;

TPM_Unseal  TPM_Unseal_dut(
  .clk(clk),
  .reset(reset),
  .start(TPM_Unseal_start),
  .ready(TPM_Unseal_valid),
  .valid(TPM_Unseal_valid),
  .tag_in(tag_in),
  .paramSize_in(paramSize_in),
  .ordinal_in(ordinal_in),
  .keyhandle(keyhandle),
  .inData(inData2),
  .inDataSize(inDataSize),
  .nonceEven_in(nonceEven_in),
  .nonceOdd_in(nonceOdd_in),
  .tag_out(tag_out_u),
  .paramSize_out(paramSize_out_u), .returnCode(returnCode_u),
  .ordinal_out(ordinal_out_u),
  .nonceEven_out(nonceEven_out_u),
  .nonceOdd_out(nonceOdd_out_u),
  .secret_size(secret_size),
  .secret(secret)
);

wire TPM_CreateWrapKey_ready,TPM_CreateWrapKey_valid;
reg TPM_CreateWrapKey_start;
reg [7:0] parentHandel;
reg [319:0] dataUsageAuth;
reg [31:0] keyinfo, authHandle;
wire [2047:0] PubAuth;
wire [2207:0] wrappedKey;
wire [7:0] tag_out_c;
wire [31:0] paramSize_out_c;
wire [3:0] returnCode_c;
wire [7:0] ordinal_out_c;
wire [159:0] nonceEven_out_c, nonceOdd_out_c;

TPM_CreateWrapKey TPM_CreateWrapKey_dut (.
.clk(clk),
.reset(reset),
.start(TPM_CreateWrapKey_start),
.ready(TPM_CreateWrapKey_ready),
.valid(TPM_CreateWrapKey_valid),
.tag_in(tag_in), .paramSize_in(paramSize_in),
.ordinal_in(ordinal_in),
.parentHandel(parentHandel),
.dataUsageAuth(dataUsageAuth),
.keyinfo(keyinfo),
.authHandle(authHandle),
.nonceEven_in(nonceEven_in),
.nonceOdd_in(nonceOdd_in),
.PubAuth(PubAuth),
.tag_out(tag_out_c),
.paramSize_out(paramSize_out_c),
.returnCode(returnCode_c),
.ordinal_out(ordinal_out_c),
.wrappedKey(wrappedKey),
.nonceEven_out(nonceEven_out_c),
.nonceOdd_out(nonceOdd_out_c),
.resAuth(resAuth_c)
);

wire TPM_GetRandom_ready, TPM_GetRandom_valid;
   reg TPM_GetRandom_start;
   reg [31:0] bytesRequested;

wire [511:0] randomBytes;
wire [31:0] randomBytesSize; wire [7:0] tag_out_g;
//TPM_TAG_RSP_COMMAND2
   wire [31:0] paramSize_out_g;
   wire [3:0] returnCode_g;
   wire [7:0] ordinal_out_g;

TPM_GetRandom TPM_GetRandom_dut (}
.clk(clk),
.reset(reset),
.start(TPM_GetRandom_start),
.ready(TPM_GetRandom_ready),
.valid(TPM_GetRandom_valid),
.tag_in(tag_in), .paramSize_in(paramSize_in),
.READY|(ordIn(ordinal_in),
.bytesRequested(bytesRequested),
.tag_out(tag_out_g),
.paramSize_out(paramSize_out_g),
.returnCode(returnCode_g),
.ordinal_out(ordinal_out_g),
.randomBytesSize(randomBytesSize),
.randomBytes(randomBytes)
);

wire TPM_sign_ready, TPM_sign_valid;

reg TPM_sign_start;
reg [319:0] areaToSign;
reg [31:0] areaToSignSize;
wire [31:0] sigSize;
wire [31:0] privAuth;
wire [31:0] sig;
wire [31:0] areaToSignSize;
wire [7:0] tag_out_i;
wire [31:0] paramSize_out_i;
wire [3:0] returnCode_i;
wire [7:0] ordinal_out_i;
wire [159:0] nonceEven_out_i, nonceOdd_out_i;
reg ind;

TPM_sign    TPM_sign_dut(
    .clk(clk),
    .reset(reset),
    .start(TPM_sign_start),
    .ready(TPM_sign_ready),
    .valid(TPM_sign_valid),
    .tag_in(tag_in), .paramSize_in(paramSize_in),
    .ordinal_in(ordinal_in),
    .areaToSignSize(areaToSignSize),
    .areaToSign(areaToSign),
    .authHandle(authHandle),
    .nonceEven_in(nonceEven_in),
    .nonceOdd_in(nonceOdd_in),
    .privAuth(privAuth),
    .tag_out(tag_out_i),
    .paramSize_out(paramSize_out_i),
    .returnCode(returnCode_i),
    .ordinal_out(ordinal_out_i),
    .sigSize(sigSize),
    .sigSize(sigSize),
always @(posedge clk) begin
    reset_1 <= reset;
end
always @(posedge clk) begin
    if (reset_1 == 1'b1)
        state <= idle;
    else
        state <= nxstate;
end

always @(state or reset_1 or start or test_valid or
TPM_Extend_valid or TPM_GetRandom_valid or
TPM_CreateWrapKey_valid or TPM_Seal_valid or TPM_Unseal_valid
or TPM_sign_valid or nonce_valid) begin
    nxstate <= state;
    case (state)
        idle: begin
            if ((reset_1 == 1'b0) & (start == 1'b1))
                nxstate <= test;
        end
        test: begin
            if (test_valid == 1'b1)
                nxstate <= nonce_generator;
        end
        nonce_generator: begin
            if (nonce_valid == 1'b1)
                nxstate <= command_exc;
        end
        command_exc: begin
            if ((TPM_Extend_valid == 1'b1) ||
                (TPM_GetRandom_valid == 1'b1) ||
                (TPM_CreateWrapKey_valid == 1'b1) ||
                (TPM_Seal_valid == 1'b1) ||
                (TPM_Unseal_valid == 1'b1) ||
                (TPM_sign_valid == 1'b1))
                nxstate <= command_output;
        end
        command_output: begin
            nxstate <= idle;
        end
always @(posedge clk) begin
    if (reset_1 == 1'b1) begin
        valid <= 1'b0;
        Buffer <= '{1'b0};
    // ready <= 1'b0;
    end
    else begin
        ready <= 1'b0;
        valid <= 1'b0;
        TPM_Extend_start <= 1'b0;
        TPM_sign_start <= 1'b0;
        TPM_Unseal_start <= 1'b0;
        TPM_Seal_start <= 1'b0;
        TPM_CreateWrapKey_start <= 1'b0;
        TPM_GetRandom_start <= 1'b0;
        nonce_Gen_start <= 1'b0;
        nonce_valid <= 1'b0;
        case (state)
            idle: begin
                ready <= 1'b1;
                if ((reset_1 == 1'b0) & (start == 1'b1))
                    ind <= 1'b1;
            end
            test: begin
                tag_in <= a [0:7];
                paramSize_in <= a [8:39];
                if (paramSize_in != 8'b0) begin
                    Command_CODE <= a[40:47];
                    ordinal_in <= a[40:47];
                    test_valid <= 1;
                end
            end
        endcase
    end
end
if( (Command_CODE==TPM_CreateWrapKey_command) ||
(Command_CODE== TPM_Seal_command) ||
(Command_CODE==TPM_Unseal_command) ||
(Command_CODE==TPM_sign_command))
begin
if (nonce_Gen_ready == 1'b1 ) begin
nonce_Gen_start <= 1'b1;
End_rng<=1'b1;
cnt<=3'b101;
end
if (nonce_Gen_valid == 1'b1 ) begin
nonce_valid <= 1'b1;
nonceOdd_in= RN_out << 2;
nonceEven_in = nonceOdd_in << 3;
end
else
nonce_valid <= 1'b1;
end//end case
command_exc: begin
case ( Command_CODE )

TPM_Extend_command: begin
if ( TPM_Extend_ready == 1'b1 )
begin
TPM_Extend_start <= 1'b1;
pcrNum <= a[48:51];
inDigest<= a[52:211];
end
if ( TPM_Extend_valid == 1'b1 )
Buffer <= {tag_out_e,paramSize_out_e, returnCode_e,ordinal_out_e,outDigest,{1'b0}};
end

TPM_GetRandom_command: begin
if ( TPM_GetRandom_ready == 1'b1 )
begin
TPM_GetRandom_start <= 1'b1;
bytesRequested<=a[48:79];
if (TPM_GetRandom_valid == 1'b1)

Buffer<={tag_out_g,paramSize_out_g,returnCode_g,ordinal_out_g,randomBytesSize,randomBytes,{1'b0}};

TPM_CreateWrapKey_command: begin

if (TPM_CreateWrapKey_ready == 1'b1)

begin
	TPM_CreateWrapKey_start <= 1'b1;
	parentHandel <= a[48:55];
	dataUsageAuth <= a[56:375];
	keyinfo<= a[376:407];
	authHandle<= a[408:439];

if (TPM_CreateWrapKey_valid == 1'b1)

Buffer<={tag_out_c,paramSize_out_c,returnCode_c,ordinal_out_c,wrappedKey,PubAuth,nonceEven_out_c,nonceOdd_out_c,resAuth_c,{1'b0}};

TPM_Seal_command: begin

if (TPM_Seal_ready == 1'b1)

begin
	TPM_Seal_start <= 1'b1;
	keyhandle <= a[48:55];
	encAuth <= a[56:375];
	inDataSize <= a[376:407];
	inData <= a[408:1431];
	pcrInfo <= a[1432:1751];

end

if (TPM_Seal_valid == 1'b1)

Buffer<={tag_out_s,paramSize_out_s,returnCode_s,ordinal_out_s,nonceEven_out_s,nonceOdd_out_s,sealedData, resAuth_s,{1'b0}};
end

TPM_Unseal_command: begin

if ( TPM_Unseal_ready == 1'b1 )
begin
    TPM_Unseal_start <= 1'b1;
    keyhandle<= a[48:55];
    inDataSize<= a[56:87];
    inData2<= a[88:3158];
end

if ( TPM_Unseal_valid == 1'b1 )
Buffer<=

TPM_sign_command: begin

if ( TPM_sign_ready == 1'b1 )
begin
    TPM_sign_start <= 1'b1;
    areaToSignSize<= a[48:79];
    areaToSign<= a[80:399];
    authHandle<= a[400:431];
    privAuth<= a[432:463];
end

if ( TPM_sign_valid == 1'b1 )
Buffer<=

endcase//case

end// command_exc

command_output: begin
valid<=1'b1;
endendcase
module TPM_Extend(
    _clk,
    reset,
    start,
    tag_in,
    paramSize_in,
    ordinal_in,
    pcrNum,
inDigest,
    tag_out,
    paramSize_out,
    returnCode,
    ordinal_out,
    outDigest,
    ready,
    valid );

//==== in/out declaration ==================================
//-------- input ----------------------------------------
input clk;
input reset,start;
output ready;
output valid;
input [7:0] tag_in;
input [31:0] paramSize_in;
input [7:0] ordinal_in;
input [3:0] pcrNum;
input [159:0] inDigest;

//-------- output ----------------------------------------
output [7:0] tag_out;
output [31:0] paramSize_out ;
output [3:0] returnCode;
output [7:0] ordinal_out;
output [159:0] outDigest;

//==== reg/wire declaration ================================
reg reset_1,test_valid;
reg ready;
reg valid;
reg [7:0] tag_out;
reg [31:0] paramSize_out;
reg [3:0] returnCode;
reg [7:0] ordinal_out;
reg [2:0] state;
reg [2:0] nxstate;
parameter idle = 0;
parameter test = 1;
parameter fetch_old_digest = 2;
parameter cal_new_digest = 3;
parameter w_new_digest = 4;
parameter digest_output = 5;
reg r_wb;
reg [159:0] data_in, outDigest, Digest_old;
reg [319:0] c_digest;
reg [3:0] addr;
reg sha_start, next_i;
wire sha_ready, sha_digest_valid;
wire [159:0] data_out, out_digest;

PCR b1 (.r_wb(r_wb), .pcr_num(addr), .digest_out(data_out), .digest_in(data_in));

sha1 dut (.clk(clk),
.reset(reset_1),
.init(sha_start),
.next(next_i),
.block({ 192'd0, c_digest}),
.ready(sha_ready),
.digest(out_digest),
.digest_valid(sha_digest_valid));

always @ (posedge clk)
begin
  reset_1 <= reset;
end
always @ (posedge clk)
begin
    if ( reset_1 == 1'b1 )
        state <= idle;
    else
        state <= nxstate;
end

always @ ( state or reset_1 or start or test_valid or sha_digest_valid or Digest_old or data_in)
begin
    nxstate <= state;
    case ( state )
        idle: begin
            if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) )
                nxstate <= test;
        end
        test: begin
            if ( test_valid == 1'b1)
                nxstate <= fetch_old_digest;
        end
fetch_old_digest: begin
    // if ( pcr_valid==1'b1)
        nxstate <= cal_new_digest;
end
cal_new_digest: begin
    if ( sha_digest_valid == 1'b1)
        nxstate <= w_new_digest;
end
w_new_digest: begin
    nxstate <= digest_output;
end
digest_output: begin
    nxstate <= idle;
end
default : begin
    nxstate <= idle;
end
endcase
always @ ( posedge clk) begin
    if ( reset_1 == 1'b1 ) begin
        valid <=1'b0;
test_valid<=0;
ready<=1'b0;
end
else begin
ready <= 1'b0;
valid <= 1'b0;
sha_start <= 1'b0;
case ( state )
idle: begin
ready <= 1'b1;

if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) )
begin
ready <= 1'b0;
test_valid <= 0;
end
end
test: begin
if ( ordinal_in == 8'h14)
if ( ( pcrNum >= 4'b0 ) & ( pcrNum <= 4'b1000 ))
begin
test_valid<=1;
addr <= pcrNum;
r_wb<=1;
end
end
fetch_old_digest: begin
Digest_old <= data_out;  r_wb<= 0;
end
cal_new_digest: begin
if ( sha_ready == 1'b1 )
begin
next_i<= 1'b0;
sha_start <= 1'b1;
c_digest <= {inDigest , Digest_old};
end
if ( sha_digest_valid == 1'b1 )
begin
outDigest <= out_digest;
module TPM_GetRandom(
  clk,
  reset,
  start,
  ready,
  valid,
  tag_in,
  paramSize_in,
  ordinal_in,
  bytesRequested,
  tag_out,
  paramSize_out,
  returnCode,
  ordinal_out,
  randomBytesSize,
  randomBytes
);

input clk;
input reset,start;
output ready,valid;

input [7:0] tag_in;
input [31:0] paramSize_in;
input [7:0] ordinal_in;
input [31:0] bytesRequested;
output [7:0] tag_out;
output [31:0] paramSize_out;
output [3:0] returnCode;
output [7:0] ordinal_out;
output [511:0] randomBytes;
output [31:0] randomBytesSize;
reg reset_1;
reg ready, valid;
reg [7:0] tag_out;
reg [31:0] paramSize_out;
reg [3:0] returnCode;
reg [7:0] ordinal_out;
reg [31:0] randomBytesSize;
reg [511:0] randomBytes;
reg [2:0] cnt;
reg [2:0] state;
reg [2:0] nxstate;
parameter idle = 0;
parameter test = 1;
parameter fetch_rng = 2;
parameter rng_output = 3;
wire rng_ready, rng_valid;
reg rng_start, test_valid, End_rng;
wiring [511:0] RN_out;
RNG_top DUT(
    .clk(clk),
    .reset(reset_1),
    .start(rng_start),
    .RN_out(RN_out),
    .valid(rng_valid),
    .ready(rng_ready),
    .cnt(cnt), .End_rng(End_rng)
);
always @(posedge clk)
begin
    reset_1 <= reset;
end
always @(posedge clk)
begin
    if (reset_1 == 1'b1)
        state <= idle;
    else
        state <= nxstate;
end
always @ ( state or reset_1 or start or rng_valid or test_valid)
begin
    nxstate <= state;
    case ( state )
    idle: begin
        if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) )
            nxstate <= test;
        end
    test: begin
        if (test_valid == 1'b1)
            nxstate <= fetch_rng;
        end
    fetch_rng: begin
        if (rng_valid == 1'b1)
            nxstate <= rng_output;
        end
    rng_output: begin
        nxstate <= idle;
    end
    default :
    begin
        nxstate <= idle;
    end
    endcase
end
always @ (posedge clk) begin
    if ( reset_1 == 1'b1 ) begin
        valid <= 1'b0;
        ready <= 1'b0;
    end
    else begin
        ready <= 1'b0;
        valid <= 1'b0;
        rng_start <= 1'b0;
        case ( state )
        idle: begin
            ready <= 1'b1;
            if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) )
                cnt <= 3'b11;
        ) begin
    end
ready <= 1'b0; end

test: begin
if ( ordinal_in == 8'h46)
test_valid<=1;
end

fetch_rng: begin
if ( rng_ready == 1'b1 )
begin
rng_start <= 1'b1;
End_rng<= 1'b1;
end
if ( rng_valid == 1'b1 )
begin
randomBytes <= RN_out;
randomBytesSize<= 32'd512;
end
end

rng_output: begin
tag_out<= tag_in;
paramSize_out <=paramSize_in;
returnCode <= 4'b10;
ordinal_out<=ordinal_in;
valid<=1'b1;
end
endcase
end
endmodule

module TPM_sign(
    clk,
    reset,
    start,
    ready,
    valid,
    tag_in,
    paramSize_in,
ordinal_in, // 3c
areaToSignSize,
areaToSign,
authHandle,
privAuth,
nonceEven_in,
nonceOdd_in,
tag_out,
paramSize_out,
returnCode,
ordinal_out,
sigSize,
sig,
nonceEven_out,
nonceOdd_out,
resAuth
);

input clk;
input reset, start;
output ready, valid;

input [7:0] tag_in;
input [31:0] paramSize_in;
input [7:0] ordinal_in;
input [159:0] nonceEven_in, nonceOdd_in;
input [319:0] areaToSign;
input [31:0] areaToSignSize, authHandle;
input [31:0] privAuth;
output [7:0] tag_out;
output [31:0] paramSize_out, resAuth;
output [3:0] returnCode;
output [7:0] ordinal_out;
output [31:0] sigSize;
output [511:0] sig;
output [159:0] nonceEven_out, nonceOdd_out;
reg reset_1;
reg ready, valid;
reg [7:0] tag_out;
reg [31:0] paramSize_out, resAuth;
reg [3:0] returnCode;
reg [7:0] ordinal_out;
reg [31:0] sigSize;
reg [511:0] sig;
reg [159:0] nonceEven_out, nonceOdd_out;
reg [2:0] state;
reg [2:0] nxstate;
parameter idle = 0;
parameter test = 1;
parameter fetch_parentkey = 2;
parameter find_Didest = 3;
parameter sign_Didest = 4;
parameter sign_output = 5;
reg test_valid;
wire [1023:0] n_out, d, e;
reg [1023:0] dn_mem [0:1];
initial $readmemh("./dat/dn1024.dat", dn_mem);
wire RSA_ready, RSA_valid;
reg RSA_start;
wire [1023:0] d_encrypt;
reg [1023:0] parent_d, parent_n, d_i;

RSA_1024 DUT2(.clk(clk), .reset(reset_1), .start(RSA_start),
.msg(d_i), .expo(parent_d), .n(parent_n), .ready(RSA_ready), .encrypt(d_encrypt), .valid(RSA_valid));

wire find_Didest_ready, find_Didest_valid;
reg find_Didest_start, next_i;
reg [319:0] c_digest;
wire [159:0] out_digest;
sha1 dut (  
.clk(clk),
.reset(reset_1),
.init(find_Didest_start),
.next(next_i),
.block({ 192'd0, c_digest}),
.ready(find_Didest_ready),
.digest(out_digest),
.digest_valid(find_Didest_valid)
);
always @ (posedge clk)
begin
reset_1 <= reset;
end
always @ (posedge clk)
begin
if (reset_1 == 1'b1)
state <= idle;
else
state <= nxstate;
end
always @ ( state or reset_1 or start or test_valid or 
RSA_valid or find_Didest_valid )
begin
  nxstate <= state;
  case ( state )
  idle: begin
    if ( (reset_1 == 1'b0) & (start == 1'b1) )
      nxstate <= test;
  end
  test: begin
    if (test_valid == 1'b1)
      nxstate <= fetch_parentkey;
  end
  fetch_parentkey: begin
    // if (parentkey_valid == 1'b1)
      nxstate <= find_Didest;
  end
  find_Didest: begin
    if (find_Didest_valid == 1'b1)
      nxstate <= signDidest;
  end
  signDidest: begin
    if (RSA_valid == 1'b1)
      nxstate <= sign_output;
  end
  sign_output: begin
    nxstate <= idle;
  end
  default : begin
    nxstate <= idle;
  end
  endcase
end
always @ (posedge clk) begin
  if (reset_1 == 1'b1) begin
    valid <= 1'b0;
    ready <= 1'b0;
  end
  else begin
    ready <= 1'b0;
    valid <= 1'b0;
    find_Didest_start <= 1'b0;
    RSA_start <= 1'b0;
  end
case (state)
idle: begin
    ready <= 1'b1;
    if (reset_1 == 1'b0 & start == 1'b1)
) begin
    ready <= 1'b0;
end
end
test: begin
    if (ordinal_in == 8'h3c)
        test_valid <= 1;
end

fetch_parentkey: begin
    parent_n <= dn_mem[0];
    parent_d <= dn_mem[1];
end

find_Didest: begin
    if (find_Didest_ready == 1'b1)
        find_Didest_start <= 1'b1;
        c_digest <= areaToSign;
        next_i <= 1'b0;
end

signDidest: begin
    if (RSA_ready == 1'b1)
        d_i <= {864'b0, out_digest};
        RSA_start <= 1'b1;
end

//if (RSA_valid == 1'b1)

sign_output: begin
    tag_out <= tag_in + 2'b10; //TPM_TAG_RSP_COMMAND2
    paramSize_out <= paramSize_in;
    returnCode <= 4'b00; //TPM_RESULT The return code of the operation.
    ordinal_out <= ordinal_in; //TPM_COMMAND_CODE
    ordinal Command ordinal
    resAuth <= authHandle;
    valid <= 1'b1;
    nonceEven_out <= nonceEven_in;
    nonceOdd_out <= nonceOdd_in;
module TPM_Seal(
    clk,
    reset, start,
    ready, valid,
    tag_in, paramSize_in,
    ordinal_in, // 17
    keyhandle,
    encAuth, inDataSize,
    inData, pcrInfo,
    nonceEven_in,
    nonceOdd_in,
    tag_out,
    paramSize_out,
    returnCode,
    ordinal_out,
    nonceEven_out,
    nonceOdd_out,
    sealedData,
    resAuth
);

input clk;
input reset, start;
input [7:0] keyhandle;
output ready, valid;
input [7:0] tag_in;
input [31:0] paramSize_in, inDataSize;
input [1023:0] inData;
input [7:0] ordinal_in;
input [319:0] pcrInfo;

sig<=d_encrypt[511:0];
sigSize<=32'd512;
endcase
end
endmodule
input [319:0] encAuth;
input [159:0] nonceEven_in,nonceOdd_in;
output [7:0] tag_out;
output [31:0] paramSize_out;
output [3:0] returnCode;
output [7:0] ordinal_out;
output [159:0] nonceEven_out,nonceOdd_out;
output [31:0] resAuth;
output [3071:0] sealedData;
reg reset_1;
reg ready, valid;

reg [7:0] tag_out;
reg [31:0] paramSize_out;
reg [3:0] returnCode;
reg [7:0] ordinal_out;
reg [159:0] nonceEven_out,nonceOdd_out;
reg [31:0] resAuth;
reg [3071:0] sealedData;
reg [3:0] state;
    reg [3:0] nxstate;
parameter idle = 0;
parameter test = 1;
parameter fetch_key = 2;
parameter encryptData = 3;
parameter find_Didest = 4;
parameter encrypt_Didest1 = 5;
parameter find_Didest2 = 6;
parameter encrypt_Didest2 = 7;
parameter data_output = 8;
reg test_valid;
wire [1023:0] n_out, d, e;
reg [1023:0] dn_mem [0:1];
initial $readmemh("./data/dn1024.dat", dn_mem);
    wire RSA_ready, RSA_valid;
    reg RSA_start;
    wire [1023:0] d_encrypt;
    reg [1023:0] n_out_i, parent_d, parent_n, d_i;
    reg [1023:0] msg_encrypt, dig1_encrypt, dig2_encrypt;
    RSA_1024 DUT2 (.clk(clk), .reset(reset_1), .start(RSA_start),
        .msg(d_i), .expo(parent_d), .n(parent_n), .ready(RSA_ready),
        .encrypt(d_encrypt), .valid(RSA_valid));

    wire find_Didest_ready, find_Didest_ready2, find_Didest_valid, find_Didest_valid2;
reg find_Didest_start, next_i, find_Didest_start2;
reg [511:0] c_digest;
wire [159:0] out_digest;
reg [159:0] out_digest1;

shal dut(
  .clk(clk),
  .reset(reset_1),
  .init(find_Didest_start),
  .next(next_i),
  .block(c_digest),
  .ready(find_Didest_ready),
  .digest(out_digest),
  .digest_valid(find_Didest_valid)
);

always @(posedge clk)
begin
  reset_1 <= reset;
end
always @(posedge clk)
begin
  if (reset_1 == 1'b1)
  
    state <= idle;
  else
    state <= nxstate;
end

always @(state or reset_1 or start or test_valid or RSA_valid or find_Didest_valid)
begin
  nxstate <= state;
  case (state)
  idle: begin
    if ((reset_1 == 1'b0) & (start == 1'b1))
      nxstate <= test;
  end
  test: begin
    if (test_valid == 1'b1)
      nxstate <= fetch_key;
  end
  fetch_key: begin
    //t if (key_valid == 1'b1)
      nxstate <= encryptData;
  end
  encryptData: begin

if (RSA_valid == 1'b1)
    nxstate <= find_Didest;
end

find_Didest: begin
    if (find_Didest_valid == 1'b1)
        nxstate <= encrypt_Didest1;
end

encrypt_Didest1: begin
    if (RSA_valid == 1'b1)
        nxstate <= find_Didest2;
end

find_Didest2: begin
    if (find_Didest_valid == 1'b1)
        nxstate <= encrypt_Didest2;
end

encrypt_Didest2: begin
    if (RSA_valid == 1'b1)
        nxstate <= data_output;
end

data_output: begin
    nxstate <= idle;
end

default:
    begin
        nxstate <= idle;
    end
endcase

always @(posedge clk) begin
    if (reset_1 == 1'b1) begin
        valid <= 1'b0;
        ready <= 1'b0;
    end
    else begin
        ready <= 1'b0;
        valid <= 1'b0;
        find_Didest_start <= 1'b0;
        RSA_start <= 1'b0;
        case (state)
            idle: begin
                ready <= 1'b1;
            end
        endcase
    end
end
if (( reset_1 == 1'b0 ) & ( start == 1'b1 ) ) begin
    ready <= 1'b0;
end
end

test: begin
if ( ordinal_in == 8'h17)
    test_valid<=1;
end

fetch_key: begin

    parent_n <= dn_mem[0];
    // parent_d <= dn_mem[1];
    parent_d <= {1004'b0,20'h10001};
end

encryptData: begin
    if ( RSA_ready == 1'b1 ) begin
        RSA_start <= 1'b1;
        d_i<=inData;
    end
    if (RSA_valid == 1'b1)
        msg_encrypt<=d_encrypt;
end

find_Didest: begin
    if ( find_Didest_ready == 1'b1 ) begin
        find_Didest_start <= 1'b1;
        c_digest<=`{192'd0,pcrInfo};
        next_i<=1'b0;
    end
end

encrypt_Didest1: begin
    if ( RSA_ready == 1'b1 ) begin
        RSA_start <= 1'b1;
        d_i<={864'b0 ,out_digest};
    end
    if (RSA_valid == 1'b1)
        dig1_encrypt<=d_encrypt;
end

find_Didest2: begin
    if ( find_Didest_ready == 1'b1 ) begin
        find_Didest_start <= 1'b1;
        c_digest<= msg_encrypt[511:0];
        next_i<=1'b0;
        out_digest1 <= out_digest;
    end
encrypt_Didest2: begin
if (RSA_ready == 1'b1) begin
    RSA_start <= 1'b1;
    d_i<= {864'b0 , out_digest};
end
if (RSA_valid == 1'b1)
    dig2_encrypt<= d_encrypt;
end

data_output: begin
    tag_out<= tag_in + 2'b10; // TPM_TAG_RSP_COMMAND2
    paramSize_out <= paramSize_in;
    returnCode <= 4'b00; // TPM_RESULT The return code of the operation.
    ordinal_out<= ordinal_in; // TPM_COMMAND_CODE
    ordinal Command ordinal
    resAuth<= {1'b0};
    sealedData<= {dig1_encrypt, dig2_encrypt, msg_encrypt};
    valid<= 1'b1;
    nonceEven_out<= nonceEven_in;
    nonceOdd_out<= nonceOdd_in;
end
endcase
end

endmodule

module TPM_Unseal(
    clk,
    reset,
    start,
    ready,
    valid,
    tag_in,
    paramSize_in,
    ordinal_in, //18
    keyhandle,
    inDataSize,
    inData,
    nonceEven_in,
    nonceOdd_in,
    tag_out,
paramSize_out,
returnCode,
ordinal_out,
nonceEven_out,
nonceOdd_out,
secret_size,
secret
);

input clk;
input reset,start;
input [7:0] keyhandle;
output ready,valid;
inout [7:0] tag_in;
input [31:0] paramSize_in,inDataSize;
inout [3071:0] inData;
inout [7:0] ordinal_in;
inout [159:0] nonceEven_in,nonceOdd_in ;
output [7:0] tag_out;
output [31:0] paramSize_out ;
output [3:0] returnCode;
output [7:0] ordinal_out;
output [159:0] nonceEven_out,nonceOdd_out ;
output [1023:0]secret;
output [31:0] secret_size;
reg reset_1;
reg ready, valid;
reg [7:0] tag_out;
reg [31:0] paramSize_out ;
reg [3:0] returnCode;
reg [7:0] ordinal_out;
reg [159:0] nonceEven_out,nonceOdd_out ;
reg [31:0] secret_size;
reg [1023:0]secret;
reg [2:0]state;
    reg [2:0]nxstate;
parameter idle = 0;
parameter test = 1;
parameter fetch_key =2;
parameter decryptData =3;
parameter find_Didest = 4;
parameter compare_digist = 5;
parameter data_output = 6;
reg test_valid,secret_matches;
// wire [1023:0] n_out,d,e;
reg [1023:0] dn_mem [0:1];
initial $readmemh("./dat/dn1024.dat", dn_mem);
wire
RSA_ready,RSA_ready1,RSA_ready2,RSA_valid,RSA_valid1,RSA_valid2;
reg RSA_start,RSA_start1,RSA_start2;
wire [1023:0] d_encrypt;
reg [1023:0] n_out_i,parent_d,parent_n,d_i;
wire [1023:0] decrypt1,decrypt2;

RSA_1024 DUT0(.clk(clk),.reset(reset_1),.start(RSA_start),
.msg(d_i),.expo(parent_d),.n(parent_n),.ready(RSA_ready),.encrypt(d_encrypt),.valid(RSA_valid));
RSA_1024 DUT1(.clk(clk),.reset(reset_1),.start(RSA_start1),
.msg(inData[1023:0]),.expo(parent_d),.n(parent_n),.ready(RSA_ready1),.encrypt(decrypt1),.valid(RSA_valid1));
RSA_1024 DUT2(.clk(clk),.reset(reset_1),.start(RSA_start2),
.msg(inData[2047:1024]),.expo(parent_d),.n(parent_n),.ready(RSA_ready2),.encrypt(decrypt2),.valid(RSA_valid2));

wire
find_Didest_ready,find_Didest_ready2,find_Didest_valid,find_Didest_valid2,compare_digist_valid;
reg find_Didest_start, next_i,find_Didest_start2;
reg [511:0] c_digest;
wire [159:0] out_digest;
reg [159:0] out_digest1;
reg [2:0] RSA_valid_3;

sha1 dut(
    .clk(clk),
    .reset(reset_1),
    .init(find_Didest_start),
    .next(next_i),
    .block(c_digest),
    .ready(find_Didest_ready),
    .digest(out_digest),
    .digest_valid(find_Didest_valid)
);

always @( posedge clk)
begin
    reset_1 <= reset;
end
always @( posedge clk)
begin
  if ( reset_1 == 1'b1 )
    state <= idle;
  else
    state <= nxstate;
end

always @( state or reset_1 or start or test_valid or RSA_valid or find_Didest_valid or secret_matches)
begin
  nxstate <= state;
  case ( state )
  idle: begin
    if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) )
      nxstate <= test;
  end
  test: begin
    if (test_valid == 1'b1)
      nxstate <= fetch_key;
  end
  fetch_key: begin
    //t   if (key_valid == 1'b1)
      nxstate <= decryptData;
  end
  decryptData: begin
    if ( (RSA_valid == 1'b1) || (RSA_valid1 == 1'b1) ||
        (RSA_valid2 == 1'b1) )
      nxstate <= find_Didest;
  end
  find_Didest: begin
    if (find_Didest_valid == 1'b1)
      nxstate <= compare_digist;
  end
  compare_digist: begin
    nxstate <= data_output;
  end
  data_output: begin
    nxstate <= idle;
  end
  default :
    begin
      nxstate <= idle;
    end
end
always @ ( posedge clk) begin
    if ( reset_1 == 1'b1 ) begin
        valid <=1'b0;
        ready<=1'b0;
    end
    else begin
        ready <= 1'b0;
        valid <= 1'b0;
        find_Didest_start <= 1'b0;
        RSA_start <= 1'b0;
        RSA_start1 <= 1'b0;
        RSA_start2<= 1'b0;
        case ( state)
            idle: begin
                ready <= 1'b1;
                if ( ( reset_1 == 1'b0 ) & ( start == 1'b1 ) ) begin
                    ready <= 1'b0;
                end
            end
            test: begin
                if ( ordinal_in == 8'h18)
                    test_valid<=1;
            end
    end
fetch_key: begin
    parent_n <= dn_mem[0];
    parent_d <= dn_mem[1];
end
decryptData: begin
    if ( RSA_ready == 1'b1 ) begin
        RSA_start <= 1'b1;
        d_i<=inData [3071:2047];
        RSA_valid_3<=1'b0;
    end
    if ( RSA_ready1 == 1'b1 ) begin
        RSA_start1 <= 1'b1;
        //  d_i<=inData [2047:3071];
if (RSA_ready2 == 1'b1) begin
  RSA_start2 <= 1'b1;
  //  \texttt{d_i<=inData[2047:3071]};
end
if (RSA_valid == 1'b1)
  RSA_valid_3 = RSA_valid_3 + 1'b1;
if (RSA_valid1 == 1'b1)
  RSA_valid_3 = RSA_valid_3 + 1'b1;
if (RSA_valid2 == 1'b1)
  RSA_valid_3 = RSA_valid_3 + 1'b1;
end

find_Didest: begin
  if (find_Didest_ready == 1'b1) begin
    find_Didest_start <= 1'b1;
    c_digest<= d_encrypt[511:0];
    next_i<=1'b0;
  end
end

compare_digist: begin
  if (out_digest == decrypt2[1023:864])
    secret_matches<=1'b1;
  else
    secret_matches<=1'b1;
end

data_output: begin
  tag_out<= tag_in + 2'b11;
  paramSize_out <=paramSize_in;
  returnCode <= 4'b00;
  ordinal_out<=ordinal_in;
  valid<=1'b1;
  nonceEven_out<=nonceEven_in;
  nonceOdd_out<=nonceOdd_in;
  if (secret_matches == 1'b1) begin
    secret_size<=32'd1024;
    secret<=d_encrypt; end
  else begin
    secret_size<=32'd0;
    secret<={1'b0}; end
end
endcase
end
endmodule