RECONFIGURABLE QUANTUM CRYPTO PROCESSOR USING FPGA

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical and Computer Engineering

By

Lavanya Gnanasekaran

2019

SIGNATURE PAGE

THESIS:	RECONFIGURABLE QUANTUM CRYPTO PROCESSOR USING FPGA
AUTHOR:	Lavanya Gnanasekaran
DATE SUBMITTED:	Fall 2019
	Department of Electrical and Computer Engineering

Dr. Mohamed El-Hadedy _____
Thesis Committee Chair
Electrical & Computer Engineering

Dr. Halima El Naga _____
Department Chair
Electrical & Computer Engineering

Dr. Anas Salah Eddin _____
Assistant Professor
Electrical & Computer Engineering

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Dr. Mohamed El-Hadedy for his support on this thesis. His encouragement has allowed me to explore further in the field of Hardware Security and Quantum Computing and achieve success with this thesis. I am grateful to Dr. Halima El Naga and Dr. Anas Salah Eddin for their guidance and valuable feedback in this research. This thesis would not be possible without continued support from my family and friends. Finally, I would like to thank IBM and Xilinx for providing the technical support needed for this research.

Research reported in this publication was supported by the MENTORES (Mentoring, Educating, Networking, and Thematic Opportunities for Research in Engineering and Science) project, funded by a Title V grant, Promoting Post-baccalaureate Opportunities for Hispanic Americans (PPOHA) — U.S. Department of Education, Washington, D.C. PR/Award Number: P031M140025. The content is solely the responsibility of the author and does not necessarily represent the official views of the Department of Education.

ABSTRACT

Cryptography is a technique followed to ensure secure communication between sender and recipient. Depending on the type of application, various methods of cryptographic techniques are used in our everyday systems. They are commonly classified as Symmetric ciphers, Asymmetric ciphers, Data integrity algorithms or Hash functions. Our traditional computers have the capability to understand the data in binary digits, which has two states – 0 and 1. Quantum computing is an advanced computing technique that operates on quantum mechanical phenomena and uses quantum bits to represent the data. Quantum cryptography is a technique that uses quantum mechanical phenomena to do cryptographic tasks. Research has shown that our current public key cryptographic algorithms like Elliptic curve cryptography (ECC) and Rivest Shamir Adleman (RSA) can be broken using quantum computers. The main objective of this research is to implement RSA algorithm in Digital and Quantum approaches to compare the performance and analyze its vulnerabilities against Quantum computing.

Contents

Chapter 1

Background

People are dependent on computers and internet in today's modern world. Computers can be broadly classified into two categories - Classical computers and Quantum computers. Classical computers stores information in the form of bits - 0's and 1's. These are used in our traditional approach to store, retrieve and process various kinds of information. Quantum computers, on the other hand, is an evolving technology based off of quantum states. Leading organizations like IBM, Google, Microsoft etc are actively researching on various aspects of Quantum computing and its availability for commercial use is not too far away. The advancement of Quantum computers pose a threat to the security of modern cryptography, as most of the commonly used algorithms depend on complex mathematical operations like prime factorization, exponentiation with large numbers, discrete logarithmic calculations. These operations are relatively easy for Quantum computers while classical computers take years to solve the problem.

1.1 Introduction to Cryptography

Security plays an important role in our everyday communication. The data transmitted over the network must be secure enough to protect ourselves from unauthorized access. With the advancements in digital technology, it has become a critical part for every system to protect user's data and privacy.

Cryptography is the study of techniques for secure communication. The process of converting the plain text into cipher text is called encryption. Decryption is the process of retrieving the original message back from cipher text. In modern cryptography, it can be broadly classified into three types - Symmetric cryptography, Asymmetric or Public key cryptography and Hash functions or Data integrity algorithms.

1.2 Types of Cryptography

This section presents an overview of various types of cryptography with an example [1].

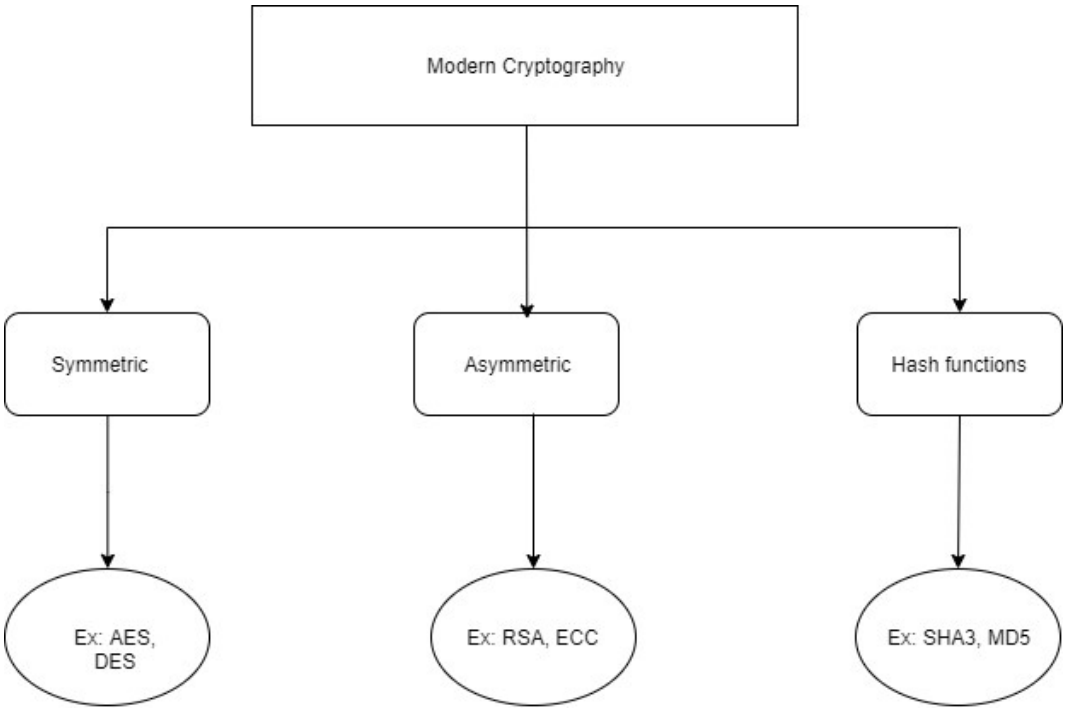

Figure 1.1: Types of Cryptography

1.2.1 Symmetric Cryptography

Symmetric cryptography uses single key for both encryption and decryption. The sender uses a key to encrypt the plaintext, thereby sending ciphertext to the receiver. The receiver applies the same key to decrypt the message and recover the plain text. In order to achieve this type of encryption both sender and receiver must share a common key via a secure medium or in person. Advanced Encryption Standard (AES) and Data Encryption Standard (DES) are good examples of Symmetric key cryptography [2].

1.2.2 Asymmetric Cryptography

Asymmetric cryptographic primitives no longer share a security key similar to the symmetric schemes. It shares a public key between sender and receiver for encryption, while a private key is used for decrypting messages.

Even though a public key is shared between both sides of the communication channel, an unauthorized user cannot decrypt the sent/received message without using a powerful computing unit to generate a private key. Recent research shows that scientists are working on using quantum computing as a powerful tool to break RSA algorithm using 20 million qubits [3].

1.2.3 Hash Functions

Hash functions are also known as one-way encryption, since it cannot be reversed. It is different from Symmetric and Asymmetric key cryptography as it doesn't use any keys. In simple terms, hashing means taking an input string of any arbitrary length and producing an output of fixed length. A secure hash function should not produce same hash value from two different inputs. If it does, this is known as a collision. In general, the hash functions are designed to be collision resistant, meaning that there is a very low probability that the same string would be created for different data. SHA1, SHA2 and SHA3 (Secure Hash Algorithms) and MD5 are good examples of Hash functions [4].

1.2.4 Quantum Cryptography

Quantum cryptography uses quantum phenomena to secure the data transmitted between sender and receiver. One-time pad is the best example of Quantum cryp-

tography which is also referred as Quantum Key Distribution. Unless the eaves dropper disturbs the message, the message transferred remains unchanged. Once the eaves dropper tries to access the data, then the data is changed since it works on the uncertainty principle of Heisenberg.

1.3 Post Quantum Cryptography and NIST

Quantum cryptography is defined by quantum mechanics principles instead of classical principles, on the other hand, Post Quantum Cryptography (PQC) uses public key cryptography and it is safe against the quantum computers mathematical power. PQC is also referred as quantum resist or quantum proof algorithms. National Institute of Standards and Technology (NIST) has provided a platform for cryptographers and mathematicians to come up with novel quantum resist algorithms [5]. Below algorithms are finalized in round 2 submission.

1.3.1 Algorithms selected for ROUND 2 submissions

1. Bike
2. Classic McEliece
3. CRYSTALS-KYBER
4. FrodoKEM
5. HQC
6. LAC
7. LEDAcrypt
8. NewHope
9. NTRU

10. NTRU prime
11. NTS-KEM
12. Rollo
13. Round 5
14. RQC
15. SABER
16. SIKE
17. Three Bears

Chapter 2

Hardware Implementation of RSA using FPGA

2.1 Introduction

In this digital era, cryptography plays a critical role in everyone's daily life. We use a wide range of electronic devices and applications to help us with our everyday routine. Data stored and transmitted between two entities needs to be done in a secure manner to prevent them from reaching untrusted sources. Cryptography is a process followed to convert original message into ciphertext as part of encryption and convert the ciphertext back to original message in decryption.

Symmetric and asymmetric cryptography are two ways to secure original message from attackers. Symmetric cryptography uses same key to encrypt and decrypt and is therefore faster than asymmetric algorithms but isn't secure enough. On the other hand, to ensure higher security, asymmetric cryptography uses a public key to encrypt the message and private key to decrypt it. RSA and Elliptic Curve

Cryptography (ECC) are some of the examples of asymmetric cryptography.

This chapter will focus on explaining the architecture of RSA and its implementation using efficient multiplication and exponentiation components on FPGA.

Related work, Theoretical concepts of RSA algorithm, Required components like Montgomery multiplication, Montgomery exponentiation, primality tests and its simulation results are explained in the following sections.

2.2 Related Work

There are multiple ways to perform RSA algorithms in efficient manner. For example, Dr. S. Nirmala, et al. [6] presented bypass multiplier technique for implementing modular units in RSA algorithm, which avoids unnecessary modulo multiplication and reduces register requirements for multiplier architecture. Xinkai Yan, et al. [7] implemented hybrid Montgomery modular multiplier which uses Karatsuba and Knuth multiplication algorithms in different levels to implement large integer multiplication. Mostafizur Rahman, et al. [8] presented RSA algorithm using interleaved modular multiplication technique and square and multiply technique for exponentiation.

2.3 RSA Algorithm

RSA is a widely-used, highly secure public key cryptographic algorithm. It generates a public key with which user's message is encrypted and a private key is generated to decrypt the message. Implementation of RSA is computationally expensive as it operates on large numbers.

Exponentiation operation is fundamental in RSA cryptosystems and it was a challenging task to implement the necessary blocks using efficient algorithms. These operations require high computations when implemented by software. To achieve high security and greater performance, it is often more advantageous to design and develop cryptosystems in hardware [9].

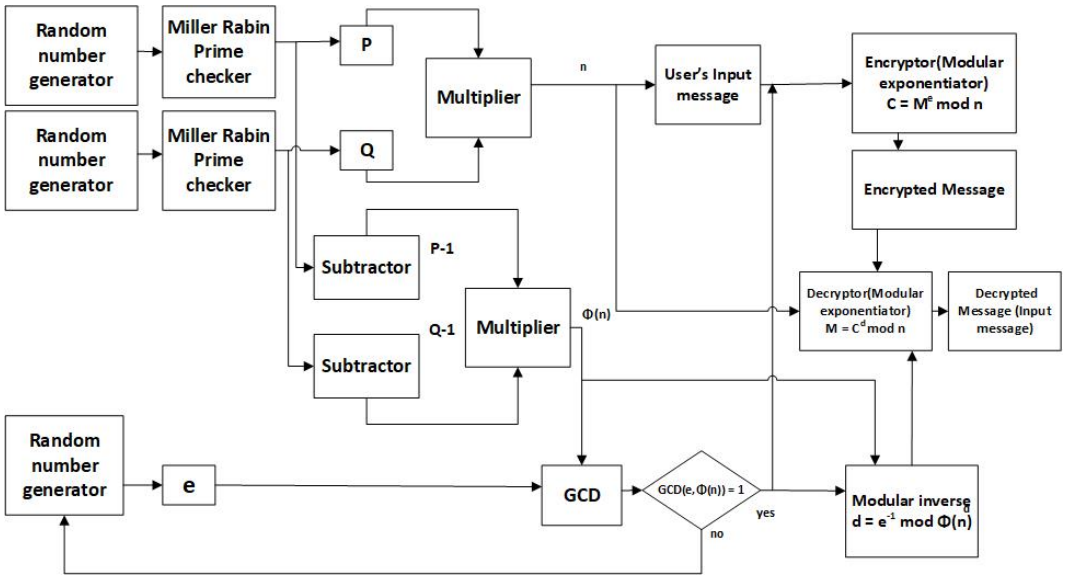

Figure 2.1: Block diagram of RSA algorithm

Fig. 2.1 explains the complete block diagram of RSA algorithm. It starts with generating two large random prime numbers p, q and n is computed by multiplying the prime numbers using add-and-shift multiplier.

$$n = p \times q \tag{2.1}$$

$\phi(n)$ is computed by multiplying the outputs of subtractor of prime numbers p and q

$$\phi(n) = (p-1) \times (q-1) \tag{2.2}$$

A random number e is generated such that it is between 1 and $\phi(n)-1$, and the Greatest Common Divisor (GCD) of e and $\phi(n)$ should be 1. A random number generator block and GCD block generates this public key in hardware.

$$e \in \{1, 2, 3...\phi(n) - 1\}, GCD(e, \phi(n)) = 1 \tag{2.3}$$

The following public key is used to encrypt the incoming message to obtain cipher-text

$$C = M^e \, mod \, n \tag{2.4}$$

A private key d is generated using modular inverse operation from e and $\phi(n)$ by

$$d = e^{-1} \, mod \, \phi(n) \tag{2.5}$$

Using the above generated private key, the original message can be decrypted back from the cipher-text by

$$M = C^d \, mod \, n \tag{2.6}$$

Figure 2.2 and Figure 2.3 shows power estimation and utilization summary of RSA respectively. Figure 2.4 shows simulation results of the RSA algorithm designed and implemented using Very High-Speed Integrated Circuit Hardware Description Language (VHDL) on Xilinx Vivado Design Suite. Random prime numbers p and q are represented as reg_p and reg_q respectively in the program, and their product is stored as reg_n. Input *message* is encrypted into *cipher_text* using public key reg_n and reg_e. *cipher_text* is then decrypted back to input message using private key reg_d.

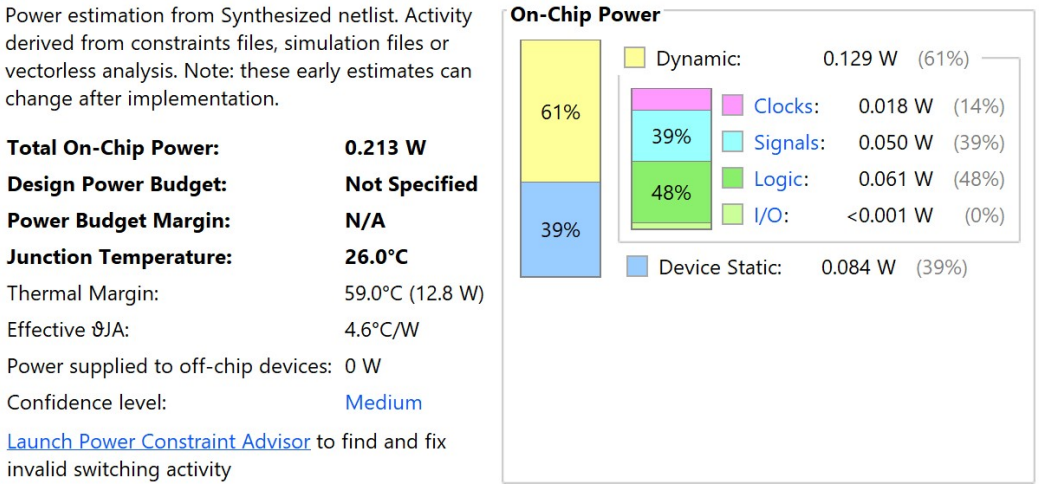

Figure 2.2: Summary of power estimation for 16-bit RSA

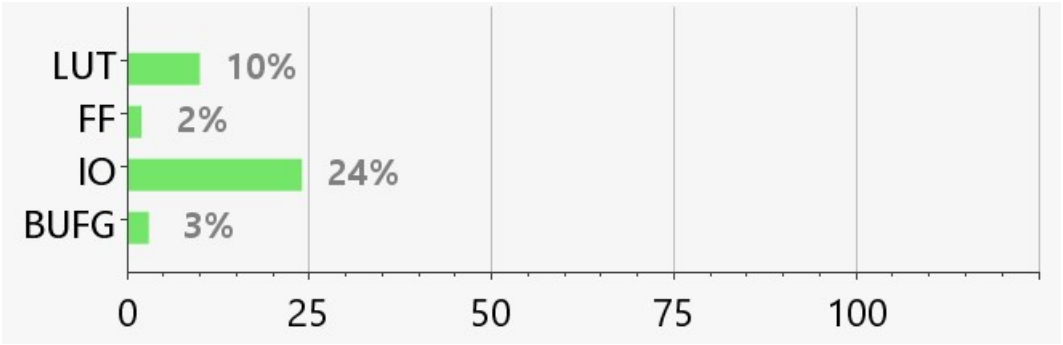

Figure 2.3: Utilization summary for RSA

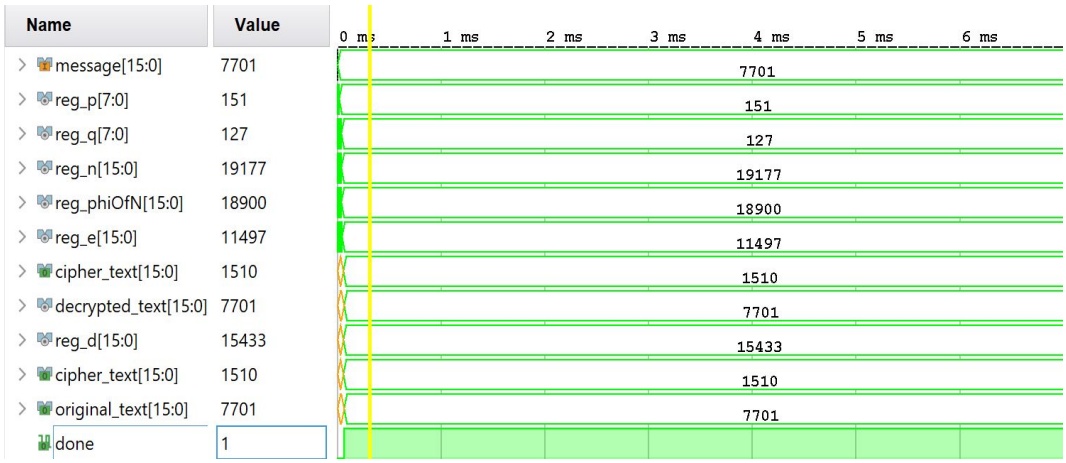

Figure 2.4: Simulation results of encryption and decryption blocks of RSA

Since this implementation is utilizing 10% of look-up tables, 10 similar blocks can be implemented on a Nexys4 FPGA board.

2.3.1 Montgomery Multiplication

Multiplication and division require complex computations when implemented in hardware platforms. In a traditional approach, modular multiplication involves multiplication followed by division [10]. Equation (2.4) and (2.6) shows the need to compute modular exponentiation in RSA, each requiring lots of modular multiplication components. There are many approaches to perform multiplication such as multiply then divide, interleaving multiplication and reduction, Brickell's method [11].

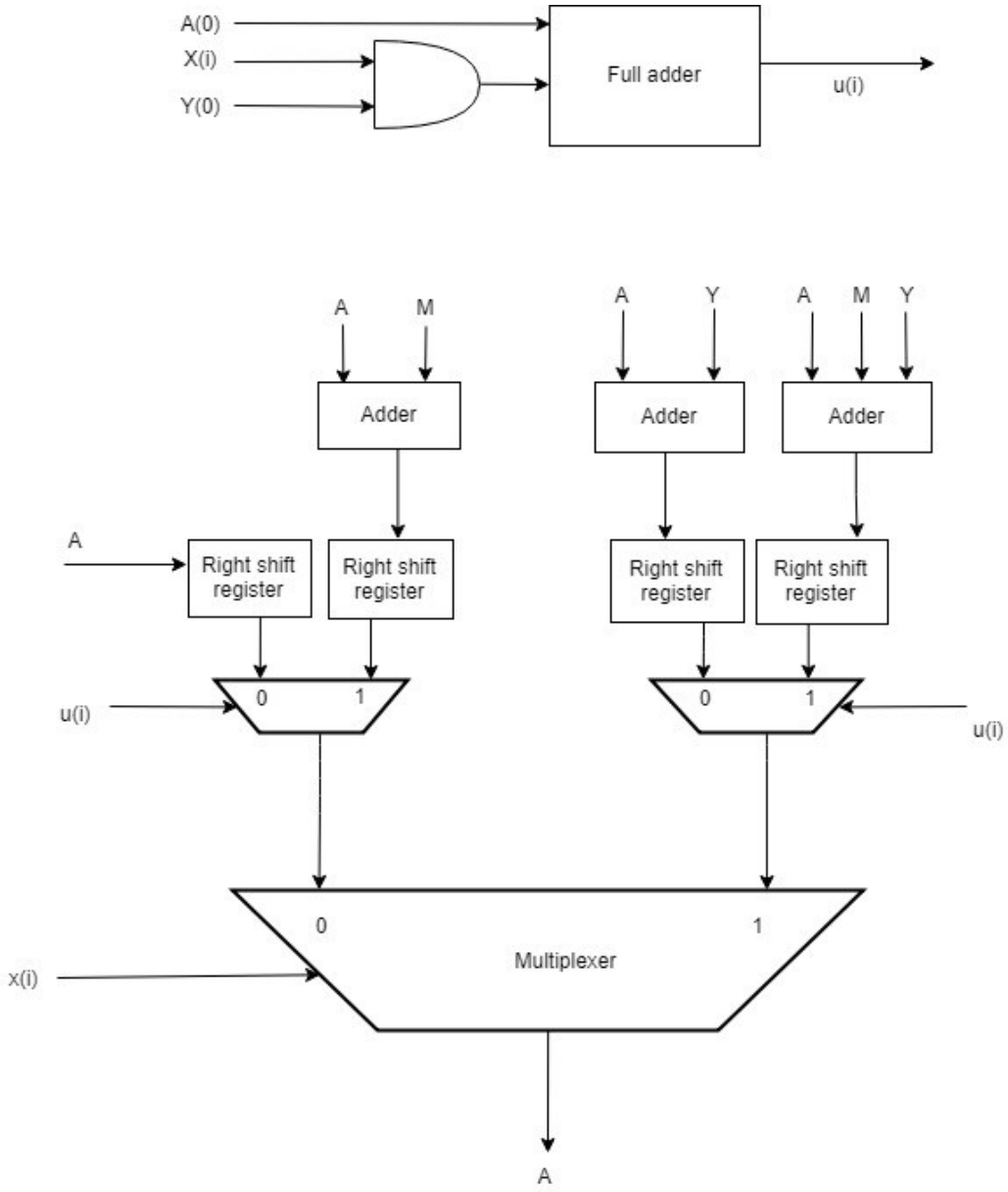

Figure 2.5: Architecture of Montgomery modular multiplier

To design and develop an efficient crypto-processor, Montgomery multiplication has been chosen for all our modular multiplication calculations. This technique avoids the traditional division operation, and replaces it with shift-and-add operation. Montgomery modular multiplication is computed as

$$A = X \times Y \times R^{-1} \, mod \, M \tag{2.7}$$

$$R = 2^n, \text{where } n \text{ is number of bits}$$

Equation (2.7) represents a modular multiplication in Montgomery domain. Results of the above computation can remain in Montgomery domain, as converting it back to real world coordinates is a costly operation. Since we use Montgomery exponentiation, it can handle the results of modular multiplication in Montgomery domain and avoids the need to convert it back [12].

Two preconditions for Montgomery multiplication technique are

- Modulus M needs to be co-prime with R

  i.e. $GCD(M, R) = 1$

- Multiplicand and multiplier need to be smaller than modulus M [13]

Montgomery multiplier requires $2n(n+1)$ single-precision multiplications due to n loop iterations as seen in Algorithm 1. As the result is computed in Montgomery domain, converting it back to real world coordinates requires additional single-precision operations, thereby having a total of $4n(n+1)$ single-precision multiplications. This is slower than the traditional multiplication algorithms, which perform at $2n(n+1)$ computational efficiency. Since we don't convert it to real world coordinates and the results of multiplication can be used in exponentiation component in Montgomery domain, this is an efficient modular multiplication technique [14].

Algorithm 1 Montgomery modular multiplication algorithm [14]

Inputs: X, Y, M

Ouput: $A = XYR^{-1} mod M$

$A \leftarrow$ zeros for $n + 1$ bits

$b = 2$

$M' = M^{-1} mod b$

for i: from 0 to n - 1 **do**

    $u(i) = (A_0 + X_i \times Y_0) \times M' mod b$

    $A = \frac{A + X_i \times Y + u(i) \times M}{2}$

if $A >= M$ **then**

    $A = A - M$

Return A

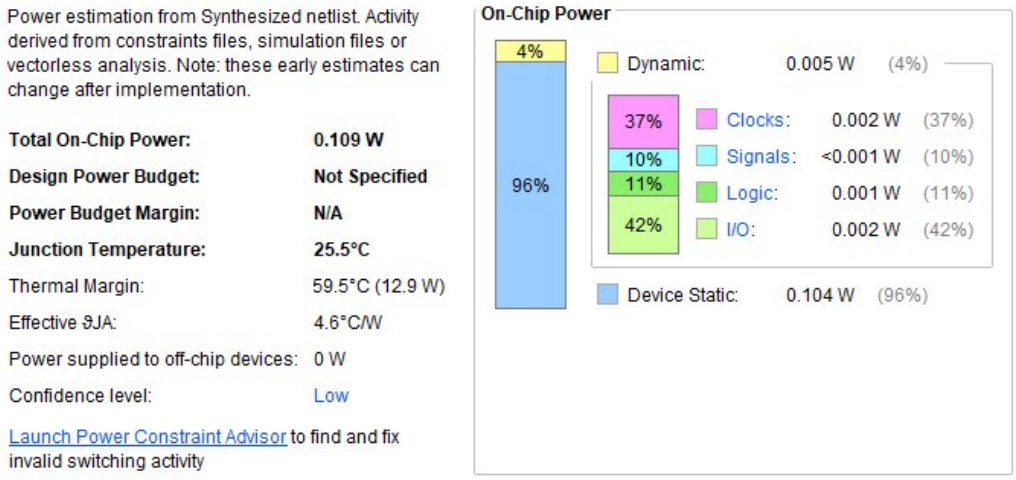

Figure 2.6: Summary of power estimation for 16-bit Montgomery multiplier

Table 2.1: Utilization summary of Montgomery multiplication

Resource	Utilization	Available	Utilization %
LUT	48	63400	0.08
FF	29	126800	0.02
IO	19	210	9.05
BUFG	1	12	3.13

For every iteration i in the algorithm, $u(i)$ is computed using a Full Adder, and the divide operation is performed using a right-shift register as shown in the architecture in the Figure 2.5. The entire algorithm is designed and implemented using Xilinx Vivado Design Suite. Figure 2.7 shows the simulation results of 16-bit Montgomery multiplication.

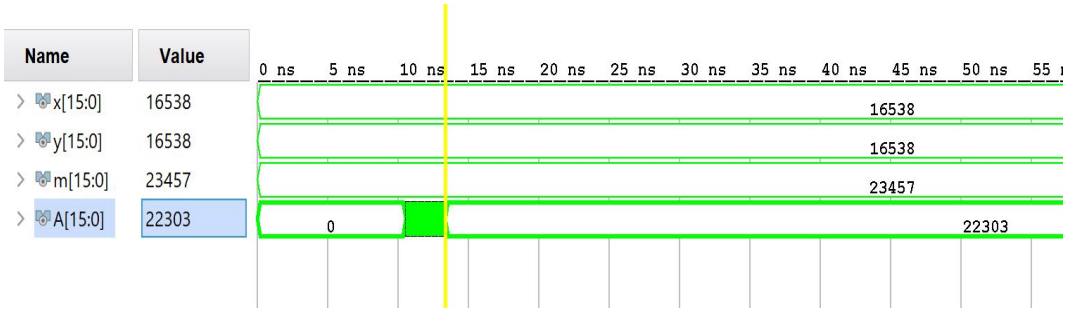

Figure 2.7: Simulation results of Montgomery multiplier from Xilinx Vivado Design Suite

Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is the language used to describe the hardware design. A test bench has been written to verify the design for variable bit sizes. The program has been then synthesized and implemented on a Nexys4 FPGA board. Table 2.1 shows the utilization summary on implementing the algorithm on FPGA board.

2.3.2 Montgomery Exponentiation

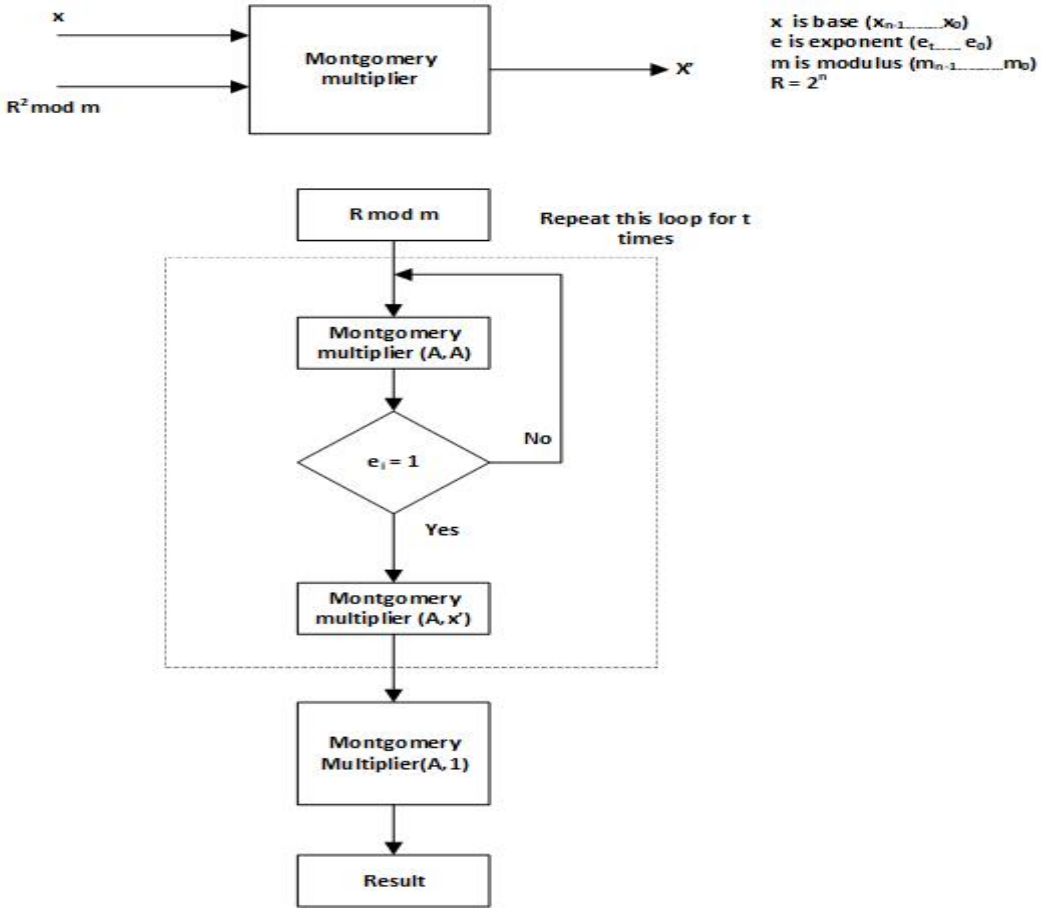

Figure 2.8: Flowchart for Montgomery Exponentiation

Figure 2.8 shows the architecture of Montgomery exponentiation. It involves 4 Montgomery multiplier blocks and is repeated over t iterations, where t is number of bits in the exponent.

$$Result(A) = x^e \, mod \, m \tag{2.8}$$

As the entire algorithm is carried out using Montgomery multiplication, result remain in Montgomery domain and the final step of equation (2.9) converts it to real world coordinates.

$$A = Mont(A, 1) \tag{2.9}$$

Figure 2.9 shows the simulation results of Montgomery exponentiation component for 16 bits using Xilinx Vivado Design Suite. Since each block of RSA algorithm was designed and developed as a stand-alone reusable component, they can be simulated to verify the design, synthesized and implemented on FPGA board. Montgomery exponentiation utilizes 3% of look-up tables on Nexys4 FPGA board as seen in Table 2.2.

Table 2.2: Utilization summary of Montgomery exponentiation algorithm

Resource	Utilization	Available	Utilization %
LUT	1796	63400	2.83
FF	2696	126800	2.13
IO	18	210	8.57
BUFG	2	32	6.25

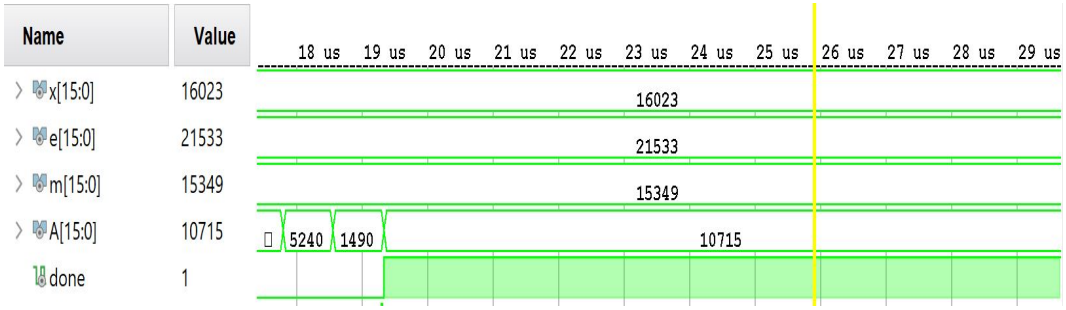

Figure 2.9: Simulation results of Montgomery exponentiation

2.3.3 Miller Rabin Primality Test

Primality tests are computationally expensive since it involves repeated exponentiation and modular multiplications. RSA algorithm begins with the need to generate two random prime numbers. Random numbers were generated on FPGA using Linear Feedback Shift Register (LFSR). These random numbers were passed onto a primality test before they can be used further in the algorithm. We chose Miller Rabin to confirm primality of random numbers. This is mainly due to the ability to configure witness count parameter in the algorithm. The higher the witness count, higher the number of iterations to generate a strong prime number. Another advantage of Miller Rabin is the way it breaks the loop as soon as it finds a composite number as shown in Figure 2.10.

Error probability of Miller Rabin is less than $(1/4)^t$, where t is witness count. This algorithm has also proven to be better than Fermet's primality test and Solovay-Strassen test due to the computational costs involved in calculating modular multiplications [14]. Figure 2.10 shows the usage of a reusable random number generator block and modular exponentiation block in Miller Rabin primality tester algorithm.

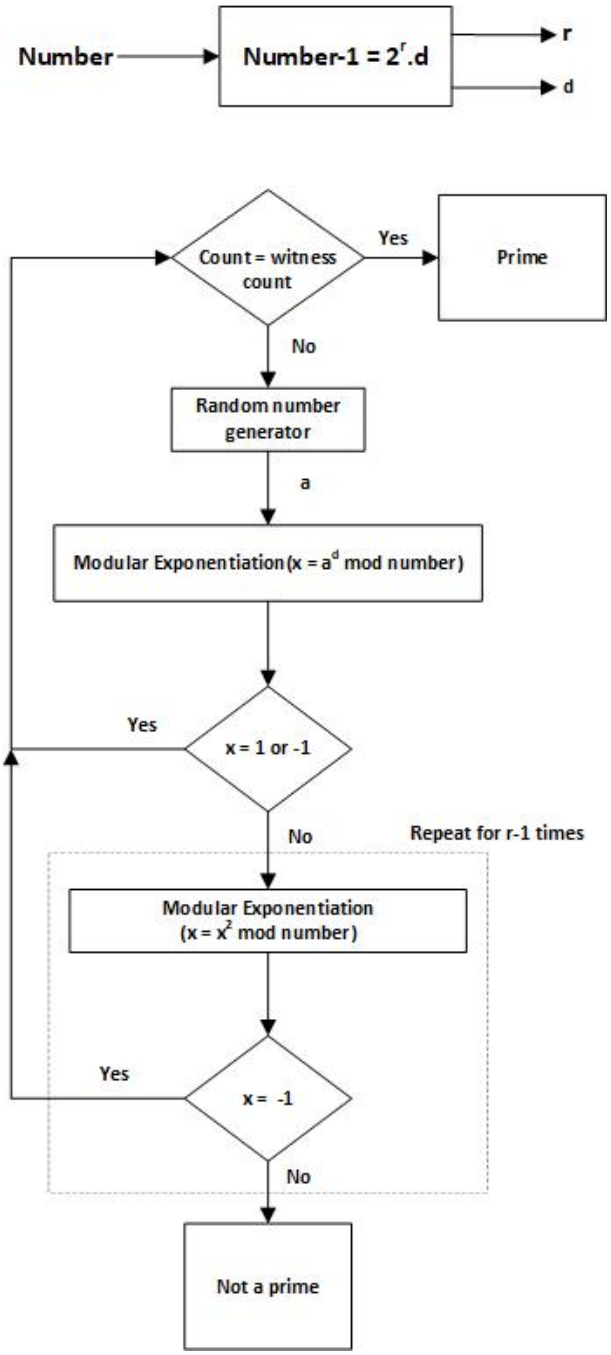

Figure 2.10: Architecture of Miller Rabin Primality Test

2.4 Evaluation

In terms of turnout time, Mostafizur Rahman's, et al. square and multiply technique [8] took 95μs to complete the exponentiation operation, but proposed Montgomery technique took only 19μs to complete the exponentiation. In terms of power, Prasanth kumar's, et al. clock cycle method [15] for exponentiation (32-bit) consumed 1.760 watt, power consumption using proposed Montgomery technique for exponentiation (32-bit) is 0.424 watt.

Chapter 3

Quantum Implementation of RSA using Qiskit

3.1 Difference between Classical computers and Quantum computers

Classical computers stores information in the form of bits - 0's and 1's. A bit can be imagined as a switch with two states - ON and OFF. But Quantum computers on the other hand, uses a special bit called Qubit. Qubit can be imagined as a point in a block sphere, where it is represented in |0> state, |1> state or a superposition state. |0> state and |1> states are equivalent of 0 and 1 bit state in classical computers. A superposition state is one, where the qubit exists in |0> state and |1> state at the same time, making it a unique aspect of quantum computers. In figure 3.1, superposition state is denoted as u3(0.5*pi,0,0) |0>. The value of the qubit can't be directly outputted. It must be measured and stored in classical bits before any read or write operations.

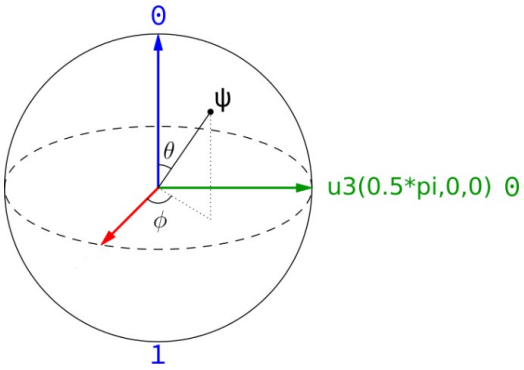

Figure 3.1: Block sphere for qubit representation

Four states can be represented using two bits in classical computers, but only one state can be represented at a time. Due to this, complex mathematical calculations take longer time to execute. Cryptographic algorithms are difficult to be broken due to this limitation. For example, security of RSA relies on factorization problem. It is easy to compute the product of two prime numbers. But, the reverse operation of trying to find the prime numbers from the product is compute-intensive. It takes millions of years to calculate in classical computers, since only one state can be representable at a time. While in quantum computers, 2^n states can be represented using n qubits simultaneously [16]. Adding one qubit to the system increases its power exponentially.

3.2 Introduction to Qiskit

Qiskit is an open source SDK (Software Development Kit) created by IBM, which allows us to access real Quantum computer via cloud [17]. Before starting to code for Quantum computers, an account needs to be created with IBM Q experience to interact with Quantum simulator. This generates a token, which provides access

to IBM's Quantum computer [18]. Programs can be written in Python in Jupyter Notebook to work with IBM's Quantum computer.

After installing Qiskit using `pip install` command, first step to start coding in the IBM's Quantum computer is to enable your account with the token provided in the IBM Q experience. This can be done by the following lines of code in Jupyter notebook.

```
from qiskit import IBMQ
provider = IBMQ.enable_account('<your-token>')
qasm_sim = provider.get_backend('ibmq_qasm_simulator')
```

3.3 Basic Quantum gates

When we execute any program in classical computers, it converts the instructions into sequence of operations called logic gates. The same rule applies for Quantum computers as well. The only difference is, we do have completely different sets of gates for Quantum computers with very rare similarities. Here are some basic gates used in Quantum computers [19].

X gate: It acts like a classical NOT, flipping a qubit |0> state to |1> state and |1> state to |0> state. This operation requires only one qubit [19].

CX gate (Controlled-X gate): This gate requires two qubits. One is called target qubit, and another is called control qubit. This gate applies an X gate to the target qubit only if control qubit is in the |1> state [19].

CCX gate (Controlled-Controlled-X gate): This gate acts on three qubits. Two qubits act as control qubits and one qubit acts as target qubit. This gate applies X gate to the target qubit, if only both control qubits are in the |1> state [19].

$SWAP$ gate: This gate acts on two qubits. A simple way to explain the effect of this gate is that all of the 0's and 1's in each state switch places. As a result, the SWAP gate has no effect on the states |00> and |11> [19].

- |00> swapped to |00>
- |01> swapped to |10>
- |10> swapped to |01>
- |11> swapped to |11>

H gate (Hadamard gate): This gate acts on single qubit. Hadamard is a half rotation of the Bloch sphere and it rotates around an axis located halfway between x and z. This gives it the effect of rotating states that point along the z axis to those pointing along x, and vice versa [19].

- H$|\ 0 > = |\ + >$
- H$|\ 1 > = |\ - >$
- H$|\ + > = |\ 0 >$
- H$|\ - > = |\ 1 >$.

3.4 Simple 7-bit Adder

Step 1: Import

Import necessary modules from Qiskit and Python. Quantum registers are needed to store and perform the calculation on our inputs. Qubits from quantum registers can't be printed directly in classical computers, hence they will be

measured and stored in classical registers. Import `ClassicalRegister` and `QuantumRegister` from Qiskit. Quantum circuit is needed to form a circuit in Quantum hardware. Module `execute` needs to be imported from Qiskit to execute a quantum circuit [18].

Once all the necessary modules are imported as below, enable your account if its not already enabled.

```
from qiskit import ClassicalRegister, QuantumRegister
from qiskit import QuantumCircuit
from qiskit import execute
from qiskit import IBMQ

provider = IBMQ.enable_account('<your-token>')
qasm_sim = provider.get_backend('ibmq_qasm_simulator')
```

Step 2: Get user input

Get two input numbers from user in the form of bits as shown in below code snippet. If the length of the first number is greater than the second number, then assign the first number length's to sum's length else do the opposite.

```
first = input("Enter a binary number with less than 8 digits")
second = input("Enter another number with less than 8 digits")


l = len(first)
l2 = len(second)
if l > l2:
     n = l
else:
     n = l2
```

Step 3: Initializing registers

```
#Initializing the registers; two quantum registers with n bits each
#1 more with n+1 bits, which will also hold the sum of the two #
   numbers
#The classical register has n+1 bits, which is used to make the sum
    #readable
a = QuantumRegister(n) #First number
b = QuantumRegister(n+1) #Second number, then sum
c = QuantumRegister(n) #Carry bits
cl = ClassicalRegister(n+1) #Classical output
#Combining all of them into one quantum circuit
qc = QuantumCircuit(a, b, c, cl)
```

Step 4: Loading the input to Quantum registers using X gate

When quantum registers are initialized, by default every qubit is in |0> state. To load appropriate value, every bit needs to be checked and X gate will be applied to make qubit in |1> state.

```
#Setting up the registers using the values inputted
for i in range(l):
    if first[i] == "1":
        qc.x(a[l - (i+1)]) #Flip the qubit from 0 to 1
for i in range(l2):
    if second[i] == "1":
        qc.x(b[l2 - (i+1)]) #Flip the qubit from 0 to 1
```

Quantum registers a and b hold the value of two numbers to be added.

Step 5: Calculate carry output and sum output

Table 3.1: Truth table to calculate carry output

C	A	B	C_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

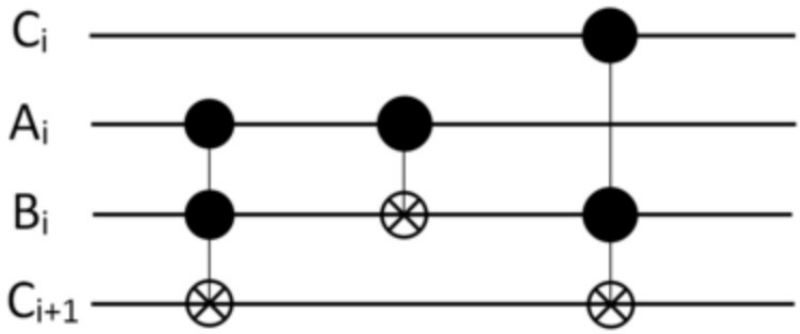

Figure 3.2: Quantum circuit for carry output

The lines with two filled in circles and one hollowed out circle with a cross represent the CCX gates, and the one with one of each circle represents the CX gate. Below code processes $n-2$ bits.

```
#Implementing a carry gate that is applied on all
#(c[i], a[i], b[i]) #with output fed to c[i+1]
for i in range(n-1):
    qc.ccx(a[i], b[i], c[i+1])
    qc.cx(a[i], b[i])
    qc.ccx(c[i], b[i], c[i+1])
```

Since b register is used to store the sum of a and b, instead of creating an extra carry bit for the last carry, and then transferring that to register b, last qubit of b was substituted for the last carry bit.

```
#For the last iteration of the carry gate,
#instead of feeding the #result to c[n],
#we use b[n], which is why c has only n bits,
#with #c[n-1] being the last carry bit
qc.ccx(a[n-1], b[n-1], b[n])
qc.cx(a[n-1], b[n-1])
qc.ccx(c[n-1], b[n-1], b[n])
```

Carry outputs are all stored in $c[i+1]$ and extra bit $b[n]$. Operation performed on input qubits needs to be reversed. Reversing is an easy operation, since all the Quantum logic gates are reversible by applying the same gate on the inputs. This is done to make sure the correct inputs are fed to calculate the sum output.

Sum can easily be calculated in classical computers using boolean logic gates. Challenge has been to find a quantum equivalent to perform the same operation. To calculate sum, a gate that can take three input qubits is needed: input carry qubit, and one qubit from each of the addends. The gate would perform an operation that would achieve the same results as shown in table 3.2

Table 3.2: Truth table to calculate sum

C	A	B	Sum
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Result couldn't be achieved by using one quantum gate, hence, multiple quantum gates will be combined as explained below:

1. Apply CX gate to C (Carry in) and B qubits where C is the control qubit and B is the target qubit. When C is in |1> state, flip the B qubit.

2. Apply CX gate to A and B qubits where A is the control qubit and B is the target qubit. When A is in |1> state, flip the B qubit.

3. Store the sum results in B qubit.

These steps are represented as a Quantum circuit as shown in figure 3.3

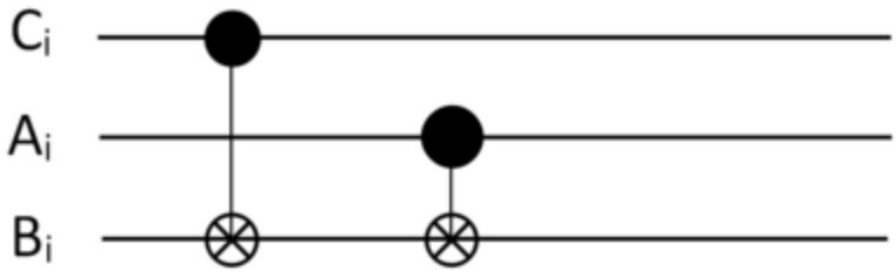

Figure 3.3: Quantum circuit for sum output

The line with one dark circle and one crossed circle represents CX gate.

```
#Reversing the gate operation performed on b[n-1]
qc.cx(c[n-1], b[n-1])
#Reversing the gate operations performed
#during the carry gate implementations
#This is done to ensure the sum gates are fed
#with the correct input bit states
for i in range(n-1):
    qc.ccx(c[(n-2)-i], b[(n-2)-i], c[(n-1)-i])
    qc.cx(a[(n-2)-i], b[(n-2)-i])
    qc.ccx(a[(n-2)-i], b[(n-2)-i], c[(n-1)-i])
    #These two operations act as a sum gate; if a control bit is at
    #the 1> state then the target bit b[(n-2)-i] is flipped
    qc.cx(c[(n-2)-i], b[(n-2)-i])
    qc.cx(a[(n-2)-i], b[(n-2)-i])
```

Step 6: Measure qubits and store the results in classical registers

Since qubits can't be extracted for any useful information, they will be measured and stored in classical registers.

```
#Measure qubits and store results in classical register cl
for i in range(n+1):
    qc.measure(b[i], cl[i])
```

Step 7: Execute job in IBM Q simulator

```
#Set chosen backend and execute job
num_shots = 2 #Setting the number of times to repeat measurement
job = execute(qc, qasm_sim, shots=num_shots)
#Get results of program
job_stats = job.result().get_counts()
print(job_stats)
```

3.5 Quantum Fourier Transform based Adder

Previous implementation of n-bit addrer shows a classical approach to solve the problem using Qiskit. To achieve fast addition, quantum-ness of qubits needs to be taken into account while programming the adder circuit. This section explains the implementation of fast adder using Quantum Fourier Transform [20].

In 2019, IBM is set to release its biggest quantum computer consisting of 53 qubits. However, at the time of this research, only 32 qubits quantum computer are available for public use, which can be used to add maximum 9 qubits because, $n + 1$ qubits is needed to store each inputs and another $n + 1$ qubits to store the output. A total of $3n + 3$ qubits are needed for quantum addition operation.

Step 1: Import modules and get user input

If user inputs are of different length, pad zeros to make it convinient to apply Quantum Fourier Transform (QFT)

```
from QFTAddition import add
from qiskit import ClassicalRegister, QuantumRegister
from qiskit import QuantumCircuit
from qiskit import execute
from qiskit import IBMQ
import math
import operator

provider = IBMQ.enable_account('<your-token>')
qasm_sim = provider.get_backend('ibmq_qasm_simulator')

#Take two numbers as user input in binary form
first = input("Enter a number with less than 10 digits.")
second = input("Enter another number with less than 10 digits.")

l1 = len(first)
l2 = len(second)
#Making sure that 'first' and 'second' are of the same length
#by padding the smaller string with zeros
if l2>l1:
    first,second = second, first
    l2, l1 = l1, l2
second = ("0")*(l1-l2) + second
```

Step 2: Create separate functions for QFT operations

There are three functions for QFT addition procedure:

1. CreateInputState()

2. evolveQFTState()

3. Inverse QFTState()

The first function CreateInputState() is used to convert the first number into QFT form by applying some of the quantum gates. Here are following rules to convert a number to QFT form.

• Let n be the length of first number.

• Apply a Hadamard gate on the qubit in register a with index n.

• Starting from the index of the qubit passed, count backwards until you reach zero. Let's call this number m.

• Each time you count down once, apply a controlled rotation gate, controlled by the qubit in register a with index n on the qubit in register a with index m.

• This rotation gate would be called with parameter equal to $\frac{\pi}{2^{n-m}}$

```
def createInputState(qc, reg, n, pie):
    """
    Apply one Hadamard gate to the nth qubit of the quantum
   register
    reg, and then apply repeated phase rotations with parameters
    being pi divided by increasing powers of two.
    """
    qc.h(reg[n])
    for i in range(0, n):
        qc.cu1(pie/float(2**(i+1)), reg[n-(i+1)], reg[n])
```

Next, evolveQFTstate() function is created to perform controlled rotations on the qubits of the register holding the first number, controlled by the qubits of the second number. Here are few rules for writing this function:

• Let n be the passed index.

• Starting from the index of the qubit passed, count backwards until you hit -1, given by the number m. Before you count down once, apply a controlled rotation gate controlled by the qubit in register b with index m on the qubit in register a with index n.

```
def evolveQFTState(qc, reg_a, reg_b, n, pie, factor):
    """
    Evolves the state |F(ψ(reg_a))> to |F(ψ(reg_a+reg_b))> using
   the
    quantum Fourier transform conditioned on the qubits of the
    reg_b. Apply repeated phase rotations with parameters being pi
    divided by increasing powers of two.
    """
    l = len(reg_b)
    for i in range(0, n + 1):
        if (n - i) > l - 1:
            pass
        else:
            qc.cu1(factor * pie / float(2**(i)), reg_b[n - i],
   reg_a[n])
```

inverseQFT() function would perform the same sequence performed in createInputState() function, but in reverse order. This would involve a new set of rules:

• Let n be the passed index.

• Starting from -1, count up until you reach n. Let's call the number you are at m.

• Each time you count up once, apply a controlled rotation gate, controlled by the qubit in register a with index n on the qubit in register a with index $n-m-1$.

• Apply a Hadamard gate to the qubit in register a with index n.

```
def inverseQFT(qc, reg, n, pie):
    """
    Performs the inverse quantum Fourier transform on a register
    reg. Apply repeated phase rotations with parameters being pi
    divided by decreasing powers of two, and then apply a Hadamard
    gate to the nth qubit of the register reg.
    """
    for i in range(0, n):
        qc.cu1(-1*pie/float(2**(n-i)), reg[i], reg[n])
    qc.h(reg[n])
```

Once all functions were defined, quantum circuits can be created to perform QFT based addition.

Step 4: Creating Quantum registers, Classical Registers and Quantum circuit

```
n = len(first)
pie = math.pi
a = QuantumRegister(n+1, "a") #Holds the first number
b = QuantumRegister(n+1, "b") #Holds the second number
cl = ClassicalRegister(n+1, "cl") #Holds the final output
qc = QuantumCircuit(a, b, cl, name="qc")
```

Step 5: Loading the input to Quantum registers

```
#Flip the corresponding qubit in register a if a bit in the
#string first is a 1
for i in range(0, n):
    if first[i] == "1":
        qc.x(a[n-(i+1)])
#Flip the corresponding qubit in register b if a bit in the
#string second is a 1
for i in range(0, n):
    if second[i] == "1":
        qc.x(b[n-(i+1)])
```

Step 6: Using QFT functions for addition operation

```
# Compute the Fourier transform of register a
for i in range(0, n):
    createInputState(circ, reg_a, n - (i+1), pie)
# Add the two numbers by evolving the Fourier transform
# F(ψ(reg_a))> to |F(ψ(reg_a+reg_b))>
for i in range(0, n):
```

```
    evolveQFTState(circ, reg_a, reg_b, n - (i+1), pie, factor)
# Compute the inverse Fourier transform of register a
for i in range(0, n):
    inverseQFT(circ, reg_a, i, pie)
```

Step 7: Executing the circuit in Quantum hardware using API token

```
for i in range(0, n+1):
    qc.measure(a[i], cl[i])

#Select backend and execute job
result = execute(qc, backend= qasm_sim, shots=512).result()
counts = result.get_counts("qc")
print(counts)
```

Quantum circuit of QFT adder is shown in figure3.4, figure 3.5 and figure 3.6.

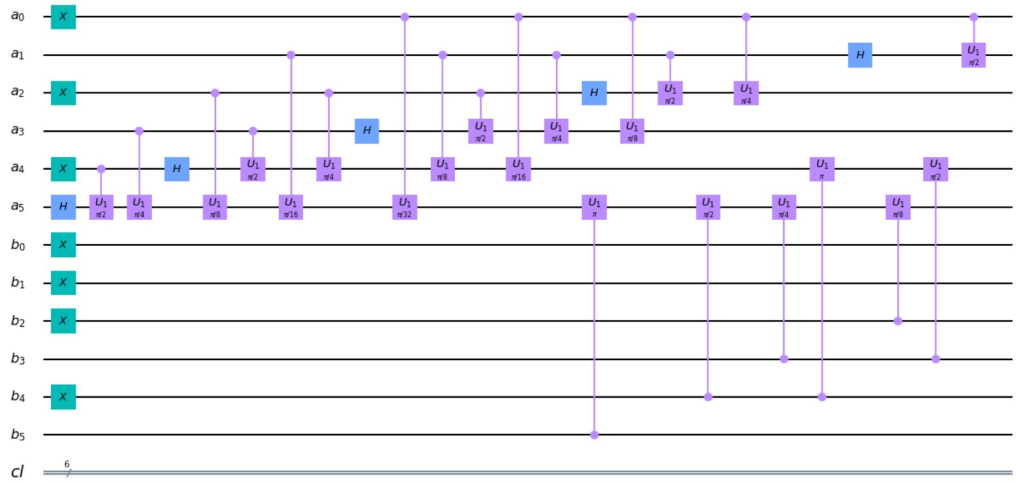

Figure 3.4: Quantum circuit of QFT adder

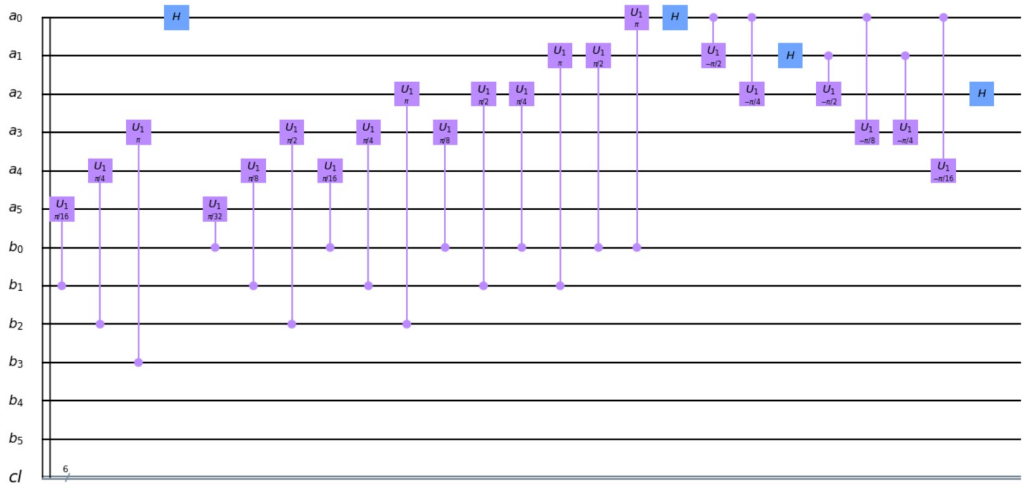

Figure 3.5: Quantum circuit of QFT adder

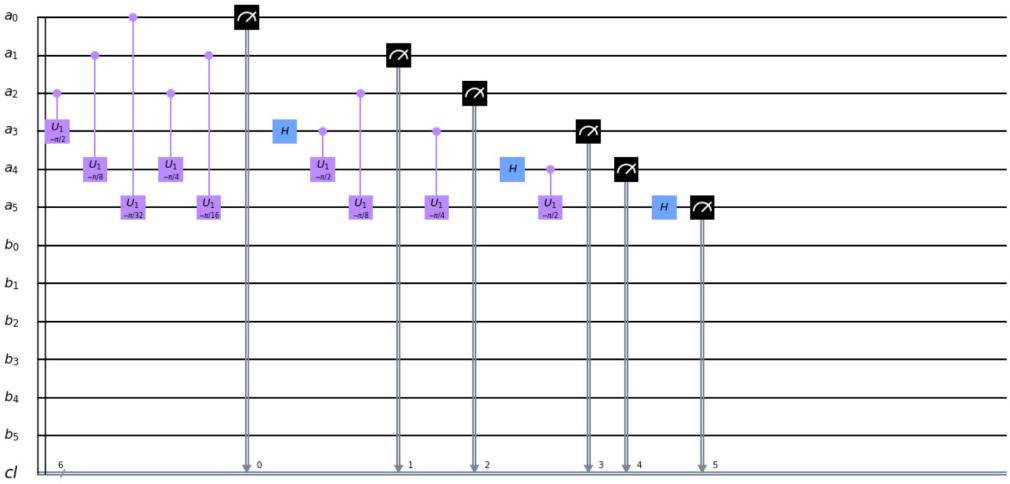

Figure 3.6: Quantum circuit of QFT adder

3.6 Multiplication using QFT adder

Multiplication operation can be performed using repeated addition. This module was developed using QFT adder described in the previous section [21].

This multiplication module is designed for multiplying p and q and to calculate $\phi(n)$ in RSA crypto algorithm. This function takes two classical numbers as arguments since the generated prime number function returns classical number as an output and then loads these numbers into Quantum registers for further Quantum operations. To form a multiplication module, below procedure is followed:

- Create a zero initialized accumulator.
- Add the multiplicand to the accumulator.
- Decrement the multiplier by one.
- Repeat the last two steps until the multiplier becomes zero.

For Example, $13 * 3 = 39$ can be performed using the above rules as shown in Table 3.3

Table 3.3: Steps to perform multiplication

Iteration	*Multiplicand*	*Muliplier*	*Accumulator*
0	1101	11	0
1	1101	10	1101
2	1101	1	11010
3	1101	0	100111

To decrement the multiplier, *factor* was set to -1 in QFT adder to perform subtraction.

```
def sub(reg_a, reg_b, circ, factor = -1):
    """
    Add two quantum registers reg_a and reg_b, and store the result
    in reg_a.
    """
    pie = math.pi
    n = len(reg_a)
    # Compute the Fourier transform of register a
    for i in range(0, n):
        createInputState(circ, reg_a, n - (i+1), pie)
    # Add the two numbers by evolving the Fourier transform
    # F(ψ(reg_a))> to |F(ψ(reg_a+reg_b))>
    for i in range(0, n):
        evolveQFTState(circ, reg_a, reg_b, n - (i+1), pie, factor)
    # Compute the inverse Fourier transform of register a
    for i in range(0, n):
        inverseQFT(circ, reg_a, i, pie)
```

```
def mul(multiplicand_in, multiplier_in):
    # Take two numbers as user input in binary form
    # multiplicand_in = input("Enter the multiplicand.")
    l1 = len(multiplicand_in)
    # multiplier_in = input("Enter the multiplier.")
    l2 = len(multiplier_in)
    # Make sure multiplicand_in holds the larger number
    if l2 > l1:
        multiplier_in, multiplicand_in = multiplicand_in,
   multiplier_in
        l2, l1 = l1, l2

    accumulator = QuantumRegister(l1 + l2)
    multiplicand = QuantumRegister(l1)
    multiplier = QuantumRegister(l2)
    d = QuantumRegister(1)

    cl_accumulator = ClassicalRegister(l1 + l2)
    cl_multiplier = ClassicalRegister(l2)

    circ_accumulator = QuantumCircuit(accumulator, multiplicand,
   cl_accumulator)
    circ_multiplier = QuantumCircuit(multiplier, d, cl_multiplier)
    circ_multiplier.x(d[0])

    for i in range(l1):
        if multiplicand_in[i] == '1':
            circ_accumulator.x(multiplicand[l1 - i - 1])
    for i in range(l2):
```

```
        if multiplier_in[i] == '1':
            circ_multiplier.x(multiplier[l2 - i - 1])

    multiplier_str = '1'
    while(int(multiplier_str) != 0):
        add(accumulator, multiplicand, circ_accumulator)
        sub(multiplier, d, circ_multiplier)
        circ_multiplier.measure(multiplier, cl_multiplier)
        result = execute(circ_multiplier, backend=Aer.get_backend('
qasm_simulator'), shots=2).result().get_counts(circ_multiplier)
        multiplier_str = list(result.keys())[0]

    circ_accumulator.measure(accumulator, cl_accumulator)
    result = execute(circ_accumulator, backend=Aer.get_backend('
qasm_simulator'), shots=2).result().get_counts(circ_accumulator
)
    total = list(result.keys())[0]
    return total
```

3.7 Montgomery Multiplication

The main advantage of Montgomery approach is that it avoids the division operation needed for modular multiplication[14]. As it is implemented in FPGA in chapter 2, algorithm 1 can be implemented for Quantum RSA using IBM's Qiskit. Different approaches have been tried to implement the algorithm in efficient way to make sure it performs better and uses lesser number of qubits, since qubits are limited in IBM Q as it is in early stages of development.

As an initial step, length of x input was measured and Quantum registers and Classical registers were assigned with extra bits as needed for the operation.

```
# Measuring the length of x
n = len(x)

# Assigning Quantum registers for Quantum operation
x_reg = QuantumRegister(n+1)
y_reg = QuantumRegister(n+2)
y_reg_0 = QuantumRegister(1)
m_reg = QuantumRegister(n+2)
a_reg = QuantumRegister(n+2)
u_reg = QuantumRegister(n+1)
onecubit = QuantumRegister(1)

# Assigning classical registers to store our results from Quantum
   registers
a_cl_reg = ClassicalRegister(n+2)
u_cl_reg = ClassicalRegister(n+1)
cl_reg = ClassicalRegister(n+1)
one_cl_reg = ClassicalRegister(1)
```

Separate Quantum circuits was created for each sets of addition in the algorithm. This way of implementing circuits makes sure the performance is faster than if they are implemented in a single circuit.

```
# creating seperate Quantum circuits for different operations to
    speedup the process
circ_u = QuantumCircuit(u_reg, y_reg_0, u_cl_reg)
circ_a = QuantumCircuit(a_reg, y_reg, m_reg, a_cl_reg, onecubit,
    one_cl_reg)
```

Inputs were loaded into Quantum registers using the below code snippet

```
# Loading inputs to Quantum registers
for i in range(n):
    if y[i] == '1':
        circ_a.x(y_reg[n - i - 1])
for i in range(n):
    if m[i] == '1':
        circ_a.x(m_reg[n - i - 1])
```

Addition was performed using QFT adder in the intermediate steps of Montgomery modular multiplication. In order to take decision according to the addition results, qubits were measured in each iteration.

```
for i in range(n):
    if x[n - i - 1] == '1':
        add(u_reg, y_reg_0, circ_u)
    circ_u.measure(u_reg[0], u_cl_reg[0])
    result = execute(circ_u,backend=Aer.get_backend('qasm_simulator
    '), shots=1).result().get_counts(circ_u)
    measure_u = int((list(result.keys())[0]))
    print('measure_u: ', measure_u)
```

```
    if x[n - i - 1] == '1':
        add(a_reg, y_reg, circ_a)
    if measure_u == 1:
        add(a_reg, m_reg, circ_a)
    rshift(circ_a, a_reg, n + 2, onecubit)
    circ_a.measure(a_reg, a_cl_reg)
    circ_a.measure(onecubit, one_cl_reg)
    result = execute(circ_a,backend=Aer.get_backend('qasm_simulator
'), shots=1).result().get_counts(circ_a)
    total = list(result.keys())[0]
    measure_a = total[2:]
    print(measure_a)

    measure_onecubit = int(total[0])
    if measure_onecubit == 1:
        circ_a.x(onecubit)

    # loading a0 to u0
    if measure_a[n + 1] == '1':
        if measure_u == 0:
            circ_u.x(u_reg[0])
    else:
        if measure_u == 1:
            circ_u.x(u_reg[0])
```

As a final step, if calculated answer is greater than the m value, then it is subtracted from m to get the final answer. Else, the obtained answer is the final result.

```
if (int(measure_a) >= int(m)):
    sub(a_reg, m_reg, circ_a)


circ_a.measure(a_reg, a_cl_reg)
result = execute(circ_a,backend=Aer.get_backend('qasm_simulator'),
   shots=1).result().get_counts(circ_a)
total = list(result.keys())[0]
final_a = total[2:]
print(final_a)
```

Results of Montgomery multiplication is shown in figure 3.7. This function outputs values in montgomery domain as they will be further used in exponentiation operation.

```
Enter the binary number for x: 110
Enter the binary number for y: 110
Enter the binary number for m: 111
Result for Montgomery modular multiplication:  00001
Time taken to execute: 0.7048022747039795 seconds
```

Figure 3.7: Results for Montgomery modular multiplication using Qiskit

3.8 Montgomery Exponentiation

Once modular multiplication is performed using Montgomery technique, the same can be used multiple times with different parameters in order to achieve exponentiation operation [14]. Refer figure 2.8 for internal details of Montgomery Exponentiation algorithm.

```
def Mont_Exp(x, e, m):
    L = len(m)

    # R calculation
    R = 2**L

    # R mod m or Initial A calculation:
    R_mod_m = R % int(m,2) #int('string', base)
    A = str(R_mod_m)
    A = str(bin(int(A))[2:]).zfill(L)

    # R^2 mod m calculation
    R_square_mod_m = str(((R**2) % int(m,2)))
    T = len(e)
    R_square_mod_m = str((bin(int(R_square_mod_m))[2:])).zfill(L)

    # writing input x in montgomery domain called x_dash
    x_dash = Mont_Mul(x, R_square_mod_m, m)

    for i in range(0,T):
        if (len(A) > L):
            A = A[2:L+2]
        A = Mont_Mul(A, A, m)
        if e[i] == '1':
            A = Mont_Mul(A[2:L+2], x_dash[2:L+2], m)


    A = Mont_Mul(A[2:L+2], '1'.zfill(L), m)
    return A
```

```
Enter the binary number for x: 110
Enter the binary number for e: 11
Enter the binary number for m: 111
Result for Montgomery modular exponentiation:  00110
Time taken to execute: 4.791259765625 seconds
```

Figure 3.8: Results for Montgomery modular exponentiation using Qiskit

3.9 Modular Inverse

Modular inverse is an essential operation to find the private key d for RSA decryption. Montgomogery modular exponentiation can be used to calculate modular inverse only if the modular value is an odd integer. Since, even modular integers are involved in finding a private key, exponentiation algorithm can't be used. This demands a separate algorithm to calculate modular inverse. The disadvantage of using a conventional GCD algorithms is that, it requires multiple precision division operations [14]. Binary extended GCD algorithm eliminates this requirement at the expense of more iterations. $271^{-1} \, mod \, 383 = 106$ is an example for modular inverse calculation, which is shown in figure 3.9

```
Enter x in binary 101111111
Enter y in binary 100001111
Results for modular inverse:  1101010
Time taken to execute: 0.0009963512420654297 seconds
```

Figure 3.9: Results for Binary Extended GCD algorithm for modular inverse

3.10 Encryption and Decryption

Since Montgomery multiplication, exponentiation and modular inversion are implemented as efficient functions, they can be used to perform encryption and decryption for RSA algorithm. C denotes encrypted/cipher text and M denotes decrypted/original text.

```
d = Mod_Inv(PhiofN,e)

# Getting message from User
M = input("Enter the message in binary")

# Encryption process or calculating cipher text
C = Mont_Exp(M, e, N)
print('Encrypted text: ', C)

# Decryption process or calculating original text
M = Mont_Exp(C, d, N)
print('Original Text: ', M)
```

```
Enter the keysize3
P =  7
Q =  5
N =  100011
PhiofN = 011000
e =  101
d =  101
Enter the message in binary010010
Encrypted text:  00010111
--- 1628.376312494278 seconds ---
Original Text:  00010010
--- 2957.5948736667633 seconds ---
```

Figure 3.10: Results for RSA Encryption and Decryption using Qiskit

3.11 Conclusion

RSA algorithm is successfully implemented on FPGA using Montgomery technique and implemented on qauntum hardware using IBM's qiskit for performance comparison. Results for digital implementation were compared with others work to prove proposed method of implementing RSA is efficient. Due to the limitations of qubits available in today's quantum computers, 6-bit RSA is implemented in this research. But, the design proposed here using Qiskit can be extended for n-bits with advancements to quantum computers.

Bibliography

[1] Gary C. Kessler: An Overview of Cryptography, Handbook on Local Area Networks, Auerbach (1998).

[2] IBM Knowledge Center: What is DES and AES, https://www.ibm.com/support/knowledgecenter

[3] Craig Gidney, Martin Eker: How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits, arXiv preprint arXiv:1905.09749(2016).

[4] Patrick Nohe: Re-Hashed - The Difference Between SHA-1, SHA-2 and SHA-256 Hash Algorithms, https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms/ (2018).

[5] NIST: Post-Quantum Cryptography, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography (2019).

[6] S. Nirmala, R. Praveena: Implementation of efficient modular bypass multiplier logic in rsa cryptographic processor, International Conference and Workshop on Electronics and Telecommunication Engineering (2016).

[7] X. Yan, G. Wu, D. Wu, F. Zheng, X. Xie: An implementation of montgomery modular multiplication on fpgas, 2013 International Conference on Information Science and Cloud Computing (2013).

[8] M. Rahman, I. R. Rokon: Efficient hardware implementation of rsa cryptography, 2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication (2009).

[9] Mario Alberto García Martínez, Guillermo Morales Luna, Francisco Rodríguez Henríquez: Hardware Implementation of the Binary Method for Exponentiation in GF(2m). Proceedings of the Fourth Mexican International Conference on Computer Science, Tlaxcala, Mexico, Mexico (2003).

[10] Hans Eberle, Nils Gura, Sheueling Chang Shantz, Vipul Gupta, Leonard Rarick: A public-key cryptographic processor for RSA and ECC, Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Galveston, TX, USA (2004).

[11] Sushanta Kumar Sahu, Manoranjan Pradhan: Implementation of Modular multiplication for RSA Algorithm. International Conference on Communication Systems and Network Technologies, Katra, Jammu, India (2011).

[12] Manish Bansal, Amit Kumar, Aakanksha Devrari, Abhinav Bhat: Implementation of Modular exponentiation using Montgomery algorithms. International Journal of Scientific and Engineering Research, Volume 6, Issue 11 (2015).

[13] Nadia Nedjah, Luiza De Macedo Mourelle: Hardware Architecture for the Montgomery Modular Multiplication. International Conference on Information Technology Coding and Computing, Las Vegas, NV, USA (2004).

[14] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone: Handbook of Applied Cryptography. 5th Edition (2001).

[15] S. P. kumar, K. kumar, B.Partibane: Efficient modular exponentiation architectures for rsa algorithm, International Journal of Engineering Research in Electronic and Communication Engineering (IJERECE) Vol 3, Issue 5 (2016).

[16] Jlspursfan: Quantum Computing, https://medium.com/swlh/quantum-computing-1d40d4ed43b2 (2019).

[17] Daniel Koch, Laura Wessing, Paul M. Alsing: Introduction to Coding Quantum Algorithms: A Tutorial Series Using Qiskit, arXiv:1903.04359v1 [quant-ph] (2019).

[18] Sashwat Anagolum: Arithmetic on Quantum Computers - Addition, https://medium.com/@sashwat.anagolum/arithmetic-on-quantum-computers-addition-7e0d700f53ae (2018).

[19] Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, David McKay, Zlatko Minev, Paul Nation, Anna Phan, Arthur Rattew, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, James Wootton: Learn Quantum Computation using Qiskit (2019).

[20] Sashwat Anagolum: Arithmetic on Quantum Computers - Addition, Faster, https://medium.com/@sashwat.anagolum/qftaddition-ce0a0b2bc4f4 (2018).

[21] Sashwat Anagolum: Arithmetic on Quantum Computers - Multiplication, https://medium.com/@sashwat.anagolum/arithmetic-on-quantum-computers-multiplication-4482cdc2d83b (2018).