National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Physics

Quantum Circuits of Quantum-Safe Encryption

Presented by

Javaria Ghafoor (207845)
Alizah Gul Memon (225523)

A thesis submitted in
partial fulfilment of the requirements of the
Degree of Bachelor of Science in Physics

May 28, 2021

Dr. Aeysha Khalique Supervisor
Dr. Sana Qadir Co-Supervisor

Abstract

In a post-quantum era, there are certain classical encryption protocols that remain safe from attacks by a quantum computer. The Advanced Encryption Standard (AES) is one such encryption protocol, that uses symmetric key distribution and a block cipher model to encode and decode critical information. In our research, we studied how to apply the AES on a quantum computer, and how to make the circuit more cost-efficient based on previous models.

We started by looking at the AES in detail, understanding how each of its components worked. For that, we explored the Galois field, which is an important element of cryptography in general, and also vital in the AES. We then went on to see the fundamentals of quantum computing, and saw certain quantum gates, as well as quantum equivalent of classical gates that can be used to implement a quantum circuit of the AES.

We studied LUP decomposition, the Itoh-Tsuji Algorithm, and affine transformation as part of operations in encryption/decryption and the S-box of the AES. Once we covered all the background details, we moved onto a detailed literature review of implementation of AES as a Quantum Circuit and its cost reduction.

Acknowledgments

All praise and thanks to God Almighty, Who Helped us complete this project even when all odds seemed against us. Special thanks to our families, who looked after us when we both had COVID, and Dr. Aeysha and Dr. Sana, for their assistance, motivation, and guidance.

This thesis is not just an amalgamation of these past few months, but instead an encapsulation of our entire degree, which challenged us to become more resilient, and persevere on the journey of physics.

Contents

List of Figures

Chapter 1

Galios Field

"Unfortunately what is little recognized is that the most worthwhile scientific books are those in which the author clearly indicates what he does not know; for an author most hurts his readers by concealing difficulties."

– *Évariste Galois*

1.1 Definition

The Galois Field [3], named after Evariste Galois, is a field for a finite number of elements. Because computers work with bits and bytes, it is much easier to represent data as vectors in this finite field, and manipulate it accordingly. Its elements can be defined as

$$\begin{aligned} gf\left(p^n\right) =& (0,1,2,\ldots,p-1) \cup \\ & (p,p+1,p+2,\ldots,p+p-1) \cup \\ & \left(p^2,p^2+1,p^2+2,\ldots,p^2+p-1\right) \cup \ldots \cup \\ & \left(p^{n-1},p^{n-1}+1,p^{n-1}+2,\ldots,p^{n-1}+p-1\right) \end{aligned} \tag{1.1}$$

where $p \in \mathbb{P}$ and $n \in \mathbb{Z}^+$. The order of the field is given by p^n while p is called the characteristic of the field. Galois Field can also be referred to as gf. The degree of polynomial of each element in this field is at most $n-1$.

1.1.1 Examples

$$gf(6) = (0,1,2,3,4,5)$$

which contains 6 elements where each is a degree 0 polynomial, while

$$\begin{aligned} gf\left(2^3\right) &= \left(0, 1, 2, 2+1, 2^2, 2^2+1, 2^2+2, 2^2+2+1\right) \\ &= (0, 1, 2, 3, 4, 5, 6, 7) \end{aligned}$$

which contains $2^3 = 8$ elements where is a degree 2 or less polynomial.

1.2 Representation of Elements

1.2.1 Decimal System

Counting as we generally know it is done in the decimal system, which is a base-10 number system. This means that each value is represented using digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

1.2.2 Binary System

The binary system represents elements in terms of 0 and 1, making it base-2.

To convert decimal values to binary, we write the decimal number as sums of $a_n 2^n$. This can be written as

$$x = \sum_{n \in \mathbb{N}} a_n 2^n$$

Example

Convert 13_{10} to binary:
13 in decimal can be written as

$$8 + 4 + 1$$

In terms of powers of 2, this translates to

$$2^3 + 2^2 + 2^0$$

which can also be written as

$$1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0$$

and now, we just add zeros for the missing terms in the series

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Once that is done, we just remove the powers of two and keep the 0's and 1's to get

$$13_{10} = 1101_2$$

1.2.3 Hexadecimal System

Just like the binary system is base-2, and decimal base-10, as the name suggests, the hexadecimal system is base-16. This means that it uses 16 numbers/characters to represent elements.

The 16 characters are: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$.

The hexadecimal system is of importance to us because that is the system used by the Advanced Encryption Standard (AES) to encrypt and decrypt plain text.

1.2.4 Bits & Bytes

Binary digits are called ***"bits"*** for short. Bits are used to quantify computing data, where all information is represented in 0's and 1's. In terms of the Galois Field, bits are elements of $gf(2)$.

Eight bits comes together to form ***"bytes"***, making them elements of $gf(2^8)$. We are considering bytes, because that is what computers use as the smallest unit to store data.

If any character less than 8 bits long, we simply add preceding zeros. For example, the number 7 in binary is 111, so in $gf(2^8)$, we write it as 00000111. It follows that the largest value a byte can store is 11111111, which is

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 255$$

1.2.5 ASCII

The American Standard Code for Information Interchange (ASCII) has exactly 255 characters, so that each character can be assigned a byte value, making each character of ASCII an element of $gf(2^8)$.

Some examples of character assignments in ASCII are decimal values 65-90 for capital letters, and 97-122 for lowercase letters. These decimal values can then be converted to binary, and written as bytes.

1.3 Finite Field Arithmetic

Working in the Galois Field (finite field) is different to the Euclidean space, and we will show how basic operations are performed.

1.3.1 Addition

Addition in the Galois Field involves converting decimal values to binary, and then adding them *modulo 2*. What that means is that if the sum of the two digits is divisible by 2, we denote it as 0, else 1. This is the same as performing an exclusive-OR (XOR) operation.

$$0 + 0 = 0$$

$$1 + 1 = 2$$

but since 2 is divisible by 2, we write $1 + 1 = 0$ *(mod 2)*

$$0 + 1 = 1 = 1 + 0$$

Example

Adding the decimal numbers 50 and 79

$$50 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110010$$

$$79 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1001111$$

In $gf(2^8)$:

$$\begin{aligned} & 50 + 79 \\ &= (2^5 + 2^4 + 2^1) + (2^6 + 2^3 + 2^2 + 2^1 + 2^0) \\ &= 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + (2 \cdot 2^1) + 2^0 \\ &= 129 \end{aligned}$$

And in binary notation,

$$\begin{aligned} & 50 + 79 \\ &= 00110010 + 1001111 \\ &= 10000001 \\ &= 1.2^7 + 0.2^6 + 0.2^5 + 0.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 1.2^0 \\ &= 129 \end{aligned}$$

1.3.2 Subtraction

Subtraction in the Galois Field is very similar to addition, because the differences are the same as the sums

$$1 - 1 = 0$$

$$0 - 0 = 0$$

$$0 - 1 = 1 = 1 - 0$$

1.3.3 Multiplication & Multiplicative Inverse

Multiplication in Galois Field, is more strenuous. Unlike the XOR operation for addition and subtraction, multiplication in the Galois Field is equivalent to applying the AND operation. If $f(p)$ and $g(p)$ are polynomials in $gf\left(p^n\right)$ and let $m(p)$ be an irreducible polynomial, of degree at least n in $gf\left(p^n\right)$. We keep $m(p)$ as a polynomial of degree at least n so that the product of $f(p)$ and $g(p)$ does not exceed $11111111 = 255$, as the product needs to be stored as a byte. If $h(p)$ denotes the resulting product,

$$h(p) = (f(p) \cdot g(p))(\text{mod}(m(p)))$$

The multiplicative inverse of $f(p)$ is given by $a(p)$ such that

$$(f(p) \cdot a(p))(\text{mod}(m(p))) = 1$$

The product and multiplicative inverse of two polynomials involves reduction of the coefficients p, and reduction of polynomials modulo $m(p)$. We can find the reduced polynomial using long division, but the best way to calculate the multiplicative inverse is by using Extended Euclidean Algorithm. Details are mentioned in the example below.

Example

In $gf\left(2^8\right)$ the irreducible polynomial modulo $m(p)$ is $p^8 + p^6 + p^5 + p^1 + p^0$. To calculate $84 \cdot 13$, we need to first find the product of the polynomial and then reduce the coefficients modulo 2.

$$\begin{aligned} 84 \cdot 13 &= \left(\left(2^6 + 2^4 + 2^2\right) \cdot \left(2^3 + 2^2 + 2^0\right)\right)(\text{mod}(m(p))) \\ &= \left(2^9 + 2^8 + 2^7 + 2 \cdot 2^6 + 2^5 + 2 \cdot 2^4 + 2^2\right)(\text{mod}(m(p))) \\ &= \left(2^9 + 2^8 + 2^7 + 2^5 + 2^2\right)(\text{mod}(m(p))) \end{aligned}$$

Then we use long division to compute the reduced polynomial as follows

Remainder	Quotient
$2^9 + 2^8 + 2^7 + 2^5 + 2^2$	
$2^8 + 2^6 + 2^5 + 2^1 + 2^0$	
2^0	$2^1 + 2^0$

where the first column's last entry is the product we were looking for. Since the product is 1, we can also deduce that 84 and 13 are multiplicative inverse pairs.

Example

Let us now assume that we do not know the multiplicative inverse of 84. We would use Extended Euclidean Algorithm to find it. We need to keep a track of the auxiliary as follows:

Remainder	Quotient	Auxiliary
$2^8 + 2^6 + 2^5 + 2^1 + 2^0$		0
$2^6 + 2^4 + 2^2$		1
$2^5 + 2^4 + 2^1 + 2^0$	2^2	2^2
2^0	$2^1 + 2^0$	$2^3 + 2^2 + 1$

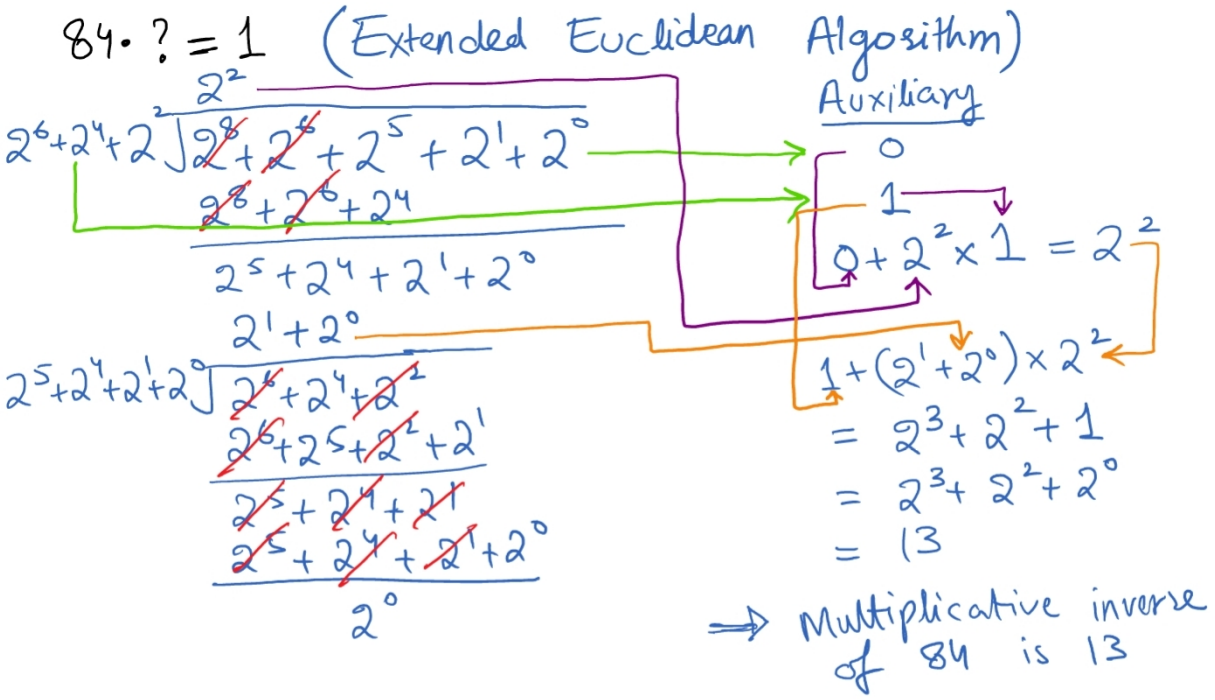

Figure 1.1: Extended Euclidean algorithm dry run

1.4 Application in Cryptography

Galois Field is most commonly used in cryptography. Every bit and byte is represented in the finite field as a vector, so it is very straight-forward to encrypt and decrypt information, using mathematical arithmetic. manipulable.

IBM developed its Data Encryption Standard (DES) in the 1970's, which uses a 56-bit key. But because technology continues to advance, a supercomputer succeeded in breaking

the key in less than 24 hours, proving the need for a more sophisticated algorithm. Vincent Rijmen and John Daemon came with a solution in 2001, when they presented the more complicated algorithm, Rijndael, and it has been the Advanced Encryption Standard (AES) ever since.

Chapter 2

Quantum Computation and Gates

"Quantum computation is... nothing less than a distinctly new way of harnessing nature... It will be the first technology that allows useful tasks to be performed in collaboration between parallel universes, and then sharing the results."

– *David Deutsch, in The Fabric of Reality (1997)*

2.1 Quantum Computation

2.1.1 Quantum Computation

Quantum computers leverage the properties of quantum mechanics to perform computation. There are three major properties of quantum mechanics that are involved in quantum computing:

1. Superposition
2. Entanglement
3. Interference

Superposition

Superposition allows a quantum system to be in multiple states simultaneously.

The famous thought experiment to demonstrate this idea is Schrödinger's cat. In it, a cat is placed into a box, along with a radioactive element and some poison. When the element decays, the poison is released, and the cat dies, but until that actually happens, the

cat is dead and alive at the same time, until someone opens the box to see the cat's final state.

On an atomic level, superposition occurs in electrons, where they have spin up or down, which can only be determined once the electrons are actually measured.

Entanglement

A group of particles are entangled when they interact with each other in a way that the quantum state of one particle of the group cannot be described independently of the others'.

A way of imagining this is having a set of gloves in two separate boxes. Even if you put one box on the Moon, if you open the other box and find the right glove in it, you would know that the other glove in the other box is the left one, even if you did not open the box on the Moon.

Again, in the case of electrons, if entangle two of them and measure one to have spin up state, we would know for certain that the other electron will be in the spin down state, even if we did not measure it, and even if it is far away from its entangled counterpart.

Interference

Interference is the phenomenon in which two waves combine to form a resultant wave of greater, lower, or the same amplitude.

Interference has been demonstrated in Young's double-slit experiment, where light passed through slits is observed on a screen, resulting in the formation of fringes - some being brighter than others. The bright fringes are a result of constructive interference, while the dimmer fringes are caused by destructive interference.

2.1.2 Quantum Supremacy

Leveraging the properties of quantum mechanics, quantum computers have a promising future ahead of them, so much that scientists are trying to achieve *quantum supremacy* - a 'quantum advantage' where quantum computers manage to solve problems which a classical computer cannot solve in a feasible amount of time.

Google [4] first claimed to achieve quantum supremacy in 2019, when its 54-qubit Sycamore processor was able to perform a calculation in 200 seconds that would have taken the world's most powerful supercomputer 10,000 years. Companies like Honeywell [5] also got into the competition, and recently, a Chinese [6] team came forth with a 62-qubit quantum processor.

2.1.3 Post-Quantum Cryptography

With the advent of quantum computers that can solve certain problems much faster than classical computers, we are now in a *'post-quantum'* era where cryptography needs to be secure against attacks by both, quantum and classical computers.

Currently popular algorithms involve mathematical problems such as integer factorization, discrete logarithm problems, or elliptic-curve discrete logarithm problems. In 1994, Peter Shor introduced Shor's algorithm, and with a strong quantum computer, Shor's algorithm can easily solve these problems, which poses a threat to such cryptographic algorithms.

However, experimental quantum computers of today do not possess sufficient processing power to break any real cryptographic algorithm. Symmetric cryptographic algorithms and hash functions are relatively secure against quantum computers, and even though Grover's algorithm may speed up attacks against symmetric ciphers, doubling the key size can effectively block these attacks. Thus, post-quantum symmetric cryptography does not need to be too different from current symmetric cryptography.

2.2 Qubits

Now that we now what quantum computing is, we will look at how quantum computing works. While classical computers have bits (0's and 1's), quantum computers work with quantum bits, or "qubits" for short. Qubits are two-state quantum mechanical systems, and are represented as $|0\rangle \, and \, |1\rangle \, , where$

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

2.3 Quantum Gates

Quantum gates are basic quantum circuits operating on qubits. Some of the main gates involved in our research are discussed below.

2.3.1 X Gate

The X gate is also called a 'bit-flip' gate. It is the quantum-equivalent of the classical NOT gate. The matrix representation of the X gate is $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

```
import qiskit
from qiskit import QuantumRegister, QuantumCircuit
q = QuantumRegister(1)
qc = QuantumCircuit(q)
qc.x(q[0])
qc.draw(output='mpl')
```

Figure 2.1: X gate in a quantum circuit on Qiskit

Applying the X gate on $|0\rangle$:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

Applying the X gate on $|1\rangle$:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

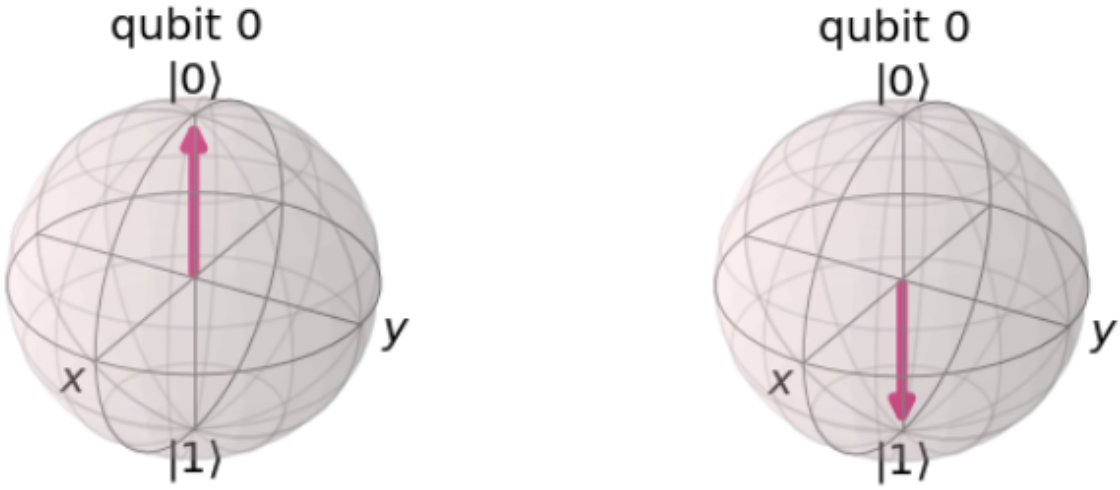

Figure 2.2: $|0\rangle$ and $|1\rangle$ on a Bloch sphere

2.3.2 CNOT Gate

The controlled NOT (CNOT or CX gate) acts on two qubits, where the first qubit is the control, based on which the NOT operation is applied on the second qubit. The matrix representation of the CNOT gate is $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

```
q2 = QuantumRegister(2)
qc = QuantumCircuit(q2)
qc.cx(q2[0],q2[1])
qc.draw(output='mpl')
```

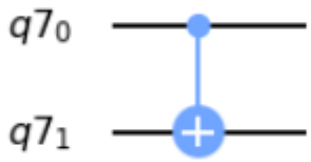

Figure 2.3: CNOT gate in a quantum circuit on Qiskit, where $q7_0$ is the control bit, and $q7_1$ is the target bit

Two qubits can be represented as a tensor product of individual qubits. The tensor product shows all possible combinations these qubits can coexist in.

$$|0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |00\rangle$$

Applying the CNOT gate on $|00\rangle$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = |00\rangle$$

Applying the CNOT gate on $|01\rangle$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |01\rangle$$

Applying the CNOT gate on $|10\rangle$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |11\rangle$$

Applying the CNOT gate on $|11\rangle$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle$$

2.3.3 Toffoli Gate

The Toffoli gate is also known as the CCX or CCNOT gate, because it acts on three qubits, where the first two qubits are the controls, based on which the NOT operation is applied on the third qubit. The matrix representation of the Toffoli gate is $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

```
q3 = QuantumRegister(3)
qc3 = QuantumCircuit(q3)
qc3.ccx(q3[0], q3[1], q3[2])
qc3.draw(output='mpl')
```

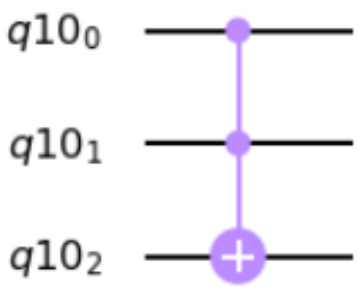

Figure 2.4: Toffoli gate in a quantum circuit on Qiskit, where $q10_0$ and $q10_1$ are the control bits, and $q10_2$ is the target bit

Applying the Toffoli gate on $|110\rangle$:

$$CCX\,|110\rangle = |111\rangle$$

Applying the Toffoli gate on $|111\rangle$:

$$CCX\,|111\rangle = |110\rangle$$

In all other cases, the qubits remain unchanged, because the Toffoli gate only works when both control qubits are in state $|1\rangle$.

The NOT (X), CNOT (CX), and Toffoli (CCX) gates are important because they are used in the quantum equivalent circuit of the AES, and because they are used in calculating the cost of quantum circuits.

2.3.4 T Gate

The T gate acts on a single qubit, and is the square of the S gate ($S = T^2$). The matrix representation of the T gate is $\begin{pmatrix} 1 & 0 \\ 0 & \exp\left(\frac{i\pi}{4}\right) \end{pmatrix}$

```
q4 = QuantumRegister(1)
qc4 = QuantumCircuit(q4)
qc4.t(q4[0])
qc4.draw(output='mpl')
```

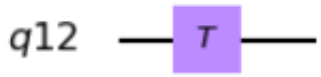

Figure 2.5: T gate in a quantum circuit on Qiskit

Applying the T gate on $|0\rangle$:

$$\begin{pmatrix} 1 & 0 \\ 0 & \exp\left(\frac{i\pi}{4}\right) \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

Applying the T gate on $|1\rangle$:

$$\begin{pmatrix} 1 & 0 \\ 0 & \exp\left(\frac{i\pi}{4}\right) \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \exp\left(\frac{i\pi}{4}\right) \end{pmatrix} = \exp\left(\frac{i\pi}{4}\right)|1\rangle$$

2.3.5 Hadamard Gate

The Hadamard (H) gate rotates a qubit by π about the axis in the middle of the x and z axes. The Hadamard gates creates superposition, resulting in equal probability of measuring a 0 or 1. The matrix representation of the Hadamard gate is $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Applying the Hadamard gate on $|0\rangle$:

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

Applying the Hadamard gate on $|1\rangle$:

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

```
q5 = QuantumRegister(1)
qc5 = QuantumCircuit(q5)
qc5.h(q5[0])
qc5.draw(output='mpl')
```

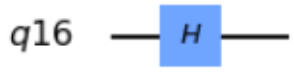

Figure 2.6: Hadamard gate in a quantum circuit on Qiskit

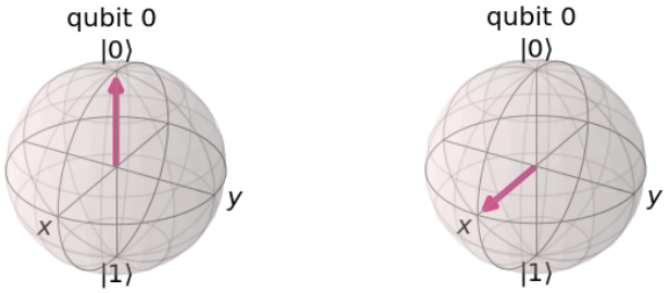

Figure 2.7: Bloch sphere representation of a Hadamard gate applied on $|0\rangle$

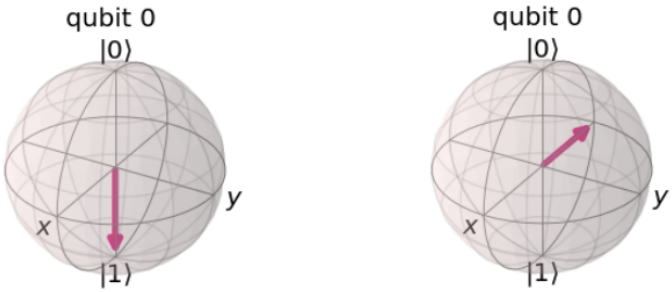

Figure 2.8: Bloch sphere representation of a Hadamard gate applied on $|1\rangle$

2.4 Quantum Equivalent of Classical Gates

After seeing some gates unique to quantum computation, we will use them to mimic certain classical gates.

2.4.1 NOT Gate

The NOT gate is straightforward, because it works the same way as the quantum X gate.

The truth table for the NOT gate is

input	output
0	1
1	0

```
from qiskit import ClassicalRegister
q6 = QuantumRegister(1)
c6 = ClassicalRegister(1)
qc6 = QuantumCircuit(q6, c6)
qc6.x(q6[0])
qc6.measure(q6[0], c6[0])
qc6.draw(output='mpl')
```

Figure 2.9: The quantum equivalent circuit of a NOT gate

2.4.2 AND Gate

The AND gate involves both input gates being 1 in order for the output to be 1 as well. This is just like the Toffoli gate (CCNOT or CCX), which needs two controls to act on the third qubit.

The truth table for the AND gate is

A (input)	B (input)	output
0	0	0
0	1	0
1	0	0
1	1	1

```
q7 = QuantumRegister(3)
c7 = ClassicalRegister(1)
qc7 = QuantumCircuit(q7, c7)
qc7.ccx(q7[0], q7[1], q7[2])
qc7.measure(q7[2], c7[0])
qc7.draw(output='mpl')
```

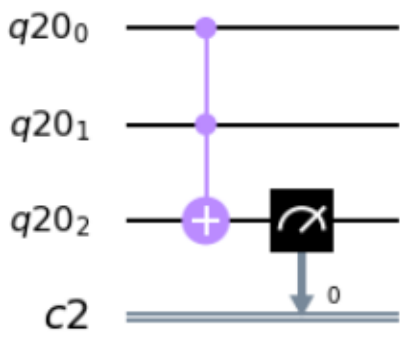

Figure 2.10: The quantum equivalent circuit of an AND gate

2.4.3 NAND Gate

The NAND gate is simply an AND gate followed by a NOT gate. On a quantum circuit, we can get the same results using an X gate after a Toffoli gate.
The truth table for the NAND gate is

A(input)	B(input)	output
0	0	1
0	1	1
1	0	1
1	1	0

2.4.4 OR Gate

Classically, the OR gate takes two inputs, and if either one of them is 1, the output is also 1. In a quantum circuit, we use a Toffoli gate to make sure that if the two inputs are 1, the output is also 1. For the other two cases, we use CNOT (CX) gates, where the first and second qubits are taken as controls respectively, and the NOT operation is applied on the third qubit.

```
q8 = QuantumRegister(3)
c8 = ClassicalRegister(1)
qc8 = QuantumCircuit(q8, c8)
qc8.ccx(q8[0], q8[1], q8[2])
qc8.x(q8[2])
qc8.measure(q8[2], c8[0])
qc8.draw(output='mpl')
```

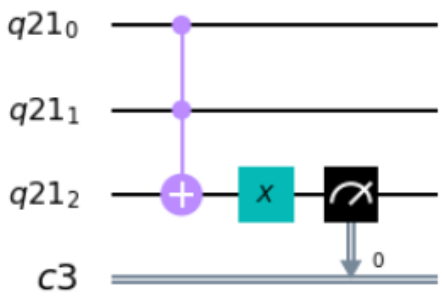

Figure 2.11: The quantum equivalent circuit of a NAND gate

The truth table for the OR gate is

A(input)	B(input)	output
0	0	0
0	1	1
1	0	1
1	1	1

2.4.5 NOR Gate

Following the OR gate, the NOR gate is straightforward: we simply add an X gate after the quantum-equivalent circuit of the OR gate.

The truth table for the NOR gate is

A(input)	B(input)	output
0	0	1
0	1	0
1	0	0
1	1	0

```
q9 = QuantumRegister(3)
c9 = ClassicalRegister(1)
qc9 = QuantumCircuit(q9, c9)

qc9.cx(q9[1], q9[2])
qc9.cx(q9[0], q9[2])
qc9.ccx(q9[0], q9[1], q9[2])

qc9.measure(q9[2], c9[0])
qc9.draw(output='mpl')
```

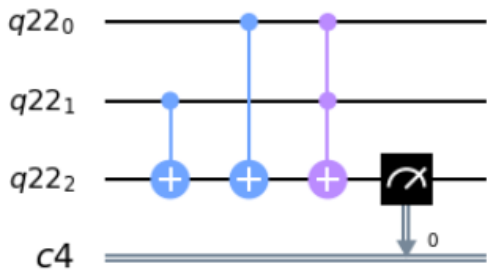

Figure 2.12: The quantum equivalent circuit of an OR gate

```
q10 = QuantumRegister(3)
c10 = ClassicalRegister(1)
qc10 = QuantumCircuit(q10, c10)

qc10.cx(q10[1], q10[2])
qc10.cx(q10[0], q10[2])
qc10.ccx(q10[0], q10[1], q10[2])
qc10.x(q10[2])

qc10.measure(q10[2], c10[0])
qc10.draw(output='mpl')
```

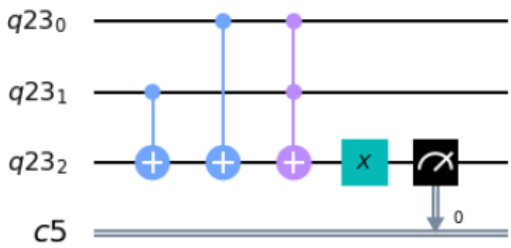

Figure 2.13: The quantum equivalent circuit of a NOR gate

2.4.6 XOR Gate

Finally, for the exclusive-OR gate, we use two CNOT (CX) gates, with the first and second qubit respectively as control, and the third qubit as the target.

The truth table for the XOR gate is

A(input)	B(input)	output
0	0	0
0	1	1
1	0	1
1	1	0

```
q11 = QuantumRegister(3)
c11 = ClassicalRegister(1)
qc11 = QuantumCircuit(q11, c11)

qc11.cx(q11[1], q11[2])
qc11.cx(q11[0], q11[2])

qc11.measure(q11[2], c11[0])
qc11.draw(output='mpl')
```

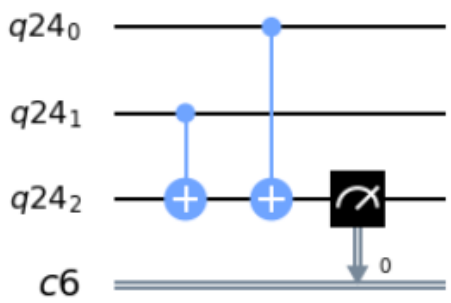

Figure 2.14: The quantum equivalent circuit of an XOR gate

Chapter 3

Quantum Cost Reduction

"Reduction is at the heart of progress in science."

– Jon Elster

3.1 Definition

The quantum cost [7] of a circuit is the cost of all the gates used in the quantum circuit. It can be measured by the number of elementary gates used to build the C^nNOT gate, which are considered as the number of two-qubit gates used.

We use the C^nNOT gate as our reference, because it is the main gate used to build any reducible circuit

$$C^nNOT\left(x_1, x_2, \ldots, x_{(n-1)}; f_{\text{in}}\right) = C^nNOT\left(y_1, y_2, \ldots, y_{(n-1)}; f_{\text{out}}\right)$$

where $y_i = x_i$ for $1 \leq i \leq n-1$ and $f_{\text{out}} = f_{\text{in}} \oplus x_1x_2 \ldots x_{(n-1)} \forall n \in Z, x \in X^n$ and $X = \{0,1\}.x_1x_2 \ldots x_{(n-1)}$ are called the control bits, and f_{in} is called the target bit.

3.1.1 Types of Cost Reduction

The two metrics that we looked at were the cost015 and cost115 metrics.

Gate	Quantum Cost	
	cost015 metric	cost115 metric
N	0	1
C	1	1
T	5	5

As in the table, the cost015 metric involves taking the cost of NOT gates to be 0, CNOT gates to be 1, and Toffoli gates to be 5. For cost115, the NOT gate also has a cost of 1, while the other gate costs remain the same. Why the Toffoli gate has a cost of 5 will be discussed in the following sections.

3.2 NCT Library for 3-Bit Systems

Since we are using the NCT quantum cost metrics, we will now look at the total circuits in the NCT library for a three-bit reversible system.

3.2.1 NOT Gate

We can apply the NOT gate on each of the qubits, and the three possibilities are stated as

$$N_1^3 : (x_1, x_2, x_3) \rightarrow (x_1 \oplus 1, x_2, x_3)$$
$$N_2^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2 \oplus 1, x_3)$$
$$N_3^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3 \oplus 1)$$

where N_i^3 shows which qubit (i) the NOT gate is acting on in the three-qubit system.

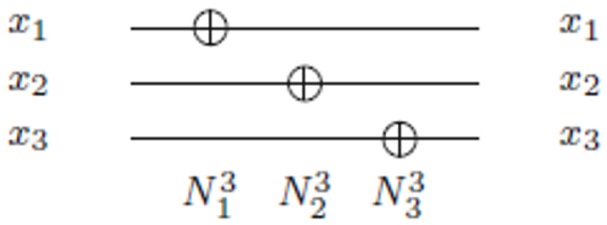

Figure 3.1: The three possible NOT gates that can be applied to a three-qubit reversible circuit

3.2.2 CNOT Gate

In the case of CNOT gates, one qubit is the control bit, while the other is the target, so there are six possible combinations in total, which have been mentioned below:

$$C_{1,2}^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2 \oplus x_1, x_3)$$
$$C_{1,3}^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3 \oplus x_1)$$
$$C_{2,3}^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3 \oplus x_2)$$
$$C_{2,1}^3 : (x_1, x_2, x_3) \rightarrow (x_1 \oplus x_2, x_2, x_3)$$
$$C_{3,2}^3 : (x_1, x_2, x_3) \rightarrow (x_1, x_2 \oplus x_3, x_3)$$
$$C_{3,1}^3 : (x_1, x_2, x_3) \rightarrow (x_1 \oplus x_3, x_2, x_3)$$

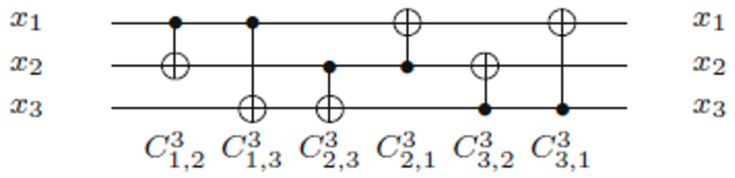

Figure 3.2: The six possible CNOT gates that can be applied to a three-qubit reversible circuit

3.2.3 Toffoli Gate

For the Toffoli gate, we have two control bits, and one target bit, resulting in three possible combinations.

$$T^3_{1,2,3} : (x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3 \oplus x_1x_2)$$
$$T^3_{1,3,2} : (x_1, x_2, x_3) \rightarrow (x_1, x_2 \oplus x_1x_3, x_3)$$
$$T^3_{3,2,1} : (x_1, x_2, x_3) \rightarrow (x_1 \oplus x_2x_3, x_2, x_3)$$

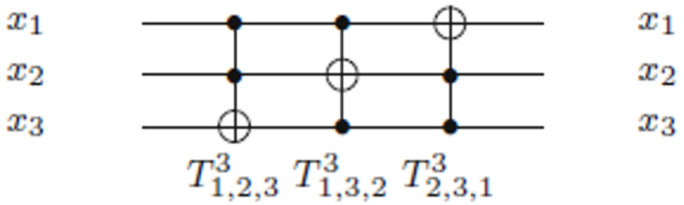

Figure 3.3: The three possible Toffoli gates that can be applied to a three-qubit reversible circuit

3.3 Reducing Controls by 1

In order to reduce the number of quantum gates involved in a circuit, we can break certain gates like Toffoli into smaller components

3.3.1 $\sqrt{N}$ Gate

The $\sqrt{N}$ gate (also called the $\sqrt{X}$ Gate) is defined such that when it acts on $|0\rangle$, it maps the state to $\frac{(1+i)|0\rangle+(1-i)|1\rangle}{2}$. Similarly, it maps $|1\rangle$, to $\frac{(1-i)|0\rangle+(1+i)|1\rangle}{2}$.

Based on this, the $\sqrt{N}$ gate can be written as

$$\sqrt{X} = \sqrt{NOT} = \frac{1}{2}\begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} \exp\left(\frac{i\pi}{4}\right) & \exp\left(\frac{-i\pi}{4}\right) \\ \exp\left(\frac{-i\pi}{4}\right) & \exp\left(\frac{i\pi}{4}\right) \end{bmatrix}$$

$$X = (\sqrt{NOT})^2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Here is the square-root gate formation in more detail:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

calculating the eigenvalues and unit eigenvectors of that matrix:

$$\lambda_1 = 1, v_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\lambda_2 = -1, v_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

This gives us the eigendecomposition of the NOT gate's operation:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \lambda_1 v_1 v_1^\dagger + \lambda_2 v_2 v_2^\dagger = \lambda_1 \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \lambda_2 \frac{1}{2} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

When applied to a matrix, the eigendecomposition form of a matrix transforms the eigenvalues. To find the square root of the matrix, we replace the eigenvalue coefficients with their square roots:

$$\sqrt{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}} = \sqrt{\lambda_1} v_1 v_1^\dagger + \sqrt{\lambda_2} v_2 v_2^\dagger = \sqrt{1} \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \sqrt{-1} \frac{1}{2} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

And, by arbitrarily picking principal square roots, we find one of the square roots of NOT:

$$\rightarrow \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + i \frac{1}{2} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$$

Verifying this: $\frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$ does in fact give you $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.

Engineer a simulation of

$$\frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$$

apply it twice, and you will have performed a NOT operation. The inverse of a unitary matrix is much easier to find than the square root: the inverse is simply the conjugate transpose of the matrix.

For example, the inverse of $\frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$ is just $\frac{1}{2} \begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}$ (another square root of NOT)

3.3.2 u and v Gates

When we take $\sqrt{N}$, the two roots we get are the u and v gates.

(a) all possible v gates (b) all possible u gates

Figure 3.4: The u and v gates can be applied on a three-bit system

As in the diagrams above, there are six possible combinations for the v and u gates, which can be written as:

$$v^3_{1,2}, v^3_{1,3}, v^3_{2,3}, v^3_{2,1}, v^3_{3,2} \text{ and } v^3_{3,1}$$

$$u^3_{1,2}, u^3_{1,3}, u^3_{2,3}, u^3_{2,1}, u^3_{3,2} \text{ and } u^3_{3,1}$$

3.3.3 Decomposing Toffoli Gates

Generally, an C^{n+1}-U gate can be decomposed as shown in Fig. 3.5 into gates with max controls reduced by 1 [8].

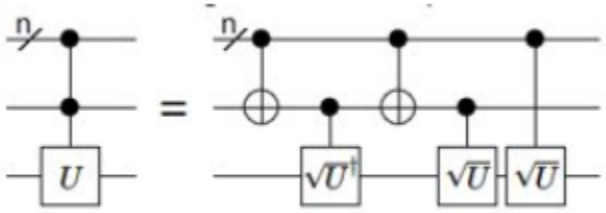

Figure 3.5: Reducing the max control by 1

The different possible Toffoli gates can thus be decomposed as shown in Figs. 3.6 3.7 and 3.8 and the cost of a single Toffoli gate is thus the sum of costs of the individual gates it is decomposed to i.e. 1+1+1+1+1=5.

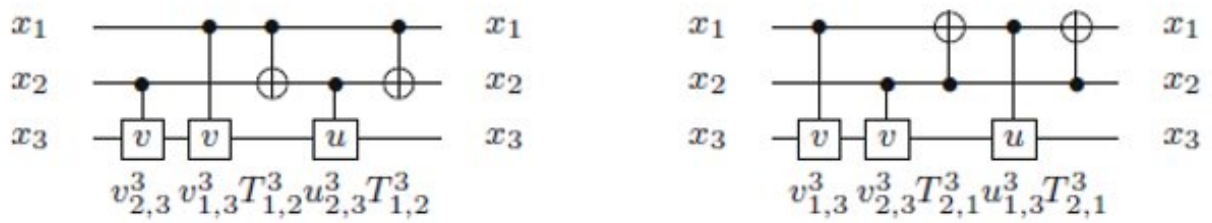

Figure 3.6: Decomposing Toffoli gates T(1, 2, 3)

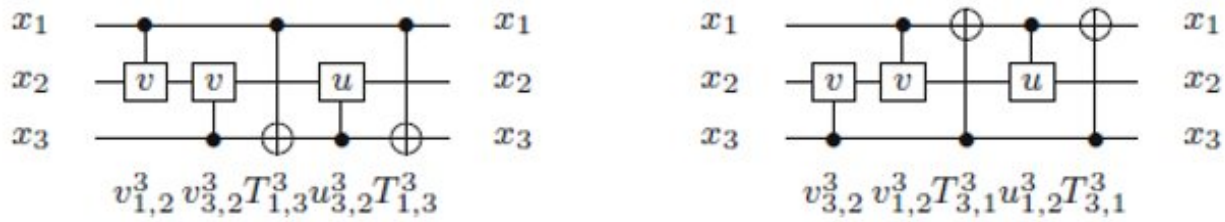

Figure 3.7: Decomposing Toffoli gates T(1, 3, 2)

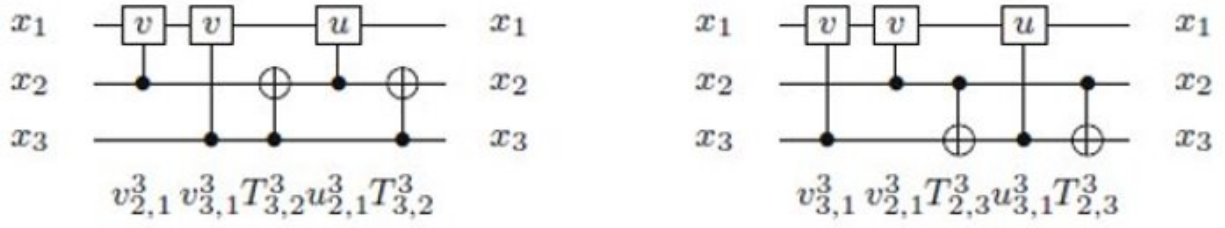

Figure 3.8: Decomposing Toffoli gates T(2, 3, 1)

3.3.4 Rules of Decomposing Toffoli Gates

Note from R7 in Fig. 3.9 that the cost of two toffoli gates T(1, 2, 3) and T(1, 3, 2) in series (placed right next to each other in a Quantum Circuit) is not 5 + 5 = 10, instead it is 8 as when this pair of Toffoli gates is decomposed to u, v, and C gates, the decomposed circuit is reduced from 10 to 8 2-qubit gates.

Similarly in R9, the cost is seen to be 9 instead of 10 which also indicates cost to be reduced by simplifying the decomposed circuit of T(1, 3, 2) and T(1, 2, 3) in series (placed right next to each other in a Quantum Circuit).

From R7 and R9 we also observe that the order in which Toffoli gates are placed in a Quantum Circuit is also important in calculating the Quantum cost.

Rule no.	Gate	Following gate	Circuit decomposition	Quantum cost
R1	$T^3_{(1,2,3)}$(a)	–	$v^3_{2,3}v^3_{1,3}C^3_{1,2}u^3_{2,3}C^3_{1,2}$	5
R2	$T^3_{(1,2,3)}$(b)	–	$v^3_{1,3}v^3_{2,3}C^3_{2,1}u^3_{1,3}C^3_{2,1}$	5
R3	$T^3_{(1,3,2)}$(a)	–	$v^3_{1,2}v^3_{3,2}C^3_{1,3}u^3_{3,2}C^3_{1,3}$	5
R4	$T^3_{(1,3,2)}$(b)	–	$v^3_{3,2}v^3_{1,2}C^3_{3,1}v^3_{1,2}C^3_{3,1}$	5
R5	$T^3_{(2,3,1)}$(a)	–	$v^3_{2,1}v^3_{3,1}C^3_{3,2}u^3_{2,1}C^3_{3,2}$	5
R6	$T^3_{(2,3,1)}$(b)	–	$v^3_{3,1}v^3_{2,1}C^3_{2,3}u^3_{3,1}C^3_{2,3}$	5
R7	$T^3_{(1,2,3)}$	$T^3_{(1,3,2)}$	$v^3_{2,3}v^3_{1,3}C^3_{1,2}[u^3_{2,3}v^3_{3,2}]v^3_{1,2}C^3_{1,3}u^3_{3,2}C^3_{1,3}$	8
R8	$T^3_{(1,2,3)}$	$T^3_{(2,3,1)}$	$v^3_{1,3}v^3_{2,3}C^3_{2,1}[u^3_{1,3}v^3_{3,1}]u^3_{2,1}C^3_{3,2}u^3_{2,1}C^3_{3,2}$	8
R9	$T^3_{(1,3,2)}$	$T^3_{(1,2,3)}$	If gate $T^3_{2,3,1}$ proceeds gates $T^3_{1,3,2}$ and $T^3_{1,2,3}$, decompose gates $T^3_{1,3,2}$ and $T^3_{1,2,3}$ into: $v^3_{3,2}v^3_{1,2}C^3_{3,1}u^3_{1,2}[C^3_{3,1}v^3_{1,3}]$ $v^3_{2,3}C^3_{1,2}u^3_{2,3}C^3_{1,2}$	9
R10	$T^3_{(1,3,2)}$	$T^3_{(1,2,3)}$	Otherwise $v^3_{3,2}v^3_{1,2}C^3_{3,1}u^3_{1,2}[C^3_{3,1}v^3_{1,3}]v^3_{2,3}C^3_{1,2}u^3_{2,3}C^3_{1,2}$	8
R11	$T^3_{(1,3,2)}$	$T^3_{(2,3,1)}$	If gate $T^3_{1,2,3}$ proceeds gates $T^3_{1,3,2}$ and $T^3_{2,3,1}$, decompose $T^3_{1,3,2}$ and $T^3_{2,3,1}$ into: $v^3_{1,2}v^3_{3,2}C^3_{1,3}u^3_{3,2}[C^3_{1,3}v^3_{3,1}]v^3_{2,1}$ $C^3_{2,3}u^3_{3,1}C^3_{2,3}$	9
R12	$T^3_{(1,3,2)}$	$T^3_{(2,3,1)}$	Otherwise $v^3_{3,2}v^3_{1,2}C^3_{3,1}[u^3_{1,2}v^3_{2,1}]u^3_{3,1}C^3_{2,3}u^3_{3,1}C^3_{2,3}$	8
R13	$T^3_{(2,3,1)}$	$T^3_{(1,2,3)}$	$v^3_{3,1}v^3_{2,1}C^3_{2,3}[u^3_{3,1}v^3_{1,3}]u^3_{2,3}C^3_{1,2}u^3_{2,3}C^3_{1,2}$	8
R14	$T^3_{(2,3,1)}$	$T^3_{(1,3,2)}$	$v^3_{2,1}v^3_{3,1}C^3_{3,2}[u^3_{2,1}v^3_{1,2}]u^3_{3,2}C^3_{3,1}u^3_{1,2}C^3_{3,1}$	8

Figure 3.9: Rules of decomposing Toffoli gates

Chapter 4

LUP Decomposition

> Tadeusz Banachiewicz (13 February 1882, Warsaw – 17 November 1954) was a Polish astronomer, mathematician and geodesist. The LU decomposition was introduced by Banachiewicz in 1938.
>
> – Wikipedia

4.1 Understanding with an Example

LUP decomposition allows us to write a matrix M as the product of three matrices that are easier to work with. We'll write M = PLU, where:

- P is pivot matrix.
- L is lower triangular.
- U is upper triangular.

We've used LUP type decomposition in different occasions in our work, so let's understand with the help of an example what it is.

Say you're given a matrix M and are required to perform LUP decomposition.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

To make the pivot matrix P, you need to apply row operations on M and make a matrix in which every column and every row has exactly one 1.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$ R1 - R2, R4 - R2, R5 - R2

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$ R3 - R6, R4 - R6, R7 - R6

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$ (-1)R4, R2 - R4, R3 + R4, R5 + R4, R6 - R4, R7 + R4, R8 - R4

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$R1 + R8, R2 - R8$

$Now, say M = PN, then P^{-1}\mathrm{M} = \mathrm{N}$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Sequentially interchanging the rows in the following manner:

- R2 interchanged with R3
- then R3 with R5
- then R4 with R7

We are left with:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

And thus,

$$P^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

This implies, we can calculate

$$N = P^{-1}M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

On applying LU [9] decomposition on N, we get N = LU such that:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \quad \& \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

4.2 Quantum Equivalent Circuit of Multiplying a Matrix M with a Vector

Considering the example matrix M in the previous section and a general vector

$$V = \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \\ a4 \\ a5 \\ a6 \\ a7 \end{bmatrix},$$

the updated vector $\begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \\ a4 \\ a5 \\ a6 \\ a7 \end{bmatrix}$ = MV can be found by using the Quantum Circuit in Fig.

4.1.

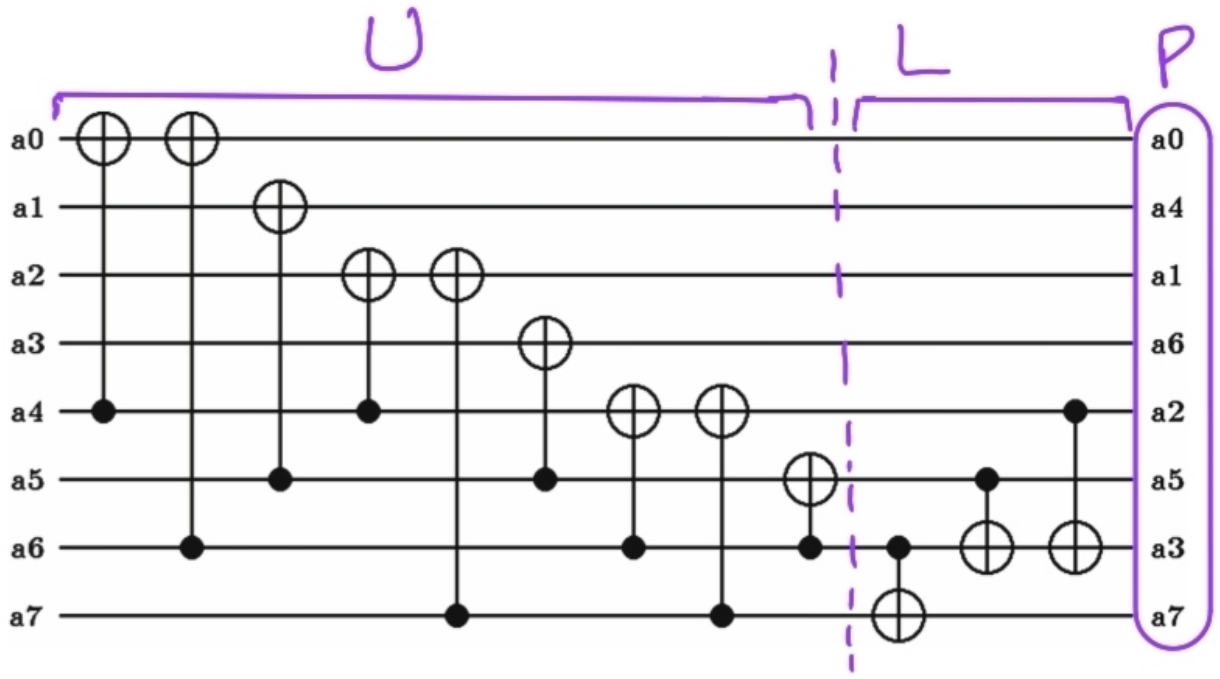

Figure 4.1: Quantum circuit for implementing multiplication of matrix M with vector V

- If you note the matrix U, the diagonal elements are controlled NOT by the off-diagonal elements and you can note this by indexing each column of the matrix U by the vector elements a0, a1, ..., a7.
- A similar observation can be made regarding matrix L.
- The matrix P dictates which wire represents which element of the updated vector. Indexing the rows and columns of P by the vector elements a0, a1, ..., a7, you can note that the original vector V (indexing the columns) pivots to some row index (element of updated vector).

Since we've now learnt how to create a link between the Quantum Circuit and the matrices, given the LUP decomposed matrices of a matrix M, we can draw its Quantum Circuit and vice versa.

Chapter 5

Computation of Inverse

5.1 Itoh Tsujii Algorithm

We're given an element $\alpha \in F_{2^m}^*$ and we're asked to find its inverse. We know that

$$\alpha^{2^m-1} = 1$$

$$\implies \alpha^{-1} = \alpha^{2^m-2}$$

Now, for $i \geq 0$, let us define $\beta_i = \alpha^{2^i-1}$, i.e. $\beta_0 = 1$, $\beta_1 = \alpha$, and so on.

$$\beta_{m-1} = \alpha^{2^{m-1}-1}$$

$$\beta_{m-1} = \alpha^{2^m 2^{-1}-1}$$

$$(\beta_{m-1})^2 = (\alpha^{2^m \cdot 2^{-1}-1})^2$$

$$(\beta_{m-1})^2 = \alpha^{2^m-2}$$

$$\implies \alpha^{-1} = (\beta_{m-1})^2$$

Also for all $i, j \in$ non-negative integers,

$$\beta_{i+j} = \beta_i \cdot \beta_j^{2^i}$$

Let's now compute the inverse [10] of $\alpha \in F_{2^8}^*$:

$$\begin{aligned}
\alpha^{-1} = \alpha^{2^8-2} = \alpha^{256-2} = \alpha^{254} &= (\beta_{8-1})^2 \\
&= (\beta_7)^2 \\
&= (\beta_{6+1})^2 \\
&= (\beta_6 \cdot \beta_1^{2^6})^2 \\
&= (\beta_{5+1} \cdot \alpha^{64})^2 \\
&= (\beta_5 \cdot \beta_1^{2^5} \cdot \alpha^{64})^2
\end{aligned}$$

$$\begin{aligned}
\alpha^{-1} = \alpha^{254} &= (\beta_{4+1} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_4 \cdot \beta_1^{2^4} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_{3+1} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_3 \cdot \beta_1^{2^3} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_{2+1} \cdot \alpha^{8} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_2 \cdot \beta_1^{2^2} \cdot \alpha^{8} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_{1+1} \cdot \alpha^{22} \cdot \alpha^{8} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\beta_1 \cdot \beta_1^{2^1} \cdot \alpha^{4} \cdot \alpha^{8} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= (\alpha \cdot \alpha^{2} \cdot \alpha^{4} \cdot \alpha^{8} \cdot \alpha^{16} \cdot \alpha^{32} \cdot \alpha^{64})^2 \\
&= ((\alpha \cdot \alpha^{2}) \cdot (\alpha \cdot \alpha^{2})^4 \cdot (\alpha \cdot \alpha^{2})^{16} \cdot \alpha^{64})^2
\end{aligned}$$

5.2 Boyar and Peralta Technique

The first step in Boyar and Paralta's method [11] [12] [13] is to derive a circuit that uses minimum AND gates and reduce the non-linearity (number of non-linear gates) of a circuit. Say we have to compute an inverse in a certain field. To build a circuit for inverses in $GF(2^{mn})$, given a circuit for inverses in $GF(2^m)$, a recursive method which involves tower fields architechture is used. In Boyer and Paralta's technique, tower field architechture has been used in the reduction from inversion in $GF(2^8)$ to inversion in $GF(2^4)$. Then, this $GF(2^4)$ inversion circuit is placed in its appropriate place in the bigger circuitry (here AES's S-Box) and an optimization technique is applied on the linear parts of the resulting circuit.

5.2.1 Step One

Tower Field Construction

A tower of fields is an extension sequence of some fields. For example:

- $GF(2^2)$ by adjoining a root W of $x^2 + x + 1$ over $GF(2)$
- $GF(2^4)$ by adjoining a root Z of $x^2 + x + W^2$ over $GF(2^2)$
- $GF(2^8)$ by adjoining a root Y of $x^2 + x + WZ$ over $GF(2^4)$

$GF(2^2)$ and $GF(2^4)$ are represented using the basis (W, W^2) and (Z^2, Z^8), respectively. Therefore, an element $\alpha \in GF(2^4)$ is written as $\alpha_1 Z^2 + \alpha_2 Z^8$, where $\alpha_1, \alpha_2 \in GF(2^2)$. Similarly an element $\beta \in GF(2^2)$ is written as $\beta_1 W + \beta_2 W^2$, where $\beta_1, \beta_2 \in GF(2)$.
This implies that an element of $GF(2^4)$ can be written as $E = (x_1 W + x_2 W^2)Z^2 + (x_3 W + x_4 W^2)Z^8$, where $x_1, x_2, x_3, x_4 \in GF(2)$.
The inverse of this element is $E' = (y_1 W + y_2 W^2)Z^2 + (y_3 W + y_4 W^2)Z^8$ where

$y_1 = x_2x_3x_4 + x_1x_3 + x_2x_3 + x_3 + x_4$
$y_2 = x_1x_3x_4 + x_1x_3 + x_2x_3 + x_2x_4 + x_4$
$y_3 = x_1x_2x_4 + x_1x_3 + x_1x_4 + x_1 + x_2$
$y_4 = x_1x_2x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_2$
and $y_1, y_2, y_3, y_4 \in GF(2^4)$

Currently, using Boyar and Paralta's Technique, an optimal circuit for each of the four equations can be constructed individually, but not for the system itself. The following approach is used:

- Select an equation and build an efficient circuit for it
- Do not discard the outputs of the circuit that stand as intermediate functions for possible use in creating the next equation's circuit
- Repeat this process till computation of all equations

These three steps are repeated for all possible orderings of $\{y_1, y_2, y_3, y_4\}$ and the most optimum result is kept. The best result was found for the ordering (y_4, y_2, y_1, y_3).
(NOTE: Before we dry run the Boyar and Peralta technique to get a circuit for this example, note that $x_i + x_i = 0$ and $x_i \cdot x_i = x_i$.)
Take

$$y_4 = x_1x_2x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_2$$
$$y_4 = x_1x_2x_3 + x_1x_3 + (x_1 + x_2) \cdot x_4 + x_2$$
$$y_4 = (x_2 + 1) \cdot x_1x_3 + (x_1 + x_2) \cdot x_4 + x_2$$
$$y_4 = (x_2 + x_1) \cdot x_1x_3 + (x_1 + x_2) \cdot x_4 + x_2$$
$$y_4 = (x_1 + x_2) \cdot (x_1 \cdot x_3 + x_4) + x_2$$

This implies the circuit for y_4 would be

$$t_1 = x_1 + x_2, \quad t_2 = x_1 \cdot x_3, \quad t_3 = x_4 + t_2, \quad t_4 = t_1 \cdot t_3, \quad y_4 = x_2 + t_4.$$

Where t_1, t_2, t_3, t_4 are intermediate outputs, y_4 is the final output, x_1, x_2, x_3, x_4 are inputs, " $+$ " corresponds to XOR gate and " $\cdot$ " corresponds to AND gate.
Now take

$$y_2 = x_1x_3x_4 + x_1x_3 + x_2x_3 + x_2x_4 + x_4$$
$$y_2 = x_1x_3x_4 + x_1x_3 + (x_3 + x_4) \cdot x_2 + x_4$$
$$y_2 = (x_4 + 1) \cdot x_1x_3 + (x_3 + x_4) \cdot x_2 + x_4$$
$$y_2 = (x_4 + x_3) \cdot x_1x_3 + (x_3 + x_4) \cdot x_2 + x_4$$
$$y_2 = (x_3 + x_4) \cdot (x_1 \cdot x_3 + x_2) + x_4$$

Circuit for y_2 would be

$$t_5 = x_3 + x_4, \quad t_6 = x_2 + t_2, \quad t_7 = t_6 \cdot t_5, \quad y_2 = x_4 + t_7$$

Where t_5, t_6, t_7 are intermediate outputs, y_4 is the final output, and t_2, x_2, x_3, x_4 are inputs. Similarly dry running the technique further, we discover the optimum circuits for y_1 and y_3 to be

$$t_8 = x_3 + y_2 \quad t_9 = t_3 + y_2, \quad t_{10} = x_4 \cdot t_9, \quad y_1 = t_{10} + t_8$$

and

$$t_{11} = t_3 + t_{10}, \quad t_{12} = y_4 \cdot t_{11}, \quad y_3 = t_{12} + t_1$$

respectively.
Thus the inversion in $GF(2^4)$ can be done by the circuit in Fig. 5.1 where (*) are the outputs of the circuit.

$t_1 = x_1 + x_2$	$t_2 = x_1 \times x_3$	$t_3 = x_4 + t_2$
$t_4 = t_1 \times t_3$	$y_4 = x_2 + t_4$ (*)	$t_5 = x_3 + x_4$
$t_6 = x_2 + t_2$	$t_7 = t_6 \times t_5$	$y_2 = x_4 + t_7$ (*)
$t_8 = x_3 + y_2$	$t_9 = t_3 + y_2$	$t_{10} = x_4 \times t_9$
$y_1 = t_{10} + t_8$ (*)	$t_{11} = t_3 + t_{10}$	$t_{12} = y_4 \times t_{11}$
$y_3 = t_{12} + t_1$ (*)		

Figure 5.1: Inversion in $GF(2^4)$

5.2.2 Step Two

Optimization to Linear parts of Circuit

Let's understand this step purely with the help of an example:
Say we have a sequence of equations and a corresponding matrix M:

$$\begin{aligned} y_0 &= x_0 + x_1 + x_2 \\ y_1 &= x_1 + x_3 + x_4 \\ y_2 &= x_0 + x_2 + x_3 + x_4 \\ y_3 &= x_1 + x_2 + x_3 \\ y_4 &= x_0 + x_1 + x_3 \\ y_5 &= x_1 + x_2 + x_3 + x_4 \end{aligned} \qquad M = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The initial base is $\{x_0, x_1, x_2, x_3\}$ which corresponds to

$$S = \{[1\ 0\ 0\ 0\ 0], [0\ 1\ 0\ 0\ 0], [0\ 0\ 1\ 0\ 0], [0\ 0\ 0\ 1\ 0], [0\ 0\ 0\ 0\ 1]\}$$

The initial distance vector is

$$D = [2\ 2\ 3\ 2\ 2\ 3]$$

(The distance vector's elements are equal to the distance between S and the corresponding row of M.)
Let the new base be expanded to contain the signal

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The new distance vector is

$$D = [2\ 1\ 3\ 1\ 1\ 2]$$

We would continue to expand the base until we find all target signals and distance vector becomes zero (see Fig. 5.2)

Step 1: $t_5 = x_1 + x_3$ (found signal = [0 1 0 1 0]). New D : [2 1 3 1 1 2]
Step 2: $t_6 = x_2 + t_5$ (found target signal y_3 = [0 1 1 1 0]). New D : [2 1 3 0 1 1]
Step 3: $t_7 = x_4 + t_6$ (found target signal y_5 = [0 1 1 1 1]). New D : [2 1 2 0 1 0]
Step 4: $t_8 = x_0 + x_1$ (found signal = [1 1 0 0 0]). New D : [1 1 1 0 1 0]
Step 5: $t_9 = x_0 + t_5$ (found target signal y_4 = [1 1 0 1 0]). New D : [1 1 1 0 0 0]
Step 6: $t_{10} = x_2 + t_7$ (found target signal y_1 = [0 1 0 1 1]). New D : [1 0 1 0 0 0]
Step 7: $t_{11} = x_2 + t_8$ (found target signal y_0 = [1 1 1 0 0]) . New D : [0 0 1 0 0 0]
Step 8: $t_{12} = t_7 + t_8$ (found target signal y_2 = [1 0 1 1 1]). New D : [0 0 0 0 0 0]
(DONE!)

Figure 5.2: Linear optimization

Chapter 6

The Advanced Encryption Standard

"We are currently at the start of a second quantum revolution; the first quantum revolution gave us new rules that govern physical reality. The second quantum revolution will take these rules and use them to develop new technologies and offer the next level of opportunities."

– Ian R. McAndrew, PhD. –
Dean of Doctorate Programs @ Capitol Technology University

6.1 Cipher

The Advanced Encryption Standard (AES) [14] is a symmetric block ciper that uses the Rijndael Algorithm to process data blocks of 128 bits. The AES uses cipher keys of lengths 128, 192, and 256 bits. While Rijndael can also handle additional block sizes and key lengths, AES only has three variants: AES-128, AES-192, and AES-256, depending on the key length.

2*	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 6.1: Key-Block-Round combinations

- The AES deals with *words*. Each word has 4 bytes, and each byte has 8 bits.
- The input block, output block, and state are all 128 bits long. We can see this by Nb = 4, which shows the number of 32-bit words in the state.

- The Cipher Key, K, can be 128, 192, or 256 bits long, depending on the variant of AES that is being used. Nk is the key length, and it can be 4, 6, or 8, based on the 32-bit words (number of columns) in the Cipher Key.
- The greater the key size. the more the rounds that need to be performed to execute the algorithm. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4, Nr = 12 when Nk = 6, and Nr = 14 when Nk = 8.

6.2 Round Function

The AES algorithm uses a round function that involves four different byte-oriented transformations:

- Substituting bytes using a substitution table (S-box).
- Shifting rows of the state array by different offsets.
- Mixing the data within each column of the state array.
- Adding a Round Key to the state

In the beginning of the cipher, the state array is initialized with the input. After an initial Round Key addition, the round function is used 10, 13, or 14 times (for AES-128, AES-192, and AES-256, respectively), to update the state array. The final round is different from the Nr-1 initial rounds. The final state is then unloaded as the output.

The round function is parameterized using a key schedule comprising of a one-dimensional array of four-byte words, derived using the Key Expansion procedure.

6.2.1 SubBytes() Transformation

The SubBytes() transformation operates independently on each byte of the state, using non-linear byte substitution. THe SubBytes() transformation uses a substitution table (S-box), which is invertible.It is constructed using two transformations:

- Take the multiplicative inverse in the finite field GF(2^8); the element {00} is mapped to itself.

- Apply the following Affine transformation (over GF(2)):

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b_6 \\ b'_9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.1}$$

Illustrating the SubBytes transformation on the state:

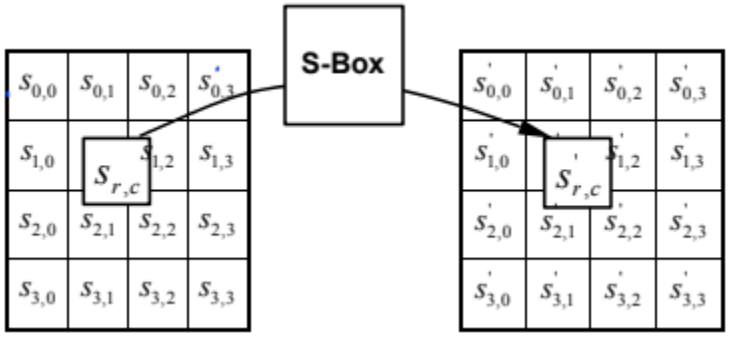

Figure 6.2: SubBytes() applies the S-box to each byte of the state

Where the S-Box is used such that say for example, if $s_{1,1} = \{ff\}$, then the substitution value would be determined by the intersection of the row with index '3' and the column with index '5' in Fig. 7. This would result in $s'_{1,1}$ having a value of $\{16\}$.

6.2.2 ShiftRows() Transformation

In the ShiftRows() transformation, the bytes in each rows of the State are cyclically shifted by different pre-defined offsets. (The nth row is shifted by an offset of n-1.)

Illustrating the ShiftRows() transformation:

6.2.3 MixColumns() Transformation

Treating each column as a four-term polynomial, the MixColumns() transformation operates on the State column-by-column. Each transformed column is a modular product of $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ and $s(x) = s_{3,c}x^3 + s_{2,c}x^2 + s_{1,c}x + s_{0,c}$, where subscript c indicates the column number.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 6.3: Using the S Box to find the multiplicative inverse of each word

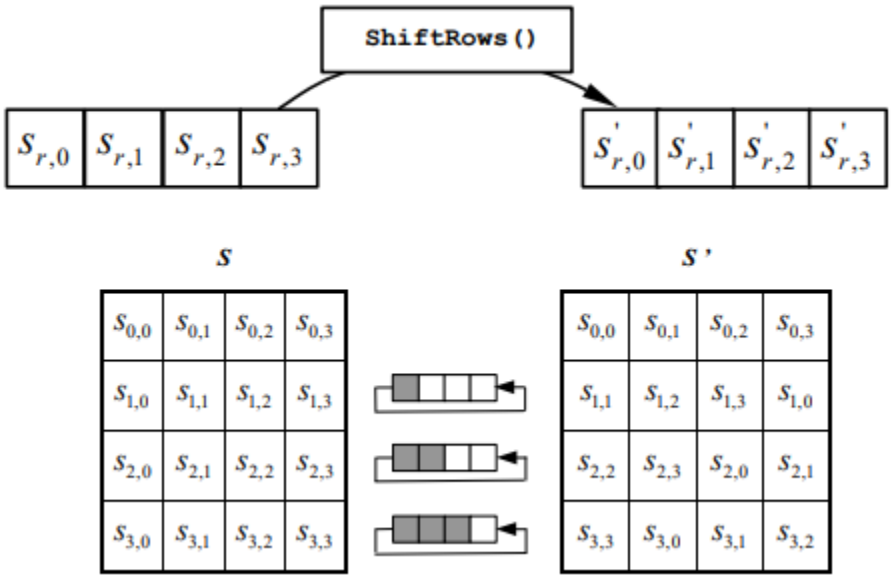

Figure 6.4: ShiftRows() cyclically shifts the rows in the state

Here $a(x)$ is coined as the fixed polynomial and the modulo polynomial is $x^4 + 1$ to confine the product to represent a four-byte word.

$$s'(x) = a(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < \boldsymbol{Nb}$$

Illustrating the MixColumns() operation:

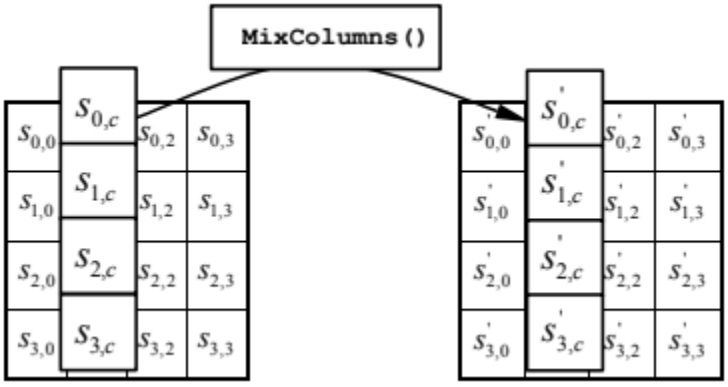

Figure 6.5: MixColumns() operates on the state column-by-column

6.2.4 AddRoundKey() Transformation

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bit-wise XOR operation.

AddRoundKey() is illustrated as follows:

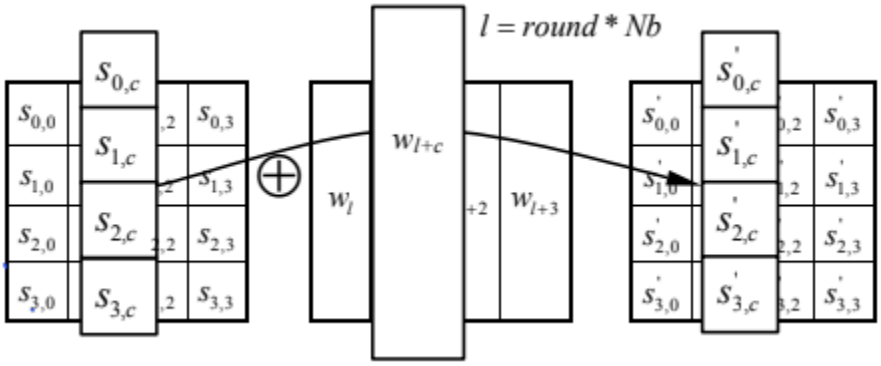

Figure 6.6: AddRoundKey() XORs each column of the state with a word from key schedule

Where $[w_i]$ are the key schedule words described in the next section.

6.3 Key Expansion

The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule $[w_i]$ (a linear array of 4-byte words with i in the range $0 \leq i < Nb(Nr+1)$). The Key Expansion generates a total of Nb(Nr + 1) words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data.

The Pseudo-Code for Key Expansion is in figure 6.7.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.   Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 6.7: Pseudocode for key expansion

6.3.1 SubWord()

The SubWord() function takes a four-byte word as input and then applies S-box to each byte producing an output word. This function is similar to SubBytes().

6.3.2 RotWord()

The RotWord() function takes a word as an input, performs a cyclic permutation with an offset of 1, and returns the word.

Input: $[a_0, a_1, a_2, a_3]$

Returned word: $[a_1, a_2, a_3, a_0]$.

6.3.3 Rcon[]

The round constant word array, Rcon[i], contains the values given by $[\{02\}^{i-1}, \{00\}, \{00\}, \{00\}]$ in the field GF (2^8), where i starts with 1.

We can see from Fig. 6.7 that the Cipher Key fills the first Nk words of the expanded key. The following words w[i] are equal to the XOR of the word Nk positions earlier w[i-Nk] and the previous word w[i-1].

For the words in the "multiple of Nk" positions, a transformation (RotWord(), then SubWord(), then XOR with RCon[i]) is applied to w[i-1] before XORing with w[i-Nk].

The Key Expansion routine for AES-256 (Nk = 8) is a bit different than for AES-128 and AES-192. If Nk = 8 and i-4 is a multiple of Nk, then SubWord() is applied to w[i-1] prior to the XOR.

Chapter 7

Quantum Circuit Implementation of AES

> "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy."
>
> – Richard P. Feynman

Let's build the quantum equivalent circuit of AES [15]!

7.1 Circuits for Basic AES Components

7.1.1 AddRoundKey()

Since we know that the equivalent quantum gate to XOR is CNOT, therefore the 128 bit state of AES, now carried in 128 qubits, would be followed by 128 CNOT gates which can be executed in parallel.

We can see a simulation of AddRoundKey() in Qiskit[16] in Fig. 7.1 where q0, q1, ... q15 represent the 16 quwords (1 quword = 16 qubits) of the state. The quwords q16, q17, ... q31 represent the round key.

7.1.2 ShiftRows()

ShiftRows() (Fig. 7.1 and 7.2) only involves simple permutation of qubits in each of the rows, so no gates are required in this part of the circuit. Instead, we only adjust the positions of subsequent gates, so that the correct wires are used.

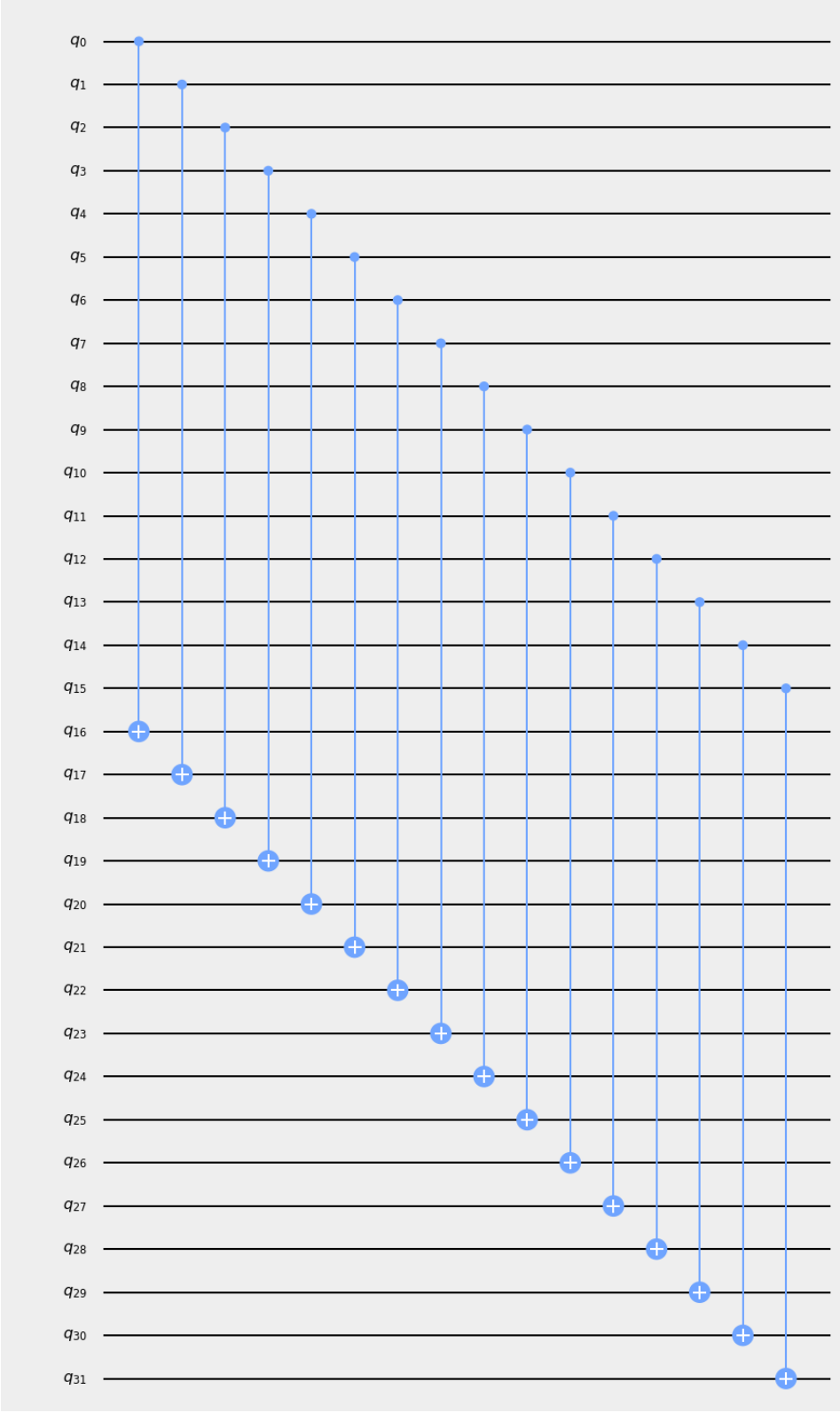

Figure 7.1: AddRoundKey() circuit

Figure 7.2: ShiftRows() circuit

q_0	q_1	q_2	q_3
q_4	q_5	q_6	q_7
q_8	q_9	q_{10}	q_{11}
q_{12}	q_{13}	q_{14}	q_{15}

$\longrightarrow$

q_0	q_1	q_2	q_3
q_5	q_6	q_7	q_4
q_{10}	q_{11}	q_8	q_9
q_{15}	q_{12}	q_{13}	q_{14}

Figure 7.3: ShiftRows() Qubit permutation

7.1.3 MixColumns()

MixColumns is applied to each column of the state to get an updated state. A column consists of 32 qubits and the MixColumns matrix M is 4x4 quwords in size.:

$$M = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Note that all of the entries of the above matrix are in hexadecimal. and that

$$\{01\} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \{02\} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\{03\} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

To make a quantum equivalent circuit of MixColumns, we can use the LUP decomposition method and obtain a circuit with 277 CNOT gates and a total depth of 39.

7.2 S-Box of AES and SubBytes()

Classically, a lookup table can be used to implement SubBytes, but this is not practical on a Quantum Computer, so we need to treat S-Box as a function. We need to use T-gate (an expensive quantum resource) to implement the S-Box function and thus the number of S-Boxes required can also be taken to be a measure of the cost.

There are 2 steps in implementation of S-Box:

- Finding the inverse of an element $\alpha \in GF(2^8)$
- Applying affine transformation on this inverse

7.2.1 Finding the Inverse

The inverse of $\alpha \in GF(2^8)$ is:

$$\alpha^{-1} = \alpha^{254} = ((\alpha \cdot \alpha^2) \cdot (\alpha \cdot \alpha^2)^4 \cdot (\alpha \cdot \alpha^2)^{16} \cdot \alpha^{64})^2$$

As the main culprit of T-gates is multiplication, and thus keeping the number of T-gates used less is a priority; even though the inverse of α appears to have 6 multiplications, the

use of the fact that $(\alpha \cdot \alpha^2)$ appears more than once, helps reduce the number of times multiplication needs to be performed.

Qubits																					
0-7	α	α	α	α	α	α	α	α	α	α^{64}	α^{64}	α^{64}	α	α	α	α	α	α	α	α	α
8-15	0	α^2	α^2	α^2	α^2	α^2	α^2	α^2	0	0	α^{127}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}	α^{254}
16-23	0	0	α^3	α^3	α^3	0	0	α^{63}	α^{63}	α^{63}	α^{63}	α^{63}	α^{63}	0	α^3	α^3	α^3	α^3	α^3	0	0
24-31	0	0	0	α^{12}	α^{12}	α^{12}	α^{48}	α^{48}	α^{48}	α^{48}	α^{48}	α^{48}	α^{48}	α^{48}	α^{48}	α^{12}	α^{12}	0	α^2	α^2	0
32-39	0	0	0	0	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	α^{15}	0	0	0	0	0

Figure 7.4: Step by step process for computing α^{-1}

Each step in Fig. 7.4 represents a multiplication.

$\alpha \cdot \alpha = \alpha^2$ (1st grey block)
$\alpha \cdot \alpha^2 = \alpha^3$ (2nd grey block)
$\alpha^3 \cdot (\alpha^2)^2 = \alpha^1 2$ (3rd grey block)
$\alpha^3 \cdot \alpha^1 2 = \alpha^1 5$ (4th grey block)
Reinitialize to 0 (5th grey block)
$\alpha \cdot \alpha^2 \cdot (\alpha^{15})^3 = \alpha^{48}$
... and so on

To realize these multiplications, we can use a general purpose multiplier in $GF(2^8)$.

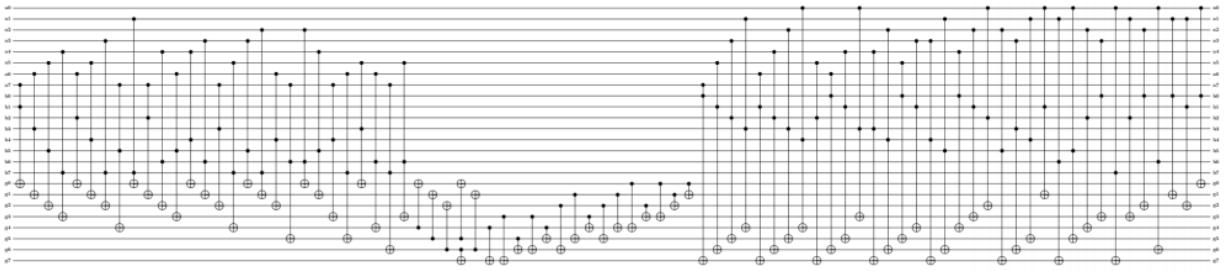

Figure 7.5: Multiplication in $F_2[x]/(1 + x + x^3 + x^4 + x^8)$

7.2.2 Affine Transformation

Affine transformation is defined as:

$$\{b'\} = M\{b\} \oplus \{v\}$$

Where $M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$ & $\{v\} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

Applying LUP decomposition on M:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(M = PLU)$$

The quantum equivalent circuit of affine transformation would thus be:

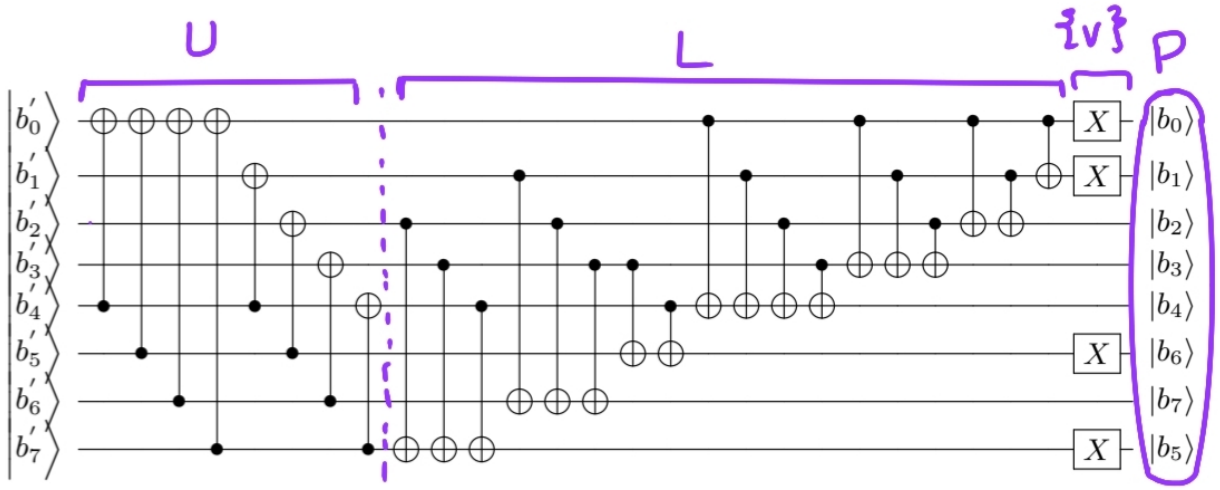

Figure 7.6: Quantum circuit of Affine transformation

7.3 KeyExpansion()

The different functions used in KeyExpansion():

- The RotWord() operation in KeyExpansion() algorithm is a simple rotation and would be structured similar to ShiftRows().
- SubWord() operation is analogous to SubBytes() and thus is costly.
- Rcon[i] adds $x^{i-1} \in F_{256}$ to the first byte of each word and thus has a simple circuit including only CNOT gates (as an equivalent to XOR gates).

The words created by this algorithm are divided into two types:

- Words not needing SubWord() in their computation
  - Can be constructed recursively from those that do by a combination of CNOT gates which makes them simple to compute
  - Word 41 is the most expensive of these words which is constructed by XORing 11 previous words costing 352 CNOT gates

- Words needing SubWord() in their computation
  - These (every 4th or 6th words) are stored as they are constructed since SubWord() is costly
  - These words are produced by starting with the previous word which must be constructed, and removed as needed.
  - Example: for w_8, first w_7 must be constructed on top of w_4, used, and then removed. (See Fig. 7.7)
  - Since SubWord() applies SubBytes() to each byte of the word independently, each of the four SubBytes computations can be done concurrently for which 96 auxiliary qubits would be needed, along with 32 qubits to store the new word

7.4 AES Rounds

Before the first round of AES, or we may term in the zeroth round of AES, a round key is XORed to the input state. Since the input state is a fixed value, we can do this step by simply flipping bits and hence approximately 64 NOT gates are used on the first four words of the key saving 128 qubits.

The circuit structure differs slightly per round for the 10, 12, or 14 rounds of AES to reduce qubits and depth despite the fact that all appply the same basic functions.

SubBytes must be computed 16 times per round. If we want to execute SubBytes simultaneously then we need 384 auxiliary qubits or else there is a need for increase in depth.

All 16 SubBytes calculations per round can be done with a maximum depth of 8 subBytes cycles by using the required 24 auxiliary qubits and 128 qubits to store the result.

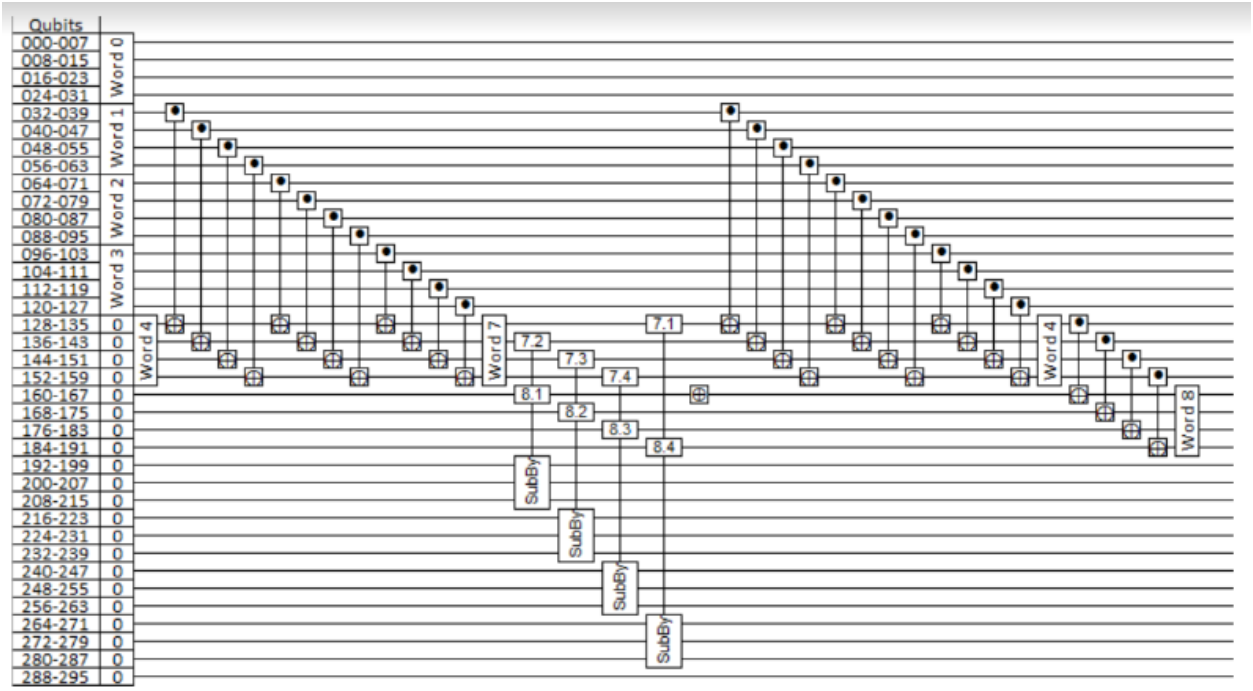

Figure 7.7: Construction of 8th word

The computation for AES-k takes 128 qubits times the number of rounds per AES, in addition to the number of qubits needed to store the original key. These qubits can be reduced in number by reversing steps between computations to clear qubits for future use. On the application of SubBytes, by reversing enough steps we can remove the input. The reverse process representation for AES-128 is shown in Fig. 7.8.

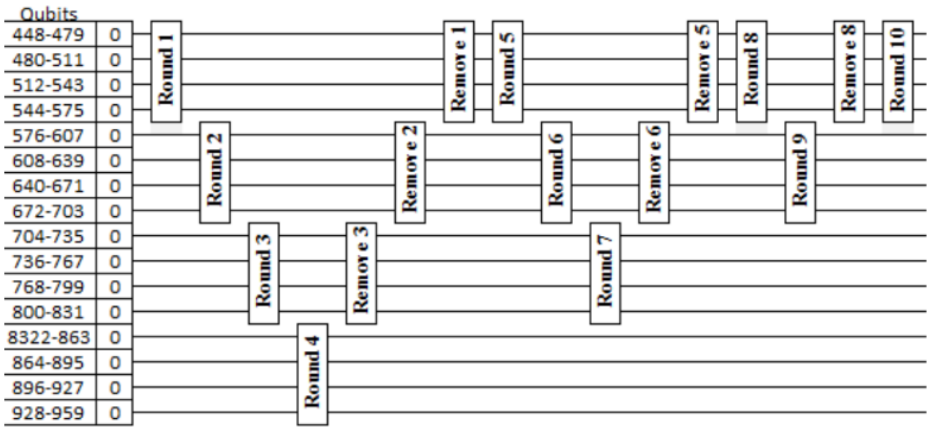

Figure 7.8: The 10 Rounds of AES-128 with 3 rounds of cleanup

Chapter 8

Reducing the Cost of Implementing AES as Quantum Circuit

"Cost is more important than quality, but quality is the best way to reduce cost."

– *Genichi Taguchi*

Since we've now somewhat realized the equivalent quantum circuit of AES, let's now discuss on reducing its quantum cost [17][18]!

8.1 Quantum Circuit for AES's S-Box

All the other components of Quantum Equivalent circuit of AES than the S-Box have a relatively simple and comparatively cost efficient circuit. The most costly circuit is of S-Box. The Boyar and Paralta technique is used for the optimization of the circuit of S-Box. The two steps of Boyar and Paralta technique serve the following purpose:

- Step 1 reduces the number of Toffoli (equivalent AND) gates in use
- Step 2 reduces the number of CNOT (equivalent XOR) gates in use

8.1.1 S-box Decomposition using Boyar and Peralta Technique

AES's S-Box can be written mathematically as:

$$S(x) = B \cdot F(U \cdot x) + [1\ 1\ 0\ 0\ 0\ 1\ 1\ 0]^T$$

where " $\cdot$ " is matrix multiplication and x is the 8-bit S-Box input.

The S-Box of AES consists of an initial linear expansion U from 8 to 22 bits ($U \in F_2^{22x8}$), followed by a non-linear contraction F from 22 to 18 bits ($F : F_2^{22} -> F_2^{18}$), and ending with a linear contraction B from 18 to 8 bits ($B \in F_2^{8x18}$). The portion of the circuit defined by U overlaps wit the $GF(2^8)$ inversion.

$$U = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Using the Boyar and Paralta Technique [12], the circuit derived is as in Fig. 8.1, where "+" translates to CNOT gate and "$\cdot$" translates to Toffoli gate. t_{25} through t_{40} in Fig. **??** is the circuit for inversion in $GF(2^4)$.

$$
\begin{array}{lll}
t_2 = y_{12} \cdot y_{15} & t_3 = y_3 \cdot y_6 & t_4 = t_3 + t_2 \\
t_5 = y_4 \cdot x_7 & t_6 = t_5 + t_2 & t_7 = y_{13} \cdot y_{16} \\
t_8 = y_5 \cdot y_1 & t_9 = t_8 + t_7 & t_{10} = y_2 \cdot y_7 \\
t_{11} = t_{10} + t_7 & t_{12} = y_9 \cdot y_{11} & t_{13} = y_{14} \cdot y_{17} \\
t_{14} = t_{13} + t_{12} & t_{15} = y_8 \cdot y_{10} & t_{16} = t_{15} + t_{12} \\
t_{17} = t_4 + t_{14} & t_{18} = t_6 + t_{16} & t_{19} = t_9 + t_{14} \\
t_{20} = t_{11} + t_{16} & t_{21} = t_{17} + y_{20} & t_{22} = t_{18} + y_{19} \\
t_{23} = t_{19} + y_{21} & t_{24} = t_{20} + y_{18} &
\end{array}
$$

$$
\begin{array}{lll}
t_{25} = t_{21} + t_{22} & t_{26} = t_{21} \cdot t_{23} & t_{27} = t_{24} + t_{26} \\
t_{28} = t_{25} \cdot t_{27} & t_{29} = t_{28} + t_{22} & t_{30} = t_{23} + t_{24} \\
t_{31} = t_{22} + t_{26} & t_{32} = t_{31} \cdot t_{30} & t_{33} = t_{32} + t_{24} \\
t_{34} = t_{23} + t_{33} & t_{35} = t_{27} + t_{33} & t_{36} = t_{24} \cdot t_{35} \\
t_{37} = t_{36} + t_{34} & t_{38} = t_{27} + t_{36} & t_{39} = t_{29} \cdot t_{38} \\
t_{40} = t_{25} + t_{39} &
\end{array}
$$

$$
\begin{array}{lll}
t_{41} = t_{40} + t_{37} & t_{42} = t_{29} + t_{33} & t_{43} = t_{29} + t_{40} \\
t_{44} = t_{33} + t_{37} & t_{45} = t_{42} + t_{41} & z_0 = t_{44} \cdot y_{15} \\
z_1 = t_{37} \cdot y_6 & z_2 = t_{33} \cdot x_7 & z_3 = t_{43} \cdot y_{16} \\
z_4 = t_{40} \cdot y_1 & z_5 = t_{29} \cdot y_7 & z_6 = t_{42} \cdot y_{11} \\
z_7 = t_{45} \cdot y_{17} & z_8 = t_{41} \cdot y_{10} & z_9 = t_{44} \cdot y_{12} \\
z_{10} = t_{37} \cdot y_3 & z_{11} = t_{33} \cdot y_4 & z_{12} = t_{43} \cdot y_{13} \\
z_{13} = t_{40} \cdot y_5 & z_{14} = t_{29} \cdot y_2 & z_{15} = t_{42} \cdot y_9 \\
z_{16} = t_{45} \cdot y_{14} & z_{17} = t_{41} \cdot y_8 &
\end{array}
$$

Figure 8.1: AES's S-Box circuit by Boyar and Paralta technique

8.1.2 Simulating the Optimized Quantum Circuit of S-Box

It is observed that after being accessed for some computations, certain wires remain idle until the end. Thus, uncomputing these wires enables us to reduce the number of qubits used and reuse these wires instead of using additional ancillas

It is also observed that the output wires of S-Box donot require to be cleaned up and thus we apply Toffoli gates directly to those wires. Also, we try to place gates in such a manner that intermediate wires are avoided.

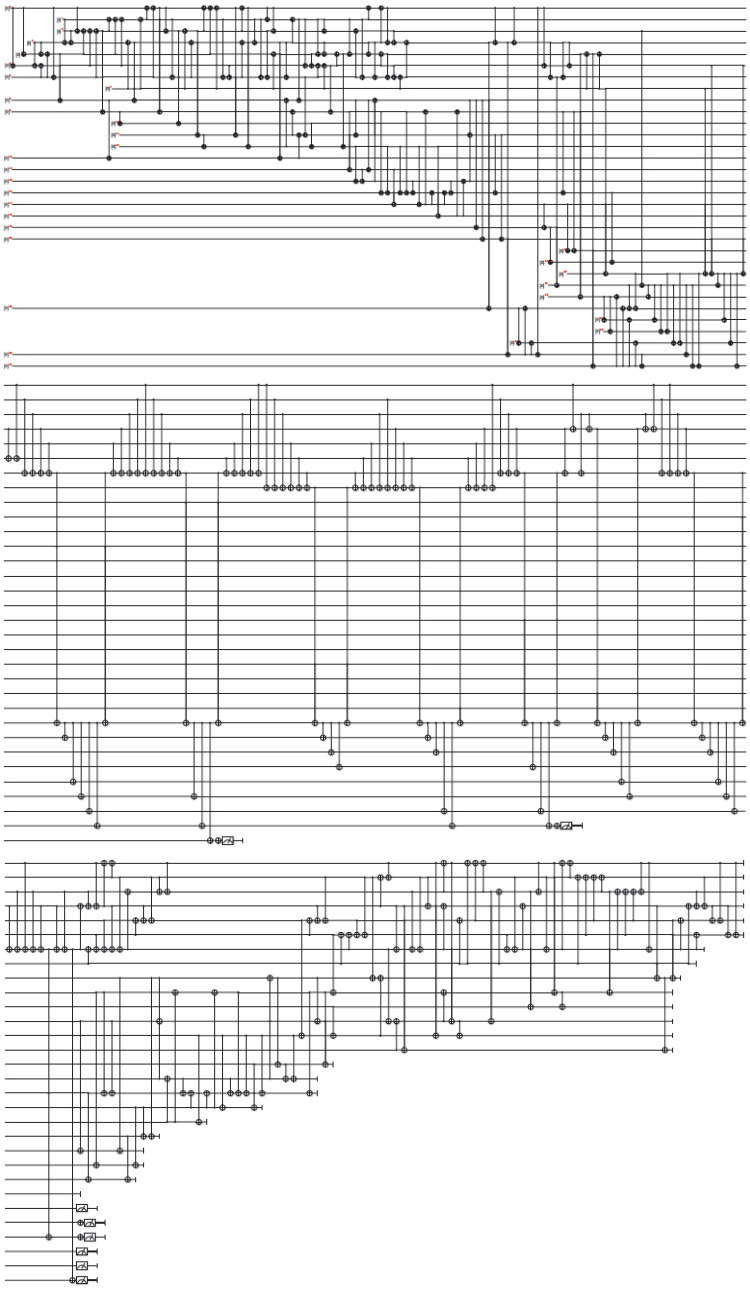

Figure 8.2: Simulation of S-Box's Quantum circuit

The final circuit thus obtained uses 32 qubits, 214 CNOT, 4 NOT, and 55 Tofolli gates with a Toffoli depth of 39 and an overall S-Box depth of 298. The high-level view of the circuit can be seen in Fig. 8.2 and the link to the code is provided in the Appendix. The code was written in an open-source software framework ProjectQ [19].

8.2 Resource Estimate for AES-k Quantum Circuit

The other components of AES than the S-Box, despite being already more cost efficient than S-Box, can also have their cost of implementation reduced by applying some further techniques

The "zig-zag" method (as described in Section 7.4.) is adopted for round generation. It's recognized that by storing all k_{4n+3} for AES-128 and AES-256 and k_{4n+5} for AES-192 where n represents the round number, not only could we use a combination to construct future keys, but also gain the ability to remove keys once they are no longer used in future constructions.

8.2.1 Savings in AES-128

Key expansion involves computation of various keywords k_i. 4 S-Boxes and an XOR of previous keys is required by every k_{4n} keyword. After k_{12}, it is observed that all keys have similar structure. Once Round n is fully computed, k_{4n+3} is stored. Each keyword is constructed at simultaneously with the round it is used in to save depth (with exception of Round 0 in which the cipher key (k_0 to k_3) and plaintext are XORed together). Parallelization greatly reduces depth for the remaining rounds. For example, during Round 2, all S-box computations for k_8 as well as Round 2 can be computed with an S-box depth of one, using 320 auxiliary qubits.

After computation of these MixColumns and S-Boxes, we can XOR k_8 onto Round 1 and then construct and XOR onto Round 1 construction the k_9, k_{10}, and k_{11} keywords. Thus, Round 2 is fully computed, and k_{11} is stored, and this entire construction took a total S-box depth of one. In absence of auxiliary qubits, parallelization cannot be done and thus S-Box depth must be increased up to 7. Round 1 (without k4), Round 2, removing Round 1, and Round 5 all are computed with an S-box depth of one.

Despite the fact that computation of keywords alongside rounds reduces depth, it still does not mean that in absence of auxiliary qubits, the 20 S-Boxes (4 for the key and 16 for the round) require an elevated S-Box depth. Even though Round 2 has an S-Box depth of 1, it is seen that Round 7 has an S-Box depth of 7 as only 16 auxiliary qubits are avaiable. Round keys can also sometimes be computed during the reversing and cleaning up of previous rounds.

In addition to cleaning up of rounds, removing the keywords that no longer serve any use for the remaining circuit also allows us to gain storage space. This removal is done using S-boxes in reverse after the keys are returned to their k_{4n} values. Both the cleaning up of rounds and removal of keys is done using the "zig-zag" method (as in Section 7.4).

Removal of keywords required the use of 12 additional S-Boxed to save qubits and this removal can be done in parallel with removal of rounds to prevent additional depth.

$k_4 : k_3, k_0$	$k_5 : k_4, k_1$	$k_6 : k_5, k_2$	$k_7 : k_6, k_3$
$k_8 : k_7, k_3, k_2, k_1$	$k_9 : k_8, k_7, k_3, k_2$	$k_{10} : k_7, k_3$	$k_{11} : k_{10}, k_7$
$k_{12} : k_{11}, k_7, k_2$	$k_{13} : k_{12}, k_{11}, k_3$	$k_{14} : k_{13}, k_{11}, k_7$	$k_{15} : k_{14}, k_{11}$
$k_{16} : k_{15}, k_{11}, k_7, k_3$	$k_{17} : k_{16}, k_{15}, k_7$	$k_{18} : k_{17}, k_{15}, k_{11}$	$k_{19} : k_{18}, k_{15}$
$k_{20} : k_{19}, k_{15}, k_{11}, k_7$	$k_{21} : k_{20}, k_{19}, k_{11}$	$k_{22} : k_{21}, k_{19}, k_{15}$	$k_{23} : k_{22}, k_{19}$
$k_{24} : k_{23}, k_{19}, k_{15}, k_{11}$	$k_{25} : k_{24}, k_{23}, k_{15}$	$k_{26} : k_{25}, k_{23}, k_{19}$	$k_{27} : k_{26}, k_{23}$
$k_{28} : k_{27}, k_{23}, k_{19}, k_{15}$	$k_{29} : k_{28}, k_{27}, k_{19}$	$k_{30} : k_{29}, k_{27}, k_{23}$	$k_{31} : k_{30}, k_{27}$
$k_{32} : k_{31}, k_{27}, k_{23}, k_{19}$	$k_{33} : k_{32}, k_{31}, k_{23}$	$k_{34} : k_{33}, k_{31}, k_{27}$	$k_{35} : k_{34}, k_{31}$
$k_{36} : k_{35}, k_{31}, k_{27}, k_{23}$	$k_{37} : k_{36}, k_{35}, k_{27}$	$k_{38} : k_{37}, k_{35}, k_{31}$	$k_{39} : k_{38}, k_{35}$
$k_{40} : k_{39}, k_{35}, k_{31}, k_{27}$	$k_{41} : k_{40}, k_{39}, k_{31}$	$k_{42} : k_{41}, k_{39}, k_{35}$	$k_{43} : k_{42}, k_{39}$

Figure 8.3: The keys required to construct each key in AES128. The leftmost column requires four S-boxes while the rightmost column is what is stored at the end of each round.

8.2.2 Savings in AES-192

AES-192 differs from AES-128 and AES-256 in key generation. It uses an S-Box only for every 6th key but only 4 keys are needed per round therefore some of the rounds need just 16 S-Boxes to be computed whereas some need additional 4 S-Boxes for key generation. So despite the fact that there are more rounds in AES-192 than in AES-128, there are lesser number of keywords generated and hence by the time k_{48} needs to be computed, for example, keywords k_{11} and prior are no longer of any use and thus we can reverse k_{11} to k_6 and then remove it using an inverted S-Box. This saves 32 qubits at the cost of 4 additional S-boxes.

It is also noted that the "zig-zag" method uses the same number of qubits for AES-192 and AES-256. Thus there is room for more space savings. This addional room can be used for the key expansion and this saves another 32 qubits for the expense of addtional 4 inverted S-Boxes.

8.2.3 Savings in AES-256

For AES-256, the methods described for AES-128 in Section 8.2.1 apply equally. But, in AES-256, removal of keys is not simple as it requires more previous keys. However, the after the construction of key k_{47} and Round 11, keys for the Rounds 2 and 3 can be removed in a similar manner and the keys for Rounds 12 and 13 can be stored. Additionally, after Round 13, we can remove and replace k_{23} with k_{59} which is key material for Round 14. In this manner we save a total of 96 qubits for an elevated cost of 12 S-Boxes and and addional S-Box depth of 3.

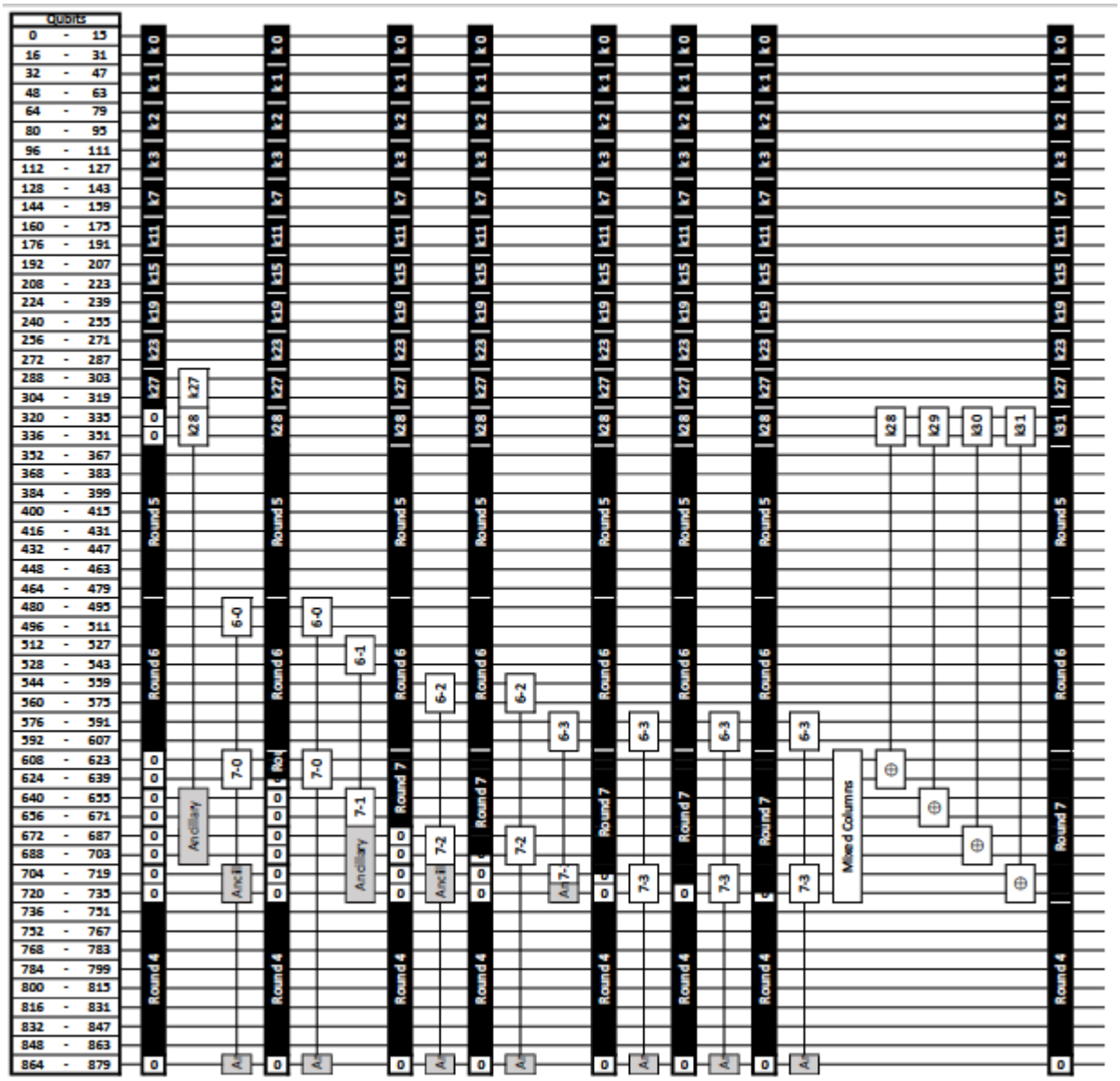

Figure 8.4: AES-128 Diagram of Round 7 computations which have an S-box depth of seven. Each column represents an S-box depth of one.

Using this method to compute keywords during round generation and just storing k_{4n+3} for AES-256 and AES-128, and k_{4n+5} for AES-192 means that the keys between this key and k_{4n} need to be computed several times. However, this method is advantageous to other methods as it produces additional keys directly in the rounds.

Fig. 8.5 summarizes the resources needed to implement AES with the approach suggested

	#NOT	#CNOT	#Toffoli	S-box Depth	Toffoli Depth	#Qubits
AES-128	1,507	107,960	16,940	47	1,880	864
AES-192	1,692	125,580	19,580	41	1,640	896
AES-256	1,992	151,011	23,760	54	2,160	1,232

Figure 8.5: Resources for AES Quantum Circuit Implementation

	Grassl et al.				
	#NOT	#CNOT	#Toffoli	Toffoli Depth	#qubits
AES-128	1,456	166,548	151,552	12,672	984
AES-192	1,608	189,432	172,032	11,088	1,112
AES-256	1,943	233,836	215,040	14,976	1,336

	Almazrooie et al.				
	#NOT	#CNOT	#Toffoli	Toffoli Depth	#qubits
AES-128	1,370	192,832	150,528	*(not reported)*	976
AES-192	*(not reported)*				
AES-256	*(not reported)*				

Figure 8.6: Resource estimates using other designs in literature [1] [2]

here. Comparing Fig 8.6 with Fig. 8.5, we see that the revised S-box design in combination with the changes to handling the key expansion enables attractive resource savings.

Chapter 9

Conclusion

"All's well that ends well."

– *William Shakespeare*

In Chapter 3, we saw how we can reduce max controls in quantum gates, and that can help us in further reducing the cost of implementing concatenated Toffoli gates. In Chapter 5 we explored in detail the Boyar and Paralta Technique which helps in reducing the number of non-linear gates (AND/Toffoli gates) used. In Chapter 7 we overviewed how in literature people have designed equivalent quantum circuits of different components of AES and how use of different techniques such as LUP type decomposition has been exploited for this purpose. In Chapter 8 we saw how in literature people have attempted to reduce the cost of implementation of quantum circuit of AES and how this improvement compares with prior work in literature.

After looking at how quantum circuits are made for different components of the AES, and then seeing different variations of these circuits, each with its own costs, we saw that each version has its merits and demerits.

The resource estimate of the quantum circuit of AES as discusses in Chapter 7 has been counted to be 1,380,420 Clifford gates, 1,060,864,984 T-gates and 984 qubits in total [15]. We can clearly see the large number of T gates that have been used in designing this circuit. While T gates are most costly of all gates, they also provide the required complexity to keep the AES secure. This means that adding more T gates to the circuit will increase the number of possible times a multiplication is being performed, and even though multiplication may not require as many gates as addition or subtraction, the fact that it is done multiple times is what increases the cost. Including these T gates ensures that even if an attempt is made

to crack the cipher, the resources will remain insufficient, because of the circuit's complexity.

On the other hand, if we reduce the number of T gates to save our cost, we will also be reducing the algorithm's complexity, and having a more cost-efficient circuit could then equate to having a circuit that has more chances of being compromised.

Some operations like ShiftRows() and RotWord() of AES do not add to the resilience of the quantum circuit implementation of AES from a quantum attack, as they involve only permutations. Additionally XORs don't always require CNOT gates to function while constructing a quantum circuit, and are also quite less in number in the design of the quantum circuit of AES. Thus, if we want to make a the Cipher more resilient, we would need to keep the count of T-gates high. On the other hand, if we want to reduce the cost of implementing the Cipher as a Quantum Circuit, we would need to reduce the T-gate count and depth, and apply different cost reduction techniques

Toffoli gates can be decomposed to a T-depth of 7 or 5, and by building on Boyar and Paralta's Technique we can reduce up to 88% of Toffoli gates required in all versions of AES circuits as compared to [20]. This also means that we have a reduced S-Box depth as the S-Box is the only component of AES that involves use of Toffoli gates. And hence, the cost of implementation of AES reduces significantly in the circuit design in [17] as compared to [15].

Furthermore, we saw that the use of "zig-zag" method and a keen observation of how and where there is space to reduce the cost, the cost of implementing components other than the S-Box of AES can also be reduced significantly.

Appendix A

An appendix

All of the codes we wrote can be found at the following github link:

https://github.com/Javaria-Ghafoor/Quantum-Circuits-of-Quantum-Safe-Encryption

Bibliography

[1] L. B. R. M. S. R. Grassl, M., *Applying grover's algorithm to aes: Quantum resource estimates. proc. post-quantum cryptography. post quantum cryptography. springer, cham, pp. 29-43* (2016).

[2] S. A. A. R. M. K. Almazrooie, M., *Quantum reversible circuit of aes-128. quantum inf. process., vol. 17, art. no. 339* (2018).

[3] C. J. Benvenuto, *Galois field in cryptography. university of washington, 1(1), pp.1-11* (2012).

[4] J. Porter, *Google confirms 'quantum supremacy' breakthrough*, `https://bit.ly/3if3zSX`, [Online; accessed 23-May-2021].

[5] L. Crane, *Honeywell claims it has built the most powerful quantum computer ever*, `https://bit.ly/2SEurkc`, [Online; accessed 23-May-2021].

[6] W. Lin, *Chinese team designs 62-qubit quantum processor with world's largest number of superconducting qubits*, `https://bit.ly/2Sxyzm9`, [Online; accessed 23-May-2021].

[7] Y. A. Montaser, R. and M. Abdel-Aty, *Improving the quantum cost of nct-based reversible circuit. quantum information processing, 14(4), pp.1249-1263.* (2015).

[8] *Using quantum gates instead of ancilla bits.*

[9] U. D. Math, *Lu decomposition*, `https://www.math.ucdavis.edu/~linear/old/notes11.pdf`.

[10] B. J. Amento, *Quantum circuits for cryptanalysis, pp.17-18*, `https://bit.ly/3wEu0VS` (2016).

[11] P. R. Boyar, J., *A new combinational logic minimization technique with applications to cryptology. in international symposium on experimental algorithms (pp. 178-189). springer, berlin, heidelberg.* (2010).

[12] M. P. Boyar, J. and R. Peralta, *Logic minimization techniques with applications to cryptology. journal of cryptology, 26(2), pp.280-312.* (2013).

[13] P. R. Boyar, J., *A depth-16 circuit for the aes s-box. cryptology eprint archive: Report 2011/322 [online]*, `https://eprint.iacr.org/2011/322` (2011).

[14] N. FIPS, *197: Announcing the advanced encryption standard (aes). information technology laboratory, national institute of standards and technology, 5(4).* (2001).

[15] B. Langenberg, *Quantum circuits for symmetric cryptanalysis (doctoral dissertation, florida atlantic university).* (2018).

[16] Qiskit, *An open-source framework for working with noisy quantum computers at the level of pulses, circuits, and algorithms*, `https://github.com/Qiskit`.

[17] H. Pham, *Contributions to quantum-safe cryptography: Hybrid encryption and reducing the t gate cost of aes (doctoral dissertation, florida atlantic university).* (2019).

[18] P. H. S. R. Landenberg, B., *Reducing the cost of implementing the advanced encryption standard as a quantum circuit. ieee transactions on quantum engineering, vol. 1, pp. 1-12* (2020).

[19] ProjectQ, *An open source software framework for quantum computing*, `https://github.com/ProjectQ-Framework`.

[20] M. R. R. S. Markus Grassl, Brandon Langenberg, *Applying grover's algorithm to aes: quantum resource estimates*, `https://arxiv.org/abs/1512.04965`.