

Implementation of Quantum Hash

Synopsis Submitted in Partial Fulfillment of
The Requirements for The Degree Of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS & COMMUNICATION ENGINEERING

Of

Maulana Abul Kalam Azad University of Technology, West Bengal

By

DEBOSMITA MONDAL – 92 – 10900320094

SNEHA SAHA – 88 – 10900320090

SHOUVIK PAUL – 66 – 10900320067

SOUMYAJYOTI SAHA – 57 – 10900320058

Under the Guidance of

PROF. ANINDITA SARKAR

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING



**NETAJI SUBHASH ENGINEERING
COLLEGE TECHNOLOGY, GARIA,
KOLKATA – 700152**

2020-2024

CERTIFICATE

This is to certify that this project report titled “**Implementation of Quantum Hash**” submitted in partial fulfilment of requirements for award of the degree Bachelor of Technology (B. Tech) in **Electronics & Communication Engineering** of West Bengal University of Technology is a faithful record of the original work carried out by,

DEBOSMITA MONDAL – 92 – 10900320094 – 201090100310002 OF 2020-21

SNEHA SAHA – 88 – 10900320090 – 201090100310006 OF 2020-21

SHOUVIK PAUL – 66 – 10900320067 – 201090100310029 OF 2020-21

SOUMYAJYOTI SAHA – 57 – 10900320058 – 201090100310038 OF 2020-21

under my guidance and supervision.

It is further certified that it contains no material, which to a substantial extent has been submitted for the award of any degree/diploma in any institute or has been published in any form, except the assistances drawn from other sources, for which due acknowledgement has been made.

Date: .2024

Guide's signature

Prof. Anindita Sarkar

Head of the Department
Electronics & Communication Engineering
NETAJI SUBHASH ENGINEERING COLLEGE
TECHNO CITY, GARIA, KOLKATA – 700 152

CERTIFICATE OF APPROVAL

We hereby approve this synopsis of the project

Implementation of Quantum Hash

carried out by

DEBOSMITA MONDAL – 92 – 10900320094 – 201090100310002 OF 2020-21

SNEHA SAHA – 88 – 10900320090 – 201090100310006 OF 2020-21

SHOUVIK PAUL – 66 – 10900320067 – 201090100310029 2020-21

SOUMYAJYOTI SAHA – 57 – 10900320058 – 201090100310038 OF 2020-21

under the guidance of Prof. Anindita Sarkar of Netaji Subhash Engineering College,
Kolkata in partial fulfilment of requirements for award of the degree Bachelor of
Technology (B. Tech) in Electronics & Communication Engineering of West
Bengal University of Technology

Date:

Examiners' signatures:

1.
2.
3.

ABSTRACT

Hash function is a very popular cryptographic function because of its high collision resistance property. It has been used for authentication purpose in many significant applications such as in IoT, block chain, on-line transactions etc. It is also used for password verification and Rootkit detection. Now a days, the potential emergence of quantum computing threatens these security solutions widely in use. Many of the novel cryptographic algorithms are found unable to resist quantum attacks. Even Hash functions having length less than 384 bits are in threat. This thesis proposes a 256-bit Quantum Hash which is a modified version of SHA-256. The modified SHA- 256 proposed in this thesis is implemented with reversible gates so that it could be simulated in both classical and quantum computer. The modified hash function simulated in python Qiskit is dynamic and lightweight in nature as well as avail the advantages of power reduction and speed maximization. The whole circuit is a combination of XOR and Feynman Gate. The comparative performance analysis with the reversible version of standard SHA – 256 algorithms is done in terms of gate count, garbage value and time complexity.

ACKNOWLEDGEMENT

We would like to take this opportunity to express our heartfelt gratitude and sincere appreciation to our esteemed supervisor, Professor Anindita Sarkar, from Netaji Subhash Engineering College. Professor Sarkar's constant guidance, expertise, understanding, and support have been instrumental in our research journey, making it possible for us to delve into the captivating topic of "Implementation of Quantum Hash."

Throughout our research endeavor, Professor Sarkar has consistently shown exceptional dedication and commitment to our growth and development. Her profound knowledge and insightful feedback have been invaluable in shaping our ideas and refining our work. Her guidance has not only provided us with a strong foundation but has also encouraged us to explore innovative approaches and think critically. Under her mentorship, we have gained a deeper understanding of the subject matter and have developed essential research skills that will undoubtedly prove invaluable in our future endeavors.

Netaji Subhash Engineering College, our esteemed institution, also deserves our gratitude for the support and resources provided during our research journey. The college's commitment to fostering a conducive academic environment and promoting research activities has been instrumental in our progress. We are grateful for the opportunities we have received to engage in research work as undergraduate researchers since August 2023. The college's infrastructure, library facilities, and access to relevant literature have all contributed to enhancing the quality of our work.

Furthermore, we would like to extend our deepest gratitude to our parents for their unwavering love, support, and blessings throughout our academic journey. Their constant encouragement and faith in our abilities have been a driving force behind our accomplishments. Their sacrifices and belief in our potential have motivated us to strive for excellence and overcome challenges.

**DEBOSMITA MONDAL
SNEHA SAHA
SHOUVIK PAUL
SOUMYAJYOTI SAHA**

CONTENTS

Chapter 1: Introduction

| | |
|--|-------|
| 1.1 Motivation | 10 |
| 1.2 Objectives of the Dissertation | 10 |
| 1.3 Major Contribution of the Dissertation | 11 |
| 1.4 Literature Survey | 11-13 |
| 1.5 Structure of the thesis | 13 |

Chapter 2: Exploring the Importance of Hash Functions in Network Security

| | |
|---|-------|
| 2.1 Introduction | 14 |
| 2.2 Importance of Hash Function in Network Security | 14 |
| 2.3 Exploring the World of Hash Function | 14-15 |
| 2.4 SHA 256 | 15-17 |
| 2.4.1 Pseudo Code for SHA 256 | 15-16 |
| 2.4.2 Explanation of the Code | 16-17 |
| 2.5 Advantages | 17 |
| 2.6 Disadvantages | 17 |
| 2.7 Application | 17-19 |

Chapter 3: Reversible Implementation of Few Important Circuits

| | |
|---|-------|
| 3.1 Introduction | 20 |
| 3.2 Classification of Reversible Logic Gates | 20-21 |
| 3.3 Application of Reversible Logic Gates | 21 |
| 3.4 Reversible Implementation of Important Circuits | 22-50 |
| i. A Register Of 8-Qbits | 22-24 |
| ii. Not Gate | 25-26 |
| iii. Cnot/Feynman Gate | 27-29 |
| iv. Swap Gate | 30-31 |
| v. Toffoli Gate | 32-34 |
| vi. CH Gate | 35-36 |
| vii. MCMT Gate | 37-38 |
| viii. Peres Gate | 39-41 |
| ix. Fredkin Gate | 42-43 |
| x. Sam Gate | 44-46 |
| xi. Double Feynman Gate | 47-49 |

Chapter 4: Exploring Original SHA-256

| | |
|----------------------------------|-------|
| 4.1 Introduction----- | 51 |
| 4.2 Original SHA 256 ----- | 51-52 |
| 4.3 Block Diagram ----- | 56 |
| 4.4 Pseudo Code for SHA 256----- | 52 |
| 4.5 Qiskit Code of SHA 256 ----- | 53-55 |

Chapter 5: Exploring Dynamic SHA-256

| | |
|--|-------|
| 5.1 Introduction----- | 57 |
| 5.2 Proposed Algorithm----- | 58 |
| 5.3 Pseudo Code for Dynamic SHA 256----- | 58-60 |
| 5.4 Qiskit Code of Dynamic SHA 256 ----- | 60-66 |

Chapter 6: Comparison Between Original SHA-256 & Exploring Dynamic SHA-256

| | |
|--|----|
| 6.1 Introduction----- | 67 |
| 6.2 A Comparative Analysis of Original SHA-256 & Exploring Dynamic SHA-256----- | 67 |

| | |
|------------------------|-----------|
| CONCLUSION----- | 68 |
|------------------------|-----------|

| | |
|-----------------------|-----------|
| REFERENCE----- | 69 |
|-----------------------|-----------|

LIST OF FIGURES

Fig 3.1 – Truth Table of a Register with 8-Qbits
Fig 3.2 – Quantum Circuit with a Register of 8-Qbits
Fig 3.3 – Truth Table of Not Gate
Fig 3.4 – Not Gate in Qiskit
Fig 3.5 – Truth Table of CNOT Gate
Fig 3.6 – CNOT Gate in Qiskit
Fig 3.7 – Truth Table in SWAP Gate
Fig 3.8 – SWAP Gate in Qiskit
Fig 3.9 – Truth Table of Toffoli Gate
Fig 3.10 – Toffoli Gate in Qiskit
Fig 3.11 – Truth Table of CH Gate
Fig 3.12 – CH Gate in Qiskit
Fig 3.13 – Truth Table of Peres Gate
Fig 3.14 – Peres Gate in Qiskit
Fig 3.15 – Truth Table for Fredkin Gate
Fig 3.16 – Fredkin Gate in Qiskit
Fig 3.17 – Truth Table for SAM Gate
Fig 3.18 – SAM Gate in Qiskit
Fig 3.19 – Truth Table for Double Feynman Gate
Fig 3.20 – Double Feynman Gate in Qiskit
Fig 3.21 – Reversible Gates in Qiskit

Fig 4.1 – Block Diagram of SHA-256

Fig 5.1 – Block Diagram of Dynamic
SHA-256

LIST OF ABBREVIATION

1. X: Pauli-X gate (also known as the NOT gate)
2. Y: Pauli-Y gate
3. Z: Pauli-Z gate
4. H: Hadamard gate
5. S: Phase gate ($\pi/2$)
6. S^\dagger : Conjugate of the Phase gate ($\pi/2$)
7. T: T gate ($\pi/4$)
8. T^\dagger : Conjugate of the T gate ($\pi/4$)
9. CX: Controlled-X gate (also known as the CNOT gate)
10. CY: Controlled-Y gate
11. CZ: Controlled-Z gate
12. CH: Controlled-Hadamard gate
13. CS: Controlled-S gate
14. CT: Controlled-T gate
15. CCX: Controlled-Controlled-X gate (also known as the Toffoli gate)
16. SWAP: Swap gate
17. U1: Phase gate (generalized)
18. U2: 2-parameter gate
19. U3: 3-parameter gate
20. RX: Rotation around the X-axis
21. RY: Rotation around the Y-axis
22. RZ: Rotation around the Z-axis
23. CRX: Controlled-Rotation around the X-axis
24. CRY: Controlled-Rotation around the Y-axis
25. CRZ: Controlled-Rotation around the Z-axis
26. CU1: Controlled-Phase gate (generalized)
27. CU2: Controlled-2-parameter gate
28. CU3: Controlled-3-parameter gate
29. RXX: Two-qubit XX rotation
30. RYY: Two-qubit YY rotation
31. RZZ: Two-qubit ZZ rotation
32. CCZ: Controlled-Controlled-Z gate (also known as the Toffoli gate)
33. CSWAP: Controlled Swap gate
34. C3X: Controlled-Controlled-Controlled-X gate (also known as the Fredkin gate)
35. MCX: Multiple-Controlled-X gate
36. MCY: Multiple-Controlled-Y gate
37. MCZ: Multiple-Controlled-Z gate
38. MCRX: Multiple-Controlled-Rotation around the X-axis
39. MCRY: Multiple-Controlled-Rotation around the Y-axis
40. MCRZ: Multiple-Controlled-Rotation around the Z-axis

CHAPTER 1: INTRODUCTION

1.1 Motivation:

In today's world, information security is of utmost importance, and cryptographic protocols are the backbone of secure communication. However, the advent of quantum computing poses a significant threat to these protocols as it has been projected that a quantum computer capable of breaking vital cryptographic schemes like RSA2048 will exist by 2035. This has led to a growing concern among experts as large-scale quantum computers are expected to become a reality in the near future, impacting all classical Cryptosystems. To address this concern, researchers have been focusing on designing and synthesizing efficient reversible implementations of security schemes that can be analyzed in both classical and quantum environments. Reversible logic gates are the critical components of quantum computing, and their implementation offers advantages such as power reduction and speed maximization.

The emergence of the Internet of Things (IoT) has enabled billions of devices that collect large amounts of data to be connected. Therefore, IoT security has fundamental requirements. One critical aspect of IoT security is data integrity. Cryptographic hash functions are cryptographic primitives that provide data integrity services. However, due to the limitations of IoT devices, existing cryptographic hash functions are not suitable for all IoT environments. As a result, researchers have proposed various lightweight cryptographic hash function algorithms. In this thesis a lightweight reversible Secured Hash Function (LRSHA) is designed with reversible logic gates. The proposed Hash function is implemented using IBM Qiskit quantum simulator. The motivation behind this research is to develop a secure and efficient hash function that can withstand quantum attacks and can be implemented in resource-constrained environments.

1.2 Objectives of The Dissertation:

The objective of the project includes:

- ❖ Designing a lightweight reversible Secured Hash Function (LRSHA) using reversible logic gates.
- ❖ Implementing the LRSHA using Python Qiskit quantum simulator.
- ❖ Evaluating the security and efficiency of the LRSHA against quantum attacks.
- ❖ Analyzing the performance of the LRSHA in resource-constrained IoT environments.
- ❖ Assessing the feasibility and practicality of using reversible logic gates in cryptographic hash function designs.
- ❖ Contributing to the field of information security by proposing a reversible hash function that addresses the limitations of existing cryptographic hash functions in IoT scenarios and provides resistance against quantum attacks.
- ❖ Investigating the potential benefits of reversible logic gates, such as power reduction and speed optimization, in the context of cryptographic hash function implementations.
- ❖ Enhancing the understanding of reversible computing and its applications in the field of information security.
- ❖ Providing insights and recommendations for future research and development in the area of reversible implementation of security schemes for quantum computing and IoT security.

1.3 Major Contribution of the Dissertation:

The major contributions of the dissertation include:

Development of a Lightweight Reversible Secured Hash Function (LRSHA): The dissertation proposes the design and implementation of LRSHA, a new reversible hash function specifically tailored for resource-constrained environments such as IoT devices. The LRSHA algorithm is designed using reversible logic gates, ensuring data integrity and security while minimizing computational overhead.

Analysis of LRSHA's Resistance against Quantum Attacks: The dissertation evaluates the security strength of LRSHA against potential quantum attacks, considering the looming threat of quantum computing to classical cryptographic schemes. Through thorough analysis and simulations, the dissertation demonstrates the robustness of LRSHA in withstanding attacks from quantum computers, ensuring long-term data security.

Exploration of Reversible Logic Gates in Cryptographic Hash Function Design: The dissertation contributes to the field of reversible computing by investigating the utilization of reversible logic gates in cryptographic hash function design. It offers insights into the potential advantages, such as reduced power consumption and increased computational speed, that reversible logic gates can bring to the implementation of secure hash functions.

1.4 Literature Survey:

In this literature review, we examine several recent studies related to the use of quantum algorithms in the field of secure hash functions. The first article by Richard Preston, from The MITRE Corporation in 2022, proposes an efficient method to implement classical hash functions MD5, SHA-1, SHA-2, and SHA3 as quantum oracles to analyze the computational resources required to conduct a preimage attack with Grover's Algorithm [1]. The article also introduces an improvement to the SHA-3 oracle that reduces the number of logical qubits needed in the Keccak block permutation by 40%.

The second article, authored by Sergi Ramos-Calderer, Emanuele Bellini, José Latorre, Marc Manzano, and Victor Mateu, and published in Springer on November 16, 2020, proposes an efficient way to implement Grover's algorithm in a quantum simulator to search for preimages of two scaled hash functions [2]. Their implementation allows for precise assessment of the scaling of the number of gates and depth of a full-fledged quantum circuit designed to find the preimages of a given hash digest.

The third study, by T.S. Thangave and A. Krishnan in IJCSE, focuses on the development of an improved secure hash function related to the syndrome decoding problem from the theory of error-correcting codes [3]. The study proposes a user interface and implementation of a browser extension called "password hash" to strengthen web password authentication, providing customized passwords to reduce the threat of password attacks with no server changes and little or no change to the user experience.

The fourth study by Hongbo Wang, Haoran Zheng, Bin Hui, and Hongwu Tang, titled "Improved Lightweight Encryption Algorithm Based on Optimized S-Box" (2013 International Conference on Computational and Information Sciences, pp 734-737, 2013), addresses the limitations of the static S-Box in the Advanced Encryption Standard (AES) by proposing the use of dynamic S-Boxes [4]. The study aims to enhance the security of AES by providing more confusion to cryptanalysts and presents an optimized lightweight encryption algorithm.

The fifth study by M. Almazrooie, A. Samsudin, R. Abdullah et al., titled "Quantum Reversible Circuit of AES-128"(Quantum Inf Process 17, 112, 2018), focuses on the explicit quantum design of AES-128 using the lowest number of qubits [5]. The study constructs the quantum version of AES-128 by designing the main components of AES-128 as quantumcircuits and combining them. It adopts efficient approaches from classical hardware implementations to design the circuitsof the multiplier and multiplicative inverse.

The seventh study, titled "Quantum Cost Optimization for Reversible Sequential Circuit," by Md. Selim Al Mamun and David Menville, published in the International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 4, No. 12, 2013, proposes the use of reversible sequential circuits as significant memory blocks for future computing devices due to their ultra-low power consumption [6]. The researchers highlight the design of various types of latches as a major objective in this field. The paper presents an efficient design of reversible sequential circuits optimized in terms of quantumcost, delay, and garbage outputs. To achieve this, the authors introduce a new 3x3 reversible gate called the SAM gate and employ it along with other basic reversible logic gates to design efficient sequential circuits.

The eighth study, authored by Sree Ramani Potluri and Lipsa Dash and titled "Design of Linear Feedback Shift Register using Reversible Logic," is published in the International Journal of Advanced Research in Computer and Communication Engineering, Vol. 4, Issue 12, December 2015 [7]. The study emphasizes the importance of reversible logic in various researchareas, such as low-power CMOS VLSI, nanotechnology, and quantum computation. The authors focus on the design and synthesis of efficient reversible logic circuits, with a specific application in linear feedback shift registers (LFSRs). They present designs of reversible LFSRs using D flip flops with asynchronous set/reset. The proposed designs are compared based on quantum cost, delay, and garbage outputs. The study utilizes important reversible gates such as Feynman gate, Fredkin gate, Toffoli gate, and SAM gate.

The ninth study, titled "On the Design of S-boxes" by A. Webster and S. Tavares, appears in Advances in Cryptology CRYPTO-1985, LNCS 218, Springer-Verlag, 1985 [8]. The authors discuss the Advanced Encryption Standard (AES), a widely used symmetric block cipher for securing data from hacking attempts. The paper focuses on the S-Box (Substitution Box), which provides confusion in the AES algorithm as its only nonlinear component. The static nature of the S-Box throughout the algorithm attracts the attention of cryptanalysts who seek to analyze its weaknesses for potential attacks. The study highlights the existence of algebraic attacks on AES since 2000, challenging its security.

The tenth study, authored by A. Peres and published in Physical Review, A32, 3266-3276, 1985, explores the potential of reversible logic design in reducing power consumption compared to traditional logic design [9] . The paper specifically addresses the application of reversible design in designing a reversible binary-coded- decimal adder. The proposed approachfocuses on selecting and arranging gates with parallel implementation to improve reversible performance parameters. The design achieves at least a 10% improvement in quantum cost compared to existing counterparts found in the literature. Additionally, the proposed design is mapped into its quantum equivalent.

The eleventh study, authored by M. S. Al Mamun and D. Menville, is titled "Quantum Cost Optimization for Reversible Sequential Circuit" and published in the International Journal of Advanced Computer Science and Applications, 4(12), 2013 [10]. It echoes the significance of reversible sequential circuits as memory blocks for future computing devices due to theirultra-low power consumption. The paper emphasizes the design of efficient reversible sequential circuits optimized in termsof quantum cost, delay, and garbage outputs. The authors introduce a new 3x3 reversible gate called the SAM gate and incorporate it with basic reversible logic gates to design efficient sequential circuits.

function based on programmable elementary cellular automata. Hash functions play a vital role in cryptographic protocols for authentication and integrity verification. The paper highlights the need for new hash function designs and strategies following the vulnerabilities identified in widely used functions such as MD5, SHA-1, and RIPEMD. The proposed hash function utilizes elementary cellular automata, known for their chaotic and complex behavior resulting from simple rule interactions, making them suitable for cryptographic applications.

The fourteenth study, authored by I. Hussain, T. Shah, M. A. Gondal, and W. A. Khan, and published in the World Applied Sciences Journal, 13(11), 2389-2395, focuses on constructing cryptographically strong 8x8 S-Boxes. The study addresses the reliability of the Advanced Encryption Standard (AES) in the face of natural and maliciously injected faults that can compromise the security and confidentiality of transmitted data. The S-Box is a key component of AES, and the paper presents a method for obtaining cryptographically strong 8x8 S-Boxes. The strength of the proposed S-Box is analyzed using various criteria such as the Strict Avalanche Criterion (SAC), Bit Independent Criterion (BIC), differential approximation probability (DP), linear approximation probability (LP), nonlinearity, and majority logic criterion.

1.5 Structure of the Thesis:

- The first chapter introduces this project by explaining the project background and objectives.
- Chapter 2 provides an overview of exploring the importance of hash functions in network security and a comprehensive analysis of different hash function.
- Chapter 3 explain the reversible implementation of few important circuits
- Chapter 4 provide a deep exploration into original SHA-256 algorithm and gives a detailed analysis of its properties and application.
- Chapter 5 discuss about the idea of advancing security with dynamic sha-256 and provides a comprehensive exploration and analysis.
- Chapter 6 provides a comparative study between original sha 256 algorithm and dynamic sha 256 algorithms.

CHAPTER 2: EXPLORING THE IMPORTANCE OF HASH FUNCTIONS IN NETWORK SECURITY

2.1 Introduction:

Hash functions are essential cryptographic tools that play a vital role in network security. They are designed to convert input data of any size into fixed-length hash values. These hash values are unique and provide a fingerprint or digital signature for the input data. In network security, hash functions are widely used for various purposes, including data integrity verification, password storage, digital signatures, and message authentication codes.

2.2 Importance of Hash function in Network Security:

Hash functions play a significant role in network security by providing a means of verifying the integrity of data being transmitted across the network. Hash functions take in data of arbitrary size and produce a fixed-size output, known as a hash or message digest. This hash is unique to the input data, which makes it a useful tool for verifying that the data has not been tampered during transmission.

Hash functions are used in a variety of network security applications, including digital signatures, password storage, and data validation. Digital signatures are used to verify the authenticity of a message, and a hash function is used to produce a unique signature for each message. Password storage systems use hash functions to convert passwords into hash values that can be stored securely. When a user enters their password, the system hashes the input and compares it to the stored hash value. If the two values match, the user is granted access.

Hash functions are also used for data validation. When data is transmitted across a network, it is vulnerable to corruption or tampering. By calculating the hash of the data before it is transmitted and then verifying the hash on the receiving end, the recipient can be sure that the data has not been modified during transmission. In summary, the importance of hash functions in network security lies in their ability to provide data integrity, authenticity, and validation. They are a fundamental tool for protecting data transmitted across a network from tampering and unauthorized access.

2.3 Exploring the World of Hash Functions: An Overview of Different Types of Hash

Hash functions are mathematical algorithms that are widely used in cryptography to produce a fixed-size output, or hash, from an input data of arbitrary size. The hash functions are designed to be one-way functions, meaning it is computationally infeasible to invert the hash function and obtain the original input data.

Here are some examples of commonly used hash functions:

- I. **MD5**: MD5 (Message-Digest algorithm 5) is a widely-used cryptographic hash function that produces a 128-bit hash value. It was developed by Ronald Rivest in 1991, and is commonly used in digital signature applications.

- II. **SHA-1**: SHA-1 (Secure Hash Algorithm 1) is a widely-used cryptographic hash function that produces a 160-bit hash value. It was developed by the National Security Agency (NSA) in the US and published as a federal information processing standard (FIPS) in 1995.
- III. **SHA-2**: SHA-2 is a family of cryptographic hash functions that includes SHA-224, SHA-256, SHA-384, and SHA-512. These hash functions produce hash values of 224, 256, 384, and 512 bits, respectively. SHA-2 was developed by the National Security Agency (NSA) in the US and published as a federal information processing standard (FIPS) in 2001.
- IV. **SHA-3**: SHA-3 (Secure Hash Algorithm 3) is the latest member of the SHA family of hash functions. It was developed by the National Institute of Standards and Technology (NIST) in response to the need for a new hash function due to vulnerabilities found in the previous SHA-2 family. SHA-3 produces hash values of 224, 256, 384, and 512 bits, respectively.
- V. **Blake 2**: BLAKE2 is a cryptographic hash function that produces hash values of 256 or 512 bits. It is designed to be faster and more secure than SHA-3, and is optimized for 64-bit platforms. These are just a few examples of commonly used hash functions. There are many other hash functions available, each with their own strengths and weaknesses.

2.4 SHA-256: Pseudo Code, Block Diagram, Advantages, Disadvantages, and Applications

SHA-256 is a widely used cryptographic hash function that generates a 256-bit message digest (or hash) for a given input. It was developed by the National Security Agency (NSA) in the United States and is one of the SHA-2 family of hash functions. SHA-256 is considered to be a secure hash function and is used in a variety of applications, including digital signatures, password storage, and blockchain technology.

2.4.1 The pseudocode for SHA-256:

1. Initialize variables:

$a0 = 0x6a09e667$; $b0 = 0xbb67ae85$; $c0 = 0x3c6ef372$; $d0 = 0xa54ff53a$; $e0 = 0x510e527ff0 = 0x9b05688c$; $g0 = 0x1f83d9ab$; $h0 = 0x5be0cd19$

2. Initialize constants:

$Kt = \{0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85, 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2\}$

3. *Initialize message schedule:*

$W_t = \{M(1), M(2), \dots, M(15), M(16), \dots, M(64)\}$
for $t = 17$ to 64 do
 $W_t = \sigma_1(W(t-2)) + W(t-7) + \sigma_0(W(t-15)) + W(t-16)$

4. *Initialize working variables:*

$a = a_0 ; b = b_0 ; c = c_0 ; d = d_0 ; e = e_0 ; f = f_0 ; g = g_0 ; h = h_0$

5. *Loop & Calculation*

For $t = 1$ to 64 do:
 $T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_t(t) + W_t(t)$
 $T_2 = \Sigma_0(a) + Maj(a, b, c)$
 $h = g$
 $g = f$
 $f = e$
 $e = d + T_1$
 $d = c$
 $c = b$
 $b = a$
 $a = T_1 + T_2$

6. *Update the hash values:*

$a_0 = a_0 + a$
 $b_0 = b_0 + b$
 $c_0 = c_0 + c$
 $d_0 = d_0 + d$
 $e_0 = e_0 + e$
 $f_0 = f_0 + f$
 $g_0 = g_0 + g$
 $h_0 = h_0 + h$

7. *Output the final hash value:*

$H = a_0 \parallel b_0 \parallel c_0 \parallel d_0 \parallel e_0 \parallel f_0 \parallel g_0 \parallel h_0$

2.4.2 Explanation: -

The pseudo code describes the SHA-256 algorithm, which is a widely used cryptographic hash function. SHA-256 takes an input message of arbitrary length and outputs a fixed-length hash value of 256 bits.

The algorithm consists of several steps, as described below:

- I. **Initialization:** The variables $a_0, b_0, c_0, d_0, e_0, f_0, g_0$, and h_0 are initialized with certain constants.
- II. **Constants:** The SHA-256 algorithm uses a set of 64 constants (K_t) which are used in the computation of the hash value.

- III. **Message schedule**: The message schedule (W_t) is initialized using the input message. The message is divided into 512-bit blocks, and each block is further divided into 16 32-bit words. The message schedule consists of these words, as well as additional words that are computed from the previous words using a set of functions.
- IV. **Working variables**: The variables $a, b, c, d, e, f, g,$ and h are initialized with the values of $a_0, b_0, c_0, d_0, e_0, f_0, g_0,$ and h_0 respectively.
- V. **Computation**: For each of the 64 rounds, a series of operations are performed on the working variables and the message schedule. These operations include logical functions such as AND, OR, and XOR, as well as arithmetic functions such as addition and rotation. The result of each round is used as input for the next round.
- VI. **Update hash value**: After all rounds have been completed, the final hash value is computed by concatenating the values of $a_0, b_0, c_0, d_0, e_0, f_0, g_0,$ and h_0 .

The SHA-256 algorithm is designed to be a one-way function, meaning that it is computationally infeasible to derive the input message from the hash value. It is also resistant to collisions, which occur when two different input messages produce the same hash value. The SHA-256 algorithm is widely used in a variety of applications, including digital signatures, message authentication, and password storage.

2.5.2 **Advantages**:

- SHA-256 produces a fixed-size output of 256 bits, making it resistant to collision attacks.
- It is widely used and recognized as a secure hash function.
- The SHA-256 algorithm is simple to implement and computationally efficient.
- It can be used for digital signatures, password storage, and data integrity checks.

2.5.3 **Disadvantages**:

- SHA-256 is susceptible to length extension attacks, which can compromise the security of digital signatures.
- It can be vulnerable to preimage attacks, where an attacker tries to find an input that produces a specific output.
- Although SHA-256 is computationally efficient, it can still be slower than other hash functions such as MD5 or SHA-1.

2.5.4 **Applications**:

- Password storage: SHA-256 can be used to securely store passwords by hashing them and storing the resulting hash value instead of the actual password.
- Digital signatures: SHA-256 can be used to generate a unique hash of a document or message, which can then be signed with a private key to produce a digital signature.
- Data integrity checks: SHA-256 can be used to verify that data has not been modified or corrupted during transmission by generating a hash of the data and comparing it to a previously calculated hash.

CHAPTER 3: REVERSIBLE IMPLEMENTATION OF FEW IMPORTANT CIRCUITS

3.1 Introduction:

Reversible gates, also known as reversible logic gates, are a class of logic gates that possess the unique property of having an equal number of inputs and outputs. In other words, these gates maintain a one-to-one mapping between their inputs and outputs. This characteristic sets them apart from conventional irreversible gates, which typically result in a loss of information during computation.

The concept of reversibility in logic gates stems from the principle of conserving energy and reducing dissipative losses in computational systems. By ensuring that the number of inputs and outputs are balanced, reversible gates minimize the generation of heat and overall energy dissipation, making them highly desirable in various fields, such as quantum computing and low-power electronic circuit design.

3.2 Classification of Reversible Gates:

Basic reversible logic gates can be classified into three categories: elementary gates, composite gates, and universal gates. Here's an overview of each category:

1) Elementary Gates:

- i) **CNOT (Controlled-NOT) Gate**: Also known as the controlled-X gate, it flips the target bit (output) if and only if the control bit (input) is set to 1.
- ii) **Fredkin (CSWAP) Gate**: Swaps the second and third inputs (bits) if and only if the first input (bit) is set to 1.
- iii) **Toffoli (CCNOT) Gate**: Similar to the CNOT gate, it flips the target bit (output) if and only if both control bits (inputs) are set to 1.

2) Composite Gates:

- i) **Peres Gate**: A composite gate constructed from CNOT and Toffoli gates, implementing an XOR operation between two bits.
- ii) **Feynman Gate**: A composite gate constructed from CNOT gates, implementing an AND operation between two bits.
- iii) **Majority Gate**: A composite gate constructed from CNOT and Toffoli gates, implementing a majority function between three bits.

3) Universal Gates:

- i) **Toffoli (CCNOT) Gate**: As mentioned earlier, the Toffoli gate is universal, meaning it can be used to construct any reversible logic circuit.
- ii) **Fredkin (CSWAP) Gate**: The Fredkin gate is also universal and can be used to build any reversible logic circuit.

These gates form the building blocks of reversible logic circuits, where the outputs can be uniquely inverted back to the inputs. They are fundamental to various applications, such as quantum computing, low-power computing, and information processing.

3.3 Application of Basic Reversible Logic Gates:

Basic reversible logic gates find applications in various fields.

Here are some notable applications:

1. **Quantum Computing**: Reversible logic gates are essential in quantum computing, where the information is processed using quantum bits (qubits). Quantum gates, such as the CNOT gate and Toffoli gate, are used for quantum information processing, quantum algorithms, and quantum error correction.
2. **Low-Power Computing**: Reversible logic gates have the advantage of zero power dissipation when the outputs are inverted back to the inputs. This property makes them suitable for low-power computing applications, such as in portable devices, IoT devices, and energy-constrained environments.
3. **Cryptography**: Reversible logic gates play a role in cryptographic algorithms, such as encryption and decryption. They are used in designing reversible cryptographic circuits for secure information processing.
4. **DNA Computing**: Reversible logic gates have been explored in DNA computing, where DNA molecules are used to perform computations. The reversibility of these gates ensures the preservation of information during DNA-based computations.
5. **Nanotechnology**: Reversible logic gates are significant in nanotechnology for designing efficient and reversible logic circuits at the molecular level. They are used in molecular computing, molecular electronics, and nanoscale information processing.
6. **Error Correction**: Reversible logic gates are employed in error correction techniques, such as the reversible fault-tolerant circuits. These circuits can detect and correct errors that occur during computation, ensuring reliable and accurate results.
7. **Neural Networks**: Reversible logic gates have been studied in the field of neural networks. They are utilized in designing reversible neural network architectures for efficient information processing, memory optimization, and reversible learning algorithms.
8. **Quantum Dot Cellular Automata (QCA)**: Reversible logic gates are used in QCA, a promising nanotechnology-based computing paradigm. QCA leverages the properties of quantum dots to implement reversible logic gates, enabling ultra-fast and energy-efficient computing.

3.4 REVERSIBLE IMPLEMENTATION OF IMPORTANT CIRCUITS: EXPLORING THE POWER OF REVERSIBILITY

i. QUANTUM CIRCUIT WITH A REGISTER OF EIGHT QBITS:

The qubits in a quantum computer are conceptually grouped together in the qubit register. Each qubit has an index within this register, starting at index 0 and counting up by 1.

Examples include the spin of the electron in which the two levels can be taken as spin up and spin down; or the polarization of a single photon in which the two states can be taken to be the vertical polarization and the horizontal polarization. In a classical system, a bit would have to be in one state or the other. So, a system with 8 qubits, for example, has a qubit register that has a width of 8 and is indexed by 0, 1, 2, 3, and 4.

Each qubit can be addressed in qubit operations by using the qubit index. The full quantum state is stored in memory and described in terms of the probability amplitude for each state. As an example, a two-qubit system is described by 4 complex parameters:

$$|\Psi\rangle = \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$$

and a three-qubit system is described by 8 complex parameters:

$$|\Psi\rangle = \alpha_0|000\rangle + \alpha_1|001\rangle + \alpha_2|010\rangle + \dots + \alpha_7|111\rangle$$

TRUTH TABLE:

| Qubit 7 | Qubit 6 | Qubit 5 | Qubit 4 | Qubit 3 | Qubit 2 | Qubit 1 | Qubit 0 | Frequency |
|------------|------------|------------|------------|------------|------------|------------|------------|----------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 ⁸ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 ⁸ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 ⁸ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 ⁸ |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 2 ⁸ |
| 0 | ... | ... | ... | ... | ... | ... | ... | ... |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 2 ⁸ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 ⁸ |

Fig 3.1 Truth Table of Quantum Circuit with A Register of Eight Qubits

WORKING:

- A quantum circuit is created with a register of eight qubits.
- The initial state of all qubits is usually set to the $|0\rangle$ state, unless otherwise specified.
- Quantum gates and operations are applied to manipulate the qubits and perform desired computations. Some commonly used gates include Hadamard (H), Pauli-X (X), Pauli-Y (Y), Pauli-Z (Z), controlled-NOT (CNOT), controlled-phase (CPhase), and more.
- The circuit can perform various operations such as entangling qubits, applying logical operations, performing measurements, and implementing quantum algorithms.
- The order and arrangement of gates matter as they determine the quantum computation being performed.
- The final state of the qubits after all the gate operations can be measured or analyzed to extract information or verify the computation.

QSKIT CODE:

```
from qiskit import QuantumCircuit, execute, Aer# Create a quantum circuit with 8 qubits
circuit = QuantumCircuit(8)
```

```
# Apply Hadamard gate to all qubitsfor qubit in range (8):
circuit.h(qubit)
```

```
# Measure all qubitscircuit.measure_all()
# Execute the circuit on a simulator backend
backend = Aer.get_backend('qasm_simulator')
job = execute (circuit, backend)
result = job.result()
# Print the measurement results
print(result.get_counts(circuit))
```

OUTPUT: -

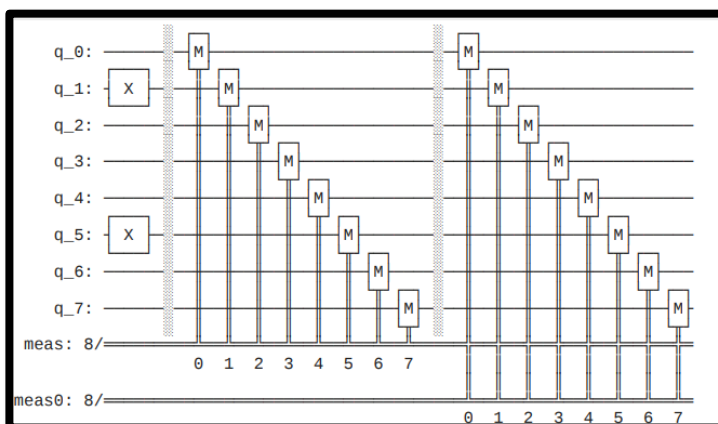


Fig 3.2 Quantum Circuit with A Register of Eight Qubits

EXPLANATION:

- A quantum circuit is created with 8 qubits using the Quantum Circuit class from Qiskit.
- The Hadamard gate (h) is applied to each qubit in a loop using qubit as the iterator variable. This gate puts the qubits in a superposition state.
- The measure_all() method is called to measure all the qubits in the circuit. This prepares the circuit for measurement and collapses the superposition states into classical outcomes.
- The circuit is executed on a simulator backend using the Aer.get_backend('qasm_simulator') function. This simulator backend allows for simulating the quantum circuit and obtaining measurement results.
- The execute function is called with the circuit and the chosen backend to run the simulation and obtain the measurement result.
- The result object is stored in the result variable.
- The measurement results are extracted using result.get_counts(circuit), which returns a dictionary with the counts of each possible outcome.
- Finally, the measurement results are printed to the console using print(result.get_counts(circuit)).

✚ **Advantages** – It is easier to implement and provides a vivid visualization of a simple quantum circuit and forms the basic foundation from which complex circuits can be designed.

✚ **Disadvantages** – As such it does not have any visible disadvantages but at the quantum level, they might be Time Limit Exceed (TLE) errors.

ii. NOT GATE

The simplest Reversible gate is NOT gate and is a 1*1 gate. The Reversible 1*1 gate is a NOT Gate with zero Quantum Cost. It is as shown in the Figure 3 Input Vector = A, Output Vector=A'.

WORKING:

- The NOT gate operates on a single qubit.
- It flips the state of the qubit from $|0\rangle$ to $|1\rangle$ and vice versa.
- Mathematically, the NOT gate is represented by the Pauli-X matrix: $\text{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
- When a qubit is in the state $|0\rangle$ and the NOT gate is applied, it transitions to the state $|1\rangle$.
- Similarly, when a qubit is in the state $|1\rangle$ and the NOT gate is applied, it transitions to the state $|0\rangle$.
- The action of the NOT gate can be visualized as a rotation of the qubit state vector around the Bloch sphere by π radians along the X-axis.
- The NOT gate is reversible, meaning that applying the NOT gate twice brings the qubit back to its original state.
- In quantum circuits, the NOT gate is often represented by the X gate notation in Qiskit or other quantum computing frameworks.
- The NOT gate is a fundamental building block in quantum computing and is used in various quantum algorithms and circuits for manipulating qubit states and performing computations.

TRUTH TABLE:

| Input (A) | Output |
|-----------|--------|
| 0 | 1 |
| 1 | 0 |

Fig 3.3 Truth Table of NOT GATE

QISKIT CODE:

```
from qiskit import QuantumCircuit, Aer, execute

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

# Apply the X gate to the qubit
qc.x(0)

# Measure the qubit
qc.measure_all()

# Execute the circuit on the local simulator
simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, simulator).result()
```

```
# Print the result
print(result.get_counts(qc))
```

OUTPUT:

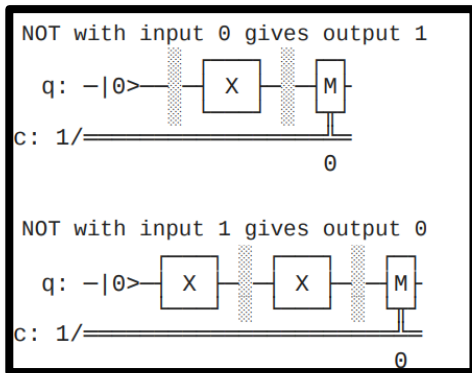


Fig 3.4 Not Gate in Qiskit

EXPLANATION:

- The code creates a quantum circuit with one qubit using Quantum Circuit(1).
- The X gate (NOT gate) is applied to the qubit using qc.x(0), where 0 represents the qubit index.
- The qubit is then measured using qc.measure_all() to obtain the measurement result.
- The circuit is executed on a local simulator backend obtained from Aer.get_backend('qasm_simulator').
- The execute function is used to run the circuit on the simulator and obtain the measurement result in the variable result.
- The measurement result is printed using print(result.get_counts(qc)), which displays the count of different measurement outcomes.

✚ **Advantages** - The Pauli-X gate is the quantum equivalent of the NOT gate for classical computers with respect to the standard basis, which distinguishes the z-axis on the Bloch sphere. It is sometimes called a bit-flip. For this reason, the X-gate is often called the quantum NOT gate as well. In an actual real-world setting, the X-gate generally turns the spin-up state $\lvert 0 \rangle$ of an electron into a spin-down state $\lvert 1 \rangle$ and vice-versa at a very low quantum cost.

✚ **Disadvantage** – Pauli Gates basically implement the theory of linear Algebra, so they might face difficulties while representing circuits with characteristic polynomial $1+x^2+x^3$

iii. CNOT /FEYNMAN GATE

Feynman gate is also known as the controlled-not gate (CNOT). It is a 2*2 reversible gate with a Quantum cost of 1. The CNOT or Feynman gate can be described as Input Vector = (A, B).

Output Vector=A XOR B. The gate acts on two qubits, a control qubit (usually denoted as q0) and a target qubit (usually denoted as q1). If the control qubit is in the state $|0\rangle$, the target qubit is left unchanged. If the control qubit is in the state $|1\rangle$, the target qubit is flipped, or NOT-ed.

WORKING:

The CNOT (Controlled NOT) gate is a fundamental two-qubit gate in quantum computing. It operates on two qubits: a control qubit (C) and a target qubit (T). The gate acts as a conditional NOT gate, where the target qubit T is flipped (or NOT ed) only if the control qubit C is in the state $|1\rangle$. If the control qubit C is in the state $|0\rangle$, then the target qubit T remains in the same state. The CNOT gate is represented by the following matrix:

The matrix form of the CNOT gate is:

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

The action of the CNOT gate can be summarized as follows:

- If the control qubit is in the state $|0\rangle$, the target qubit remains unchanged.
- If the control qubit is in the state $|1\rangle$, the target qubit is flipped (NOTed).

In terms of basis states, the CNOT gate can be represented as follows:

$$\text{CNOT}(|00\rangle) = |00\rangle$$

$$\text{CNOT}(|01\rangle) = |01\rangle$$

$$\text{CNOT}(|10\rangle) = |11\rangle$$

$$\text{CNOT}(|11\rangle) = |10\rangle$$

where the first bit is the control qubit and the second bit is the target qubit.

The CNOT gate is an essential building block for many quantum algorithms, such as the quantum teleportation protocol and the quantum error correction code.

TRUTH TABLE:

| Control | Target 1 | Target 2 | Output 1 | Output 2 |
|---------|----------|----------|----------|----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Fig 3.5 Truth Table for CNOT Gate

QISKIT CODE:

```
from qiskit import QuantumCircuit, Aer, execute

# Create a quantum circuit with 3 qubits and 3 classical bits
qc = QuantumCircuit(3, 3)

# Apply the CNOT gate between qubit 0 and qubit 1
qc.cx(0, 1)

# Apply the CNOT gate between qubit 1 and qubit 2
qc.cx(1, 2)

# Measure the qubits and store the results in classical bits
qc.measure([0, 1, 2], [0, 1, 2])

# Use the local simulator to run the circuit and get the results
simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, simulator).result()

# Print the results
print(result.get_counts(qc))
```

OUTPUT:

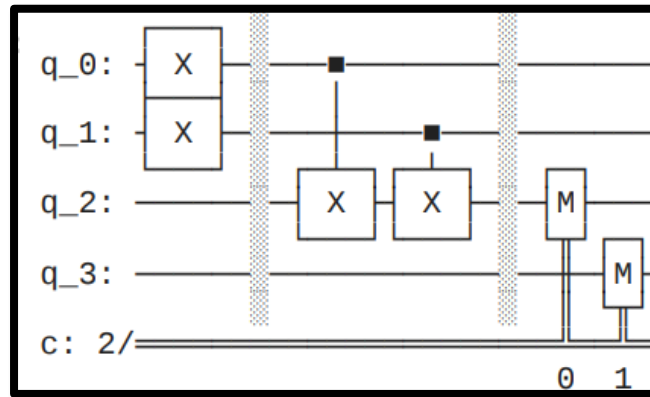


Fig 3.6 CNOT Gate in Qiskit_

EXPLANATION:

- The code creates a quantum circuit with 3 qubits and 3 classical bits using `Quantum Circuit(3, 3)`.
- The CNOT gate is applied between qubit 0 and qubit 1 using `qc.cx (0, 1)`.
- Similarly, the CNOT gate is applied between qubit 1 and qubit 2 using `qc.cx (1, 2)`.
- The qubits are measured, and the measurement results are stored in the classical bits using `qc.measure([0, 1, 2], [0, 1, 2])`.
- The local simulator backend is obtained using `Aer.get_backend('qasm_simulator')`.
- The `execute` function is used to run the circuit on the simulator and obtain the measurement results in the variable `result`.
- The measurement results are printed using `print(result.get_counts(qc))`, which displays the count of different measurement outcomes.

✚ **Advantages** - The CNOT gate can be used to prepare a maximally entangled two- qubit state, or to unambiguously distinguish between all four possible Bell-state inputs: hence, the CNOT gate's importance to quantum computation evidently stems from its effect on superposition-state inputs.

✚ **Disadvantage** - CNOT error rates are higher than error rates of 1-qubit gates, qubits are allocated to positions where errors of required CNOTs are high.

iv. SWAP GATE

The SWAP gate is a two-qubit operation. Expressed in basis states, the SWAP gate swaps the state of the two qubits involved in the operation. Sometimes we need to move information around in a quantum computer. For some qubit implementations, this could be done by physically moving them. Another option is simply to move the state between two qubits. This is done by the SWAP gate.

WORKING:

- The dRWA (double Rydberg wavefunction ansatz) swap gate is a two-qubit gate used in quantum computing. It can be implemented using two CNOT gates and four single-qubit rotations. The gate swaps the wavefunction of two qubits, and it is useful in quantum algorithms such as quantum Fourier transform and quantum phase estimation.

To implement the dRWA swap gate using CNOT gates, we first apply a Hadamard gate to the first qubit, followed by a CNOT gate with the second qubit as the target and the first qubit as the control. We then apply two Ry rotations to the first qubit, followed by another CNOT gate with the first qubit as the target and the second qubit as the control. Finally, we apply two more Ry rotations to the first qubit and a Hadamard gate to the second qubit.

The resulting circuit swaps the wavefunction of the two qubits.

TRUTH TABLE:

| Input Qubits | Output Qubits |
|--------------|---------------|
| 00 | 00 |
| 01 | 10 |
| 10 | 01 |
| 11 | 11 |

Fig 3.7 Truth Table for SWAP Gate

QISKIT CODE:

```
from qiskit import QuantumCircuit, Aer, execute

# create a quantum circuit with two qubits
qc = QuantumCircuit(2)

# apply SWAP gate on qubits 0 and 1

# swap a qubit from b to a
qc.cx(b,a)
# copies 1 from b to a
qc.cx(a,b)
# uses the 1 on a to rotate the state of b to 0
qc.draw()
```

```
# run the circuit on the local simulator
```

```
backend = Aer.get_backend('statevector_simulator')  
result = execute(qc, backend).result()
```

```
# print the final state vector  
print(result.get_statevector())
```

OUTPUT:

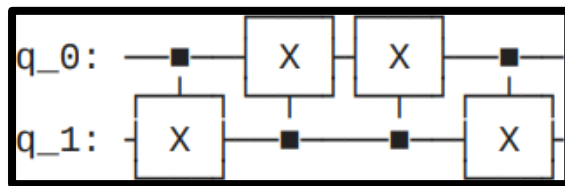


Fig 3.8 SWAP Gate in Qiskit

EXPLANATION: -

- The code creates a quantum circuit with two qubits using Quantum Circuit (2).
- The SWAP gate is applied between qubits 0 and 1. In the code snippet, the SWAP operation is implemented using two CNOT gates: `qc.cx(b, a)` applies a CNOT gate with qubit b as the control and qubit a as the target, which copies the state of qubit b to qubit a. `qc.cx(a, b)` applies a CNOT gate with qubit a as the control and qubit b as the target, which uses the copied state of qubit a to rotate the state of qubit b to the original state of qubit a.
- The circuit is drawn using `qc.draw()`, which visualizes the circuit structure.
- The local simulator backend is obtained using `Aer.get_backend('statevector_simulator')`.
- The circuit is executed on the simulator using `execute(qc, backend).result()`, and the measurement result is stored in the variable `result`.
- The final state vector of the circuit is printed using `result.get_statevector()`, which displays the state vector representation of the final quantum state after applying the gates.

🚦 **Advantages** -The SWAP gate interchanges the input state.

🚦 **Disadvantages** – SWAP Gate sometime faces memory overlapping and memory run-offs. Register Circuit of 8 qubits – In quantum computers, our basic variable is the *qubit*: a quantum variant of the bit. These have exactly the same restrictions as normal bits do: they can store only a single binary piece of information, and can only ever give us an output of 0 or 1.

v. TOFFOLI GATE

TOFFOLI gate which is a 3*3 reversible gate with inputs (A, B, C) and outputs P=A, Q=B, R=AB XOR C. It has Quantum cost five. It is a fundamental gate in quantum computing and is commonly used as a building block for various quantum algorithms and circuits.

WORKING: -

- The Toffoli gate, also known as the Controlled-Controlled-Not (CCX) gate, is a three-qubit gate.
- It performs a bit-flip operation on the target qubit (the third qubit) if and only if both control qubits (the first and second qubits) are in the |1⟩ state.
- The Toffoli gate leaves the state of the control qubits unchanged.
- When the control qubits are in the |0⟩ state, the Toffoli gate acts as an identity gate, leaving the target qubit unchanged.
- The Toffoli gate can be implemented using a combination of two CNOT gates:
- The first CNOT gate has the first control qubit as the control and the target qubit as the target.
- The second CNOT gate has the second control qubit as the control and the target qubit as the target, with the first control qubit as an additional control.
- The Toffoli gate is reversible, meaning it can be undone by applying it again.

$$Toffoli = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

TRUTH TABLE: -

| Control 1 | Control 2 | Target | Output |
|-----------|-----------|--------|--------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Fig 3.9 Truth Table for Toffoli Gate

In the table, the first two columns represent the control qubits (Control 1 and Control 2), the third column represents the target qubit, and the last column represents the output qubit. The gate flips the state of the target qubit if both control qubits are in the state $|1\rangle$, otherwise, the target qubit remains unchanged.

QISKIT CODE: -

```
from qiskit import QuantumCircuit
```

```
# Toffoli Gate
```

```
qc_ha = QuantumCircuit(4,2)
```

```
# Encode inputs in qubits 0 and 1
```

```
qc_ha.x (0)
```

```
# For a=0, remove this line. For a=1, leave it.qc_ha.x(1)
```

```
# For b=0, remove this line. For b=1, leave it.qc_ha.barrier()
```

```
# use cnots to write the XOR of the inputs on qubit 2qc_ha.cx (0,2)
```

```
qc_ha.cx (1,2)
```

```
# use ccx to write the AND of the inputs on qubit 3qc_ha.ccx(0,1,3)
```

```
qc_ha.barrier()
```

```
# extract outputs
```

```
qc_ha.measure(2,0)
```

```
# extract XOR valueqc_ha.measure(3,1)
```

```
# extract AND valueqc_ha.draw()
```

```
# draw the circuitprint(circ.draw())
```

OUTPUT -

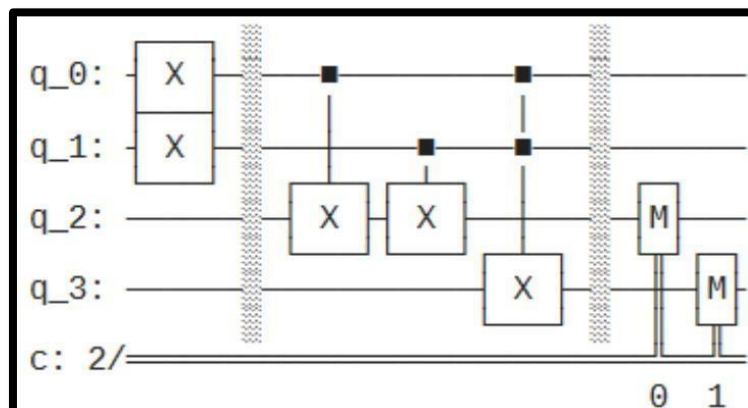




Fig 3.10 Toffoli Gate in Qiskit

EXPLANATION –

- The code creates a quantum circuit qc_ha with 4 qubits and 2 classical bits.
- The inputs a and b are encoded in qubits 0 and 1, respectively. Setting a=1 is done by applying an X gate to qubit 0, and setting b=1 is done by applying an X gate to qubit 1.
- A barrier is added to separate the encoding of the inputs from the gate operations.
- Two CNOT gates are used to write the XOR of the inputs onto qubit 2. The first CNOT gate has qubit 0 as the control and qubit 2 as the target, and the second CNOT gate has qubit 1 as the control and qubit 2 as the target.
- A Toffoli gate (CCX gate) is used to write the AND of the inputs onto qubit 3. The Toffoli gate has qubits 0 and 1 as the control qubits and qubit 3 as the target.
- Another barrier is added to separate the gate operations from the measurement.
- Measurements are performed on qubits 2 and 3, and the results are stored in classical bits 0 and 1, respectively.
- The circuit is then drawn using qc_ha.draw().
- The circuit represents a half-adder, which computes the sum and carry of two input bits. The XOR result is stored in classical bit 0, and the AND result is stored in classical bit 1.

 **Advantages** – The advantage over regular Toffoli gate is their smaller circuit size. However, their use has been often limited to a demonstration of quantum control in designs such as those where the Toffoli gate is being applied last or otherwise for some specific reasons the relative phase does not matter.

 **Disadvantages** – The main disadvantage is the lack of hermicity in the representation of quantum states.

vi. CH- GATE

The Controlled Hadamard (CH) gate is a two-qubit gate in quantum computing. It applies a Hadamard gate (H gate) to the target qubit if and only if the control qubit is in the state $|1\rangle$. Otherwise, it leaves the target qubit unchanged.

TRUTH TABLE:

| C | T | CH(C,T) |
|---|---|---------|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | -1 |

Fig 3.11 Truth Table for CH Gate

WORKING:

- If the control qubit (C) is in the state $|0\rangle$ and the target qubit (T) is in any state, the CH gate has no effect on the target qubit. The state of the target qubit remains the same.
- If the control qubit (C) is in the state $|1\rangle$ and the target qubit (T) is in the state $|0\rangle$, the CH gate applies the Hadamard gate (H gate) to the target qubit. This transforms the target qubit from $|0\rangle$ to the superposition state $(|0\rangle + |1\rangle)/\sqrt{2}$.
- If the control qubit (C) is in the state $|1\rangle$ and the target qubit (T) is in the state $|1\rangle$, the CH gate applies the Hadamard gate (H gate) followed by a phase flip gate (Z gate) to the target qubit. This transforms the target qubit from $|1\rangle$ to the state $-(|0\rangle + |1\rangle)/\sqrt{2}$, which is equivalent to multiplying the state by -1.

QISKIT CODE:

```
from qiskit import QuantumCircuit

# Create a Quantum Circuit with 3 qubits
qc = QuantumCircuit(3)

# Apply the CH gate to qubit 0 controlled by qubit 1
qc.ch(1, 0)
```

```
# Apply the CH gate to qubit 0 controlled by qubit 2
qc.ch (2, 0)

# Print the circuit
print(qc.draw())
```

OUTPUT-

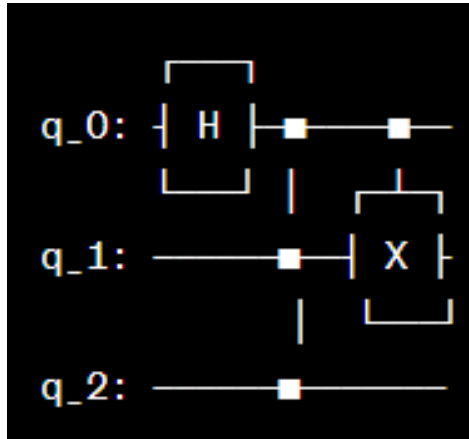


Fig 3.12 CH Gate in Qiskit

EXPLANATION:-

- from qiskit import QuantumCircuit: This line imports the necessary classes from the Qiskit library, including the QuantumCircuit class for creating quantum circuits.
- qc = QuantumCircuit(3): This line creates a QuantumCircuit object named qc with 3 qubits. The circuit is initially empty.
- qc.ch(1, 0): This line applies the CH gate to qubit 0, controlled by qubit 1. The ch() method is used to add the CH gate to the circuit.
- qc.ch(2, 0): This line applies another CH gate to qubit 0, controlled by qubit 2. This gate is independent of the previous one and is added sequentially to the circuit.
- print(qc.draw()): This line prints the visual representation of the quantum circuit using the draw() method. The output will show the CH gates applied to the corresponding qubits.
- The resulting circuit will have the CH gate applied to qubit 0, controlled by qubits 1 and 2, as shown by the gate symbols in the visual representation.

vii. MCMT Gate

"MCMT" stands for "Multi-Controlled Multi-Target" gate, which is a general term used to describe a gate that operates on multiple control qubits and multiple target qubits.

In quantum computing, a typical multi-controlled gate, such as the Toffoli gate (CCNOT), involves multiple control qubits and a single target qubit. The Toffoli gate performs a controlled-NOT operation on the target qubit if and only if all the control qubits are in the state $|1\rangle$. Similarly, a multi-controlled T gate (Toffoli gate with a phase factor) is often referred to as the MCMT gate.

WORKING: -

- The MCMT gate is a generalization of the Controlled gate, allowing for multiple control qubits and multiple target qubits.
- It performs a quantum operation on the target qubits based on the states of the control qubits.
- The gate applies the specified operation only when all the control qubits are in the desired state.
- If the control qubits are not in the desired state, the gate leaves the target qubits unchanged.
- The specific quantum operation performed by the MCMT gate depends on the gate being implemented. Examples of MCMT gates include the Toffoli gate (CCNOT) and the Fredkin gate (CSWAP).
- The MCMT gate can be implemented using a combination of single-qubit and two-qubit gates, such as CNOT gates and Toffoli gates, depending on the desired operation.
- The working of the MCMT gate involves applying the gate operation to the target qubits if and only if all the control qubits are in the specified state.
- The MCMT gate is commonly used in quantum algorithms and quantum error correction to perform complex operations on multiple qubits based on their control states.
- The gate can be represented using a quantum circuit, where the control qubits are connected to the target qubits through appropriate gates, ensuring the desired operation is applied when the control conditions are met.
- The MCMT gate plays a crucial role in implementing various quantum algorithms, such as quantum arithmetic, quantum search algorithms, and quantum error correction codes.

QISKIT CODE

```
from qiskit import QuantumCircuit
```

```
def mcmt_gate(circuit, controls, targets):  
    num_controls = len(controls)  
    num_targets = len(targets)
```

```

# Apply X gate to the target qubits
for target in targets:
    circuit.x(target)

# Apply controlled-X gate with multiple controls
circuit.mcx(controls, targets)

# Apply X gate to the target qubits again
for target in targets:
    circuit.x(target)

# Create a Quantum Circuit with 3 qubits
qc = QuantumCircuit(5)

# Define the control qubits
control_qubits = [0, 1]

# Define the target qubits
target_qubits = [2, 3, 4]

# Apply the MCMT gate
mcmt_gate(qc, control_qubits, target_qubits)

# Print the circuit
print(qc.draw())

```

EXPLANATION –

- The provided code implements the MCMT (Multi-Control Multi-Target) gate in Qiskit.
- It defines a function called `mcmt_gate` that takes three parameters: `circuit` (the quantum circuit to apply the gate to), `controls` (a list of control qubits), and `targets` (a list of target qubits).
- The function first determines the number of control qubits (`num_controls`) and the number of target qubits (`num_targets`).
- It applies an X gate to each target qubit to prepare them in the $|1\rangle$ state.
- Then, it applies the multi-controlled X gate (`mcx`) with the specified control qubits (`controls`) and target qubits (`targets`).
- Finally, it applies an X gate to each target qubit again to restore their original states.
- The code creates a 5-qubit quantum circuit, defines control qubits `[0, 1]` and target qubits `[2, 3, 4]`, and applies the MCMT gate using the `mcmt_gate` function. It then prints the circuit diagram.

viii. PERES GATE

Peres gate which is a 3*3 reversible gate having inputs (A, B, C) and outputs $P = A$; $Q = A \text{ XOR } B$; $R = AB \text{ XOR } C$. It has Quantum cost four. The Peres gate is a two-qubit quantum gate that is named after its inventor, Asher Peres. It is also known as the SWAP-test gate or the entanglement test gate. The Peres gate is used to test whether two qubits are entangled or not.

The Peres gate takes two qubits as input, say qubit A and qubit B. It performs a controlled-NOT (CNOT) operation, with qubit A as the control qubit and qubit B as the target qubit. Then, it applies a Hadamard gate to qubit A and another CNOT gate, with qubit B as the control qubit and qubit A as the target qubit. Finally, it applies another Hadamard gate to qubit A. The resulting state of the two qubits can be measured to determine whether they are entangled or not.

WORKING:

- Initialize the two qubits A and B to the state $|00\rangle$.
- Apply a CNOT gate with qubit A as the control qubit and qubit B as the target qubit. This operation entangles the two qubits, so that their state becomes: $|00\rangle \rightarrow |00\rangle + |11\rangle$
- Apply a Hadamard gate to qubit A, which puts it into a superposition of $|0\rangle$ and $|1\rangle$. $|00\rangle + |11\rangle \rightarrow (|0\rangle + |1\rangle) \otimes |1\rangle$
- Apply another CNOT gate with qubit B as the control qubit and qubit A as the target qubit. This operation depends on whether the two qubits are entangled or not. If the two qubits are not entangled, then the resulting state is: $(|0\rangle + |1\rangle) \otimes |1\rangle \rightarrow |01\rangle$
- If the two qubits are entangled, then the resulting state is: $(|0\rangle + |1\rangle) \otimes |1\rangle \rightarrow (|00\rangle + |11\rangle) / \sqrt{2}$
- Apply another Hadamard gate to qubit A, which brings it back to its original state. $(|00\rangle + |11\rangle) / \sqrt{2} \rightarrow |00\rangle + |11\rangle$
- Measure the two qubits. If the measurement result is $|00\rangle$ or $|11\rangle$, then the two qubits are entangled. If the measurement result is $|01\rangle$ or $|10\rangle$, then the two qubits are not entangled.

TRUTH TABLE:

| Control 1 | Control 2 | Target | Output |
|-----------|-----------|--------|--------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Fig 3.13 Truth Table for Peres Gate

In the table, the first two columns represent the control qubits (Control 1 and Control 2), the third column represents the target qubit, and the last column represents the output qubit. The gate flips the state of the target qubit if all three qubits are in the special W state, otherwise, the target qubit remains unchanged.

QISKIT CODE:

```
from qiskit import QuantumCircuit

# Create a 3-qubit quantum circuit
qc = QuantumCircuit(3)

# Apply CNOT gates to create a Toffoli gate
qc.cx(0, 1)
qc.cx(1, 2)
qc.cx(0, 1)
qc.cx(1, 2)

# Apply Hadamard gates to the first two qubits
qc.h([0, 1])

# Apply Toffoli gate to the first three qubits
qc.ccx(0, 1, 2)

# Apply Hadamard gates to the first two qubits
qc.h([0, 1])

# Apply X gates to the second and third qubits
qc.x([1, 2])

# Apply controlled Z gate to the first and second qubits
qc.cz(0, 1)

# Apply X gates to the second and third qubits
qc.x([1, 2])

# Draw the circuit
qc.draw()
```

OUTPUT:

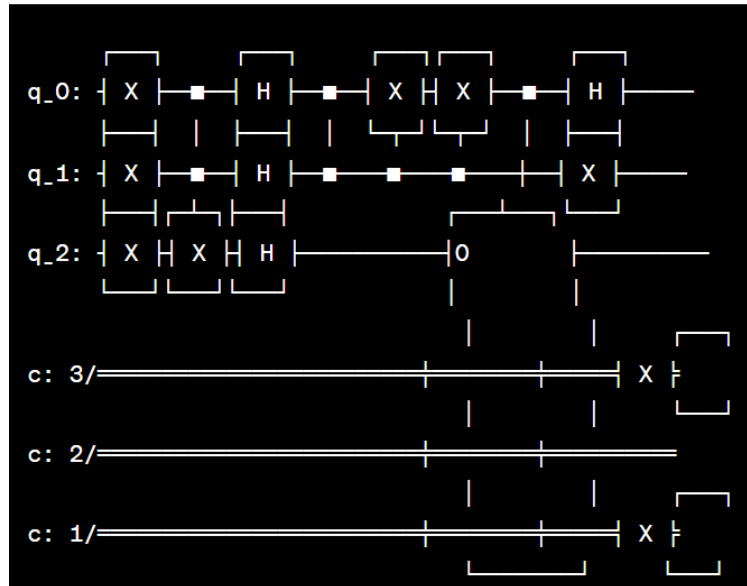


Fig 3.14 Peres Gate in Qiskit

EXPLANATION:

- The code creates a 3-qubit quantum circuit using Qiskit.
- It applies CNOT gates to create a Toffoli gate, which is a three-qubit gate that performs a controlled-NOT operation.
- Hadamard gates are applied to the first two qubits to create superposition states.
- A Toffoli gate is applied to the first three qubits using the `ccx` function in Qiskit.
- Additional Hadamard gates are applied to the first two qubits to reverse the superposition.
- X gates are applied to the second and third qubits to flip their states.
- A controlled-Z gate is applied to the first and second qubits.
- X gates are applied again to the second and third qubits to revert their states.
- Finally, the circuit is drawn to visualize its structure using the `qc.draw()` function.

✚ **Advantages** – Peres gate is a basic reversible logic gate used in various reversible circuit. The various applications of Peres gate such as full-adder circuit and flip-flop.

✚ **Disadvantages** – As such it does not have any visible disadvantages but at quantum level they might be time limit exceed (TLE) errors.

ix. FREDKIN GATE

Fredkin gate is a 3*3 reversible gate with inputs (A, B, C) and outputs $P=A$, $Q=A'B+AC$, $R=AB+A'C$. It has Quantum cost five. The Fredkin gate, also known as the Controlled-SWAP gate or the CSWAP gate, is a three-qubit gate in quantum computing. It performs a conditional swap operation on its input qubits based on the state of the control qubit.

TRUTH TABLE:

| Input A | Input B | Control | Output A | Output B |
|---------|---------|---------|----------|----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Fig 3.15 Truth Table for Fredkin Gate

QISKIT CODE:

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

```
# Create a quantum circuit with three qubits
```

```
qr = QuantumRegister(3, 'q')
```

```
circuit = QuantumCircuit(qr)
```

```
# Apply Fredkin gate
```

```
circuit.cswap(qr[0], qr[1], qr[2])
```

```
# Draw the circuit
```

```
circuit.draw('mpl')
```


OUTPUT:

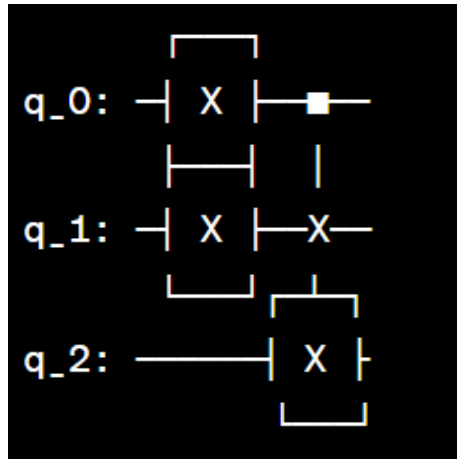


Fig 3.16 Fredkin Gate in Qiskit

EXPLANATION:

- The code imports the necessary modules from Qiskit to create and manipulate quantum circuits.
- A QuantumRegister named qr is created with 3 qubits.
- A QuantumCircuit named circuit is created using the qr register.
- The Fredkin gate (also known as the Controlled-SWAP or CSWAP gate) is applied to the circuit. It is a three-qubit gate that swaps the states of qubits 1 and 2 if and only if qubit 0 is in the state $|1\rangle$.
- The circuit is drawn using the 'mpl' (Matplotlib) backend to visualize it.
- The circuit represents the application of the Fredkin gate, which is a controlled version of the SWAP gate. It swaps the states of qubits 1 and 2 if and only if qubit 0 is in the state $|1\rangle$.

✚ **Advantages** – Fredkin gate is reversible and conservative in nature, that is, it has unique input and output mapping and also has the same number of 1s in the outputs as in the inputs. Fredkin gate is also called a controlled swap gate as it can swap two input bits i_0, i_1 when $c_0 = 1$. Implementing the reversible logic has the advantages of reducing gate counts, garbage outputs as well as constant inputs.

✚ **Disadvantages** - The major drawback of the proposed EA MRR-based Fredkin configuration is that the control signal is in the electrical domain, which complicates the performance as a universal gate, in which the control signal is used as one of the operands (as in the case of the two-input AND and OR gates based on the Fredkin Gate).

x. SAM GATE

SAM gate is a 3*3 reversible gate with inputs A, B, C and outputs $P=A'$, $Q=A'B \text{ XOR } AC'$, $R=A'C \text{ XOR } AB$. The quantum cost of SAM gate is 4. The SAM (Square-Root of Anti-Phase-Mixed) gate is a single-qubit gate that was introduced to quantum computing by Maslov and Dueck in 2008. The gate is designed to be universal, meaning that it can be used to perform any single-qubit operation.

The SAM gate is defined as:

$$\text{SAM}(\theta, \phi) = \text{Rz}(\theta/2) * \text{Rx}(\pi/2) * \text{Rz}(\phi) * \text{Rx}(-\pi/2)$$

where **Rz** and **Rx** are the rotation gates around the z and x axes, respectively, and **theta** and **phi** are real-valued parameters.

The SAM gate is composed of a sequence of rotations that can be visualized as follows:

Rx(-pi/2) - rotate the qubit by -pi/2 around the x-axis

Rz(phi) - rotate the qubit by phi around the z-axis

Rx(pi/2) - rotate the qubit by pi/2 around the x-axis

Rz(theta/2) - rotate the qubit by theta/2 around the z-axis

The SAM gate can be used to perform any single-qubit operation by appropriately choosing the values of **theta** and **phi**. In particular, the **theta** parameter can be used to rotate the qubit around the z-axis, while the **phi** parameter can be used to rotate the qubit around the x-axis.

The SAM gate is useful in quantum computing because it is universal, meaning that any single-qubit operation can be approximated to arbitrary accuracy using a sequence of SAM gates.

TRUTH TABLE:

| Input A | Input B | Input C | Output A | Output B | Output C |
|---------|---------|---------|----------|----------|----------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Fig 3.17 Truth Table for SAM Gate

QISKIT CODE:

```
from qiskit import QuantumCircuit, QuantumRegister
```

```
def sam_gate(qc, a, b, c, d, e, f):  
    # define ancilla and auxiliary qubits  
    ancilla = QuantumRegister(2, name='ancilla')  
    aux = QuantumRegister(2, name='aux')
```

```
    # add ancilla and auxiliary qubits to the circuit  
    qc.add_register(ancilla, aux)
```

```
    # apply the SAM gate  
    qc.cx(a, ancilla[0])  
    qc.cx(b, ancilla[0])  
    qc.toffoli(a, b, ancilla[1])  
    qc.cx(c, aux[0])  
    qc.cx(d, aux[0])  
    qc.toffoli(c, d, aux[1])  
    qc.toffoli(ancilla[0], aux[0], e)  
    qc.toffoli(ancilla[1], aux[1], f)  
    qc.toffoli(ancilla[1], aux[0], f)  
    qc.toffoli(ancilla[0], aux[1], e)
```

```
    # create a 6-qubit quantum circuit  
    qc = QuantumCircuit(6)
```

```
    # apply the SAM gate  
    sam_gate(qc, 0, 1, 2, 3, 4, 5)
```

```
    # draw the circuit  
    qc.draw()
```

OUTPUT:



Fig 3.18 SAM Gate in Qiskit

EXPLANATION:

- The "sam_gate" function takes a quantum circuit (qc) and six qubit indices (a, b, c, d, e, f) as inputs.
- It defines two additional quantum registers: an ancilla register with 2 qubits and an auxiliary register with 2 qubits.
- The ancilla and auxiliary registers are added to the circuit using the qc.add_register() method.
- The code applies a series of quantum gates to implement the SAM gate:
 - Controlled-X gates (cx) are applied between qubits a and the first qubit of the ancilla register, and between qubits b and the first qubit of the ancilla register.
 - A Toffoli gate is applied between qubits a, b, and the second qubit of the ancilla register.
 - Controlled-X gates are applied between qubits c and the first qubit of the auxiliary register, and between qubits d and the first qubit of the auxiliary register.
 - A Toffoli gate is applied between qubits c, d, and the second qubit of the auxiliary register.
 - A series of Toffoli gates and controlled-X gates are applied to implement the desired computation involving the ancilla, auxiliary, and target qubits.
- The final circuit is drawn using the qc.draw() method.

✚ **Advantages** – SAM gate is a basic reversible logic gate used in various reversible circuit. The various applications of SAM gate such as full-adder circuit and flip-flop.

✚ **Disadvantages** – As such it does not have any visible disadvantages but at quantum level they might be time limit exceed (TLE) errors.

xi. DOUBLE FEYNMAN GATE

The double Feynman gate is 3*3 reversible gate having inputs (A, B, C) and outputs $P=A$, $Q=A \text{ XOR } B$, $R=A \text{ XOR } C$ with a quantum cost of 2.

The double Feynman can be described as:

- The output bit P is simply equal to input bit A.
- Output bit Q is the result of an XOR gate between input bits A and B.
- Output bit R is the result of an XOR gate between input bits A and C.

TRUTH TABLE:

| A | B | C | P | Q | R |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Fig 3.19 Truth Table for Double Feynman Gate

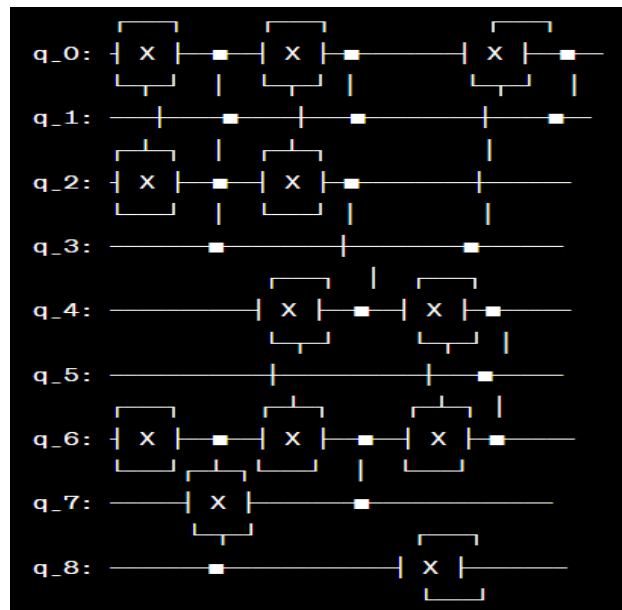
QISKIT CODE:

```
from qiskit import QuantumCircuit

# create a 9-qubit quantum circuitqc = QuantumCircuit(9)

# apply the 3x3 Double Feynman gate
qc.cx(0, 3)
qc.cx(1, 4)
qc.cx(2, 5)
qc.toffoli(0, 1, 6)
qc.toffoli(3, 4, 7)
qc.toffoli(6, 7, 8)
qc.toffoli(2, 1, 6)
qc.toffoli(5, 4, 7)
```

```
# draw the circuit
qc.draw()
```



- CNOT gates are applied to qubits 0-2 and 3-5, with the output qubits connected to qubits 6-8 respectively.
- Toffoli gates are used to compute the output qubits Q and R.
- The Toffoli gates used in the previous step are repeated in reverse order to obtain the final outputs.
- The circuit is then drawn using the draw method.
- Overall, this code demonstrates how to apply the Double Feynman gate in Qiskit and how to create and draw a quantum circuit in Python.

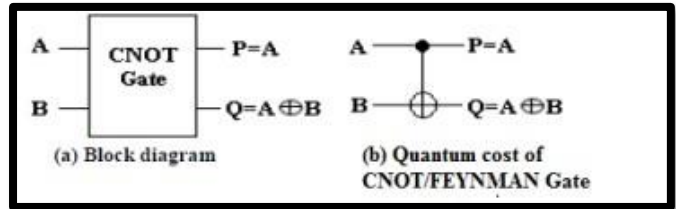
✚ **Advantage** – Double Feynman Gate is a basic reversible logic gate used in various reversible circuits. The various applications of SAM gate such as full-adder circuit and flip-flop.

✚ **Disadvantages** – As such it does not have any visible disadvantages but at quantum level, they might be time limit exceed (TLE) errors.

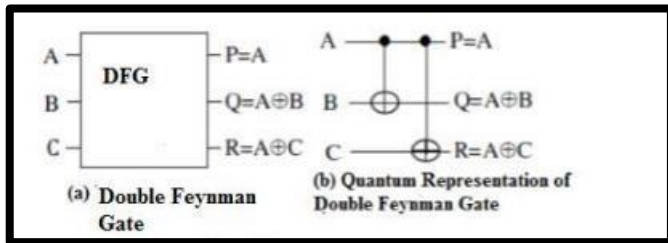
BLOCK DIAGRAM OF ALL THE REVERSIBLE CIRCUITS



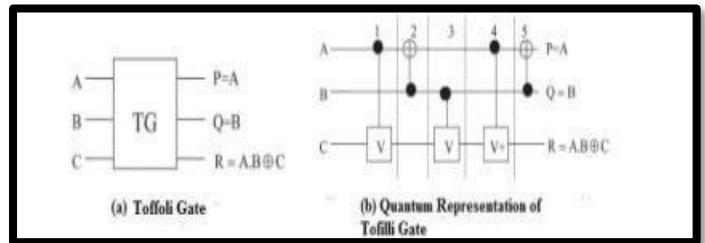
NOT GATE



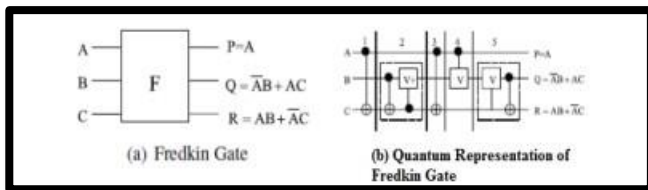
FEYNMAN GATE



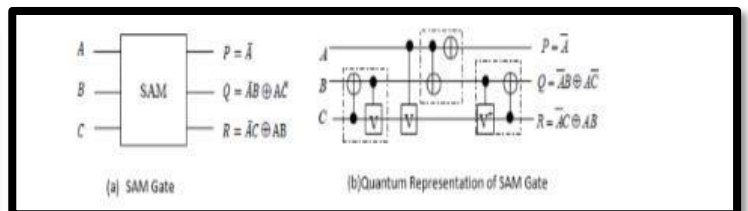
DOUBLE FEYNMANN GATE



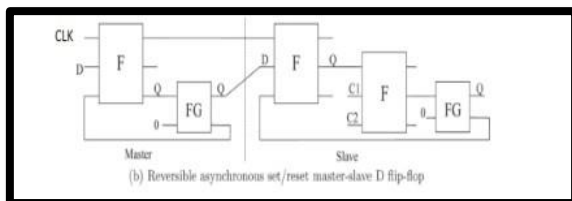
TOFFOLI GATE



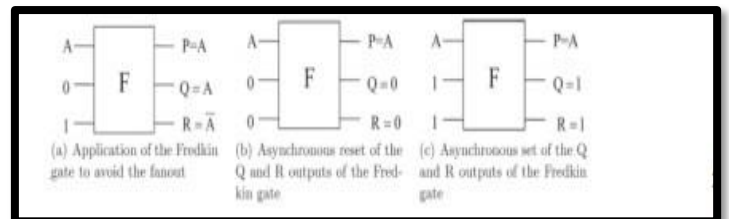
FREDKIN GATE



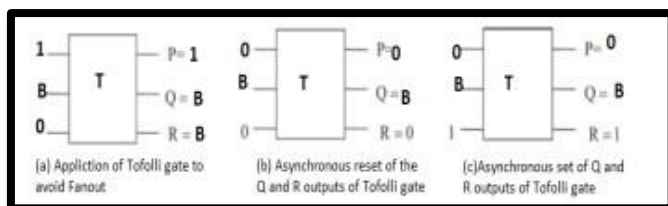
SAM GATE



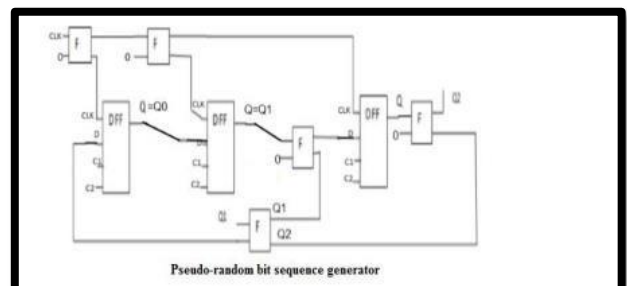
MASTER SLAVE D FLIPFLOP



APPLICATION OF FREDKIN GATE



APPLICATION OF TOFOLLI GATE



PN SEQUENCE GENERATOR

Fig 3.21 Reversible Circuits in Qiskit

CHAPTER 4: EXPLORING ORIGINAL SHA-256

4.1 Introduction:

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function designed to produce a fixed-size, 256-bit (32-byte) hash value from an arbitrary amount of input data. It is part of the SHA-2 (Secure Hash Algorithm 2) family, which was designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. SHA-256 is widely regarded for its robustness and security, making it a cornerstone of modern cryptographic practices.

SHA-256 is a fundamental cryptographic hash function known for its security and efficiency. It plays a crucial role in ensuring data integrity, authentication, and security in various applications and protocols. Despite the emergence of more advanced cryptographic techniques, SHA-256 remains a trusted and widely used hash function in the digital world.

4.2 Original SHA-256:

In recent years with the growing network, communication between people has become very easy but along with that, it also matters how secure the communication is, because of all the important and confidential data that are being exchanged over the network. Cryptography is the practice and study of techniques for secured communication in presence of the third parties called adversaries. Modern cryptography is broadly classified into three categories symmetric cryptography, asymmetric cryptography, and hash functions. Symmetric-key cryptography refers to encryption methods in which both the sender and receiver will have the same key. Asymmetric or public key cryptography is a type of cryptography in which two different keys (private key and public key) are used for sender and receiver. The public key is used for encryption, while the private or secret key is used for decryption. Hash functions are the algorithms which do not use any key for encryption but a fixed-length hash value is computed based upon the plain text that makes it impossible to recover data from it. They are used to provide the digital fingerprint of a file's content. They have very special properties like pre-image resistance, collision resistance and second pre-image resistance which make them harder to crack even by means of brute-force attack. The MD5 algorithm is a widely used hash function producing a 128-bit hash value. Message digest (MD5) was designed by Ronald Rivest in 1991 to replace an earlier hash function MD4. Secure hash algorithms (SHA) are one of the best examples of cryptographic hash function with major applications in the field of security. SHA-1 is the hash function which takes an input and produces 160-bit hash value as output. It uses similar structure as that of MD4 and MD5. SHA-2 includes significant changes compared to SHA-1. It consists of a family of six hash functions that are with different output bit lengths and they are 224, 256, 384, 512, 512/224 and 512/256 bit. SHA-256 is computed with 32-bit word and SHA-512 is computed with 64-bit word. They both have a similar structure with only difference in shift amounts, additive constants and number of rounds. SHA-224 and SHA-384 are simply truncated versions of the first two, respectively, computed with different initial values. SHA-256 based on MD5 structure with input length less than 264 bit is the widely used hash function as it has enough high output length for most of the practical applications. SHA-256 algorithm has major steps like padding, word expansion and main loop which will run for 64 times with initial hash values as input. It involves 64 constants each of 32 bit derived from the fractional parts of the cube roots of the first sixty-four prime numbers. Each time after main loop runs, there is swapping of registers involved in the round and after the completion of last round, it is added with initial hash values to get the final output hash. SHA-2 hash functions are widely used in protocols like TLS, SSL, and PGP. Currently, the best public attacks break pre-image resistance for 52 out of 64 rounds of SHA-256 or 57 out of 80 rounds of SHA-512, and collision resistance for 46 out of 64 rounds of SHA-256. SHA-3 is the new member of the SHA family

released by NIST in 2015. SHA-3 is based on the cryptographic family KECCAK, which is based on a novel approach called sponge construction. In the recent years, due to their ability to reduce the power dissipation, reversible logic has received attention in the field of VLSI and research. Reversible logic supports the unique way of generation of input from the output which is one-to-one mapping. The actual idea behind this logic is energy dissipation can be reduced if the computation becomes information lossless. Reversible logic has found a major application in cryptography because this field needs a method to address and reduce the power consumed while carrying out such many operations in loop.

4.3 BLOCK DIAGRAM:

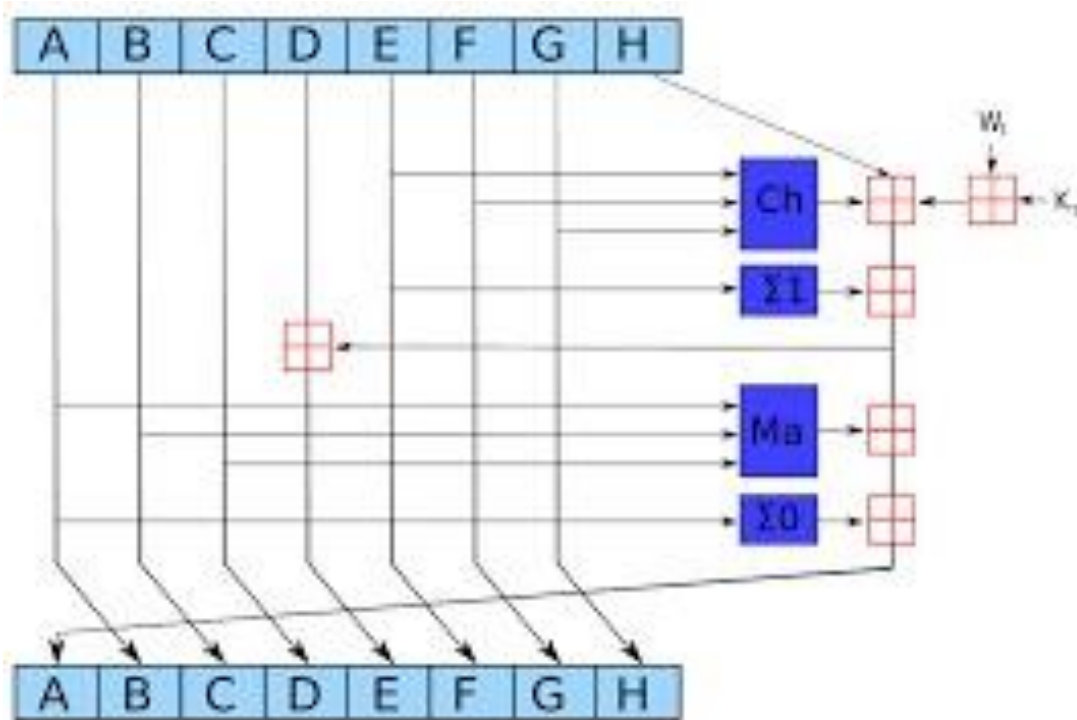


Fig 4.1 Block Diagram of SHA 256

4.4 Pseudo Code for SHA-256:

Step 1: Initialize Hash Values

The hash values (H0 to H7) are initialized to the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. These values are constants specified in the SHA-256 standard.

Step 2: Initialize Round Constants

The array of round constants (K) consists of 64 values, each being the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers. These constants are also specified in the SHA-256 standard.

Step 3: Pre-processing (Padding)

Original Message Length: The length of the original message is calculated in bits.

Padding: A single '1' bit is appended to the message, followed by '0' bits until the length of the message (in bits) is congruent to 448 modulo 512. This ensures the final message length (including padding) is a multiple of 512 bits but leaves 64 bits free.

Append Length: The 64-bit big-endian representation of the original message length is appended to the message. This finalizes the padding process.

Step 4: Process the Message in 512-bit Chunks

The padded message is processed in 512-bit chunks.

Step 4.1: Break Chunk into 32-bit Words

Each 512-bit chunk is divided into sixteen 32-bit words (W[0] to W[15]).

Step 4.2: Extend Words

The first 16 words are extended into 64 words (W[0] to W[63]). Each word from W[16] to W[63] is computed based on a combination of previous words

4.5 QISKIT CODE:

```
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```

```
def generate_hash(message: bytearray) -> bytearray:
```

```
    """Return a SHA-256 hash from the message passed.
```

```
    The argument should be a bytes, bytearray, or
    string object."""
```

```
    if isinstance(message, str):
```

```
        message = bytearray(message, 'ascii')
```

```
    elif isinstance(message, bytes):
```

```
        message = bytearray(message)
```

```
    elif not isinstance(message, bytearray):
```

```
        raise TypeError
```

```
    # Padding
```

```
    length = len(message) * 8 # len(message) is number of BYTES!!!
```

```
    message.append(0x80)
```

```
    while (len(message) * 8 + 64) % 512 != 0:
```

```
        message.append(0x00)
```

```
    message += length.to_bytes(8, 'big') # pad to 8 bytes or 64 bits
```

```
    assert (len(message) * 8) % 512 == 0, "Padding did not complete properly!"
```

```
    # Parsing
```

```
    blocks = [] # contains 512-bit chunks of message
```

```
    for i in range(0, len(message), 64): # 64 bytes is 512 bits
```

```
        blocks.append(message[i:i+64])
```

```
    # Setting Initial Hash Value
```

```
    h0 = 0x6a09e667
```

```
    h1 = 0xbb67ae85
```

```
    h2 = 0x3c6ef372
```

```
    h3 = 0xa54ff53a
```

```
    h5 = 0x9b05688c
```

```
    h4 = 0x510e527f
```

```
    h6 = 0x1f83d9ab
```

```
    h7 = 0x5be0cd19
```

```
    # SHA-256 Hash Computation
```

```
    for message_block in blocks:
```

```
        # Prepare message schedule
```

```
        message_schedule = []
```

```

for t in range(0, 64):
    if t <= 15:
        # adds the t'th 32 bit word of the block,
        # starting from leftmost word
        # 4 bytes at a time
        message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
    else:
        term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
        term2 = int.from_bytes(message_schedule[t-7], 'big')
        term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
        term4 = int.from_bytes(message_schedule[t-16], 'big')

        # append a 4-byte byte object
        schedule = ((term1 + term2 + term3 + term4) % 2**32).to_bytes(4, 'big')
        message_schedule.append(schedule)

assert len(message_schedule) == 64

# Initialize working variables
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7

# Iterate for t=0 to 63
for t in range(64):
    t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
           int.from_bytes(message_schedule[t], 'big')) % 2**32)

    t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32

    h = g
    g = f
    f = e
    e = (d + t1) % 2**32
    d = c
    c = b
    b = a
    a = (t1 + t2) % 2**32

# Compute intermediate hash value
h0 = (h0 + a) % 2**32
h1 = (h1 + b) % 2**32
h2 = (h2 + c) % 2**32
h3 = (h3 + d) % 2**32
h4 = (h4 + e) % 2**32
h5 = (h5 + f) % 2**32
h6 = (h6 + g) % 2**32
h7 = (h7 + h) % 2**32

```

```

return ((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
        (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
        (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
        (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big'))

def _sigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 7) ^
           _rotate_right(num, 18) ^
           (num >> 3))
    return num

def _sigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 17) ^
           _rotate_right(num, 19) ^
           (num >> 10))
    return num

def _capsigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 2) ^
           _rotate_right(num, 13) ^
           _rotate_right(num, 22))
    return num

def _capsigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 6) ^
           _rotate_right(num, 11) ^
           _rotate_right(num, 25))
    return num

def _ch(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (~x & z)

def _maj(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (x & z) ^ (y & z)

def _rotate_right(num: int, shift: int, size: int = 32):
    """Rotate an integer right."""
    return (num >> shift) | (num << size - shift)

if __name__ == "__main__":
    hexadecimal_input=generate_hash("Hello").hex()
    print(hexadecimal_input)

# Convert hexadecimal to binary using built-in function bin()
binary_output = bin(int(hexadecimal_input, 16))[2:] # [2:] to remove the '0b' prefix
# Print the binary equivalent
print("Binary equivalent:", binary_output)

```

INPUT:-

Hello

OUTPUT:-

```
11000010111111000110110110011001000100111000111111110001001011111010101100
00110100110111111001001001110001011001011100010011001000011000001101110110
00011000001001110110110100101000110000000000001111101000101110110010010000
0100110001110000001100101101001
```

CHAPTER 5: *EXPLORING DYNAMIC SHA-256*

5.1 Introduction:

The algorithm SHA-256 has initial hash values fixed (32 bits) which are derived from the square root of the first eight prime numbers and there are no specific reasons for the selection of such fixed values. So, in order to make it more dynamic, it is proposed to design the hash values as a function of input text message instead of those directly chosen fixed values. The proposed algorithm can be added as module to the initial pre-processing stage of the current SHA-256 implementation leaving less changes in the rest of the rounds. Keeping hardware, power requirements and cost of implementation in mind, we have modified initial hash values based on the incoming message and reduced further rounds (if standard implementation has r rounds) in main loop to $(r-16)$ to maintain the complexity of the hashing and security level as that of SHA standard implementation. Flow chart of the proposed algorithm is given in Figure.

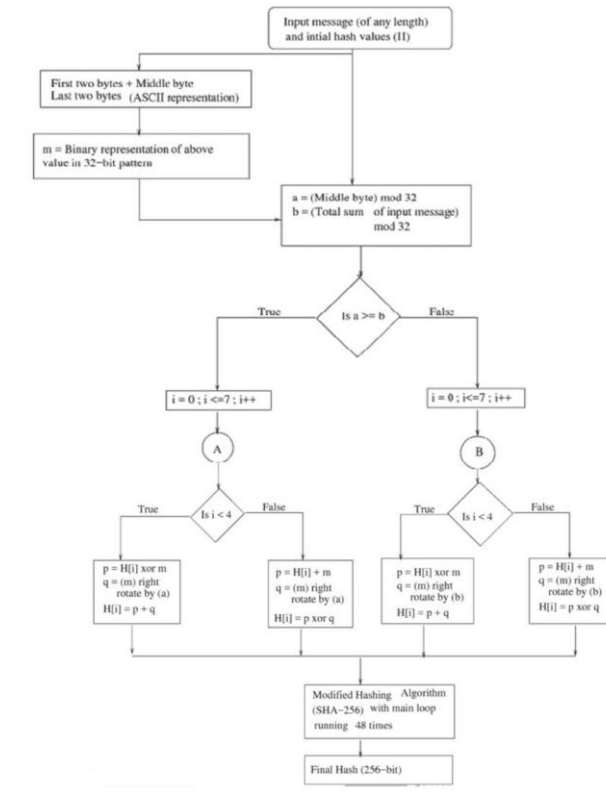


Fig 5.1 Block Diagram of Dynamic SHA 256

5.2 Proposed Algorithm:

1. Calculate two values:
 - $A = (\text{Middle byte of message}) \bmod 32$
 - $B = (\text{Total sum of the message}) \bmod 32$
2. Update the initial hash values ($H(0)_1$ - $H(0)_8$) based on four conditions:
 - a. If $(A \geq B)$ and $(i < 5)$, then $H(0)_i = ((H(0)_i \text{ XOR } m) + (m \text{ rotate right by } A))$
 - b. If $(A \geq B)$ and $(i \geq 5)$, then $H(0)_i = ((H(0)_i + m) \text{ XOR } (m \text{ rotate right by } A))$
 - c. If $(A < B)$ and $(i < 5)$, then $H(0)_i = ((H(0)_i \text{ XOR } m) + (m \text{ rotate right by } B))$
 - d. If $(A < B)$ and $(i \geq 5)$, then $H(0)_i = ((H(0)_i + m) \text{ XOR } (m \text{ rotate right by } B))$
3. Reduce the number of rounds in the main loop of the SHA algorithm to $(r-16)$, i.e., for SHA-256, reduce to 48 rounds.
4. Feed the updated hash values into the main loop of the SHA algorithm.

5.3 Pseudo Code for Dynamic SHA-256:

```
function dynamicSHA256(input_data):
    # Step 1: Derive dynamic IVs
    IVs = deriveDynamicIV(input_data)

    # Step 2: Derive dynamic constants
    constants = deriveDynamicConstants(input_data)

    # Step 3: Determine number of rounds
    num_rounds = determineNumberOfRounds(input_data)

    # Step 4: Split input data into blocks
    blocks = splitInput(input_data, 64) # Assuming 512-bit (64-byte) blocks

    # Step 5: Initialize hash value
    hash_value = IVs

    # Step 6: Process each block
    for block in blocks:
        # Dynamic message scheduling
        message_schedule = dynamicMessageSchedule(block)

        # Initialize working variables with current hash value
        a, b, c, d, e, f, g, h = hash_value

        # Compression function main loop
        for i from 0 to num_rounds - 1:
            # Example dynamic round constant for this iteration
            K = constants[i % len(constants)]

            # Perform the main SHA-256 compression steps
            T1 = h + Sigma1(e) + Ch(e, f, g) + K + message_schedule[i]
            T2 = Sigma0(a) + Maj(a, b, c)
```

```

    h = g
    g = f
    f = e
    e = d + T1
    d = c
    c = b
    b = a
    a = T1 + T2

    # Add the compressed chunk to the current hash value
    hash_value = [(x + y) mod 2^32 for (x, y) in zip(hash_value, [a, b, c, d, e, f, g, h])]

    # Step 7: Produce the final hash value
    final_hash = concatenate(hash_value)

    return final_hash

function deriveDynamicIV(input_data):
    initial_hash = sha256(input_data)
    dynamic_IVs = splitInput(initial_hash, 4)
    return dynamic_IVs

function deriveDynamicConstants(input_data):
    constants = [someTransformation(part) for part in splitInput(input_data)]
    return constants

function determineNumberOfRounds(input_data):
    length = lengthOf(input_data)
    return max(64, length % 128) # Minimum 64 rounds

function dynamicMessageSchedule(input_block):
    scheduled_message = [transform(part) for part in splitInput(input_block)]
    return scheduled_message

function splitInput(data, size):
    return [data[i:i+size] for i in range(0, lengthOf(data), size)]

function sha256(data):
    # Returns the SHA-256 hash of the input data
    return standardSHA256Hash(data)

function Sigma0(x):
    # SHA-256 Sigma0 function
    return (rotateRight(x, 2) ^ rotateRight(x, 13) ^ rotateRight(x, 22))

function Sigma1(x):
    # SHA-256 Sigma1 function
    return (rotateRight(x, 6) ^ rotateRight(x, 11) ^ rotateRight(x, 25))

function Ch(x, y, z):
    # SHA-256 Ch function
    return ((x & y) ^ (~x & z))

```

```

function Maj(x, y, z):
    # SHA-256 Maj function
    return ((x & y) ^ (x & z) ^ (y & z))

function rotateRight(value, bits):
    # Rotates the value to the right by the specified number of bits
    return (value >> bits) | (value << (32 - bits))

function concatenate(values):
    # Concatenates the list of values into a single binary string
    return ''.join([intToBinaryString(value) for value in values])

function lengthOf(data):
    # Returns the length of the data
    return len(data)

function intToBinaryString(value):
    # Converts an integer to a binary string
    return format(value, '032b')

function standardSHA256Hash(data):
    # Placeholder for a function that computes the standard SHA-256 hash
    return standard_sha256_library_hash_function(data)

function someTransformation(part):
    # Placeholder for any transformation logic
    return part

function transform(part):
    # Placeholder for message scheduling transformation logic
    return part

```

5.4 QISKIT CODE:

```

user_input = input("Please enter something: ")

# Step 1: Calculate the length of the binary representation
binary_representation = ''.join(format(ord(char), '08b') for char in user_input)

# Step 2: Calculate the number of zeros needed for padding
padding_length = (512 - (len(binary_representation) + 24 + 1)) % 512 # 24 bits for message length + 1 for
the '1' appended earlier

# Step 3: Ensure the message length is represented in a 24-bit binary format
message_length_binary = format(len(user_input), '024b')

# Step 4: Append the necessary zeros

```

```

padded_binary_representation = binary_representation + '1' + '0' * padding_length + message_length_binary

#print("Padded binary representation:", padded_binary_representation)

# Extract the parts of the binary string
first_two_bytes = padded_binary_representation[:16] # First two bytes (16 bits)
middle_byte = padded_binary_representation[16:24] # Middle byte (8 bits)
last_two_bytes = padded_binary_representation[-16:] # Last two bytes (16 bits)

# Convert the binary strings to integers and compute the sum
m = int(first_two_bytes, 2) + int(middle_byte, 2) + int(last_two_bytes, 2)

#print("Sum of first two bytes, middle byte, and last two bytes:", m)

a = int(middle_byte, 2) % 32

# Compute b (sum of specified bytes mod 32)
b = m % 32

#print("a =", a)
#print("b =", b)

"""#Initial Hash Value declaration
H1 = 0x6a09e667
H2 = 0xbb67ae85
H3 = 0x3c6ef372
H4 = 0xa54ff53a
H5 = 0x510e527f
H6 = 0x9b05688c
H7 = 0x1f83d9ab
H8 = 0x5be0cd19*/"""

# Given initial hash values
H = [
    0x6a09e667,
    0xbb67ae85,
    0x3c6ef372,
    0xa54ff53a,
    0x510e527f,

```

```

0x9b05688c,
0x1f83d9ab,
0x5be0cd19
]

```

```

if a >= b:

```

```

    for i in range(8):

```

```

        if i < 4:

```

```

            p = H[i] ^ m

```

```

            q = (m >> a) | (m << (32 - a)) # Right rotate by a

```

```

            H[i] = p + q

```

```

            #print(f'H[{i}] = {hex(H[i])}')

```

```

        else:

```

```

            p = H[i] + m

```

```

            q = (m >> a) | (m << (32 - a)) # Right rotate by a

```

```

            H[i] = p ^ q

```

```

            #print(f'H[{i}] = {hex(H[i])}')

```

```

else:

```

```

    for i in range(8):

```

```

        if i < 4:

```

```

            p = H[i] ^ m

```

```

            q = (m >> b) | (m << (32 - b)) # Right rotate by b

```

```

            H[i] = p + q

```

```

            #print(f'H[{i}] = {hex(H[i])}')

```

```

        else:

```

```

            p = H[i] + m

```

```

            q = (m >> b) | (m << (32 - b)) # Right rotate by b

```

```

            H[i] = p ^ q

```

```

            #print(f'H[{i}] = {hex(H[i])}')

```

```

# Original SHA-256

```

```

K = [

```

```

    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

```

```

    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,

```

```

    0xc19bf174,

```

```

    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

```

```

    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

```

```

    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,

```

```

    0x92722c85,

```

```

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]

```

```

def generate_hash(message: bytearray) -> bytearray:

```

```

    """Return a SHA-256 hash from the message passed.
    The argument should be a bytes, bytearray, or
    string object."""

```

```

    if isinstance(message, str):
        message = bytearray(message, 'ascii')
    elif isinstance(message, bytes):
        message = bytearray(message)
    elif not isinstance(message, bytearray):
        raise TypeError

```

```

    # Padding

```

```

    length = len(message) * 8 # len(message) is number of BYTES!!!
    message.append(0x80)
    while (len(message) * 8 + 64) % 512 != 0:
        message.append(0x00)

```

```

    message += length.to_bytes(8, 'big') # pad to 8 bytes or 64 bits

```

```

    assert (len(message) * 8) % 512 == 0, "Padding did not complete properly!"

```

```

    # Parsing

```

```

    blocks = [] # contains 512-bit chunks of message
    for i in range(0, len(message), 64): # 64 bytes is 512 bits
        blocks.append(message[i:i+64])

```

```

    # Setting Initial Hash Value

```

```

    h0 = H[0]
    h1 = H[1]
    h2 = H[2]
    h3 = H[3]
    h5 = H[4]
    h4 = H[5]
    h6 = H[6]
    h7 = H[7]

```

```

# SHA-256 Hash Computation
for message_block in blocks:
    # Prepare message schedule
    message_schedule = []
    for t in range(0, 64):
        if t <= 15:
            # adds the t'th 32 bit word of the block,
            # starting from leftmost word
            # 4 bytes at a time
            message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
        else:
            term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
            term2 = int.from_bytes(message_schedule[t-7], 'big')
            term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
            term4 = int.from_bytes(message_schedule[t-16], 'big')

            # append a 4-byte byte object
            schedule = ((term1 + term2 + term3 + term4) % 2**32).to_bytes(4, 'big')
            message_schedule.append(schedule)

    assert len(message_schedule) == 64

# Initialize working variables
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7

# Iterate for t=0 to 63
for t in range(48):
    t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
           int.from_bytes(message_schedule[t], 'big')) % 2**32)

    t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32

    h = g

```

```

    g = f
    f = e
    e = (d + t1) % 2**32
    d = c
    c = b
    b = a
    a = (t1 + t2) % 2**32

    # Compute intermediate hash value
    h0 = (h0 + a) % 2**32
    h1 = (h1 + b) % 2**32
    h2 = (h2 + c) % 2**32
    h3 = (h3 + d) % 2**32
    h4 = (h4 + e) % 2**32
    h5 = (h5 + f) % 2**32
    h6 = (h6 + g) % 2**32
    h7 = (h7 + h) % 2**32

    return ((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
            (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
            (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
            (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big'))

def _sigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 7) ^
           _rotate_right(num, 18) ^
           (num >> 3))
    return num

def _sigma1(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 17) ^
           _rotate_right(num, 19) ^
           (num >> 10))
    return num

def _capsigma0(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 2) ^
           _rotate_right(num, 13) ^

```



```

        _rotate_right(num, 22))
    return num

def _capsignal(num: int):
    """As defined in the specification."""
    num = (_rotate_right(num, 6) ^
           _rotate_right(num, 11) ^
           _rotate_right(num, 25))
    return num

def _ch(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (~x & z)

def _maj(x: int, y: int, z: int):
    """As defined in the specification."""
    return (x & y) ^ (x & z) ^ (y & z)

def _rotate_right(num: int, shift: int, size: int = 32):
    """Rotate an integer right."""
    return (num >> shift) | (num << size - shift)

if __name__ == "__main__":
    hexadecimal_input=generate_hash("Hello").hex()
    print(hexadecimal_input)

# Convert hexadecimal to binary using built-in function bin()
binary_output = bin(int(hexadecimal_input, 16))[2:] # [2:] to remove the '0b' prefix

# Print the binary equivalent
print("Binary equivalent:", binary_output)

```

INPUT:

HELLO

OUTPUT:

```

1010110010001100010010000100001000011011100000010001001110000011111111011
1001101101011011001000011110110011100000101010010000111110011010010011000
1001010111100011101111000110111010011111101001111111001001110110000001000
0101110000101000010011100110011000101

```

CHAPTER 6: COMPARISON BETWEEN ORIGINAL SHA-256 AND DYNAMIC SHA-256

6.1 Introduction:

The hash function SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic algorithm that generates a unique representation of input data. It is known for its time efficiency, low storage requirements, and resistance against computational attacks. However, with the emergence of reversible computing and the need for quantum-resistant cryptographic solutions, researchers have proposed a modified version of SHA-256 called Reverse SHA-256.

In terms of time complexity, Reverse SHA-256 has a complexity of $O(n)$, indicating a linear time requirement. On the other hand, Reverse SHA-256 using DRS-BOX has a complexity of $O(n)$, indicating exponential time requirements, which can be computationally expensive.

6.2 A Comparative Analysis of Original SHA-256 & Dynamic SHA-256:

| <u>Parameter</u> | <u>ORIGINAL SHA-256</u> | <u>MODIFIED SHA-256</u> |
|-------------------------|--------------------------------|--|
| Time Complexity | $O(n)$ | $O(n)$ |
| Garbage Values | 0 | 384 (Modifying initial hash- 256 + Word expansion- 128) |
| Gate Count | 96 | 96 |
| Space Complexity | $O(1)$ | $O(1)$ |
| Round | 64 | 48 |

CONCLUSION

As the concept of a mechanical quantum computer was proposed theoretically in the '80s and the mid-'90s two algorithms were coined which showed devastating effects of its calculation capacity that cripple the cryptographic world with Shor's and Grover's algorithms. After such algorithms were introduced the researchers tried to make the existing cryptographic models quantum resistant. The classical RSA, DES, and AES failed to protect but the modification of AES 128 bit to AES 256 bit which becomes quantum resistant. After that, the researches moved along the asymmetric algorithm that leads to the development of Lattice-based systems. This thesis explains in detail the devastating impact of Shor's and Grover and explains how it works. This paper also focuses on the quantum-safe symmetric algorithm such as 256 bit in details and it tries to differentiate quantum symmetric algorithms from asymmetric algorithms which have led to a detailed review of latticed based algorithms. This paper can help upcoming researchers to understand the differences and perform better in their respective research fields. Attack analysis of the proposed Hash Function is left for the future work.

REFERENCES:

1. [1] A. Barenco et al., "Elementary gates for quantum computation" Phys. Review, vol - A52, 3457. 1995.
2. [2] A. G. Aruna et al, "A Study on Reversible Logic Gates of Quantum Computing", (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 7 (1), pp 427-432, 2016.
3. [10] Amandeep Singh, Praveen Agarwal, Mehar Chand, "Analysis of Development of Dynamic S-Box Generation", International Conference on Computer Science and Information Technology 5(5), pp: 154-163, November, 2017. DOI: 10.13189/csit.2017.050502.
4. [11] Hongbo Wang, Haoran Zheng, Bin Hui and Hongwu Tang, "Improved Lightweight Encryption Algorithm Based on Optimized S-Box", 2013 International Conference on Computational and Information Sciences, pp 734-737, 2013.
5. [13] M. Almazrooie, A. Samsudin, R. Abdullah et al. Quantum reversible circuit of AES-128. Quantum Inf Process 17, 112 (2018). <https://doi.org/10.1007/s11128-018-1864-3>
6. [14] A. Peres, "Reversible Logic and Quantum Computers", Physical Review, vol. A32, pp. 3266-3276, 1985.
7. [15] Md. Selim Al Mamun and David Menville, "Quantum Cost Optimization for Reversible Sequential Circuit", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 4, No.12, 2013.
8. [16] Sree Ramani Potluri and Lipsa Dash, "Design of Linear Feedback Shift Register using Reversible Logic", International Journal of Advanced Research in Computer and Communication Engineering, Vol.4, Issue 12, December 2015.
9. [17] A. Webster, S. Tavares, "On the Design of S-boxes", Advances in Cryptology CRYPTO-1985, LNCS 218, Springer-Verlag, 1985. [18] Advances in cryptology, in: S. Goldwasser (Ed), Crypto88, Lec.
10. [15] Al Mamun, M. S., & Menville, D. (2013). "Quantum Cost Optimization for Reversible Sequential Circuit." International Journal of Advanced Computer Science and Applications, 4(12).
11. [16] Potluri, S. R., & Dash, L. (2015). "Design of Linear Feedback Shift Register using Reversible Logic." International Journal of Advanced Research in Computer and Communication Engineering, 4(12).
12. [17] Webster, A., & Tavares, S. (1985). "On the Design of S-boxes." In Advances in Cryptology CRYPTO-1985, LNCS 218, Springer-Verlag.
13. [18] Goldwasser, S. (Ed.). (1990). "Advances in cryptology." In Crypto88, Lecture Notes in Computer Science, 403, 450-468.