Quantum Implementation of SHA-1

Hansung University
Seyoung Yoon

Contents

1. Introduction

2. Background

3. Quantum Circuit of SHA-1

4. Benchmark

5. Conclusion

HSU HANSUNG UNIVERSITY

Introduction

Introduction

- **Quantum Computing**
  - Unlike classical computers, quantum computers leverage the quantum-mechanical phenomena of superposition and entanglement to perform calculations, thereby possessing expanded computational capabilities.

- **Shor's Algorithm**
  - Shor's algorithm can compromise widely used public-key cryptosystems such as RSA and ECC by solving fundamental mathematical problems in polynomial time.

- **Grover's Algorithm**
  - Grover's algorithm is known to be able to break symmetric-key cryptography by accelerating bruteforce search operations by a square root.

Contribution

- **Complete Quantum Circuit of the SHA-1**
  - We present a full quantum circuit implementation of the SHA-1 hash function, designed to serve as a verifiable oracle for quantum cryptanalysis.

- **Resource Optimization with Minimal Circuit Depth**
  - Our circuit achieves a low quantum circuit depth while using a minimal number of qubits.

Background

Grover's Algorithm

- **State initialization**
  - Apply Hadamard gates to each of the n input qubits to prepare a uniform superposition state $|\psi\rangle$, where all 2^n basis states are equally probable.

$$H^{\otimes n}|0\rangle^{\otimes n} = |\psi\rangle = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes n} = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}|x\rangle$$

- **Oracle**
  - The oracle encodes the function to be searched as a quantum circuit. It evaluates the function on all inputs in superposition and flips the sign of the amplitude corresponding to a solution state.

$$f(x) = \begin{cases} 1, & \text{if Hash}(x) = \text{target output} \\ 0, & \text{otherwise} \end{cases}$$

- **Diffusion operator**

$$U_f(|\psi\rangle|-\rangle) = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle|-\rangle$$

  - This operator amplifies the amplitude of the marked solution states identified by the oracle, thereby increasing the probability of measuring the correct result.

SHA-1

- **SHA-1**
  - SHA-1 is a cryptographic hash function designed by the United States National Security Agency (NSA).
  - It was used in DSS (Digital Signature Standard) and played a vital role in ensuring data integrity in major security protocols such as TLS and SSH.

Secure Hash Standard

Date Published: April 17, 1995

Supersedes: FIPS 180 (05/11/1993)

Author(s)

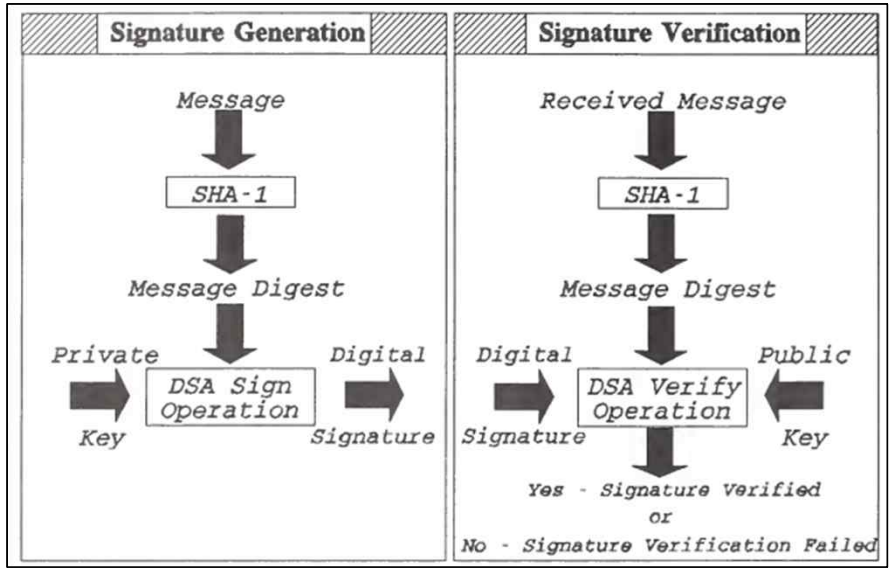
National Institute of Standards and Technology

Abstract

This standard specifies a Secure Hash Algorithm (SHA-1) which can be used to generate a condensed representation of a message called a message digest. The SHA-1 is required for use with the Digital Signature Algorithm (DSA) as specified in the Digital Signature Standard (DSS) and whenever a secure hash algorithm is required for federal applications. The SHA-1 is used by both the transmitter and intended receiver of a message in computing and verifying a digital signature.

Keywords

computer security; digital signatures; Federal Information Processing Standard; hash algorithm



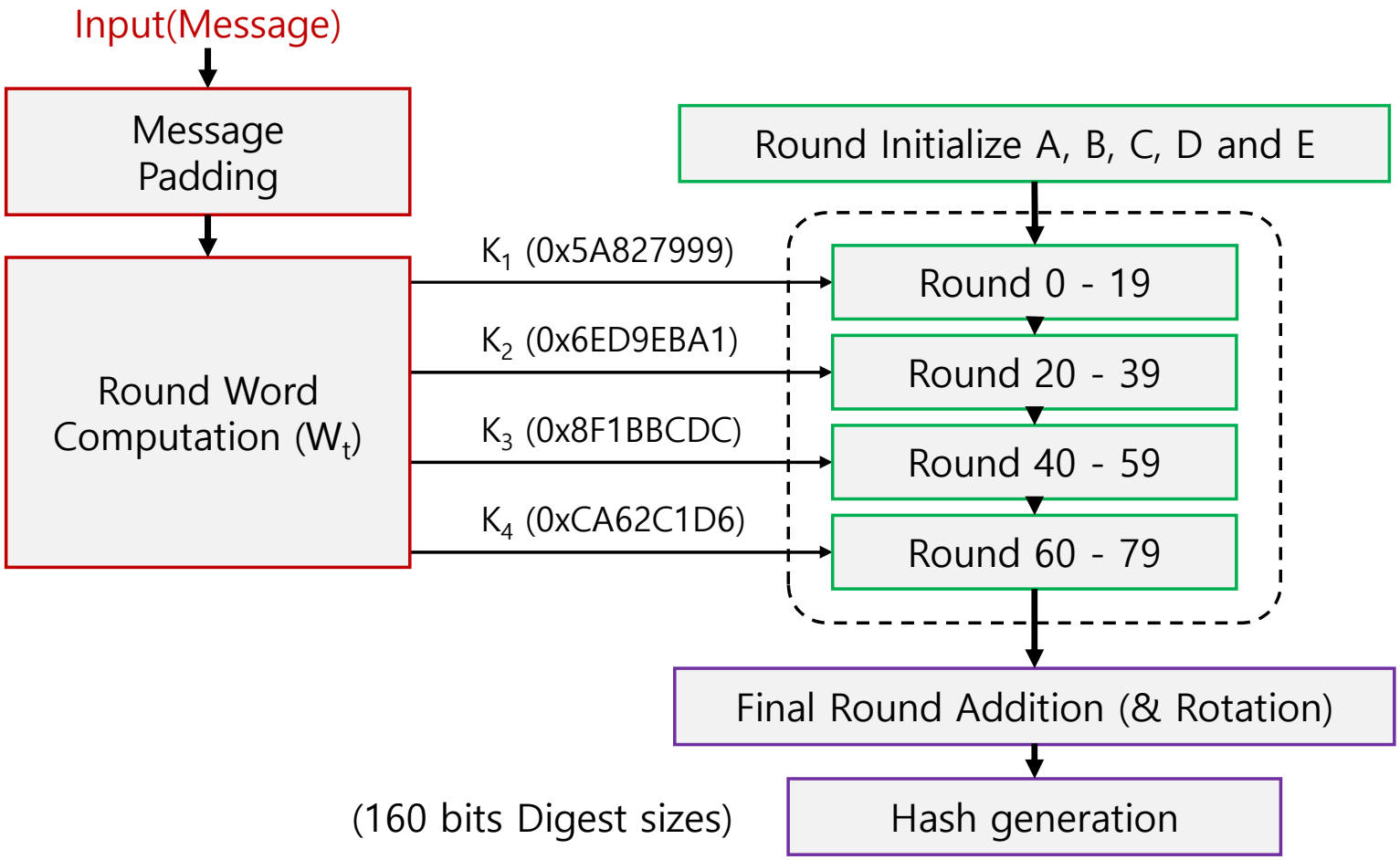
SHA-1

- **Finding Collisions in the Full SHA-1**
  - Despite its widespread adoption, the security of SHA-1 began to face scrutiny in the early 2000s. In 2005, a research team led by Xiaoyun Wang presented a theoretical attack that could find collisions in SHA-1 with a complexity significantly lower than the brute-force benchmark of 2^{80} operations.

- **The Instructive Value of Analyzing SHA-1**
  - Although SHA-1 is currently considered cryptographically weak for mainstream use cases, particularly digital signatures, it has been widely used for some time in Keyed Hash Message Authentication Codes (HMACs), as specified in FIPS 198-1.

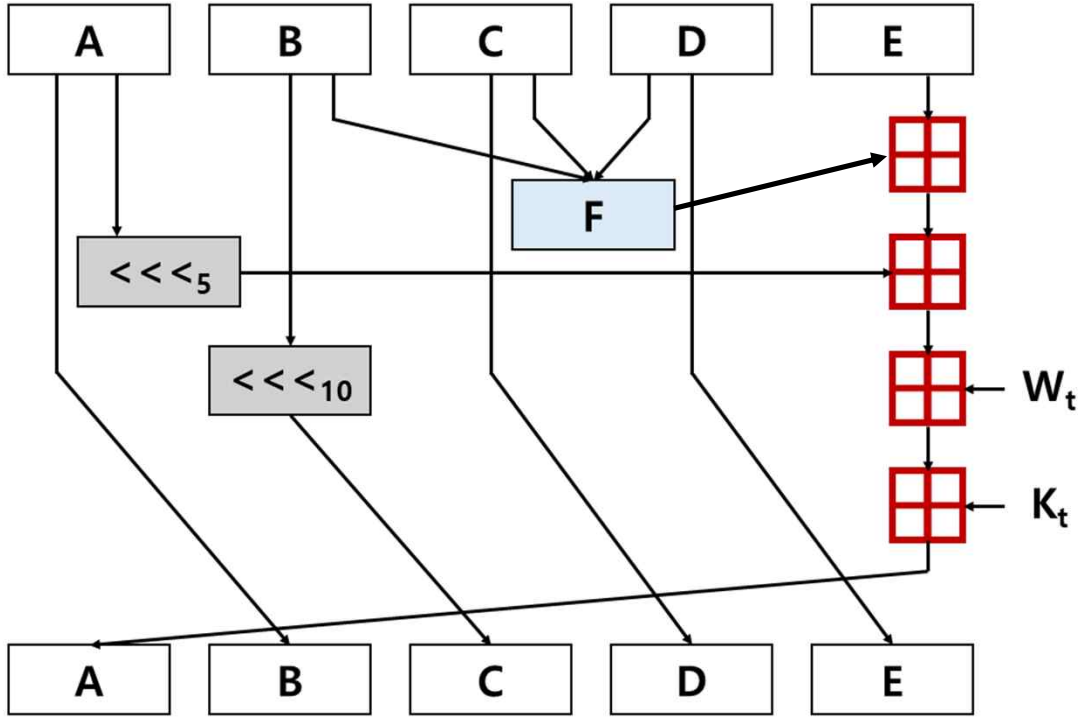
SHA-1

- **SHA-1**


Input(Message)
Message Padding
Round Word Computation (W_t)
K_1 (0x5A827999)
K_2 (0x6ED9EBA1)
K_3 (0x8F1BBCDC)
K_4 (0xCA62C1D6)
Round Initialize A, B, C, D and E
Round 0 - 19
Round 20 - 39
Round 40 - 59
Round 60 - 79
Final Round Addition (& Rotation)
(160 bits Digest sizes)
Hash generation

SHA-1

- **SHA-1**
  - The following figure shows the compression function that forms the core of the iterative operation of the SHA-1 algorithm.



Quantum Circuit of SHA-1

Message Padding and Round Word Computation

- **Classical Preprocessing**
  - The input message is padded into 512-bit blocks and then preprocessed. This process uses classical bits.

- **Dynamic Qubit Allocation and Qubit Reuse**
  - Each 512-bit message block is loaded into the quantum circuit. This is achieved by allocating a 512-qubit register, organized into 16 blocks of 32 qubits, to store the initial words W_0 through W_{15}. The subsequent 64 words (W_{16} through W_{79}), required for the 80 compression rounds of the algorithm, are fully computed within the quantum circuit.

  - We used only 512 qubits (32*16) instead of 2560 qubits (32*80) by taking advantage of SHA-1's expansion rules.

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1$$

$$W_{16} = ROTL(W_{13} \oplus W_8 \oplus W_2 \oplus W_0)$$

$$W_{17} = ROTL(W_{14} \oplus W_9 \oplus W_3 \oplus W_1)$$

$$\vdots$$

$$W_{32} = ROTL(W_{29} \oplus W_{24} \oplus W_{18} \oplus W_{16})$$

Rounds (0-79)

- **SHA-1 Round Structure**
  - This process updates five 32-bit working variables, denoted a, b, c, d, and e, which collectively form the internal state of the hash.

▷ 2. Main loop: 80 rounds
for $t = 0 \rightarrow 79$ **do** ▷ *Round Logic*
  if $0 \leq t \leq 19$ **then**
    $f \leftarrow \text{FuncF1}(b, c, d)$ ▷ Choose function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x5A827999})$
  else if $20 \leq t \leq 39$ **then**
    $f \leftarrow \text{FuncF2}(b, c, d)$ ▷ Parity function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x6ED9EBA1})$
  else if $40 \leq t \leq 59$ **then**
    $f \leftarrow \text{FuncF3}(b, c, d)$ ▷ Majority function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x8F1BBCDC})$
  else
    $f \leftarrow \text{FuncF2}(b, c, d)$
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0xCA62C1D6})$
  end if
  ▷ Compute temporary value T using sequential quantum additions
  $T \leftarrow \text{ROTL}(a, 5)$
  $T \leftarrow \text{QuantumAdder}(T, f, carry)$
  $T \leftarrow \text{QuantumAdder}(T, e, carry)$
  $T \leftarrow \text{QuantumAdder}(T, K_t, carry)$
  $T \leftarrow \text{QuantumAdder}(T, W[t \pmod{16}], carry)$
  ▷ Update state variables (quantum register swaps)
  $e \leftarrow d$
  $d \leftarrow c$
  $c \leftarrow \text{ROTL}(b, 30)$
  $b \leftarrow a$
  $a \leftarrow T$
  ▷ Uncompute this round's temporary values (f, K_t) to free qubits
end for

Resource-Efficient Quantum Circuit Implementation

- **Application of the Compute-Uncompute Paradigm**
  - We employ the Compute-Uncompute paradigm for the temporary qubits that store the output of the non-linear function F_t (denoted as the 'result' variable).

```
if 0 <= t <= 19:
    with Compute(eng):
        FunctionF1(result, b, c, d, temp)  # (B AND C) OR ((NOT B) AND D)
    classictoquantum(eng, 0x5A827999, K)
elif 20 <= t <= 39:
    with Compute(eng):
        FunctionF2(result, b, c, d)  # B XOR C XOR D
    classictoquantum(eng, 0x6ED9EBA1, K)
elif 40 <= t <= 59:
    with Compute(eng):
        FunctionF3(result, b, c, d, temp)  # (B AND C) OR (B AND D) OR (C AND D)
    classictoquantum(eng, 0x8F1BBCDC, K)
else:
    with Compute(eng):
        FunctionF2(result, b, c, d)  # B XOR C XOR D
    classictoquantum(eng, 0xCA62C1D6, K)
Uncompute(eng) # Uncompute Function F
```

Resource-Efficient Quantum Circuit Implementation

- **Resetting the Constant Register After Use**
  - Once the constant is no longer needed for the round's computation, we uncompute this register to reset it.

procedure ClassicToQuantum($op1, op2$)
  for $i \leftarrow 0$ to 31 **do**
    $b \leftarrow (op2 \gg i) \wedge 1$
    if $b = 1$ **then**
      X | $op1[i]$
    end if
  end for
end procedure

```
if 0 <= t <= 19:
    with Compute(eng):
        FunctionF1(result, b, c, d, temp)  # (B AND C) OR ((NOT B) AND D)
    classictoquantum(eng, 0x5A827999, K)
elif 20 <= t <= 39:
    with Compute(eng):
        FunctionF2(result, b, c, d)  # B XOR C XOR D
    classictoquantum(eng, 0x6ED9EBA1, K)
elif 40 <= t <= 59:
    with Compute(eng):
        FunctionF3(result, b, c, d, temp)  # (B AND C) OR (B AND D) OR (C AND D)
    classictoquantum(eng, 0x8F1BBCDC, K)
else:
    with Compute(eng):
        FunctionF2(result, b, c, d)  # B XOR C XOR D
    classictoquantum(eng, 0xCA62C1D6, K)
```

```
# Optimization: Initialize the K value to 0 and reuse it
if (0 <= t <= 19):
    classictoquantum(eng, 0x5A827999, K)  # '00000000000000000000000000000000'
elif (20 <= t <= 39):
    classictoquantum(eng, 0x6ED9EBA1, K)  # '00000000000000000000000000000000'
elif (40 <= t <= 59):
    classictoquantum(eng, 0x8F1BBCDC, K)  # '00000000000000000000000000000000'
else:
    classictoquantum(eng, 0xCA62C1D6, K)  # '00000000000000000000000000000000'
```

Operator

- **AND / OR / XOR**

AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Quantum Gate

- **CNOT / Toffoli / SWAP**

CNOT

Contorl	Target		Target
0	0	=	0
0	1	=	1
1	0	→	1
1	1	→	0

Toffoli

Inputs			Outputs		
Control	Control	Target	Control	Control	Target
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Internal Functions

- **(X AND Y) OR ((NOT X) AND Z)Internal Functions**

$$(x \wedge y) \vee (\bar{x} \wedge z)$$

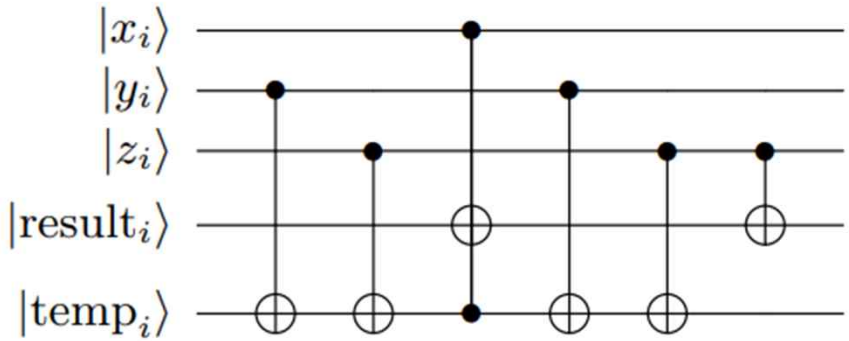
procedure FuncF1(x, y, z, *temp1*, *temp2*, *result*)
for $i \leftarrow 0$ to 31 **do**
    Toffoli | ($x[i]$, $y[i]$, temp1[i])
    X | ($x[i]$)
    Toffoli | ($x[i]$, $z[i]$, temp2[i])
    X | ($x[i]$)
    X | (temp1[i])
    X | (temp2[i])
    Toffoli | (temp1[i], temp2[i], result[i])
    X | (temp1[i])
    X | (temp2[i])
    X | (result[i])
end for
end procedure

Internal Functions

- **(X AND Y) OR ((NOT X) AND Z)Internal Functions**
  - Inside this function, temp was initialized using CNOT.

$$(x \wedge y) \vee (\bar{x} \wedge z)$$

procedure FuncF1(*result*, *x*, *y*, *z*, *temp*)
for $i \leftarrow 0$ to 31 **do**
    CNOT | $(y[i], temp[i])$
    CNOT | $(z[i], temp[i])$
    Toffoli | $(x[i], temp[i], result[i])$
    CNOT | $(y[i], temp[i])$
    CNOT | $(z[i], temp[i])$
    CNOT | $(z[i], result[i])$
end for
end procedure

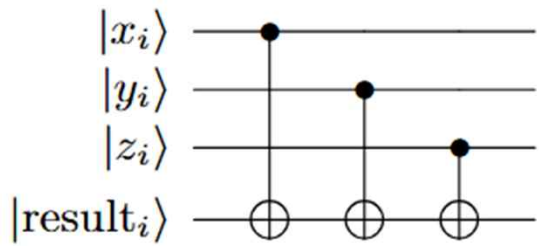


Internal Functions

- **X XOR Y XOR Z**
  - Function F2 is an internal function of SHA-1, which means (6), and this algorithm implements a three-input XOR logic by applying a sequence of three CNOT gates. Each input qubit (x[i], y[i], z[i]) acts as a control for the target qubit (result[i]).

$$x \oplus y \oplus z$$

procedure FuncF2(*result*, x, y, z)
for $i \leftarrow 0$ to 31 **do**
    CNOT | $(x[i], result[i])$
    CNOT | $(y[i], result[i])$
    CNOT | $(z[i], result[i])$
end for
end procedure

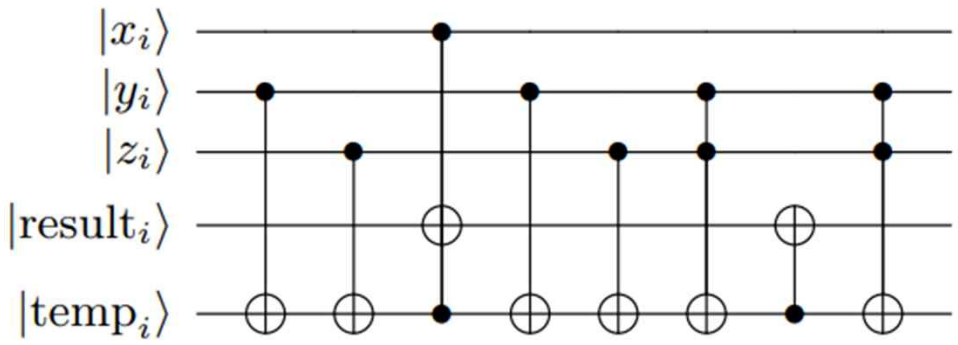


Internal Functions

- **(X AND Y) OR (X AND Z) OR (Y AND Z)**
  - In this function, after saving the final result, we use Toffoli to reset temp to 0.

$$\boxed{(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)}$$

procedure FuncF3(*result*, *x*, *y*, *z*, *temp*)
for $i \leftarrow 0$ to 31 **do**
    CNOT | $(y[i], temp[i])$
    CNOT | $(z[i], temp[i])$
    Toffoli | $(x[i], temp[i], result[i])$
    CNOT | $(y[i], temp[i])$
    CNOT | $(z[i], temp[i])$
    Toffoli | $(y[i], z[i], result[i])$
    CNOT | $(temp[i], result[i])$
    Toffoli | $(y[i], z[i], temp[i])$
end for
end procedure



Rounds (0-79)

- **SHA-1 Round Structure**
  - This process updates five 32-bit working variables, denoted a, b, c, d, and e, which collectively form the internal state of the hash.

▷ 2. Main loop: 80 rounds
for $t = 0 \rightarrow 79$ **do** ▷ *Round Logic*
  if $0 \leq t \leq 19$ **then**
    $f \leftarrow \text{FuncF1}(b, c, d)$ ▷ Choose function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x5A827999})$
  else if $20 \leq t \leq 39$ **then**
    $f \leftarrow \text{FuncF2}(b, c, d)$ ▷ Parity function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x6ED9EBA1})$
  else if $40 \leq t \leq 59$ **then**
    $f \leftarrow \text{FuncF3}(b, c, d)$ ▷ Majority function
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0x8F1BBCDC})$
  else
    $f \leftarrow \text{FuncF2}(b, c, d)$
    $K_t \leftarrow \text{ClassicToQuantum}(\texttt{0xCA62C1D6})$
  end if
    ▷ Compute temporary value T using sequential quantum additions
  $T \leftarrow \text{ROTL}(a, 5)$
  $T \leftarrow \text{QuantumAdder}(T, f, carry)$
  $T \leftarrow \text{QuantumAdder}(T, e, carry)$
  $T \leftarrow \text{QuantumAdder}(T, K_t, carry)$
  $T \leftarrow \text{QuantumAdder}(T, W[t \pmod{16}], carry)$
    ▷ Update state variables (quantum register swaps)
  $e \leftarrow d$
  $d \leftarrow c$
  $c \leftarrow \text{ROTL}(b, 30)$
  $b \leftarrow a$
  $a \leftarrow T$
    ▷ Uncompute this round's temporary values (f, K_t) to free qubits
end for

Final Round Addition (& Rotation)

- **Addition and Rotation**

```
with Compute(eng):
    ROTL(a, 5)
Adder(result, e, carry, ancilla)
Adder(a, e, carry, ancilla)
Adder(w[t % 16], e, carry, ancilla)
Adder(K, e, carry, ancilla)
Uncompute(eng)  # Uncompute ROTL a
```

```
ROTL(b, 30)
for bit in range(32):
    Swap | (e[bit], d[bit])
    Swap | (d[bit], c[bit])
    Swap | (c[bit], b[bit])
    Swap | (b[bit], a[bit])
```

temp = (a leftrotate 5) + f + e + k + w[i]
e = d
d = c
c = b leftrotate 30
b = a
a = temp

Final Round Addition (& Rotation)

- **Final Addition**
  - The new h0 is the sum of the old h0 and a, the new h1 is the sum of the old h1 and b, and so on for all five variables. These newly computed hash values then serve as the initial hash values for the processing of the next message chunk. It ensures that the entire message content sequentially influences the final output.

```
Adder(a, h0, carry, ancilla)
for bit in range(32):
    CNOT | (h0[bit], a[bit])

Adder(b, h1, carry, ancilla)
for bit in range(32):
    CNOT | (h1[bit], b[bit])

Adder(c, h2, carry, ancilla)
for bit in range(32):
    CNOT | (h2[bit], c[bit])

Adder(d, h3, carry, ancilla)
for bit in range(32):
    CNOT | (h3[bit], d[bit])

Adder(e, h4, carry, ancilla)
for bit in range(32):
    CNOT | (h4[bit], e[bit])
```

Final Hash Value Computation

- Once the very last message chunk has been processed, the five final 32-bit hash values (h0, h1, h2, h3, h4) are concatenated in order to form the classical 160-bit message digest.
- Each value is connected after determining the qubit value using 'measure'.

```
# quantum measurement function (final hash)
def q_measure(q, eng):
    All(Measure) | q
    eng.flush()

    result = 0
    n = len(q)

    for i in range(n):
        bit = int(q[i])
        shifted = bit << i
        result = result + shifted
    return result

# final hash measurement
if(resource_check != 1):
    H0 = q_measure(h0, eng)
    H1 = q_measure(h1, eng)
    H2 = q_measure(h2, eng)
    H3 = q_measure(h3, eng)
    H4 = q_measure(h4, eng)
    return f"{H0:08X}{H1:08X}{H2:08X}{H3:08X}{H4:08X}"
else:
    return 0
```

Benchmark

SHA-1 Quantum Circuit With Draper Adder

- **Addition on a Quantum Computer (Thomas G. Draper)**
  - The Draper adder, known as the Quantum Carry-Lookahead Adder (QCLA), is a widely recognized quantum arithmetic circuit notable for its logarithmic depth performance.

	Draper
Qubits	**985**
Toffoli (CCX) Gates	90,940
CNOT (CX) Gates	67,879
X Gates	20,339
Circuit Depth	**9,026**

SHA-1 Quantum Circuit With TTK Adder

- **Quantum Addition Circuits and Unbounded Fan-Out (Yasuhiro Takahashi)**
  - An alternative approach to quantum addition is presented by Takahashi, Tani, and Kunihiro (TTK), with a primary focus on minimizing the number of required qubits.

	TTK
Qubits	**929**
Toffoli (CCX) Gates	24,315
CNOT (CX) Gates	78,279
X Gates	189
Circuit Depth	**43,232**

Evaluation

- **Draper and TTK Adders**
  - In the era of Noisy Intermediate-Scale Quantum (NISQ) computers and beyond, circuit depth is often a more critical limiting factor than the absolute qubit count. Consequently, we selected the Draper adder as the foundational arithmetic component for our final, optimized SHA-1 circuit.

	Draper	TTK
Qubits	**985**	**929**
Toffoli (CCX) Gates	90,940	24,315
CNOT (CX) Gates	67,879	78,279
X Gates	20,339	189
Circuit Depth	**9,026**	**43,232**

Conclusion

Conclusion

- **A Resource-Optimized Quantum Circuit for SHA-1**
  - This research builds a comprehensive, resource-optimized quantum circuit for SHA-1. It focuses on reducing circuit depth, a key performance limiter for current and future quantum hardware, while also minimizing the use of qubits.

- **Quantum adder evaluation**
  - We measure the performance of a quantum circuit combining two adders, Draper and TTK, and find that it is a robust choice for minimizing decoherence on Noisy Intermediate-Scale Quantum (NISQ) hardware.

- **A Practical Framework for Quantum Cryptographic Implementations**
  - Our resource management and systematic Compute-Uncompute strategy provide a practical framework for implementing other complex cryptographic algorithms on quantum computers. The presented algorithms and circuit diagrams provide a transparent foundation for future quantum cryptography research.

Future works

- **Refinement and Extension**
  - Our subsequent research will investigate better depth-qubit tradeoffs for SHA-1 quantum circuits and accurate resource estimation for Grover's algorithm. We will also focus on quantum circuit implementations for other cryptographic algorithms.

- **Advancing Quantum Threat Research**
  - We are confident that the circuit implementation methodology proposed in this study can be applied to quantum circuit optimization for various cryptographic functions, including other hash functions, which will advance quantum threat research for modern cryptography.

Thank you for your attention

E-mail: sebbang99@gmail.com