CMPT 981
Quantum Circuits & Compilation

Mini-project Reflection Sara Babaee

1 Introduction

Shor's algorithm is one of the most well-known quantum algorithms because it can efficiently factor integers—a problem that is very difficult for classical computers. This advantage could break RSA and other cryptographic systems that rely on the hardness of integer factoring.

The quantum part of Shor's algorithm is a period-finding subroutine, which, given integers a and N, finds the smallest integer r such that:

$$a^r \equiv 1 \mod N$$

This technique is also a common pattern in a broader class of quantum algorithms.

In this project, we compile the period-finding subroutine into the Clifford+T gate set. We implement a program that takes a and N as inputs and generates a quantum circuit to find the multiplicative order of a modulo N.

We use *Qiskit 1.x* for our implementation and generate the final circuit in both *OpenQASM 2* format and as a visual circuit diagram.

2 Implementation Process

The period-finding subroutine in Shor's algorithm (at a high-level abstraction) consists of two n-qubit registers:

- x register, initially in $|0\ldots0\rangle$
- b register, initially in $|0\ldots1\rangle$

The process is divided into three main phases:

1. Creating a uniform superposition of n-bit binary strings in the x register.
2. Applying an oracle U_{ax} that maps $|x\rangle|1\rangle \to |x\rangle|a^x \mod N\rangle$.
3. Applying the QFT_n^{-1} followed by measurement of x.

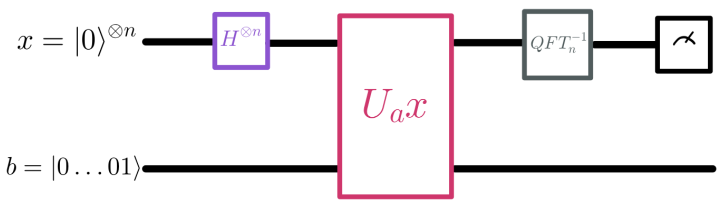

Fig (1). High-level schema of Shor's period finding subroutine.

Phase 1: Superposition Creation

The first step is straightforward: we apply Hadamard gates (H) to all qubits in the x register to create an equal superposition of all possible n-bit states. This is already within the Clifford+T gate set.

Phase 2: Modular Exponentiation Oracle

Then, we want to compute a^x for all $x \in \mathbb{Z}_N$ as generated in the previous step. To achieve this, we implement a reversible modular exponentiation circuit U_{a^x}, which forms the main computational challenge in terms of both complexity and compilation overhead.

As described in the project instructions, we can construct U_{a^x} by leveraging the binary representation of x. This reduces the problem to a sequence of controlled modular multiplications, $U_{a^{2i}}$, where:

$$U_{a^{2i}} : |b\rangle \rightarrow |a^{2^i} \cdot b \mod N\rangle$$

Each reversible $U_{a'}$ operation consists of a sequence of additions and shifts. The details of the modular multiplication implementation are covered in the next section.

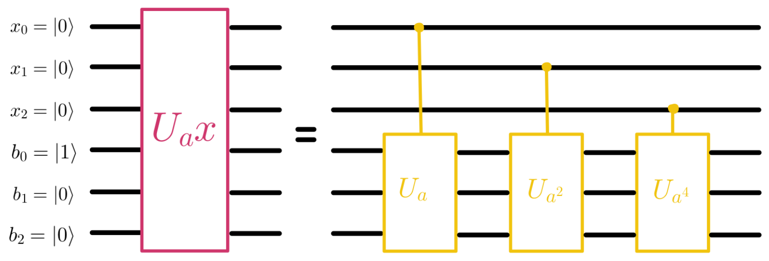

Fig (2). Implementation of the modular exponentiation gate using controlled-modular multiplication gates.

Phase 3: Inverse QFT and Measurement

To amplify periodicity and extract useful information, we apply the Quantum Fourier Transform (QFT). The QFT consists of a sequence of Hadamard gates and controlled rotations. We use *Qiskit*'s built-in QFT function and omit the final swaps for optimization, as they are unnecessary before measurement.

Finally, we measure the qubits in the x register and store the results in a classical register.

2.1. Implementation of Modular Multiplication

For each integer a, we construct the in-place modular multiplication circuit U_a using out-of-place modular multipliers U'_a as described in the project instructions. The operation of U'_a is defined as:

$$U'_a : |b\rangle|0\rangle \rightarrow |b\rangle|a \cdot b \mod N\rangle$$

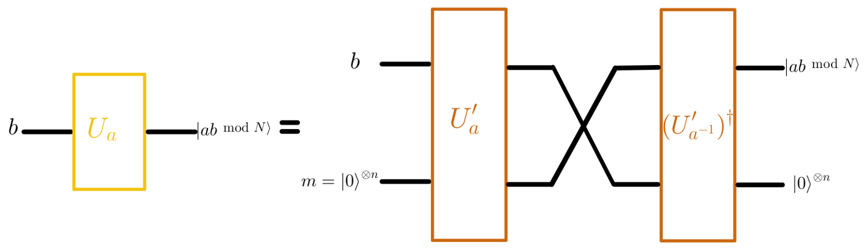

Fig (3). Implementation of the in-place modular multiplication gate using out-of-place modular multiplication gates.

Constructing U'_a

To implement U'_a, we introduce a new n-qubit register m. By leveraging the binary representation of a, modular multiplication is performed using a sequence of shifted additions. However, to ensure reversibility, we do not simply add a shifted b to the sum. Instead, we shift the sum and then add b, leading to the following transition sequence:

- Reversible Approach (preserving b):

$$|0\rangle \mapsto |b\rangle \mapsto |2b+b\rangle \mapsto |2^2b+2b+b\rangle \mapsto \ldots \mapsto |2^{(n-1)}b+\ldots+2b+b\rangle$$

- Standard Approach (not preserving b):

$$|0\rangle \mapsto |b\rangle \mapsto |b+2b\rangle \mapsto |b+2b+2^2b\rangle \mapsto \ldots \mapsto |b+2b+\ldots+2^{(n-1)}b\rangle.$$

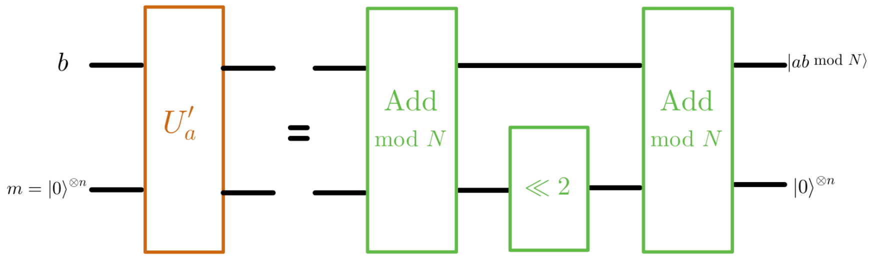

Fig (4). Implementation of the out-of-place modular multiplication gate using shift and addition gates for $a = 5 = (101)_2$.

Modular Addition

For modular addition, we implement the Cuccaro adder exactly as described in its original design. However, the standard n-qubit adder performing addition modulo 2^n, requires $n \geq 4$. To accommodate smaller values of n, we adjusted the construction for $n = 3$ and simplified it for $n = 1$ and $n = 2$.

This Cuccaro adder implementation requires $2n+1$ qubits, including one ancillary qubit in state $|0\rangle$ at the beginning and at the end. And for $n \geq 4$:

- $2n-3$ Toffoli gates,
- $5n-7$ CNOT gates.

Reversible Left Shift

To reversibly shift an n-bit string left by k positions, we use k clean ancilla qubits initialized to $|0\rangle$ and dirty at the end, and apply:

- $2n$ CNOT gates to perform the shift.

This process results in k leading zeros, while the ancilla qubits store the most significant bits.

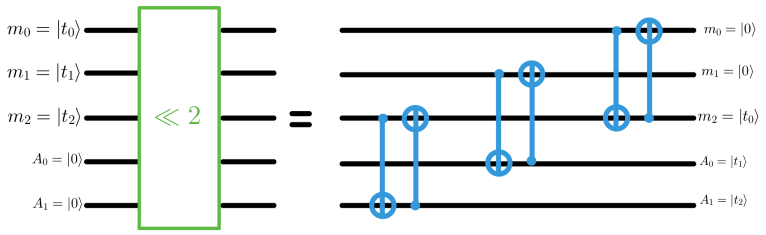

Fig (5). Implementation of a 3-qubit left shift by 2 using 2 ancilla qubits and $2 \times 3 = 6$ CNOT gates.

Resource Requirements for U'_a

Since we may need to shift the register m up to $n-1$ times, the implementation of U'_a requires $n-1$ ancilla qubits for shifts, and 1 ancilla qubit reusable for all additions.

Additionally, after each shift by k, we know the k least significant bits in m are zero, so we avoid unnecessary operations in addition by copying only the first k qubits of x to m and using an $n-k$-qubit addition circuit.

Thus, for an integer a with l ones in its binary representation, modular multiplication modulo $N = 2^n$ applies l addition and $l-1$ shifts, which requires $3n$ qubits and:

- $l(5n-7) + (l-1)2n$ CNOT gates,
- $l(2n-3)$ Toffoli gates.

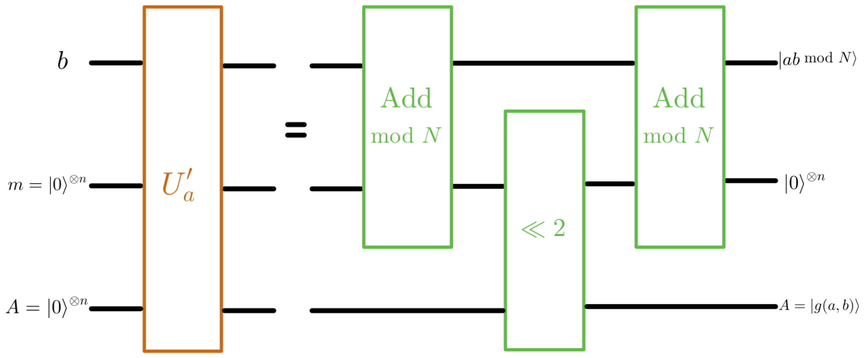

Fig (6). Implementation of the out-of-place modular multiplication gate for $a = 5 = (101)_2$ showing the required qubit registers.

Assembling the Pieces

After each use of U'_a, the dirty ancillas must be uncomputed. We do this using Bennett's trick, which efficiently removes garbage qubits without affecting the computation.

Also, for the inverse operation $(U'_{a-1})^\dagger$ to function correctly, we first need generate its dirty ancillas by applying U'_{a-1}.

Putting all the pieces together, Figure (7) presents the final implementation of the in-place modular multiplication circuit, which, for an integer a with multiplicative inverse a and l and l'ones in their binary representations, requires $4n$ qubits, along with:

- $2(l + l')(5n - 7) + 2(l + l' - 2)2n + 4n$ CNOT gates
- $2(l + l')(2n - 3)$ Toffoli gates

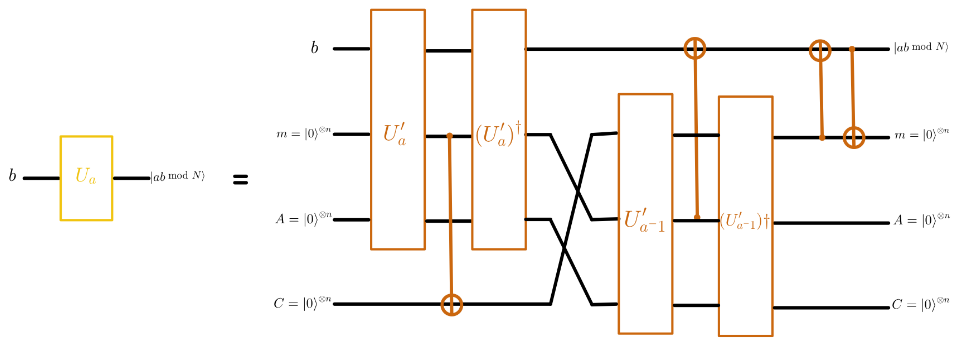
b
U_a
$|ab \bmod N\rangle$
=
b
$m = |0\rangle^{\otimes n}$
$A = |0\rangle^{\otimes n}$
$C = |0\rangle^{\otimes n}$
U'_a
$(U'_a)^\dagger$
$U'_{a^{-1}}$
$(U'_{a^{-1}})^\dagger$
$|ab \bmod N\rangle$
$m = |0\rangle^{\otimes n}$
$A = |0\rangle^{\otimes n}$
$C = |0\rangle^{\otimes n}$

Fig (7). Implementation of in-place modular multiplication, showing all required qubit registers.

It should be noted that the mid-process swap is a logical reordering of wires rather than a computational swap operation, meaning it has no real cost. However, at the end of the process, an actual swap with 0 is necessary to move the multiplication result into the first register.

Since the entire process is controlled by another qubit in the period-finding subroutine, a logical reordering instead of a swap would disrupt the circuit, affecting the next step.

3 Optimization

Since the modular multiplication circuit is controlled by another qubit in the period-finding subroutine, even CNOT gates become Toffoli gates, significantly increasing the cost. To improve the T-count, several problem-specific optimizations were implemented to minimize the number of CNOT and Toffoli gates:

- Optimized shift operations, leveraging ancillary qubits to use 2 CNOTs instead of 3 CNOTs in a typical SWAP.
- Modular addition instead of standard addition, reducing the need for 1 ancilla and saving 4 CNOTs.
- Copying instead of addition with 0, avoiding 2 Toffoli and 5 CNOTs per qubit.
- Using swaps with $|0\rangle$ requiring only 2 CNOTs instead of 3 CNOTs in a general swap, where applicable.

Additionally, more general quantum optimization techniques, such as phase folding, could be applied to further reduce circuit depth.

I also considered the possibility of using measurement-based uncomputation of Toffoli gates, but did not implement.

4 Debugging

During implementation, we used Pauli operators and the *Qiskit* Estimator to simulate measurements for debugging purposes. Specifically, to perform a computational basis measurement on qubit *j* in an *n*-qubit system, we applied the operator:

$$P = \bigotimes_{i=1}^{n} P_i, \quad \text{where} \quad P_i = \begin{cases} Z & \text{if } i = j \\ I & \text{o.w.} \end{cases}$$

Since the eigenvalues of Z are ± 1, this approach returned $+1$ if the measured qubit was in $|0\rangle$ and -1 if it was in $|1\rangle$. This technique allowed us to test each module individually and verify its correctness before integrating it into the full circuit.

5 Final Results & Observations

We compiled the circuit to the 3-controlled-X level and leave the compilation of controlled gates to Clifford+T to the Qiskit transpiler.

Implementation for inputs $N = 32 = 2^5$, $a = 3$, $\epsilon = 10^{-7}$ was run and the generated circuit in open qasm2 is attached. It uses 25 qubits and the T count is reported in table (1).

Optimization Level	**0**	**1**
T for multi-qubit unitaries	5404	4632
Single qubit rotations	1488	2098
Total T	**5452**	**4680**

Table (1). T-count for $N = 32 = 2^5$, $a = 3$, $\epsilon = 10^{-7}$, using 25 qubits.

6 Reflection on the Experience

Compiling a quantum algorithm was significantly more challenging than I expected. While the mathematical expressions appeared straightforward, translating them into actual gate implementations and circuit synthesis required much deeper consideration and attention to detail.

One of the main challenges was the lack of expressiveness in quantum programming languages. Even the simplest operations, such as a binary shift, required a large number of low-level instructions. The process would be much more efficient with a higher-level quantum programming language, ideally with built-in arithmetic operations to serve as a foundation for more complex tasks.

Another major challenge was debugging. For qubits in classical states, debugging was possible but difficult and time-consuming. However, for qubits in quantum superposition, debugging became even more complex due to limited simulation capabilities and hardware constraints. This highlights the importance of formal verification techniques and the potential benefits of abstract interpretation in quantum computing.