CMPT 478/981: Quantum Circuits and Compilation
Mini-project

Due March 6th
by email to the instructor

You may collaborate with your classmates on this assignment, but you should write and submit individual solutions

1 Compiling Shor's algorithm [50% of assignment grade]

In this assignment you will compile an instance of Shor's period finding algorithm. The goal is to implement a program which **generates**, for a particular choice of parameters, a circuit over Clifford+T (with some simplifying assumptions) performing the quantum period finding subroutine.

1.1 Background

Recall that Shor's algorithm for integer factorization relies on a quantum subroutine for computing the *multiplicative order* of an integer modulo N. Specifically, given an integer N, after eliminating trivial factors Shor's algorithm consists in:

1. Picking some integer a coprime to N,
2. Finding the order of a modulo N — that is, the smallest integer r such that $a^r \equiv 1 \mod N$
3. If $r = 2k$ and N is not a factor of $a^k + 1$, then $a^k + 1$ and N share a non-trivial factor

For a simple (in fact, trivial) example, note that $3^8 \equiv 1 \mod 32$. Since 32 does not divide $3^4 + 1 = 82$, we compute $\gcd(82, 32) = 2$, which is in fact a factor of 32!

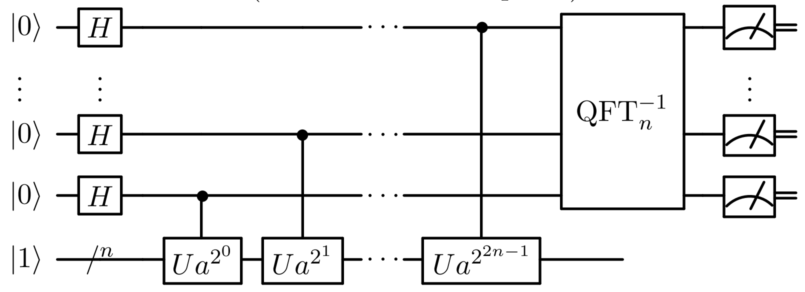
The quantum part of Shor's algorithm, the period finding algorithm, involves creating an initial n-bit (for our purposes) superposition $\frac{1}{\sqrt{2^n}} \sum_{x \in \mathbb{Z}_2^n} |x\rangle$, then applying modular exponentiation $U_{a^x} : |x\rangle|1\rangle \mapsto |x\rangle|a^x \mod N\rangle$, before taking an inverse Fourier transform on the superposition register and measuring to obtain an integer multiple of the order. The crux of implementing Shor's algorithm is then implementing the modular exponentiation circuit. Noting that

$$\begin{aligned} a^x &= a^{x_0 + 2x_1 + \cdots + 2^{n-1}x_n} \\ &= a^{x_0}(a^2)^{x_1} \cdots (a^{2^{n-1}})^{x_{n-1}} \\ &\equiv (a \mod N)^{x_0}(a^2 \mod N)^{x_1} \cdots (a^{2^{n-1}} \mod N)^{x_{n-1}} \mod N \end{aligned}$$

we find that modular exponentiation reduces to performing a series of *controlled* constant modular multiplications Ua^{2^i} where

$$Ua^{2^i} : |b\rangle \mapsto |b \cdot a^{2^i} \mod N\rangle$$

The resulting circuit is shown below (modified from wikipedia):



1.2 Modular multiplication

The modular multiplication involved in Shor's algorithm is more subtle than it initially seems. At face value, multiplication of b by a simply involves a number of shifted additions of b,

$$a \cdot b = a_0 \cdot b + a_1 \cdot 2b + a_2 \cdot 2^2 b + \cdots + a_{n-1} 2^{n-1} b.$$

where the shifted addition $x + 2^i b$ can be implemented simply by adding b to the most significant $n - i$ bits of x.

Complication arises from the fact that mutliplication in Shor's algorithm is **in-place** — i.e. $Ua^{2^i} : |b\rangle \mapsto |b \cdot a^{2^i} \mod N\rangle$ which is reversible by the fact that a^{2^i} is coprime to N and hence has a multiplicative inverse. In place addition is relatively straightforward — you may use the so-called Cuccaro adder for instance[1] However, as modular multiplication involves a *series* of shifted additions, we need to preserve the original value b along the way!

The simple solution is to note that the multiplicative inverse of $a^{2^i} \mod N$ can be efficiently computed using the extended Euclidean algorithm. Given that, define out-of-place multipliers

$$\begin{aligned} Ua^{2^i} &: |b\rangle|0\rangle \mapsto |b\rangle|a^{2^i} \cdot b \mod N\rangle \\ U(a^{2^i})^{-1} &: |b\rangle|0\rangle \mapsto |b\rangle|(a^{2^i})^{-1} \cdot b \mod N\rangle \end{aligned}$$

which can be used together with a register swap to implement in-place modular multiplication[2]:

$$|b\rangle|0\rangle \xrightarrow{Ua^{2^i}} |b\rangle|a^{2^i} \cdot b \mod N\rangle \xrightarrow{\text{swap}} |a^{2^i} \cdot b \mod N\rangle|b\rangle \xrightarrow{(U(a^{2^i})^{-1})^\dagger} |a^{2^i} \cdot b \mod N\rangle|0\rangle$$

where the last step follows because $(U(a^{2^i})^{-1})^\dagger : |x\rangle|(a^{2^i})^{-1} \cdot x \mod N\rangle \mapsto |x\rangle|0\rangle$.

[1]Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, David Petrie Moulton, *A new quantum ripple-carry addition circuit* . arXiv:quant-ph/0410184.

[2]Stephane Beauregard, *Circuit for Shor's algorithm using 2n+3 qubits.* quant-ph/0205095

1.3 Tasks

1. Implement a program (e.g. in Python, Q#, Qiskit, etc.) which compiles a quantum circuit for the parameters
   - N (the modulus being factored)
   - a (the integer coprime to N which we are finding the period of), and
   - ϵ (the error which the circuit is to be compiled to). **Note: see below for simplifying assumptions**

   You may produce the circuit in any way you like, but you should be able to (1) run it, and (2) inspect it to count the number of T gates, for instance. A simple solution is to write a Python program which generates openQASM[3] code, as openQASM can be run on many existing simulators.

2. Generate a circuit for $N = 32 = 2^5$, $a = 3$, $\epsilon = 10^{-7}$ and report the number of T gates and qubits used.

3. Run the compiled code in a simulator and **attempt** to factor 32. A simple simulator capable of simulating openQASM code is quantum++[4].

1.4 Simplifying assumptions

1. You may assume that $N = 2^n$. This allows you to forgo modular reduction and simply perform truncated arithmetic (e.g. only computing the lowest order n bits of a sum)

2. You do **not** need to compile single-qubit rotations to Clifford+T, and may instead upper-bound the T-count by assuming $3\log_2(1/\epsilon)$ T gates per single-qubit rotation, as per the Ross-Selinger algorithm.

3. You do not need to use the extended Euclidean algorithm to compute the multiplicative inverse of a^{2^i} mod N. In particular, you may brute force by checking whether $b \cdot a^{2^i} = 1$ mod N for every $b \in \{2, \ldots, N-1\}$.

2 Reflection [50% of assignment grade]

Reflect on your experience programming and compiling the algorithm. Did your implementation correctly produce a multiple of r? What difficulties arose? Did you do any debugging and if so how and did you find any bugs? If not, how could you go about debugging your implementation? What type of programming/compilation support would make this process easier?

3 What to submit

Submit your code, generated circuit, and reflections. **There will be a prize for the functioning implementation with the lowest T-count.**

[3] Andrew W. Cross, Lev S. Bishop, John A. Smolin, Jay M. Gambetta, *Open Quantum Assembly Language.* arXiv:1707.03429.

[4] qpp