Simulation of Modular Exponentiation Circuit for Shor's Algorithm in Qiskit

Harashta Tatimma Larasati[1,2], Howon Kim[3]
[1,3]Pusan National University, Republic of Korea
[2]Institut Teknologi Bandung, Indonesia
{harashta[1], howonkim[3]}@pusan.ac.kr

***Abstract*—This paper discusses and demonstrates the construction of a quantum modular exponentiation circuit in the Qiskit simulator for use in Shor's Algorithm for integer factorization problem (IFP), which is deemed to be able to crack RSA cryptosystems when a large-qubit quantum computer exists. We base our implementation on Vedral, Barenco, and Ekert (VBE) proposal of quantum modular exponentiation, one of the firsts to explicitly provide the aforementioned circuit. Furthermore, we present an example simulation of how to construct a $7^x \bmod 15$ circuit in a step-by-step manner, giving clear and detailed information and consideration that currently not provided in the existing literature, and present the whole circuit for use in Shor's Algorithm. Our present simulation shows that the 4-bit VBE quantum modular exponentiation circuit can be constructed, simulated, and measured in Qiskit, while the Shor's Algorithm incorporating this VBE approach itself can be constructed but not yet simulated due to an overly large number of QASM instructions.**

Keywords—quantum modular exponentiation circuit, quantum computing, Qiskit, quantum circuit

I. Introduction

The development of quantum computing field has been on the rise due to the technology advances and the apparent advantages of implementing quantum algorithm over its classical counterpart. One of the most influential quantum algorithms are Shor's algorithm for integer factorization problem (IFP) which offers super-polynomial execution speedup [1], enabling a fast cracking of the widely-used RSA cryptosystems when a quantum computer with sufficiently large number of qubits exists in the future [2]. Currently, researchers are still on the race to find the most optimized quantum circuit for realizing modular exponentiation function, the most computationally intensive part of Shor's algorithm [1].

The proposal by Vedral, Barenco, and Ekert (VBE) [3] is one of the earliest in literature that elaborate on how to build quantum arithmetic circuits, including the modular exponentiation circuit. The authors of [3] explain in detail the various aspects of their proposed quantum circuits, such as reversibility, change of state of each operations, and even provides explicit constructions of their proposed quantum circuits . Hence, [3] often serves as the baseline implementation for a more recent quantum circuit constructions and optimizations [4][5].

Despite the importance of VBE modular exponentiation circuit, how to fully build it in the existing quantum simulators is not trivial and has never been found in the literature. Available simulations mostly use an optimized circuit intended for small specific values, as in the example in [6], or utilizes the Beauregard's version of modular exponentiation [7], which uses Fourier-basis arithmetic based on Draper's adder [8]. To the best of our knowledge, there has not been any literature that detail the construction of binary arithmetic based VBE modular exponentiation circuit in a quantum computer simulator.

This paper is intended as an exposition to VBE modular exponentiation circuit and its demonstration in a quantum computer simulator. Since the main purpose of this paper is to give an insight on how to realize the circuit in a simulator, optimization is outside of the scope of this paper. Regardless, the simulation details provided in this paper will be beneficial to the readers start delving into quantum simulations.

II. Literature Review

A. Arithmetic Circuit in Quantum Computers

1) *Layers of quantum computers*: Quantum computers consists of several layers [9]: From top to bottom are the abstract layer (quantum algorithm level, example: Shor's algorithm), logical layer (quantum circuit level), error correction layer (fault-tolerant level), and physical layer (hardware level). At the level of quantum circuit, considerations related to the implementation such as qubit layout and error correction can be overlooked.

2) *Quantum arithmetic circuits*: Generally, arithmetic operations on quantum computers can be realized using the binary arithmetic or Fourier-basis arithmetic [10]. The former approach, in principle, only uses classical reversible gates (i.e., Toffoli gate, CNOT gate, NOT gate/X gate) and mostly very similar to that of in the classical computing (i.e., the same classical circuit, but made reversible), while the latter typically employ controlled-rotation (cR_n) gates which do not exist in classical computation.

Modular exponentiation circuits utilizing the latter approach is known to give the lowest width (i.e., smallest qubit size) [7], which make it a great candidate for implementation. However, the classically-inspired circuit has advantages over the Fourier-based [2] due to its efficiency in testing and debugging, and no overhead from single-qubit rotation synthesis when error correction is taken into account [11].

978-1-7281-7598-0/20/$31.00 ©2020 IEEE

B. Quantum Modular Exponentiation Circuit

1) *Construction of Modular Exponentiation Circuit in Quantum Simulators:* There are abundant literatures covering Shor's algorithm for integer factorization problem (IFP). For the modular exponentiation function in simulators, one of the commonly used are the circuit proposed by Beauregard [7], a Fourier-based arithmetic comprising a series of cR_n gates. This approach eases the circuit orchestration, as found in the demonstration using Qiskit [12] and Microsoft Quantum Development Kit (QDK) [13]. In terms of binary arithmetic-based circuit, the popular approach is by employing the circuit featured in [6], which presents reversible classical circuit for a specific value mapping by utilizing swap gates and X gates, as used in [14]. Recently, Qiskit also presents the controlled modular exponentiation circuit blocks [15] referring to [16], but also built for a very specific value, hence cannot be used for different bit size exponentiation.

2) *VBE Quantum Modular Exponentiation in Literatures:* There are several literatures that elaborates how to construct a modular exponentiation circuit based on Vedral, Barenco, and Ekert (VBE) approach. Two notable literatures are [5] which explains in detail and clarify several parts of VBE, including redrawing the schematic of the circuit. Another insightful resource of VBE approach can be found in [17], in which the author also presents the simulation of adder and modular adder. However, to date, there has not been any literatures which faithfully follows VBE approach up to the modular exponentiation circuit.

III. VBE-based Modular Exponentiation Circuit for Shor's Algorithm

In this section, we discuss the construction of VBE modular exponentiation circuit, which commonly referred to as the standard implementation [5]. This circuit is a classically-inspired circuit, i.e., a reversible classical circuit that can be implemented in a quantum computer, hence only utilizes gates that exist in classical computers: Toffoli, CNOT, NOT, and swap gates. Essentially, VBE modular exponentiation circuit is built using a series of modular multiplier circuits. The steps to construct modular exponential circuit are as follows:

1. Adder, which outputs $a + b$, $0 \leq a, b$;
2. Modular adder, which outputs $a + b$ mod N;
3. Modular multiplexer, which outputs ax mod N;
4. Modular multiplier, which outputs a^x mod N.

We redraw the block diagram presented in [5] to ease the explanation. The circuit is built in an inside-out approach, starting from the underlying adder, modular adder, all the way out to the modular exponentiation.

A. VBE Adder

VBE adder is a reversible ripple-carry adder which maps the quantum state $|a, b\rangle \rightarrow |a, a + b\rangle$. The result is computed in-place, i.e., it has introduced a space saving from the very general implementation of $|a, b, 0\rangle \rightarrow |a, b, a + b\rangle$. Three registers are required: Register A and B, each to contain the input a and b, respectively, and a carry register to hold the carry value during computation. As shown in the quantum circuit presented in Fig. 1, VBE adder works by first propagating the carry from the last significant (qu)bit to the most significant (qu)bit (i.e., from $|s_0\rangle$ to $|s_3\rangle$ in the figure), then computing the sum for each bit, which is then stored in Register B. Note that since Register B will hold the output at the end of the computation, it should accommodate the last carry. Hence, the size is n+1. Furthermore, using the reversed carry block proposed in VBE, the carry bits (except the last one) is directly uncomputed while computing the sum, resulting in the carry register's state back to $|0\rangle$ before computation finishes.

When applied backwards, this circuit performs the subtraction $|a, b\rangle \rightarrow |a, b - a\rangle$. When $b < a$, meaning that subtraction will yield negative number, the circuit will output the result in two's complements form: the n+1 bit will be in state $|1\rangle$ in this case. This reversed circuit, which is also the inverse for this case, is often referred to as *subtractor, reversed adder,* or *adder*$^{-1}$.

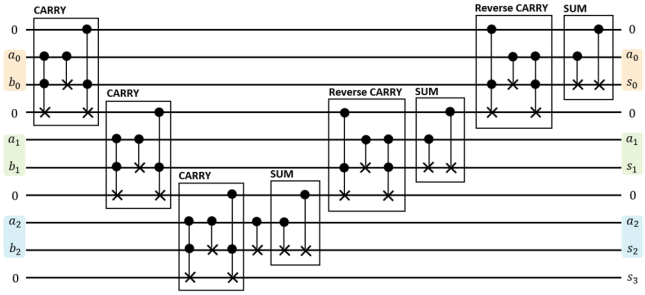

Fig. 1. A 3-bit VBE adder, redrawn from [3]

B. Modular Adder

For modular adder, the circuit should output $a + b - N$ when $a + b - N > 0$, otherwise outputs $a + b$, with the input bound of $0 \leq a, b < N$. Intuitively, it can be achieved by trying to subtract N from $a + b$. If it yields negative, add N back. However, due to the nature of quantum computation which cannot perform measurement without collapsing the quantum state, we need a way to compare without measuring the circuit.

VBE modular adder makes use of the concept of two's complements to detect the comparison. As shown in Fig. 2, it uses 5 adders (3 adders and 2 reversed adder to be specific, but we refer them generally as adders for this case), in which the first two are for performing $a + b - N$, the third is for adding back N if the former operation yield negative values (indicated by the values stored in the Temp Qubit, which is controlled by the MSB of Register B, b_{n+1}), and another two are for restoring the temp qubit bit back to $|0\rangle$. As with the adder, the computation result is stored in Register B.

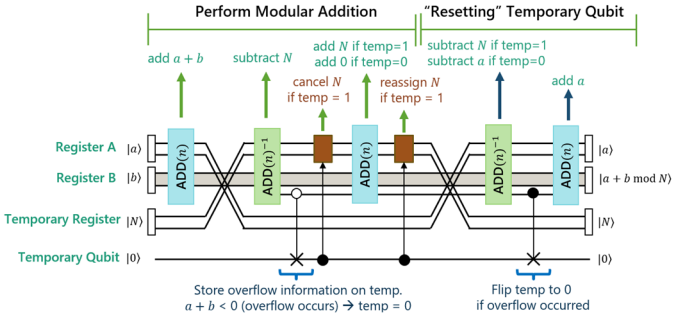

Fig. 2. Modular adder circuit, redrawn from [1][2].

At the end of the second adder, if $a + b - N < 0$, we need to re-add N back to the circuit. This is done as follows: b_{n+1} will be in the state $|1\rangle$ (due to the two's complements form), hence the Temp Qubit, initially in the state $|0\rangle$, will not be "activated" by the negative-control CNOT gate, and consequently, the next block in brown, which assigns N to Register A, will be bypassed. Therefore, the third adder will re-add N back to $a + b - N$, yielding the expected result of $a + b$.

On the other hand, if $a + b - N \geq 0$, there is basically no other computation needs to be done. In this case, since the result is positive, the negative-control CNOT (controlled by b_{n+1}, which now is in state $|0\rangle$) will flip the temp qubit to $|1\rangle$, which in turn "activate" the brown block and thus, cancels out the N currently stored in Register A. Hence, the third adder will perform no-op (i.e., only addition by 0).

The last two adders are for uncomputing the Temp Qubit. For the first case (i.e., $a + b - N < 0$), Temp Qubit stays in state $|0\rangle$, thus no uncomputation is required. The last two adders will just subtract a and then add a back without "activating" the last CNOT gate. The reason why Temp Qubit stays in state $|0\rangle$ after subtraction by a is because the subtraction will leave Register B in the state $|b\rangle$, which is only n bit in size (as the input is bounded to be lower than N) and is positive in value (thus, will not present in two's complements form), hence b_{n+1} will always be in state $|0\rangle$. On the other hand, for the latter case where Temp Qubit has been flipped (i.e., $a + b - N \geq 0$), the fourth adder will leave Register B in the state $|b - N\rangle$, which certainly is negative in values and hence, b_{n+1} will be in state $|1\rangle$. The next CNOT will then flip the Temp Qubit, restoring its state back to $|0\rangle$.

C. Modular Multiplier

Modular multiplier performs the operation $ax \bmod N$ where a is a constant. This can be obtained by the binary expansion of x as shown in (1).

$$ax = ax_{n-1}2^{n-1} + \cdots + ax_1 2^1 + ax_0 2^0 = \sum_{x_k} a2^k \quad (1)$$

Since the binary of x can be expanded, we can perform the multiplication by the repeated addition of $a2^k \bmod N$ for each bit of x if $x_k = 1$, as illustrated in Fig. 3. The gold block assigns $a2^k \bmod N$ to Register A as the input of modular adder, then uncompute it, to be assigned again by the value for the subsequent modular adder. Additionally, the value $a2^k \bmod N$ itself is precomputed, hence will always be lower than N.

We require the multiplier to be a controlled circuit. This is because we will incorporate the circuit into the modular exponentiation, whose input value will be specified by user (or in the Shor's algorithm case, will be simultaneously 1 and 0 due to the use of Hadamard gate). Furthermore, when the control qubit is $|0\rangle$, we would like the output of Register 1 to be $|x\rangle$ as opposed to also $|0\rangle$. This can be overcome by adding the last block, which just "copies" the value of x in the case where the control qubit is $|0\rangle$. Hence, the overall mapping is (2).

$$\text{CMODMULT}(n)|c, x, 0, 0\rangle = \begin{cases} |c, x, 0, ax \bmod N\rangle & (c = 1) \\ |c, x, 0, x\rangle & (c = 0) \end{cases} \quad (2)$$

Note that the state of Register 1 should be $|x\rangle$ at both the initial input and the output. This property will be the base of the modular exponentiation circuit.

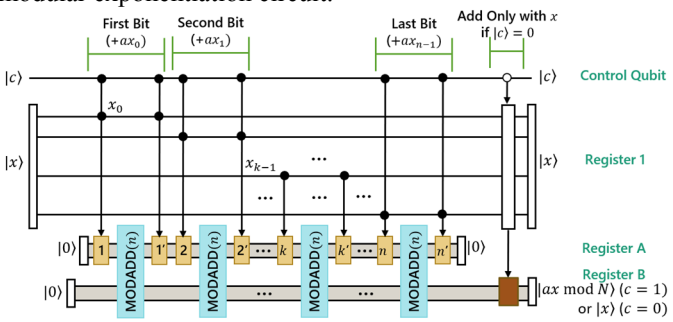

Fig. 3. Modular multiplier circuit, redrawn from [1][2].

D. Modular Exponentiation

To construct a modular exponentiation circuit that performs $a^x \bmod N$ where a and N is a constant, VBE utilizes the binary expansion of x but as the power in a as shown in (2) and (3).

$$a = a_{n-1}2^{n-1} + \cdots + a_1 2 + a_0 \quad (2)$$

$$x = x_{n-1}2^{n-1} + \cdots + x_1 2 + x_0 \quad (3)$$

By using this approach, exponentiation modulo N can be thought as a series of multiplication of the binary expansion as in (4).

$$a^x = a^{x_{n-1}2^{n-1}} \times a^{x_{n-2}2^{n-2}} \times \ldots a^{x_1 2} \times a^{x_0} = \prod_{\substack{k=0 \\ x_k=1}}^{n-1} a^{2^k} \quad (4)$$

To realize the circuit, VBE proposes a quantum circuit as illustrated in Fig. 4. Specifically, it performs the mapping $|x, 1, 0\rangle = |x, a^x \bmod N, 0\rangle$. The circuit consists of a series of modular multiplier pair (in a pair, one is the corresponding inverse circuit) with swap gates in the middle of each pair to preserve each multiplication result. The inverse circuit is for uncomputation of multiplication operation, preparing the circuit for the multiplication of the next bit. Note that the inverse is $a^{-1} \bmod N$, which can be precomputed classically using techniques such as the Euclidean algorithm [18]. Additionally, since the multiplication is performed bitwise, n-bit exponentiation requires n pairs of modular multipliers.

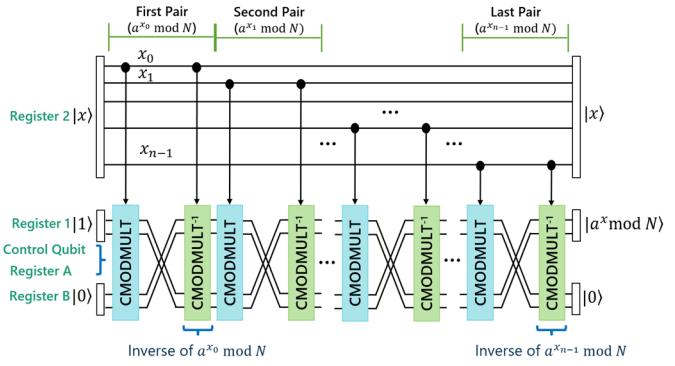

Fig. 4. Modular exponentiation circuit, redrawn from [1][2].

IV. Modular Exponentiation Circuit Simulation in Qiskit

In this section, we elaborate on our simulation of modular exponentiation circuit in Qiskit. The simulated circuit closely follows VBE's approach (in particular, the "perfected" schematic by [5]) with total number of qubit of $6n+3$ for

modular exponentiation circuit and $7n+3$ for use in Shor's algorithm. Since the circuit is general, it is scalable; it can be expanded for use in cracking RSA-1024 by only expanding the register. For ease of discussion, we explain with example of constructing a circuit that performs the computation $7^x \bmod 15$, with input size n=4.

A. General Idea

Suppose we want to build a modular exponentiation circuit $a^x \bmod N$ with a predetermined value of a=7 and N=15 for use in Shor's algorithm. There are several remarks. Firstly, one same modular exponentiation circuit should be able to cover any valid value of x (i.e., within the specified range of $0 \leq x \leq 2^n - 1$). This is because in Shor's algorithm, Hadamard gates are employed as the input rather than a "standard" input of zeros and ones, meaning that we need to build a circuit that works for 2^n variation of inputs simultaneously. Furthermore, in the exponentiation circuit, the input of the second multiplier pair is the result of the first multiplier pair, which cannot be measured during the computation. Hence, even though which variation of modular multiplier (i.e., value of a in each $ax \bmod N$) to be appended next is predetermined prior to the run of the algorithm, the input to each multiplier (i.e., value of x in each $ax \bmod N$) is a quantum state unknown to the observer.

Secondly, even though they bear the same variable name in [3] that we also keep in this paper, the constant in underlying modular multiplier circuit (i.e., a in $ax \bmod N$) is not necessarily same as the one in the modular exponentiation (i.e., a in $a^x \bmod N$). Rather, its value depends on the binary expansion of $a^x \bmod N$, which could be $a_{modmult_0} = a_{modexp}{}^{x_0 \cdot 2^0}$, $a_{modmult_1} = a_{modexp}{}^{x_1 \cdot 2^1}$, and so on. Breaking down into smaller parts, the underlying modular multiplier circuits to be incorporated are as shown in (5).

$$a^x \bmod N = (a^{x_0 \cdot 2^0} \bmod N) \times (a^{x_1 \cdot 2^1} \bmod N) \times (a^{x_2 \cdot 2^2} \bmod N) \times (a^{x_3 \cdot 2^3} \bmod N) \quad (5)$$

For our example of $7^x \bmod 15$, the circuit to be appended is as shown in (6), with the value of x in each multiplier circuit is the output of the preceding circuit. Conceptually, for 4-bit input, the illustration is as presented in Fig. 5.

$$\begin{aligned} &7^x \bmod 15 \\ &= (7^{x_0 \cdot 2^0} \bmod 15) \times (7^{x_1 \cdot 2^1} \bmod 15) \times (7^{x_2 \cdot 2^2} \bmod 15) \times (7^{x_3 \cdot 2^3} \bmod 15) \\ &= (7\boldsymbol{x} \bmod 15) \times (4\boldsymbol{x} \bmod 15) \times (1\boldsymbol{x} \bmod 15) \times (1\boldsymbol{x} \bmod 15) \end{aligned} \quad (6)$$

Note that the appended circuits are fixed for any value of modular exponentiation input (e.g., state from the Hadamard gates, or an arbitrary number chosen by user). Thus, the input only acts like a control. For instance, for exponentiation input x=3 (i.e., $x_3x_2x_1x_0$=0011 in binary), the first and second input will set the control bit to on, hence only the first two multipliers is active, multiplying $a2^i$ (mod N) for x_0 and x_1. On the other hand, the last two which are not "activated", will instead multiply 1 to the circuit. This can be achieved by using controlled gates such as CNOT or Toffoli gates depending on the implementation. Note that implementation-wise, the inverse of each modular multiplier is also required. We address the discussion directly on the implementation in the next section.

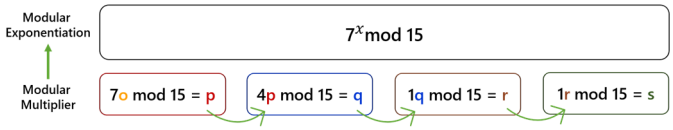

Fig. 5. Constructing modular exponentiation circuit from modular multiplier

B. Constructing Modular Exponentiation Circuit in Qiskit

We run our simulation on Qiskit 0.14.0 on a Windows 10 Pro PC, with Intel® Core™ i7-8700 CPU @ 3.20GHz processor and 64 GB of RAM. For composing the modular exponentiation circuit, we first refer to the Qiskit adder implementation from [19], then develop the rest on our own. Since we use a relatively large circuits, we simulate them on Qiskit local simulator rather than on the actual quantum hardware. Note that the circuits presented here have already been tested for any value of inputs (a, b, $x_{modmult}$, x_{modexp}) of *valid* range on the local simulator and yielded correct output.

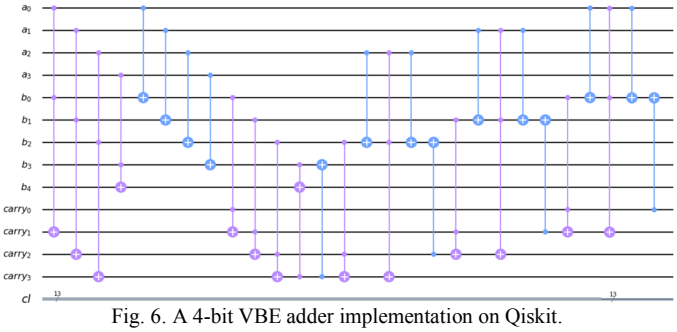

Fig. 6. A 4-bit VBE adder implementation on Qiskit.

1) *Adder and Adder^{-1}:* The implementation of these two circuits are straightforward, hence no tinkering is required. We define three registers that accounts for $3n+1$ qubits in total: Register A and a carry register of size n each, and Register B of size $n+1$. For a standalone use, Register A and Register B will be initialized with the inputs a and b. At the end of the computation, the result $a+b$ will be contained in Register B while Register A will stay in the same state. As for carry register, it acts as a temporary register which starts with the state $|0\rangle$ and will be returned to that initial state at the end. Fig. 6 and Fig. 7 shows the implemented `Adder` and `Adder`$^{-1}$ circuit in Qiskit, respectively.

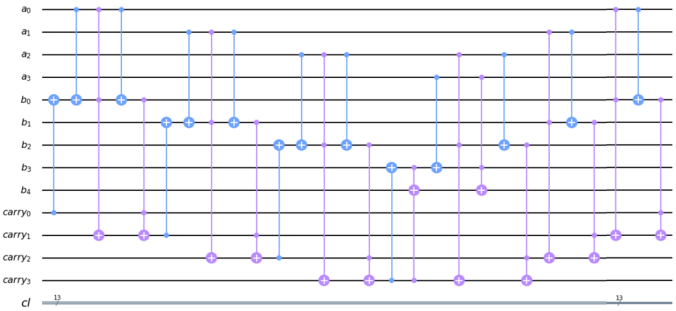

Fig. 7. A 4-bit VBE subtractor (reversed adder) implementation on Qiskit

To build `Adder`$^{-1}$, we define a function which performs adder operation backward. Regardless, Qiskit provides methods for easy dealing with circuit reversal and inversion [20]: `QuantumCircuit.mirror()` and `QuantumCircuit.inverse()`, respectively. The difference between the two is that mirror works by reversing the instructions without conjugating the

gates. For our case, since `Adder` does not employ any phase gates, the circuit of inverse and reverse will be the same.

2) *Modular Adder and Modular Adder*$^{-1}$*:* For modular adder circuit (`Modadd`), we can make it almost similar to that of in [18], with qubit size of $4n$+2. Specifically, we define another register, namely ModulusN, to contain the modulus value N, and a temporary qubit Temp for controlling the addition of N before and after the third adder. For negative-control CNOT gate, we simply add an X gate before and after the CNOT gate, as presented in Fig. 8. To conditionally append N to Register A (located just after the negative-control CNOT gate), we arrange a series of CNOT gates with Temp as the control and qubits corresponding to the value of N in Register A as the targets. Note that since what we simulate is N=15, CNOTs are appended for all qubits of Register A. For other case such as N=12 ($x_3x_2x_1x_0$=1100 in binary), the targets of CNOT gates are only the last two qubits.

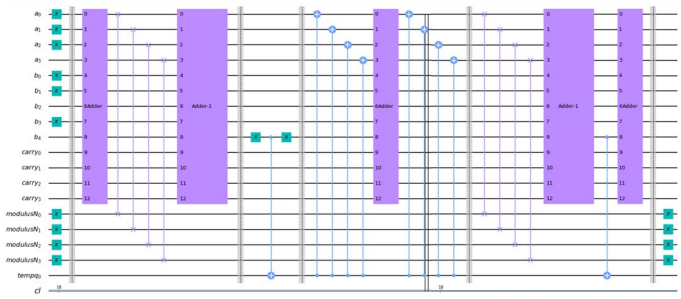

Fig. 8. A 4-bit VBE modular adder implementation on Qiskit.

In the presented figures, gray lines (called as a "barrier" in Qiskit) at the first and last part of the circuit is used to separate the inputs with the "internal" part of the circuit. Additionally, the adder blocks are incorporated to `Modadd` by utilizing `QuantumCircuit.to_instruction()` method. In Fig. 8, the inputs a and b are 7 ("0111") and 11 ("1011"), respectively, and are applied to the circuit by the X gate (the turquoise boxes in the picture). Regarding the assignment of modulus value N, it can be treated as an internal or external part of the circuit depending on the implementation. In terms of `Modadd`$^{-1}$, the construction is also by reversing the corresponding operation, as of `Adder`$^{-1}$.

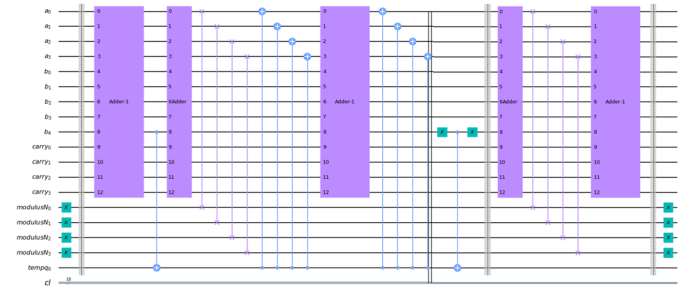

Fig. 9. A 4-bit VBE reversed modular adder circuit on Qiskit.

3) *Modular Multiplier and Modular Multiplier*$^{-1}$*:* The constructed modular multiplier circuit (`Modmult`) requires $6n$+3 qubits in total. We add another register (referred to as Register 1 in Fig. 3, and as Reg X in Fig. 10) for a placeholder for input x, and a control qubit Ctrl to switch between two parts: circuit performing $|x,0\rangle \rightarrow |x, ax \bmod N\rangle$ when the control bit is set, otherwise performing $|x,0\rangle \rightarrow |x,x\rangle$. A series of Toffoli gates assigns $a2^i$ value as the input of each `Modadd`, then retracts the value by appending the same gates after each `Modadd` to be used again to assign $a2^{i+1}$ in the next `Modadd`. These Toffoli gates employ Reg X and Ctrl for the control, as shown in Fig. 10, a circuit performing ax mod N = 7x3 mod 15.

In the figure, Ctrl is set, and Reg X holds the value of 3 ("0011"). Since Toffoli gate flips the value of the target qubit only if both control qubits are set (i.e., Ctrl and Reg X_i), assignment of $a2^i$ only occurs on the first and second `Modadd`, (which assigns 7 (mod 15) and 14 (mod 15), respectively), while the other two will not undergo state manipulation by their Toffoli gates. Note that in Fig. 10, each Toffoli gate series points to different targets for different `Modadd`, depending on the $a2^i$ to be added.

In another case when the control is not set, the whole assignment to modular adder will be bypassed (yields $|0\rangle$ at Register B), then proceed to the second part of reassigning back $|x\rangle$ via the Toffoli gates with an X gate at both sides to temporarily activate the control qubit.

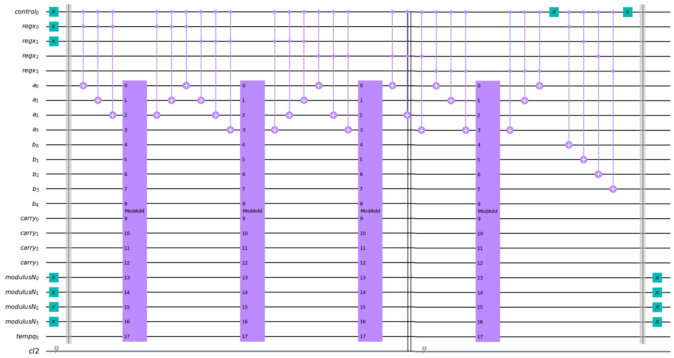

Fig. 10. A 4-bit 7x mod 15 VBE modular multiplier circuit on Qiskit.

The *reversed* version of the circuit is as shown in Fig. 11, similar to the previous approach. This reversed circuit, when appended to the original 7x mod 15, will return the state back to the original inputs. However, it is essential to understand that for the use in the modular exponentiation circuit, what we require is the *multiplicative inverse* of 7x mod 15, that is 13x mod 15 (hence, this circuit is what we refer to as `Modmult`$^{-1}$). This is because we save the computation result by switching it with another register (i.e., Register 1, which previously is in state $|1\rangle$) via a series of swap gates. In this case, appending reversed 7x mod 15 will not properly uncompute `Modmult` back to $|0\rangle$. Fortunately, state $|1\rangle$ (which now is at the result register, Register B), is a multiplicative identity. Hence, applying the inverse circuit in reverse will perform the job of uncomputation.

4) *Modular Exponentiation:* Our modular exponentiation circuit (`Modexp`) employs another register of size n (referred to as Register 2 in Fig. 4, as Reg X_{exp} in Fig. 12) for containing the x in $a^x mod\ N$, hence requires total qubit of $6n$+3. This register controls the Ctrl qubit via CNOT gates, which in turn will control the workings of the underlying modular multipliers. Reg X is always initialized with state $|1\rangle$, which functions as the input of the first multiplier. For our circuit of 4-bit exponentiation, four modular multiplier pairs are employed.

In our example of $7^3 mod\ 15$, after the first `Modmult` (7x mod 15) for computing the first bit, we need to: 1) clear out

the current `Modmult` in order to compute the second bit in-place (using the same sets of qubits); 2) store the current multiplication result. Reversing the first `Modmult` right away is not an option since this will also uncompute the result. Therefore, a series of swap gates switches the state of Register B (containing the result of first `Modmult`) with the state of Reg X (currently in the state $|1\rangle$). This preserves the current result since the state of Reg X (Register 1) does not change throughout the multiplication.

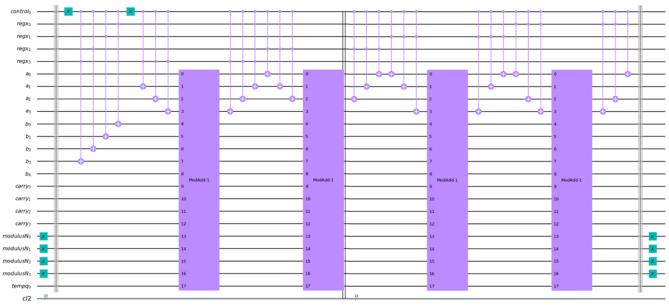

Fig. 11. A 4-bit 7x mod 15 VBE reversed modular multiplier circuit on Qiskit.

Consequently, uncomputing the multiplier is done by `Modmult`$^{-1}$ (13x mod 15 arranged in reverse order), which clears the value in `Modmult`. A subsequent CNOT gate disentangles `Modmult` from the first bit of Reg X_{exp}, then another CNOT gate entangles to the second bit, activating the second `Modmult` of 4x mod 15. The above process continues until the end of the computation. Note that by calculation, the inverse of the succeeding multipliers (4x mod 15, 1x mod 15, 1x mod 15 again) is same as the multiplier itself. Hence, their `Modmult`$^{-1}$ is just the same circuit in reverse.

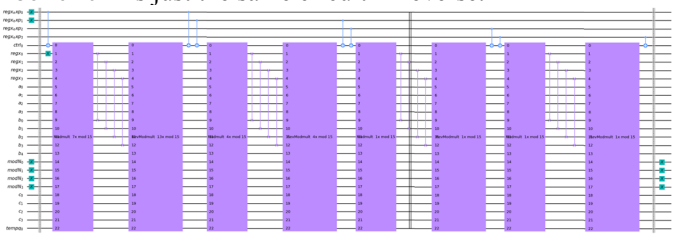

Fig. 12. A complete 4-bit modular exponentiation circuit on Qiskit.

C. *Modular Exponentiation Circuit in Shor's Algorithm*

For use in Shor's algorithm for IFP, the size of Reg X_{exp} should satisfy the condition $N^2 \leq 2^{size} < 2N^2$, which is 8 bit for the case N=15. Hence, the modular exponentiation circuit employs 8 pairs of modular multipliers, as shown in Fig. 13. Furthermore, unlike the previous examples of assigning the input via X gate, Hadamard gates are used here to create superposition—combination of all inputs simultaneously. Additionally, after the modular exponentiation, an inverse Quantum Fourier Transform (IQFT) and measurement is applied on Reg X_{exp}. The next step is the classical post-processing to obtain the period of the function.

Unfortunately, the circuit in Fig. 13 is too large to simulate on Qiskit (i.e., exceeding the allowed number of QASM instructions) at the time of our simulation. What we were able to simulate was with the Reg X_{exp} of size 4, which greatly reduced the number of QASM instructions. Regardless, our simulation shows that VBE modular exponentiation can be completely realized in Qiskit as one of the quantum simulators.

V. Remarks and Future Work

The circuits provided in this paper is the baseline implementation without considering optimization. A more recent literatures have proposed various techniques to be more efficient, such as utilizing the semi-classical Fourier Transform (qubit recycling), various tinkering to the modular exponentiation circuit to reduce circuit depth and width, or utilizing a whole new approach using QFT-based circuit for performing exponentiation. In future work, we plan to simulate the existing proposals of circuit optimization and also work on our own optimization approach.

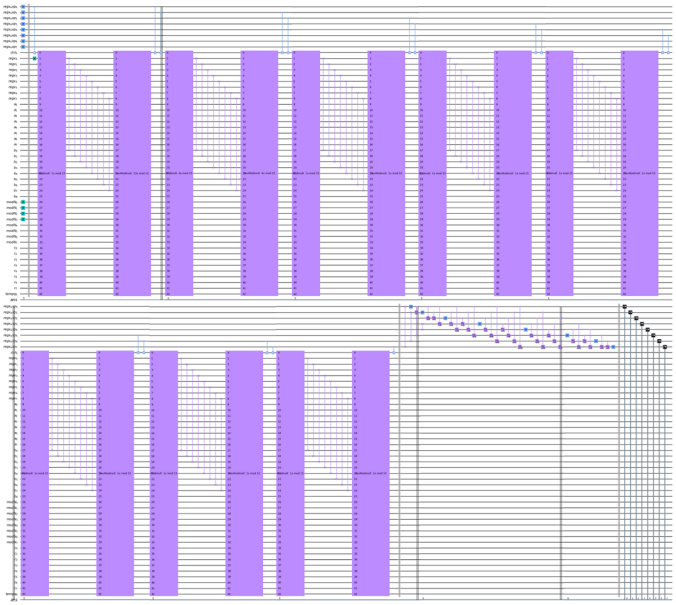

Fig. 13. Shor's algorithm utilizing VBE modular exponentiation circuit

Acknowledgment

This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity).

References

[1] A. Pavlidis and D. Gizopoulos, "Fast quantum modular exponentiation architecture for shor's factoring algorithm," *Quantum Inf. Comput.*, vol. 14, no. 7–8, pp. 649–682, 2014.

[2] E. Gerjuoy, "Shor's factoring algorithm and modern cryptography. An illustration of the capabilities inherent in quantum computers," *Am. J. Phys.*, 2005, doi: 10.1119/1.1891170.

[3] V. Vedral, A. Barenco, and A. Ekert, "Quantum networks for elementary arithmetic operations," *Phys. Rev. A - At. Mol. Opt. Phys.*, vol. 54, no. 1, pp. 147–153, 1996, doi: 10.1103/PhysRevA.54.147.

[4] R. Van Meter and K. M. Itoh, "Fast quantum modular exponentiation," *Phys. Rev. A - At. Mol. Opt. Phys.*, vol. 71, no. 5, pp. 1–12, 2005, doi: 10.1103/PhysRevA.71.052320.

[5] M. Nakahara and T. Ohmi, *Quantum computing: From linear algebra to physical realizations*. CRC press, 2008.

[6] I. L. Markov and M. Saeedi, "Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation," pp. 1–29, 2012, [Online]. Available: http://arxiv.org/abs/1202.6614.

[7] S. Beauregard, “Circuit for Shor’s algorithm using 2n+3 qubits,” pp. 1–14, 2002, [Online]. Available: http://arxiv.org/abs/quant-ph/0205095.
[8] T. G. Draper, “Addition on a Quantum Computer,” pp. 1–8, 2000, [Online]. Available: http://arxiv.org/abs/quant-ph/0008033.
[9] V. Gheorghiu and M. Mosca, “Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes,” pp. 1–19, 2019, [Online]. Available: http://arxiv.org/abs/1902.02332.
[10] R. Rines and I. Chuang, “High Performance Quantum Modular Multipliers,” pp. 1–48, 2018, [Online]. Available: http://arxiv.org/abs/1801.01081.
[11] T. Häner, M. Roetteler, and K. M. Svore, “Factoring using 2n+2 qubits with Toffoli based modular multiplication,” 2016, [Online]. Available: http://arxiv.org/abs/1611.07995.
[12] Qiskit Development Team, “qiskit.aqua.algorithms.factorizers.shor — Qiskit 0.19.6 documentation,” Jul. 17, 2020. https://qiskit.org/documentation/_modules/qiskit/aqua/algorithms/factorizers/shor.html#Shor (accessed Jul. 26, 2020).
[13] T. Matsuzaki, “Programming Quantum Arithmetics (Adder, Multiplier, and Exponentiation) – tsmatz.” https://tsmatz.wordpress.com/2019/05/22/quantum-computing-modulus-add-subtract-multiply-exponent/ (accessed Jul. 26, 2020).
[14] A. Phan, J. Gambetta, and R. Morales, “Shor’s Algorithm for Integer Factorization.” https://github.com/qiskit-community/qiskit-community-tutorials/blob/b9e487126dbd3f4f65696261fa85a7bd769b3ee8/algorithms/shor_algorithm.ipynb (accessed Jul. 26, 2020).
[15] Qiskit Development Team, “Shor’s Algorithm.” https://qiskit.org/textbook/ch-algorithms/shor.html (accessed Jul. 26, 2020).
[16] P. Padalia and A. Yadav, “Running Shor ’ s Algorithm on IBM Quantum Experience.” .
[17] D. Hutama, “An Exposition on Shor ’ s Factoring Algorithm and Quantum Circuits for Modular Exponentiation Description of Shor ’ s Algorithm,” pp. 1–21, 2018.
[18] V. Vedral, A. Barenco, and A. Ekert, “Quantum networks for elementary arithmetic operations,” *Phys. Rev. A - At. Mol. Opt. Phys.*, vol. 54, no. 1, pp. 147–153, 1996, doi: 10.1103/PhysRevA.54.147.
[19] S. Anagolum, “Arithmetic on Quantum Computers: Addition.” https://medium.com/@sashwat.anagolum/arithmetic-on-quantum-computers-addition-7e0d700f53ae (accessed Jul. 26, 2020).
[20] Qiskit Development Team, “QuantumCircuit — Qiskit 0.19.6 documentation,” Jul. 17, 2020. https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html (accessed Jul. 26, 2020).