QHLS: An HLS Framework to Convert High-Level Descriptions to Quantum Circuits

Chao Lu, *Student Member, IEEE,* Christian Pilato, *Senior Member, IEEE,* Kanad Basu, *Senior Member, IEEE*

***Abstract*—Quantum Computing has been shown to provide exponential performance improvements in several tasks, such as cryptography, healthcare, etc. This paper presents a new framework for quantum high-level synthesis, called QHLS, that aims to facilitate programmers using quantum computers. Currently, quantum-computer programmers need extensive linear algebra and quantum mechanics knowledge, which can be challenging for traditional software programmers. Additionally, the current quantum programming paradigm is not scalable, and it can be difficult to combine quantum circuits to create a more complex functionality. QHLS addresses these issues by enabling the automatic generation of quantum circuit descriptions directly from high-level behavioral specifications (using languages like C or C++). This simplifies the programming of a quantum computer, making it more accessible to a wider range of programmers. Our experiments show that QHLS can successfully translate high-level software programs containing various types of statements (such as arithmetic, logical, and conditional operations) into functionally equivalent quantum circuits.**

***Index Terms*—High-level synthesis (HLS), Quantum Circuits.**

I. Introduction

Quantum computers have the potential to perform certain computational tasks much faster than classical computers that rely on CMOS technology. This is because quantum computers use principles such as quantum entanglement and superposition, which allow them to perform certain operations more efficiently than their classical counterparts, quantum computers utilize quantum mechanic effects, including quantum superposition and entanglement, enabling qubits to perform computation on any states from $|0\rangle$ to $|1\rangle$ [1]. There has been a great deal of research focused on developing quantum algorithms that take advantage of these quantum principles to achieve exponential speed-ups over classical algorithms [2]. For example, Shor's algorithm showed that a quantum computer could factorize large numbers in polynomial time, potentially compromising current encryption standards [3]. There are various approaches to building quantum computers, including using superconducting materials, trapped ions, quantum annealing, and photonics to create quantum entanglement and superposition [4]–[7].

Python libraries like Qiskit, Cirq, and Tket provide quantum computing platforms that enable the generation of quantum circuits at the quantum gate or pulse level [8]–[10]. The workflow is demonstrated in Figure 1 These platforms offer diverse grammars and tools tailored for different quantum computing systems. However, programming high-level logic on a quantum computer can be challenging for conventional software developers. Moreover, it necessitates a strong grasp of quantum mechanics and linear algebra to create efficient quantum circuits [1]. This challenge becomes even more pronounced when designing complex quantum circuits. Therefore, there is a demand for a framework that simplifies quantum computer programming and reduces its complexity.

Chao Lu, and Kanad Basu are with the Department of Electrical and Computer Engineering, University of Texas at Dallas, Richardson, TX USA. e-mail: (cxl200053, kxb190012@utdallas.edu)

Christian Pilato is with Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy. e-mail:christian.pilato@polimi.it

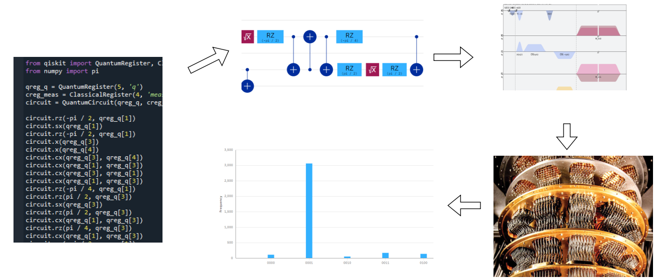


Fig. 1: The workflow of quantum circuits execution. The quantum circuit execution process involves several key steps. It begins with classical programming, where the circuit is designed and represented at the gate level on a classical computer. Next, the circuit is transpiled and converted into a pulse waveform to match a specific quantum computer's requirements. After this preparation, the quantum circuit is executed on the quantum computer, and the resulting measurement data is returned to the user.

High-level synthesis (HLS) is a widely adopted technique in CMOS-based hardware design. It translates a behavioral specification into a corresponding register-transfer-level (RTL) description that realizes the intended behavior [11], [12]. HLS tools accept C/C++ code as input and handle various behavioral instructions such as arithmetic and logical operations, conditional statements, and loops, which are then converted into their RTL equivalents. Several commercial and academic HLS frameworks, including Xilinx Vivado HLS, Stratus HLS, Bambu, and HDL Coder, utilize C/C++ and Matlab to generate RTL code [11], [13]–[15]. HLS greatly enhances hardware design efficiency and simplifies the development of sophisticated hardware systems.

In this paper, we introduce the Quantum HLS (QHLS) framework, which is the first HLS framework for quantum circuits. QHLS allows designers to generate quantum circuits from high-level software languages like C, without the need for expert knowledge of quantum mechanics. Our framework follows a modified design flow for creating quantum circuits. We start by taking a high-level behavioral code as input and use a Python framework to parse this description. The

core engine of QHLS generates an Open Quantum Assembly language (OpenQASM) file that describes the quantum circuit at the gate level. The resulting OpenQASM description is compatible with all current quantum programming tools.

Therefore, a designer can use a platform like Qiskit to execute the computation with the generated circuit [8]. The main contributions of this paper are:

- We propose and enhance the Quantum High-Level Synthesis (QHLS) framework, which translates high-level behavioral programming languages into corresponding quantum circuits.
- We have developed quantum circuit primitives for use in the QHLS framework, including those for emulating arithmetic circuits mentioned in Section IV-A, logic operations in Section IV-B, conditional statements in Section IV-C, iteration loops in Section IV-D, and array operations in Section IV-E.
- We have evaluated the performance of the QHLS framework on small benchmark programs typically used in traditional HLS frameworks. Since real quantum computers only have a limited number of noisy qubits and classical computers are unable to efficiently simulate quantum circuits with a large number of qubits, we have used tailored benchmark programs that can be simulated on a noise-free quantum simulator to evaluate the QHLS framework. We have also estimated the quantum resources required for these benchmark programs.

The remainder of the paper is organized as follows: In Section II, we provide background information on quantum computing, including an introduction to qubits and quantum gates, and a discussion of current issues of quantum circuit design. Section III shows the related work of quantum arithmetic circuit design and optimization. In Section IV, we describe the methodology for designing the QHLS framework. In Section V, we present our experimental results. Finally, in Section VI, we conclude the paper and suggest potential directions for future research.

II. Background

In this section, we introduce the concept of quantum circuits and explain the motivation for developing the Quantum High-Level Synthesis (QHLS) framework.

A. Quantum Superposition and Entanglement

Quantum entanglement is a quantum mechanical phenomenon where the state of one qubit can instantaneously change the state of other qubits in a predetermined way. A pair of qubits can be entangled by connecting them together. They are related in such a way that if the measurement value of one is known, the state of the other qubit can be determined based on the state of the measured qubit. Quantum computers make use of this phenomenon through the use of two-qubit gates like the CX gate to perform computations [1]. The principles of quantum superposition and entanglement allow two qubits to represent 4 parameters, for example, $|ab\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$, where $\alpha^2 + \beta^2 + \gamma^2 + \delta^2 = 1$. This means that the number of parameters grows exponentially with a linear increase in the number of qubits, known as the Hilbert space [1]. This can enable quantum computers to perform certain tasks faster than classical algorithms, sometimes achieving an exponential speed-up [3].

A quantum circuit consists of quantum bits (qubits) and quantum gates. Unlike classical bits, qubits can exist in a superposition state and are typically denoted using a bra-ket notation, expressed as $|\rangle$. A qubit in a superposition state can be represented as $|a\rangle$, where $a = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$, and $\alpha^2 + \beta^2 = 1$.

The state space of a single qubit encompasses all values between 0 and 1 and is visually represented using a Bloch Sphere, depicted in Figure 2. The Bloch Sphere's x-y plane corresponds to real states, while the z-axis represents the imaginary component. When representing the state of a single qubit, the basis states are identified as opposite points on the Bloch Sphere, as illustrated in Figure 2.

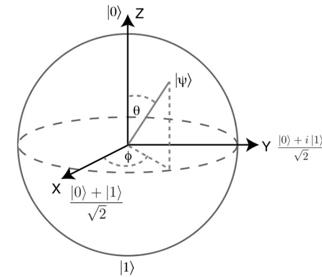


Fig. 2: Bloch Sphere Single Qubit Visualisation.

When designing quantum circuits, each quantum gate can be represented as a matrix, and the computation of the quantum circuit is represented as matrix multiplication. Tensor products are used for multiple qubits in the quantum circuit. However, quantum gates can perform logic operations similar to those in classical computers based on CMOS technology. For example, the Pauli-X gate flips the phase of qubits from $|0\rangle$ to $|1\rangle$ or from $|1\rangle$ to $|0\rangle$. The Control-X (CX) gate uses a control and a target bit, flipping the target bit if the control bit is $|1\rangle$. The Toffoli (CCX) gate has two control bits and a target bit, flipping the target bit when both control bits are 1. The SWAP gate swaps the states of two qubits, exchanging the information on those two qubits.

In the present day, quantum computers have limited qubits, which are prone to errors due to decoherence and noise. As a result, quantum computers are not yet able to outperform classical computers in all tasks. However, both industry and academia are working on developing error-resilient quantum computers, and it is expected that they will eventually be able to perform certain tasks faster than classical computers.

B. Motivation

Currently, programming a quantum computer requires extensive knowledge of quantum mechanics and linear algebra. This can make it difficult for traditional software programmers to program a quantum computer. In addition, the current

quantum programming paradigm is not easy to scale and integrate quantum circuits to achieve complex functionality.

To address these challenges, we propose the Quantum High-Level Synthesis (QHLS) framework. QHLS is a framework that allows quantum programmers to start with high-level behavioral descriptions (such as C or C++) and automatically generate the corresponding quantum circuit. This can reduce the complexity of programming a quantum computer and make it more accessible to traditional software programmers.

Moreover, to expand the functionality of quantum computers, our proposed QHLS framework can function complementary to the quantum circuit design and automation task that addresses the design of classical algorithms. Several related works also attempted to address the problem while generating classical algorithms to achieve similar goals that this work addresses [16]–[18]. For example, recently, atomic silicon logic was proposed to be one potential replacement for CMOS logic [19], where the design of atomic silicon is based on quantum logic gates. The proposed QHLS strategy could be potentially applied to the future atomic silicon binary logic.

To the best of our knowledge, this is the first HLS framework for quantum circuits. Our proposed QHLS framework allows designers to start with a high-level description and generate a quantum circuit design without the need for expert knowledge of quantum mechanics. In our framework, we modified the flow for designing a quantum circuit. We take a high-level behavioral code as input and use a Python framework to parse this behavioral description. An Open Quantum Assembly Language (OpenQASM) file is generated that describes the quantum circuit at the gate level. This OpenQASM description is compatible with all current quantum programming tools, and a designer can use a quantum programming platform like Qiskit to execute the computation with the generated circuit.

Our proposed Quantum High-Level Synthesis (QHLS) framework aims to simplify the process of designing quantum circuits by allowing the use of high-level software languages, such as C, for input. Currently, quantum programming languages only support gate-level programming, which requires expert knowledge of quantum mechanics and linear algebra and can be inefficient for generating complex quantum circuits. Inspired by the success of High-Level Synthesis (HLS) in automating the generation of RTL code from high-level software descriptions in CMOS-based hardware design, our QHLS framework aims to raise the programming abstraction level for quantum circuits by allowing the use of software language elements such as arithmetic and logical operations, conditional statements, and loops. To the best of our knowledge, no HLS framework currently exists for generating quantum circuits from high-level behavioral languages. Our proposed QHLS framework aims to fill this gap and facilitate the design of quantum circuits without requiring expert knowledge of quantum mechanics.

III. Related Work

The exponential computing power of quantum computing, combined with the linear stacking of qubits, holds great promise. Researchers have proposed various quantum circuits to perform arithmetic calculations [20]–[22]. Quantum arithmetic circuits are just one example of th many quantum implementations that have the potential to provide quantum advantage over classical algorithms [23]. Numerous research efforts are focused on various perspectives, including optimizing qubit requirements, reducing quantum gate complexity, and utilizing quantum data compression techniques to compress data. Additionally, there is considerable attention given to the design of complex arithmetic circuits [24]. The advantage of quantum computers in executing algorithms involving arithmetic is particularly relevant since arithmetic algorithms find wide application in numerous real-world situations [25].

Currently, many quantum toolkits are developed for the application for the implementation of quantum computers. Qiskit from IBM, Tket from Quantinuum, Cirq from Google, and Q# from Microsoft makes tremendous process on advancement on quantum computing [8]–[10], [26]. These toolkits enables local access to the quantum computers to the quantum computer remotely from the local classical computers with built-in compilation techniques for easier quantum circuit design and implementation on a quantum computer. However, a more abstract level of computation is lacking in their toolkit assembly. More abstract levels might lead to easier design and coding on more complex circuits with sophisticated functions.

The SILQ [27] framework achieved the uncomputation of temporary quantum values to ensure that the body computation is not affected by the measurement of the qubits, which does not include any classical synthesis with C code conversion. QCL is the first quantum programming language that enables the definition of quantum operators, and the function is similar to qiskit. Thus, those two libraries provide different functions from the proposed QHLS.

High-Level Synthesis (HLS) is an extensively researched area focused on automating the generation of RTL code from high-level software descriptions such as C or C++ [28]. HLS simplifies hardware design by enabling the same software code to produce multiple Register Transfer Level (RTL) designs. It finds valuable applications in the development of machine learning hardware accelerators using FPGA or ASICS [29]–[32], as well as in security chip and FPGA design [33], [34]. The HLS process involves three phases: input program compilation, micro-architecture development, and generation of RTL circuit descriptions. The output from HLS can be further utilized in subsequent stages, such as logic synthesis.

In contrast, the field of quantum circuit implementation currently lacks an equivalent framework, with existing quantum programming languages only supporting gate-level programming [11], [13]. This approach proves inefficient for generating complex quantum circuits and necessitates advanced knowledge of programming on quantum computers. To address this, the proposed Quantum High-Level Synthesis (QHLS) seeks to enhance the programming paradigm by introducing a higher level of programming that enables direct synthesis from a behavioral coding language to a quantum circuit. By leveraging software language elements like conditional statements, loops, and arithmetic and logical computations, QHLS allows for the generation of a quantum circuit without any additional requirement of quantum computing expertise.

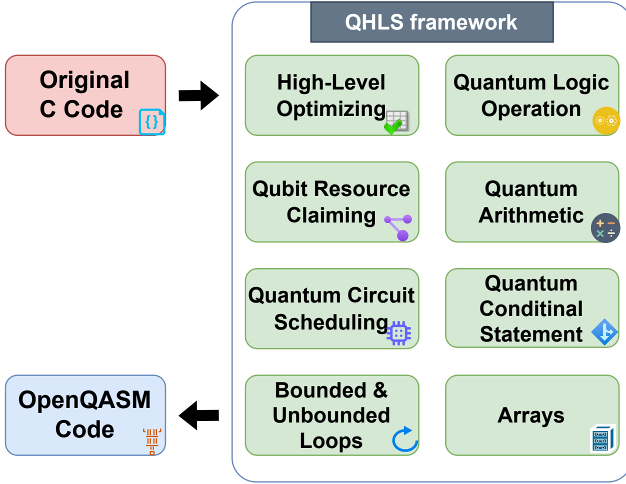
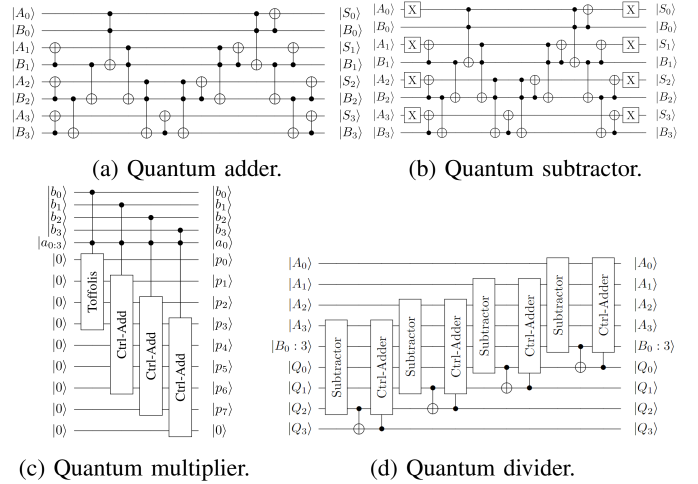


Fig. 3: Overall QHLS Workflow. We have incorporated numerous functions within the framework, enabling it to accept behavioral language inputs (C code) and automatically generate OpenQASM code.



(a) Quantum adder. (b) Quantum subtractor. (c) Quantum multiplier. (d) Quantum divider.

Fig. 4: Quantum arithmetic blocks.

IV. Proposed QHLS

The proposed QHLS framework aims to simplify the process of programming a quantum computer by using high-level behavioral software languages to automatically generate quantum circuits. The framework utilizes existing programming languages to convert input specifications into a gate-level quantum circuit, allowing for more efficient computation on a quantum computer.

In this section, we will mention the details of our proposed Quantum High-Level Synthesis. Figure 3 demonstrates the overflow of the synthesis process of a quantum circuit. Our proposed framework takes the original C code as input, and the QHLS will process the quantum circuit utilizing integrated functions including High-level Optimizing, Qubits Resource Claiming, Quantum Circuit Scheduling, Quantum Logic Operation, Quantum Arithmetic and Quantum Conditional Statements, in order to generate the Open Quantum Assembly (OpenQASM) Code automatically.

In the following subsections, we will discuss how QHLS translates the various facets of traditional behavioral description languages.

A. Quantum Arithmetic Circuit

The focus of this study is on quantum arithmetic operations, for which we have designed the following quantum circuits: quantum adder, quantum subtractor, quantum multiplier, and quantum divider (refer to Figure 4). The proposed QHLS uses existing high-level descriptions to generate these quantum circuits that carry out arithmetic computations on a quantum computer [20]–[22].

1) Quantum Adder: Figure 4a showcases the quantum adder design employed in this study. The addition operation utilizes CNOT gates and CCX gates [35]. Two binary numbers, denoted as A and B, are added together, and the result is stored in input A as $A \leftarrow A + B$. This approach minimizes qubit usage by overwriting one of the input registers while leaving the other unchanged. An additional qubit is introduced to handle the carry bit of the result.

This work utilized a half adder without a carry bit for easier calculation and synthesis process. However, the full adder is available to use in [35] with carry-in and carry-out qubits. The QHLS works well with a half adder without a carry bit.

2) Quantum Subtractor: Figure 4b depicts the quantum subtractor circuit utilized in our design. The subtraction operation leverages the property $A - B = \overline{\overline{A} + B}$ [35]. The input $\overline{A}$ and $\overline{B}$ are initialized by applying X gates to the desired bit, as shown in Figure 4b, to invert the quantum state.

3) Quantum Multiplier: In this paper, we utilize the quantum multiplier illustrated in Figure 4c [36]. It consists of Toffoli gates and the *quantum Ctrl-Add operation*. The Ctrl-Add circuit adds two numbers and replaces one of the input registers when the control point is $|1\rangle$. However, if the control point is $|0\rangle$, no operation is performed, and the state remains unchanged. The quantum multiplier calculates the product of two binary numbers, A and B, with the result stored in a new register C, while the input registers A and B remain unchanged. If the input bits A and B have the same bit length n, the output C will have a bit length of $2n + 1$.

4) Quantum Division: Figure 4d presents a generic quantum integer division circuit, which can also perform the modulo calculation [22]. It enables arbitrary integer division, where the dividend, divisor, and quotient have the same bit lengths. The circuit employs a restoring division algorithm for modulus calculation [22]. The quantum division circuit requires the quantum subtraction circuit and the *quantum Ctrl-Add* operation that contains one more control qubit to control whether the division calculation will be performed based on the value of the computation. The quantum division circuit comprises three parts: the dividend A, the divisor B, and the quotient Q. The circuit performs the operation A divided by B and stores the quotient in register Q, while the remainder is stored in the dividend register A.

B. Logic Operation

Quantum computers employ quantum logic gates, which differ from classical logic gates, offering unique computational

capabilities. While fundamental classical logical operations like "logic AND" and "logic OR" are not directly supported on quantum computers, they can still be achieved by combining multiple quantum gates. Figure 5 illustrates two variations of quantum circuits implementing "logic AND" and "logic OR" operations. In these circuits, the output is appended to a new register while leaving the input register unchanged. These quantum logic operations can be performed within conditional statements and logic operations without altering the state of the original input qubits.

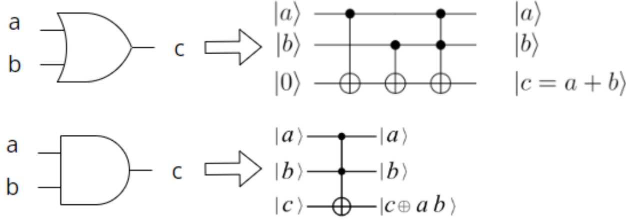


Fig. 5: Quantum logic operation equivalence.

C. Conditional Statement Branches

The conditional statement is a vital feature of classical computers, allowing calculations to be performed with different inputs based on specific predetermined conditions. In light of this, we put forth the concept of quantum conditional statement branches, which enable comparable computations to those performed by classical conditional statements. To achieve this, we develop sub-modules of conditional statement quantum circuits that emulate the computations corresponding to classical *if* statements on a quantum computer. Through our proposed Quantum High-Level Synthesis (QHLS) approach, we can generate arbitrary conditional statement branches to perform computations, thereby surpassing the limitations of small conditional statement prototypes that are unable to handle real-world scenarios.

If we want to perform the computation when $a < b$, we must first compare a and b in the quantum circuit and then project the result onto the target bit. Next, we can append a controlled-U gate to the target bit to execute the computation. For cases where $a \geq b$, we can compute $a < b$ and project the result onto the target bit. We can flip the state using an X gate on the target bit to perform *"a is not less than b"*. The remaining part of the circuit remains unchanged. In the following sections, we will introduce several other quantum circuits that demonstrate various types of conditional statements that are used in behavioral descriptions.

1) General Conditional Statements: One of the fundamental logical operations in high-level programming is the conditional statement, which allows for different computations based on the evaluation of a condition. In QHLS, we leverage the phenomenon of quantum entanglement to represent conditional statements. This approach utilizes a control bit to enable distinct computations on a target quantum gate, effectively implementing an "if-then" operation. Figure 7a illustrates this concept. In addition, QHLS employs the quantum subtractor, as described in Section IV-A, to design quantum comparators

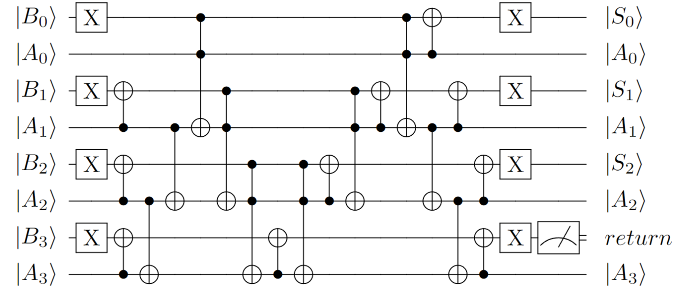


Fig. 6: Quantum Comparator to compute if $a > 3$, where a is initialized in binary state of 5.

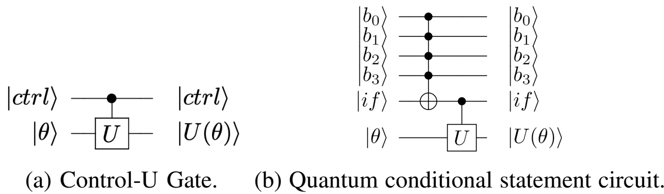
as shown in Figure 6. These comparators are instrumental in implementing conditional statements. We consider various scenarios, such as $a == b$, $a \neq b$, $a < b$, $a > b$, $a \geq b$, and $a \leq b$, where certain cases can be combined. In this section, we initially focus on two crucial situations: $a == b$ and $a > b$. The remaining scenarios can be derived from these cases, as explained later.

To begin with, let us consider the simplest scenario: $a == b$. Figure 7b demonstrates how this function can be achieved using the MCX gate. Initially, the input variable a is encoded as a binary state. Subsequently, the original register is used to encode the complement of b. After encoding, if the two variables are equal, the register value should consist entirely of 1s. We then apply the MCX gate to the original register, and the comparison results are transferred to an ancilla qubit for further computations.

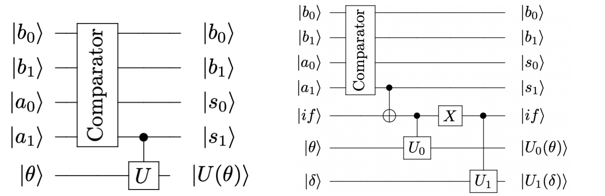
As for the scenario $a < b$, the quantum circuit first compares a and b and projects the result onto the target bit. Then, a controlled-U gate is applied to the target bit to perform the desired computation. For cases like $a \geq b$, we can accomplish the computation by evaluating $a < b$ and projecting the result onto the target bit. To compute "a is not less than b", we can append an X gate to the target bit to flip the state. The remaining part of the circuit remains unchanged. In the following sections, we will introduce additional quantum circuits that demonstrate various types of conditional statements, commonly used in behavioral descriptions. Figures 7d and 7e illustrate the simple quantum circuit framework for implementing conditional operations, enabling the realization of 'if and else' as well as 'if, else-if, and else' statements."

2) Complex Meshed Conditional Statements: In the previous sections, we focused on designing simple conditional statements, as depicted in Figure 7. However, real-world applications often demand more intricate decision branches. Figure 8 presents an example of a conditional statement used for handling complex meshed cases.

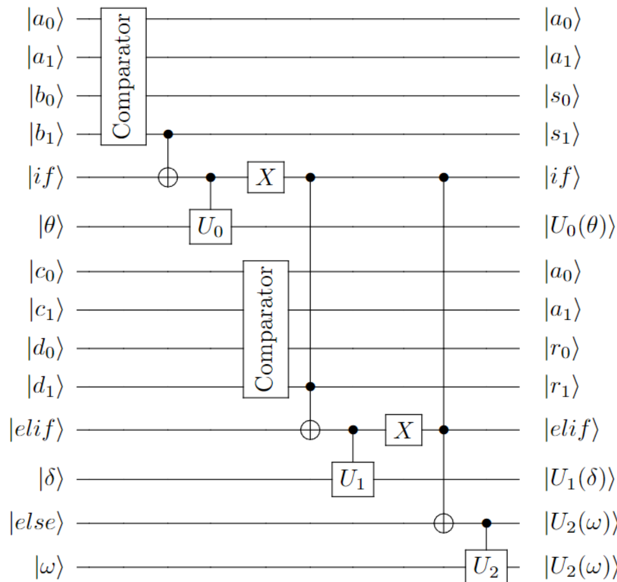
To accommodate the need for more complex conditionals within these statements, the model can be executed recursively. This involves replacing the consequent circuit or alternative circuits with new conditional branches, enabling more sophisticated computations. For instance, if an additional conditional branch is required in Figure 7e, the algorithm can seamlessly append a new conditional statement branch to the existing circuit in order to achieve the desired outcome. The



(a) Control-U Gate. (b) Quantum conditional statement circuit.



(c) An if statement with a 2-bit comparator. (d) Conditional statement including if and else statements.



(e) Conditional statements including if, else-if, and else statements.

Fig. 7: Quantum conditional statement variants.

resulting circuit, shown in Figure 9, incorporates an extra conditional statement branch with a simple if-else statement as demonstrated in Figure 7d. Please note that if the conditional statement branch necessitates an "else-if" statement with one Multiple-Controlled X (MCX) gate, additional modification is necessary that is on each ancilla qubit for the conditional operations, ensuring that the clauses are executed only once, preventing any redundant computations.

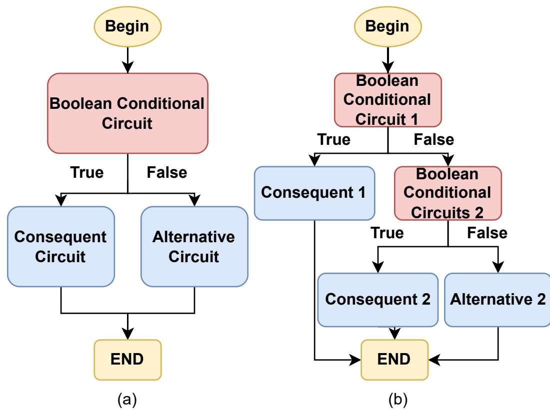


Fig. 8: (a) Conditional Statement for IF-THEN-ELSE. (b) Nested Conditional Statement by combining multiple IF-THEN-ELSE.

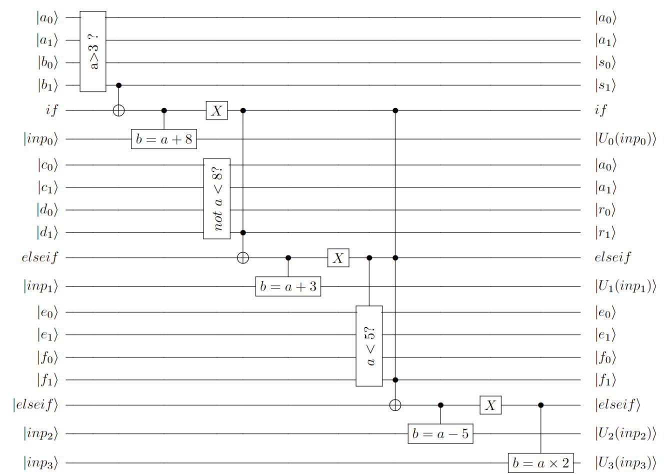


Fig. 9: Complex meshed conditional statement circuit example.

Listing 1: Complex meshed behavioral code.

```
int main() {
    int a = 5;
    if (a > 3) {
        int b = a + 8;
    } else if (a >= 8){
        int b = a + 3;
    } else if (a<5) {
        int b = a - 5;
    }
      else {
        int b = a x 2
      }
    }
    return b;
}
```

Listing 1 demonstrates a meshed conditional statement in behavioral language. To generate a quantum circuit with the corresponding behavioral code, first, the QHLS generates a quantum circuit by initializing the binary representation of the integer number a. The circuit then proceeds with the conditional statements. Initially, QHLS generates a quantum comparator circuit to determine whether $a > 3$. Here, we illustrate the quantum subtraction circuit, as depicted in Figure 6. This circuit performs the calculation $3 - 5$, as shown in Figure 10, and the return bit indicates whether the result is positive or negative. If the measurement yields 1, indicating that the condition is satisfied, the subsequent quantum circuit performs the computation $b = a + 8$. Therefore, the quantum circuit proceeds with this calculation. Once the computation is completed, the output of the quantum circuit should contain the measured values of a and b. These measured values can then be utilized for further calculations or returned to the user, depending on the specific requirements.

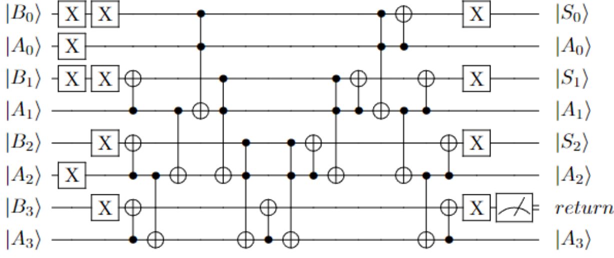


Fig. 10: Quantum Circuit that performs the arithmetic calculation 3-5 and returns the logic computation if $5 > 3$.

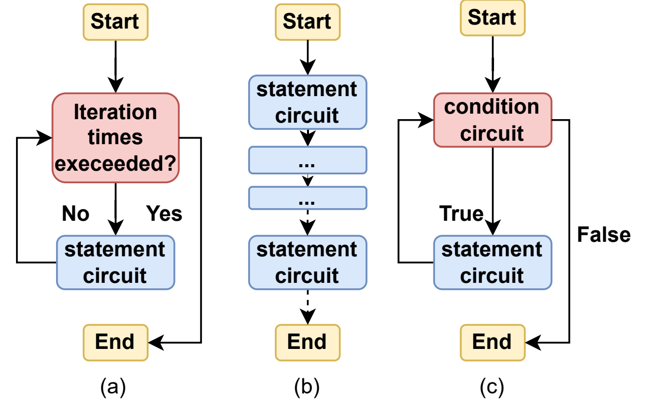


Fig. 11: (a). Bounded loop flow chart. (b). Unrolled Bounded Loop. (c) Unbounded Loop flow chart.

D. Iteration Loops

Loops are fundamental programming structures widely used in classical programming. To enable the quantum computer to perform iterative tasks akin to classical computers, we have incorporated loops into the proposed QHLS. In doing so, we have identified two types of loops that a program may contain: bounded loops and unbounded loops.

The first "For" loop showcased in Listing 2 allows the program to calculate the exact number of iterations needed to complete the execution of the loop's contents. To optimize the program within the QHLS framework, our initial step involves applying high-level optimizations that unroll the loop. This unrolling process entails executing the body statement within the loop five times consecutively. Subsequently, the quantum circuit performs the division computation five times within a single circuit, prior to proceeding with the measurement.

Listing 2: Behavioral Code example showing bounded loop in C.

```
int main() {
    int a = 7;
    for(i=1; i<=5; ++i){
        a = a / 3;
    }
    return 0;
}
```

Listing 3: Behavioral Code example showing unbounded loop in C.

```
int main() {
    int a = 15;
    while (a >= 2){
        a = a / 3;
    }
    return 0;
}
```

The program's iteration times for the second while loop in Listing 3 are determined by the computation results. The quantum circuit executes the body statement each time and utilizes the statement's outcome to evaluate the conditional statements, which determine if the unbounded loop has completed its computation, as depicted in Figure 11. The computation continues until the stopping conditional statement is satisfied. More detailed explanations of these types of loops can be found in Sections IV-D1 and IV-D2 correspondingly.

1) Bounded Loops: When quantum circuits involve loops, they need to be measured to determine whether the program should continue executing the loop. However, this approach is inefficient as each iteration of the loop requires communication between the quantum processor and classical processor for measurement. Additionally, multiple quantum circuits may be required to perform quantum entanglement.

To design a quantum circuit with bounded loops, the number of iterations can be readily determined without relying on the computation results from the loop's body statements (as depicted in Figure 11 (a)). In such cases, optimizations can be employed to consolidate the repeated statements into a single quantum circuit. These optimizations are typically performed at a higher level, directly analyzing the original C code, examining variables, and unrolling the bounded loops. By reducing the number of iterations, the computation can be completed with just one execution of the quantum circuit, eliminating the need to measure the result after each iteration. After optimization, the loop count is converted into a quantum circuit representation for computation. This allows the quantum circuit to proceed with the computation without requiring measurement at each iteration, continuing until the end of the computation. Consequently, the statement clauses can be generated and appended to the previous quantum circuits, completing the computation efficiently.

2) Unbounded Loops: In classical computers, HLS utilizes various approaches to implement unbounded loops, including using HLS pragmas to allow providing hints to the loop behaviors. Some HLS optimization algorithms are performed to get rid of unbounded loops by moving them to a software layer where it is easier to address with a CPU [14].

Unbounded loops pose a challenge as the calculation of the loop body is required to determine when the loop should terminate. As a result, it is not easy to determine the number of iterations for unbounded loops through high-level optimization or loop unrolling techniques. When measurement is necessary within unbounded loops, conditional constructs are employed to determine if the loop has completed its computation. This implementation involves iterating the conditional statements discussed in section IV-C, as illustrated in Figure 11 (c). Due to the difficulty of measuring the quantum state during the intermediate stages of quantum circuits, we propose a

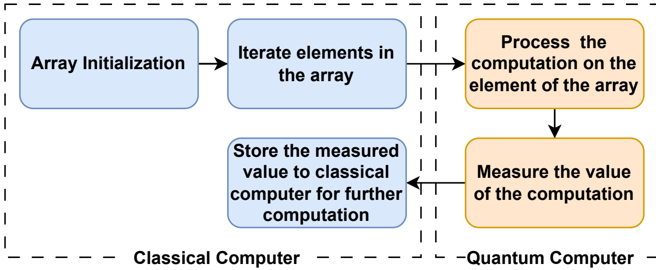


Fig. 12: Workflow of quantum array operation demonstration.

hybrid framework that combines both classical and quantum computers to complete this task. In this framework, the quantum computer performs the computational operations within the loop body and returns the desired variable for the conditional judgment to decide whether the iteration should continue. Thus, the computation is accomplished through the collaboration of classical and quantum computers. An example code showcasing an unbounded loop is demonstrated in Section IV-G.

E. Arrays

Within the QHLS framework, we have incorporated array functions. However, it is not efficient to store arrays directly in a quantum computer due to the instability of qubits for information storage. The presence of noise in a quantum computer further complicates the storage of information. Therefore, we store the array in a classical computer and transfer the required information into the quantum computer when computation is needed. The workflow of this computation process is depicted in Figure 12. In this framework, each element in the array is initialized and the computation is performed on the quantum computer. The resulting values are then stored back in the classical computer. For a simple C code example illustrated in Listing 4, the classical computer feeds each element into the quantum computer. The quantum circuit remains unchanged except for the initialization of values.

Listing 4: The behavioral code for array operation.

```
int main() {
    int myNumbers[] = {1, 2, 3, 4};
    for ( int i = 0; i < 10; i++ ) {
        myNumbers[ i ] = i + 5;
    }
    return myNumbers;
}
```

F. Qubits Resource Determination

In order to automatically generate quantum circuits using QHLS, it is essential to determine the necessary qubit resources. This information is crucial for organizing the quantum gates within the circuit effectively. Our proposed QHLS framework employs three distinct scenarios to ascertain the required qubits. The first scenario focuses on input variables. The number of qubits is determined by the count of input variables present in the program. The second scenario arises during calculations, specifically for multiplication and division operations. In multiplication circuits, there must be enough available qubits to store the product of the multiplication operation. Similarly, division circuits require empty qubits to store the quotient. The third scenario revolves around logical operations, including computations involving conditional statements. To determine the required number of qubits for a given high-level program, one needs to consider the count of inputs and operations that will be executed until the quantum circuit produces the output. By considering these three scenarios, QHLS is capable of accurately determining the appropriate number of qubits required for automatic quantum circuit generation.

TABLE I: Total gate count of quantum circuits with n qubit length.

Quantum Algorithms	X gate	CX gate	CCX gate
Adder	0	$5n+6$	$2n-2$
Subtractor	$2n$ or $3n$	$5n+6$	$2n-2$
Quantum Modulus	$2n^2$	$7n^2-11n$	$5n^2-4n$
Multiplication	0	$4n^2-10n+6$	$3n^2-n-1$
And Logic	0	$2n$	n
Or Logic	0	0	n

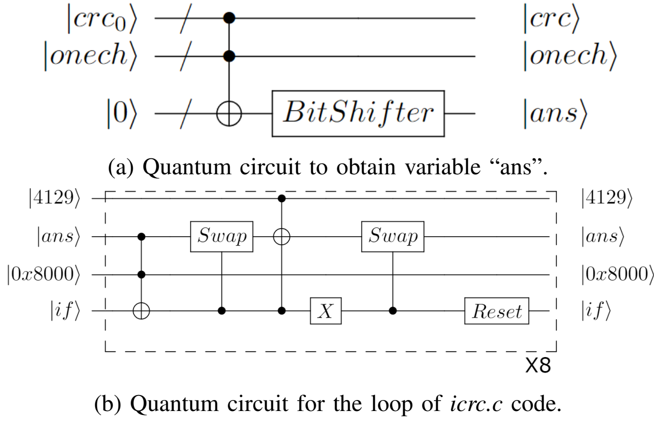
Table I shows the quantum resource required for each operation concerning the qubit length of the quantum circuits. From the table, the multiplication and integer division circuit increases in the complexity of $O(n^2)$ for qubit length, and the other operations increase linearly.

G. Examples

The QHLS framework utilizes Python script to parse the command lines from the C language to generate the quantum circuits that correspond to the operation functions. The connection between each operator appends the quantum operators on the qubit where the qubits are assigned for different variables or empty qubits to assist with the computation. For the specific synthesis methodology, the proposed QHLS will parse the names of variables and assign each variable to certain qubits. The quantum operators can be appended to the corresponding qubits to finish the required computations. Extra qubits are imported if the required operation requires ancilla qubits to assist with the computation. For the unbounded loops, we will generate the quantum circuits corresponding to body operators, and measure the corresponding termination criterion to determine the next step of operations.

The proposed QHLS framework simplifies the creation of quantum circuits that correspond to logical operations, arithmetic calculations, and conditional statements on a quantum computer. By utilizing these three circuit types in combination, it is possible to produce a wide range of quantum circuits capable of performing complex tasks. To illustrate the potential of our proposed QHLS framework, we will provide two example quantum circuit designs in this section, *ICRC* and *GCD* benchmark code that is originally written in C language.

1) ICRC Benchmark Circuit: The *ICRC* circuit is utilized to evaluate the performance of the HLS software with bit operations in a bounded loop. Listing 5 displays the C programming language that we will perform the *ICRC* benchmark circuit. To begin with, the QHLS examines the original code by



(a) Quantum circuit to obtain variable "ans".

(b) Quantum circuit for the loop of *icrc.c* code.

Fig. 13: Quantum *ICRC* circuit.

expanding the loops. In this instance, the loop is executed eight times. The variable i used for iteration is taken out of the code once the number of iterations is determined. Then, the variable *'ans'* is calculated by performing bitwise operations such as logical 'XOR' and bit shifting. Quantum SWAP gates can be utilized to perform the bit-shifting operation. The QHLS resets the first 7 bits of the variable and swaps the first bit with the 8th bit, the second bit with the 9th bit, and so on until all bits have been swapped. The quantum circuit needed to obtain the variable *ans* is depicted in Figure 13a.

Listing 5: The behavioral benchmark code "*icrc.c*".

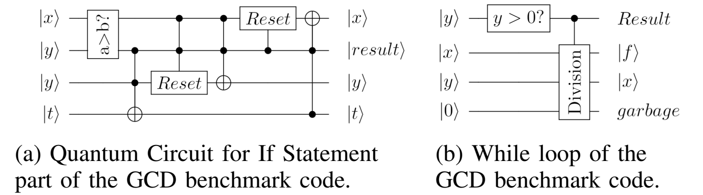
```
unsigned short icrc(unsigned short crc, unsigned char onech
    )
{
    int i;
    unsigned short ans=(crc^onech << 8);

    for (int i=0;i<8;i++) {
        if (ans & 0x8000)
            ans = (ans << 1) ^ 4129;
        else
            ans <<= 1;
    }
    return ans;
}
```

Afterward, we will make use of the proposed quantum *if-else* circuit, which has been explained in detail in Section IV-C, for the *if* statement. To perform the logic operation *ans* & 0x8000, we perform a bitwise AND operation between ans and 0x8000. Since only one bit in 0x8000 is significant to the result (the other bits are all zeros), we can use only one AND operator to perform the computation. This optimization significantly reduces the quantum gate overhead. The generated quantum circuit is displayed in Figure 13b. To construct the circuit, registers are required for all input variables, one bit for the *if* statement, and 32 bits for the logical 'XOR' operations. After the computation is completed, the circuit only needs to measure the qubits that correspond to the variable *ans* in order to acquire the desired output.

2) GCD Benchmark Circuit: The *GCD* benchmark circuit, depicted in Listing 6, showcases the use of an unbounded loop in determining the Greatest Common Divisor (GCD) through an if statement and a while loop. Initially, the quantum circuit executes an if statement to swap the values of variables *x* and *y* if *x* is found to be less than *y*. Subsequently, the quantum circuit proceeds with the modulus operation and stores the result for subsequent operations. The quantum circuit for the first if statement is illustrated in Figure 14a. Within the unbounded while loop, the quantum circuit performs the modulus computation, which can be realized using the integer division circuit, while retaining the remainder. The other two operations can be completed on a classical computer to finalize the computation. Thus, the while loop portion of the quantum circuit is presented in Figure 14b.

Listing 6: The behavioral benchmark code *gcd.c*.

```
int gcd(int x, int y )
{
   int t;
   if( x < y ) {
      t = y;
      y = x;
      x = t;
   }
   while( y > 0 ) {
      int f = x % y;
      x = y;
      y = f;
   }
   return x;
}
```



(a) Quantum Circuit for If Statement part of the GCD benchmark code.

(b) While loop of the GCD benchmark code.

Fig. 14: Quantum Circuit for the GCD benchmark circuit.

V. Experimental Evaluation

This section presents an evaluation of the proposed Quantum High-Level Synthesis (QHLS) using standard benchmark algorithms that involve arithmetic and logical operations in traditional High-Level Synthesis (HLS) frameworks like Bambu. We utilized four benchmark programs, namely *ARF*, *GSM_NORM*, *GSM_DIV* (i.e., two subfunctions of the *GSM* benchmark), *ICRC*, and *GCD* using QHLS. These high-level behavioral codes were used to assess the capabilities of QHLS and the quantum computer's resource requirements. QHLS is the first HLS framework for quantum circuits, Hence, there is no prior research to compare it to. However, the authors anticipate that future researchers will build on QHLS and improve its performance and efficiency.

The reason that we did not integrate Deutsch-Jozsa, Grover search, or Shor's factoring algorithms into the QHLS framework is that there remains a notable absence of classical modeling algorithms for quantum algorithms. Numerous quantum oracle designs are still being explored to achieve scalable and efficient automation. Therefore, we do not integrate the mentioned quantum algorithm in the QHLS framework.

Simulating complicated tasks on a quantum computer is a challenging task for a classical computer. At present, the Qiskit

TABLE II: Benchmark C code with program statistics and quantum resource requirement. From the code, it only requires a maximum of 2 seconds to generate the quantum circuits, showing the scalability of such computation.

Benchmark	Program Statistics						Quantum Resource Requirements					
	Variables	Loops	Iteration Count	If Statements	Arithmetic Operations	Logic Operations	Qubits	X	CX	CCX	SWAP	Time consumption on generating quantum circuit(s)
ARF	16	0	N/A	0	28	0	1056	0	13896	17529	0	2.00
GSM_NORM	11	0	N/A	5	5	7	484	175	857	534	48	1.15
GSM_DIV	8	1	15	16	30	46	272	960	7470	5642	930	0.72
ICRC	4	1	8	8	0	26	104	8	0	272	248	0.59
GCD	4	1	N/A	2	1	0	128	2028	6983	5150	0	N/A

TABLE III: Reduced benchmarks with quantum resource and simulation time of small quantum circuits.

Reduced Benchmark Circuits	Quantum Resource Requirements								
	Qubits	X	CX	CCX	SWAP	Circuit generation time (s)	Qiskit Simulation time (s)	Lines of code required for Qiskit (estimation)	Lines of code required for QHLS
Reduced ARF	27	0	116	49	0	0.24	0.16	165	3
Reduced GSM_NORM	24	22	108	534	69	0.13	0.11	733	3
Reduced GSM_DIV	28	60	430	218	8	0.15	0.10	716	3
Reduced ICRC	18	1	0	36	248	0.23	0.15	285	3
Reduced GCD	16	16	42	32	0	N/A	N/A	90	7

library only supports a maximum of 32 qubits for quantum simulations. However, regular integer representation needs 32 bits to represent a single number. As a result, even though our benchmarks are small for HLS, they are still too big for a quantum simulator. Therefore, we reduced the bit length to 4 and made some adjustments to the parameters to simulate our benchmark programs. We use quantum X gates to initialize the qubits to a binary number.

A. Results

Table II presents the quantum resource required for the corresponding benchmark code. The first column of the table displays the name of the benchmark program, followed by six columns showing the program statistics and four columns showing the resource requirements in terms of qubits and quantum gates. Columns 8 to 12 furnish the quantum resource requirements to generate the corresponding quantum circuits. The last column shows the time consumption for generating the corresponding quantum circuits in seconds. Table II shows the time consumption of the original circuit synthesis of standard quantum circuits to show the scalability of the proposed synthesis methodology. Table III demonstrates quantum resource requirement and simulation time consumption of the reduced benchmark code, which are based on the original benchmark circuit mentioned in Table II. The first column in Table III shows the name of the reduced benchmark circuit and columns 2 to 6 demonstrate the quantum gates required to generate the corresponding quantum circuit. Columns 7 and 8 demonstrate the time consumption for quantum circuit generation and quantum circuit simulation time. Notice that different input values of GCD benchmark circuit will influence the time consumption for circuit generation and simulation because of the uncertainty of iteration numbers. The last two columns demonstrate the lines of code required for Qiskit and QHLS framework. Reduced GCD requires a few more lines to deal with the unbounded loop and initialization of the next quantum circuit cycle. Based on Table III, the QHLS can reduce lines of code from the maximum of 716 to only 3 lines with around **238.6X** reduction of code requirement to generate the corresponding quantum circuit.

The values in Table II correspond to the actual resource requirements when operating on 32-qubit data. For instance, the *ARF* program requires 11 addition operations and 17 multiplication operations for computation. In addition, initializing the 16 input variables requires $16 \times 32 = 512$ qubits, and extra qubits are needed to store the product for the 17 multiplication operations. As a result, the total number of qubits required for *ARF* is $512 + 17 \times 32 = 1056$ qubits. Furthermore, for 32-bit inputs, each quantum adder necessitates 166 CX gates and 62 CCX gates, while the quantum multiplier necessitates 710 CX gates and 991 CCX gates to execute the program.

We conducted simulations for two subfunctions of *GSM*: *GSM_NORM* and *GSM_DIV*. For *GSM_NORM*, the quantum circuit includes one input variable, denoted as *'a'*, which is used seven times. The circuit also uses an integer value of -1073741824, four hexadecimal numbers (0xff000000, 0xff00, 0xFF, and 0xffff0000), and other integers such as 7, 15, 23, and 8, along with four *if* statements. To initialize the variables, we require $15 \times 32 = 480$ qubits. The four *if* statements require 32 CCX gates and four qubits for conditional operations. Moreover, 128 X gates and $2 \times 127 = 254$ CCX gates and 1 CX gate are required to perform the logical computation. To generate the quantum circuit, we need 43 X gates, 760 CX gates, 248 CCX gates, and 48 SWAP gates for value initialization, logical operation, and calculation, as shown in Table II.

For *GSM_DIV*, QHLS performs a high-level optimization that determines the number of loop iterations and unrolls the loop accordingly. To process two variable inputs, 32 bits are needed for each input, and for two longword format inputs, 64 bits are required for each. In addition, an internal variable called *"div"* needs 32 bits for initialization. The circuit also requires two sets of bit shifting operations, one *if* statement with a comparator, a subtractor, and an adder for each loop to execute. Therefore, for *GSM_DIV*, a total of 7470 CX gates, 5642 CCX gates, and 930 SWAP gates are necessary to

complete the computation.

For *ICRC*, the circuit design is illustrated in Figure 13. The circuit utilizes XOR logic operations for the variables *crc* and *onech* and projects the output on the variable *ans*. A bit-shifter function block consisting of SWAP gates generates the variable *ans*. For each bit, two CX gates and one CCX gate are necessary to perform a *logic XOR* operation. The loop contains one logic operation, two sets of *controlled-SWAP* operations, one CCX operation, one X gate, and a reset gate to reset the if statement. All of these statements are repeated eight times, requiring a total of 8 X gates, 32 CX gates, 272 CCX gates, 248 SWAP gates, and 8 reset gates to complete the computation.

In the case of the *GCD* benchmark circuit, executing it as an unbounded loop poses challenges for the quantum circuit. Without classical control gates, it becomes difficult for the quantum circuit to perform this task efficiently. To overcome this limitation, we employ multiple instances of the same quantum circuit and return the desired result when the algorithm reaches its termination condition. The original C code for this benchmark circuit includes two distinct quantum circuits. The first circuit is responsible for swapping the values of variables x and y if x is greater than y. This quantum circuit requires four registers to complete the computation. On the other hand, the classical computer requires an additional register to achieve the same computation. By utilizing SWAP gates, the quantum computer can perform the computation without the need for extra registers. Consequently, executing the C code on a quantum computer without specific code optimization for generating a quantum circuit incurs additional resource consumption. As for the while loop, it primarily consists of the logical operation of checking whether y is greater than 0 and the integer division calculation within the quantum circuit. Therefore, the quantum circuit requires four variables, an unbounded loop, two conditional statements (including the unbounded while loop), and one arithmetic operation to finalize the computation. The corresponding quantum circuit for this benchmark circuit requires 128 qubits, 2028 X gates, 6983 CX gates, and 5150 CCX gates to complete the computation.

VI. Conclusion

This paper introduces a novel Quantum High-level Synthesis (QHLS) framework that automates the conversion of high-level behavioral languages into quantum circuits, which can alleviate the burden of programming more complex quantum circuits. To the best of our knowledge, this is the first approach towards High-level synthesis of quantum computing, with various quantum arithmetic, logical operations, conditional statement circuits, bounded loops, unbounded loops, as well as arrays that correspond to software language constructs. We evaluated our framework on HLS benchmark programs using a quantum simulator, considering the limitations of qubits and noise levels on current quantum hardware. To limit the complexity, only small circuits were analyzed. In the future, we aim to enhance the QHLS framework by reducing the quantum resource usage and introducing quantum algorithms for more lightweight circuit design and generation, as well as improving this technique by adding pragmas and optimization directives, which are similar to traditional HLS.

VII. Acknowledgment

This research is supported by NSF grant #2228725.

References

[1] M. A. Nielsen *et al.*, *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.
[2] F. Arute *et al.*, "Quantum Supremacy using a Programmable Superconducting Processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019.
[3] T. Monz *et al.*, "Realization of a Scalable Shor Algorithm," *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.
[4] P. Jurcevic *et al.*, "Demonstration of quantum volume 64 on a superconducting quantum computing system," *Quantum Science and Technology*, vol. 6, no. 2, p. 025020, 2021.
[5] J. M. Pino *et al.*, "Demonstration of the qccd trapped-ion quantum computer architecture," *arXiv preprint arXiv:2003.01293*, 2020.
[6] R. D. Somma *et al.*, "Quantum speedup by quantum annealing," *Physical review letters*, vol. 109, no. 5, p. 050501, 2012.
[7] S. Takeda *et al.*, "Toward large-scale fault-tolerant universal photonic quantum computing," *APL Photonics*, vol. 4, no. 6, p. 060902, 2019.
[8] IBM Research, "Qiskit: Open-Source Quantum Development," Accessed: March 2022. [Online]. Available: https://qiskit.org.
[9] V. Omole *et al.*, "Cirq: A python framework for creating, editing, and invoking quantum circuits," 2020.
[10] S. Sivarajah *et al.*, "t— ket¿: a retargetable compiler for nisq devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014003, 2020.
[11] F. Ferrandi *et al.*, "Bambu: an open-source research framework for the high-level synthesis of complex applications," in *2021 IEEE/ACM DAC*, 2021.
[12] S. Lahti *et al.*, "Are we there yet? a study on the state of high-level synthesis," *IEEE TCAD*, vol. 38, no. 5, pp. 898–911, 2018.
[13] D. Pursley *et al.*, "High-level low-power system design optimization," in *2017 VLSI-DAT*. IEEE, 2017, pp. 1–4.
[14] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: Case studies," *ISSC 2014*, 2014.
[15] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, "Hw/sw co-design toolset for customization of exposed datapath processors," *Computing platforms for software-defined radio*, pp. 147–164, 2017.
[16] Z. Alwardi, R. Wille, and R. Drechsler, "Synthesis of reversible circuits using conventional hardware description languages," in *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE, 2018, pp. 97–102.
[17] S. Adarsh, L. Burgholzer, T. Manjunath, and R. Wille, "Syrec synthesizer: An mqt tool for synthesis of reversible circuits," *Software Impacts*, vol. 14, p. 100451, 2022.
[18] A. Zulehner and R. Wille, "One-pass design of reversible circuits: Combining embedding and synthesis for reversible logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 996–1008, 2017.
[19] T. Huff, H. Labidi, M. Rashidi, L. Livadaru, T. Dienel, R. Achal, W. Vine, J. Pitters, and R. A. Wolkow, "Binary atomic silicon logic," *Nature Electronics*, vol. 1, no. 12, pp. 636–643, 2018.
[20] H. Thapliyal, "Mapping of subtractor and adder-subtractor circuits on reversible quantum gates," in *TCS*. Springer, 2016, pp. 10–34.
[21] E. Muñoz-Coreas *et al.*, "Quantum circuit design of a t-count optimized integer multiplier," *IEEE TC*, vol. 68, no. 5, pp. 729–739, 2018.
[22] H. Thapliyal *et al.*, "Quantum circuit designs of integer division optimizing t-count and t-depth," *IEEE TETC*, pp. 1045–1056, 2019.
[23] P. Gokhale, J. M. Baker, C. Duckering, F. T. Chong, N. C. Brown, and K. R. Brown, "Extending the frontier of quantum computers with qutrits," *IEEE Micro*, vol. 40, no. 3, pp. 64–72, 2020.
[24] C. Lu *et al.*, "Design and logic synthesis of a scalable, efficient quantum number theoretic transform," in *ACM/IEEE ISLPED*, 2022, pp. 1–6.
[25] V. Bhatia *et al.*, "An efficient quantum computing technique for cracking rsa using shor's algorithm," in *IEEE ICCCA*, 2020, pp. 89–94.
[26] J. Hooyberghs and J. Hooyberghs, "Azure quantum," *Introducing Microsoft Quantum Computing for Developers: Using the Quantum Development Kit and Q#*, pp. 307–339, 2022.
[27] B. Bichsel *et al.*, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 286–300.

[28] R. Nane *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE TCAD*, vol. 35, no. 10, pp. 1591–1604, 2015.

[29] X. Ma *et al.*, "The application of wi-fi rtls in automatic warehouse management system," in *2011 ICAL*. IEEE, 2011, pp. 64–69.

[30] S. Dai *et al.*, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *IEEE FCCM*, 2018, pp. 129–132.

[31] F. Winterstein *et al.*, "High-level synthesis of dynamic data structures: A case study using vivado hls," in *IEEE FPT*, 2013, pp. 362–365.

[32] J. Zhao *et al.*, "Machine learning based routing congestion prediction in fpga high-level synthesis," in *IEEE DATE*, 2019, pp. 1130–1135.

[33] D. Soni *et al.*, "Power, area, speed, and security (pass) trade-offs of nist pqc signature candidates using a c to asic design flow," in *IEEE ICCD*, 2019, pp. 337–340.

[34] A. Tumeo *et al.*, "Hw/sw methodologies for synchronization in fpga multiprocessors," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 265–268.

[35] H. Thapliyal, "Mapping of subtractor and adder-subtractor circuits on reversible quantum gates," in *TCS*. Springer, 2016, pp. 10–34.

[36] E. Muñoz-Coreas *et al.*, "Quantum circuit design of a t-count optimized integer multiplier," *IEEE TC*, vol. 68, no. 5, pp. 729–739, 2018.