Towards High-Level Synthesis of Quantum Circuits

Chao Lu
Dept. of Electrial & Computer Engineering
University of Texas at Dallas
Richardson, TX, USA
cxl200053@utdallas.edu

Christian Pilato
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
christian.pilato@polimi.it

Kanad Basu
Dept. of Electrial & Computer Engineering
University of Texas at Dallas
Richardson, TX, USA
kxb190012@utdallas.edu

***Abstract*—In recent years, there has been a proliferation of quantum algorithms, primarily due to their exponential speedup over their classical counterparts. Quantum algorithms find applications in various domains, including machine learning, molecular simulation, and cryptography. However, extensive knowledge of linear algebra and quantum mechanics are required to program a quantum computer, which might not be feasible for traditional software programmers. Moreover, current quantum programming paradigm is difficult to scale and integrate quantum circuits to achieve complex functionality. To this end, in this paper, we introduce QHLS, a quantum high-level synthesis (HLS) framework. To the best of our knowledge, this is the first HLS framework for quantum circuits. The proposed QHLS allows quantum programmers to start with high-level behavioral descriptions (*e.g.*, C, C++) and automatically generate the corresponding quantum circuit; thus, reducing the complexity of programming a quantum computer. Our experimental results demonstrate the success of QHLS in translating high-level behavioral software programs containing arithmetic, logical, and conditional statements.**

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

I. Introduction

Quantum Computing can expedite the performance of several computational tasks compared to classical CMOS-based computers. Specifically, quantum entanglement and superposition empower a quantum computer to compute more efficiently than a classical computer. There has been a plethora of research that explores the quantum advantages by developing new algorithms that have exponential speed-up over their classical counterparts [1]. For example, Shor's algorithm demonstrated that a quantum computer could factorize large numbers in polynomial time, which could potentially break current encryption standards [2], [3]. Various scientific approaches, including superconducting, trapped ion, quantum annealing, and photonics can generate quantum entanglement and superposition efficiently [4]–[7].

Many quantum computing platforms, including Qiskit, Cirq, and Tket, are integrated as Python libraries that generate quantum circuits at quantum-gate-level [8]–[10]. These platforms utilize many grammars and suit different quantum computing systems. However, implementing high-level programming logic in a quantum computer requires additional knowledge of quantum mechanics and linear algebra to furnish an efficient quantum circuit. Acquiring this knowledge might not be feasible for traditional software programmers. The problem exacerbates when designing complex quantum circuits. Thus, it is imperative to develop a framework that can reduce the complexity of programming a quantum computer. To this end, we propose an approach based on high-level synthesis concepts for quantum computers.

High-Level Synthesis (HLS) is widely applied in CMOS-based hardware design processes to translate a behavioral specification into the corresponding Register-Transfer-Level (RTL) description that implements such behavior. HLS tools usually accept C/C++ code as the input. Such descriptions include multiple behavioral instructions, including arithmetic-logical operations, conditional statements, and loops that are eventually translated into their RTL counterparts [11]. Many commercial or academic HLS frameworks, including Vivado HLS, Stratus HLS, Bambu, and HDL Coder, use C/C++ and Matlab to generate the RTL code [12], [13]. HLS improves the efficiency of hardware design and reduces the complexity of designing sophisticated hardware.

In this paper, for the first time, we introduce the Quantum HLS (QHLS) framework that generates quantum circuits from high-level software languages like C. To the best of our knowledge, currently, there are no HLS framework to generate quantum circuits from high-level behavioral languages. The proposed QHLS framework will aid designers, starting from a high-level description, in generating a quantum circuit design without expert knowledge of quantum mechanics.

In our framework, we modified the flow for designing a quantum circuit. To begin with, we take a high-level behavioral code as input. Next, we use a Python framework that parses this behavioral description. The QHLS framework generates a file in the Open Quantum Assembly (OpenQASM) language that describes the quantum circuit at the gate level. This OpenQASM description is compatible with all current quantum programming tools. A designer can use a quantum programming platform like Qiskit to execute the computation with the generated circuit. To this end, our major contributions:

- We proposed a Quantum High-Level Synthesis (QHLS) framework, which translates behavioral programming language to a corresponding quantum circuit.
- We proposed quantum circuit primitives to emulate arithmetic circuits in Section III-A, logic operations in Section III-B, conditional statements in Section III-C, and loops in Section III-D in the QHLS framework.
- We evaluated our proposed QHLS flow on small high-level benchmark programs, used by traditional HLS

This work is supported by the National Science Foundation (OMA-2228725). (*Corresponding Author*: *Chao Lu*, Email: *Chao.Lu@utdallas.edu*).

978-3-9819263-7-8/DATE23/© 2023 EDAA

frameworks [13]. Since real quantum computers only contain limited noisy qubits [14], and classical computers cannot simulate quantum circuits efficiently, while being limited by the number of qubits, we utilized tailored benchmark programs that can be simulated on a noise-free quantum simulator to evaluate our QHLS framework. We also estimated the quantum resources required for these benchmark programs, as shown in Section IV.

The rest of this paper is organized as follows. Section II provides background on quantum computing, including the introduction of qubits and quantum gates, along with a discussion of programming issues. Section III explains the methodology for designing the proposed QHLS flow. Section IV presents our experimental results. Finally, Section V concludes the paper with possible future research directions.

II. Background and Motivation

In this section, we will introduce quantum circuits and the motivation for developing QHLS.

A. Quantum Circuit

A quantum circuit comprises *quantum bits* (qubits) and *quantum gates*. In contrast to a classical bit, the qubit can form a superposition state, and it is usually expressed as a bra-ket form, written as $|\rangle$. A qubit in a superposition state can be expressed as $|a\rangle$, where $a = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$, where $\alpha^2 + \beta^2 = 1$. The state space of a single qubit can be in any form from 0 to 1, and it is geometrically represented using a Bloch Sphere, as shown in Figure 1. The x-y plane of the Bloch Sphere represents real states, and the z-axis represents the imaginary part. While representing the state of a single qubit, the basis states are referred to as the opposite points on Bloch Sphere, as seen in Figure 1.

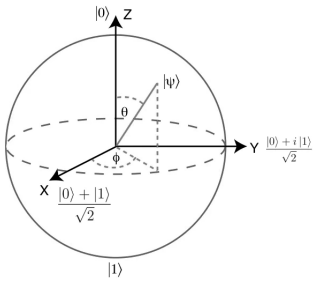


Fig. 1: Bloch Sphere Single Qubit Visualisation.

Quantum entanglement is a quantum mechanical phenomenon whereby a state change in one qubit instantaneously changes the state of others in an anticipated fashion. A pair of qubits are connected by entangling them together. They correspond to each other such that if the measurement value of one is known, the state of another qubit is determined by the state of the measured qubit. Quantum computers utilize such phenomena as two-qubit gates like CX gate to perform computation. Based on the specification of quantum superposition and entanglement, two qubits can generate 4 parameters, *e.g.* $|ab\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$, where $\alpha^2 + \beta^2 + \gamma^2 + \delta^2 = 1$. Therefore, the parameters scale exponentially with a linear increase of qubits, known as Hilbert Space [15]. Such specifications can speed up specific tasks over classical algorithms exponentially [3].

When designing quantum circuits, each quantum gate can be expressed as a matrix form, and the computation of the quantum circuits is expressed as a matrix multiplication. A tensor product is required for multiple qubits in the quantum circuits. However, the quantum gates can perform logic operations similar to CMOS-based classical computers. For instance, a Pauli-X gate flips the phase of the qubits from $|0\rangle$ to $|1\rangle$ or from $|1\rangle$ to $|0\rangle$. The Control-X gate (CX gate) utilizes a control and a target bit. It flips the target bit if the control bit is $|1\rangle$. The Toffoli gate (CCX gate) contains two control bits and a target bit. The CCX gate flips the target bit when both control bits are one. A SWAP gate swaps the states of two qubits, so the information on those two qubits is interchanged.

When measuring a quantum circuit, the phase of the qubits will collapse into "0" or "1" depending on the phase of the measured qubits. Multiple measurements are required for the quantum circuit to estimate the original qubit phase. When a qubit in phase $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ is measured, the probability of obtaining zero is α^2 and one is β^2, such that $\alpha^2 + \beta^2 = 1$.

B. Programming Quantum Circuits and High-Level Synthesis

Quantum computing is promising due to its exponential computing power with a linear increase in the number of qubits. Researchers have proposed several quantum circuits to perform integer arithmetic calculations [16]–[20]. Current quantum programming languages only facilitate gate-level programming. Most quantum programming platforms, including Qiskit, Cirq, and Tket [8]–[10], require the direct design of qubits and quantum gates to achieve a certain function. When using these programming tools, the program includes the number of qubits required by the quantum circuit initialization. Next, the program uses the quantum gates to generate the circuits. However, gate-level programming is inefficient for generating complex quantum circuits. Moreover, it requires designers to have expert knowledge of quantum mechanics and linear algebra. To address this issue, we propose QHLS, which takes inspiration from the improvements to the CMOS-based hardware design steps achieved by raising the abstraction level to high-level synthesis.

High-Level Synthesis (HLS) is a popular Electronic Design Automation (EDA) technique that automates the generation of RTL code from high-level software descriptions ($e.g.$, C, C++, etc.) [21]. It facilitates hardware design since the same software code can be used to generate multiple RTL descriptions [22]. HLS has been utilized in several application domains, including computer vision, machine learning, and hardware security [23], [24]. It is mainly composed of three phases: compilation of the input program, creation of the micro-architecture, and generation of the Hardware Description Language (HDL) description ready for logic synthesis.

In this work, we aim to raise the programming abstraction level and use software language elements like arithmetic and

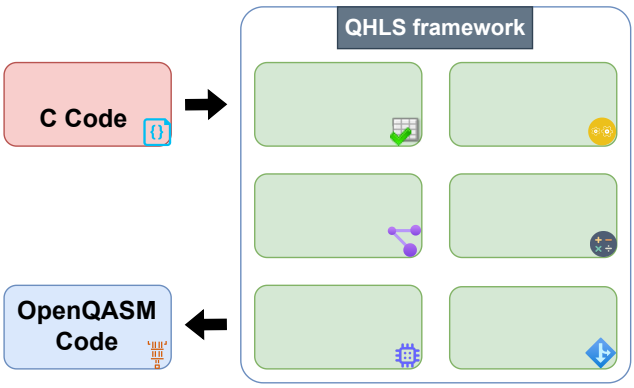


Fig. 2: Overall QHLS Workflow.

logical operations, conditional statements, and loops for generating quantum circuits, akin to HLS. Despite the popularity of HLS frameworks [12], [13], the development of such a framework for quantum circuits is still missing. Our proposed QHLS addresses this issue by enabling direct synthesis from the behavioral coding language to a quantum circuit without requiring additional knowledge from the designer.

III. Proposed QHLS

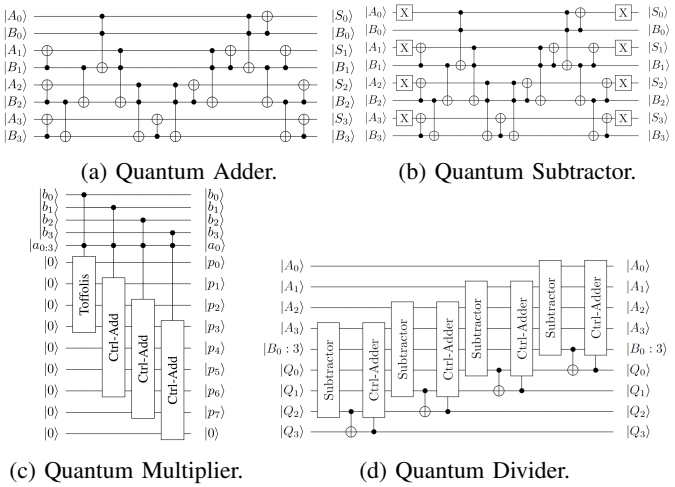
We aim to develop a design framework that utilizes high-level behavioral software languages to generate complicated quantum circuits, like in HLS. Our QHLS framework can lower the barrier and hence, increase the efficiency of programming on a quantum computer. It utilizes existing high-level programming languages to automatically design the gate-level quantum circuit corresponding to the input specifications to perform the computation on a quantum computer.

Figure 2 shows the proposed QHLS framework. First, It parses the input C code to determine the induction variables and unrolls the iteration loops when possible. This step corresponds to the classic *front-end* phase in traditional HLS frameworks. Next, we claim the qubits requirement and assemble the quantum gates to achieve specific functions. This phase corresponds to the HLS engine. Finally, the QHLS will generate an OpenQASM file that describes the corresponding quantum circuits (*backend phase*). The generated OpenQASM file defines the details of qubits and quantum gates design and is compatible with most of the existing quantum programming platforms like Qiskit, Cirq, and Tket. The QHLS does not perform any logic and layout synthesis on the quantum circuits, since these steps can be performed easily by current quantum computing platforms.

In this section, we describe several quantum circuit primitives for traditional high-level computations, including arithmetic, logical and conditional statements to be used in our proposed QHLS flow.

A. Quantum Arithmetic Circuits

First, we focus on quantum arithmetic operations. To this end, we developed the following quantum circuits: quantum adder, quantum subtractor, quantum multiplier, and quantum divider (see Figure 3). The proposed QHLS utilizes existing



(a) Quantum Adder. (b) Quantum Subtractor.

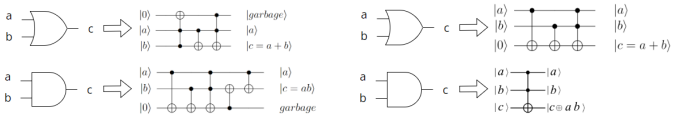
(c) Quantum Multiplier. (d) Quantum Divider.

Fig. 3: Quantum arithmetic blocks.

high-level descriptions to generate these quantum circuits to perform arithmetic computations on a quantum computer [16]–[18]. The quantum adder and subtractor utilize two binary numbers as input, and the generated output replaces one of the input registers. The quantum integer multiplier and quantum division circuits utilize two input numbers and reserve the qubits for the multiplication product and division quotient, respectively. Since some arithmetic operations replace the output with some input registers, CX gates are required to copy the information that will be replaced by the new qubits, so that the information can be utilized for other operations.

B. Logic Operation

In contrast to classical computers, quantum computers utilize quantum logic gates that are different from classical logic gates. Although simple logical operations, including "logic AND" and "logic OR" are not available on a quantum computer, they can still be implemented by combining several quantum gates to realize logical operations. Figure 4 demonstrates two versions of logic operations of "logic AND" and "logic OR" using quantum circuit elements.



(a) Quantum logic operation with result appended on a new register.

(b) Quantum logic operation with result that replaces one input register.

Fig. 4: Quantum logic operation equivalence.

C. Conditional Statement

One of the most important logical operations for high-level programming is the conditional statement. Conditional statements enable different computations upon the evaluation of a condition. To this end, we utilize the quantum entanglement phenomenon for representing conditional statements in QHLS. The control bit enables different computations of the target quantum gate, thus, performing an *if-then* operation, as shown in Figure 5a. Furthermore, QHLS also designs the quantum

comparators utilizing the quantum subtractor, described in Section III-A. For conditional statements, we have multiple scenarios, including $a == b$, $a \neq b$, $a < b$, $a > b$, $a \geq b$, and $a \leq b$, where some cases could be merged together. In this section, first, we demonstrate two important situations, namely, $a == b$, and $a > b$. The others can be derived from these scenarios, as explained later.

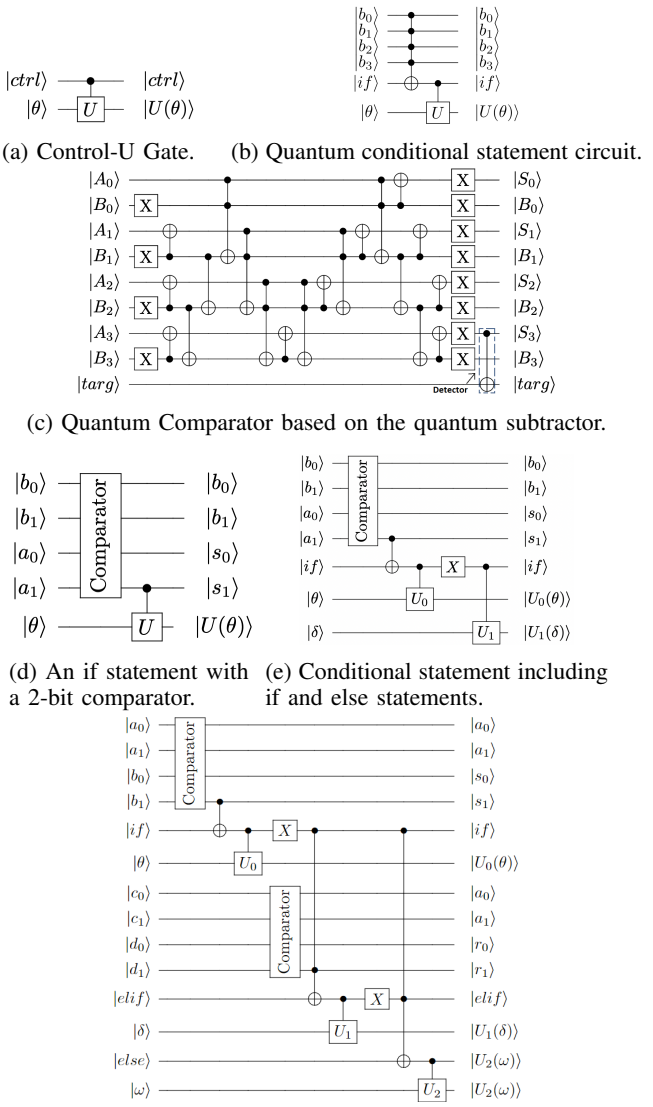
For the simplest scenario, if $a == b$, the function can be achieved using the MCX gate, as shown in Figure 5b. First, the input a is encoded as a binary state. Next, the original register encodes $\bar{b}$ again at the same register. After the encoding, the register value should be all 1s if the two variables are equal. The MCX gate is applied to the original register, and the comparison results are furnished to an ancilla qubit to perform further computation.

For scenarios like $a > b$, a different representation is necessary, so that the quantum circuit can compare the two values. The design of a relevant quantum comparator is shown in Figure 5c. In this case, we use a quantum subtractor to achieve such functionality. The quantum subtractor utilizes $one's$ complement code to perform the computation. In this case, we perform the subtraction of the two inputs, $i.e.$, $a - b$. If $a > b$, the Most Significant Bit (MSB) of the result should be 0; if $a < b$, the result should be negative, which means the MSB is 1. A quantum circuit that performs such an operation is shown in Figure 5d.

If we want to perform the computation when $a < b$, the quantum circuit will compare a and b first and project the result to the target bit. Next, a controlled-U gate is appended to the target bit of the quantum circuit to perform the computation. For cases like $a \geq b$, the computation can be executed by calculating $a < b$ and projecting the result to the target bit. An X gate can be appended to the target bit to flip the state to compute *"a is not less than b"*. The rest of the circuit remains unchanged. Next, we will introduce several other quantum circuits to demonstrate various types of conditional statements used in behavioral descriptions.

1) If-else statement: In this section, we introduce quantum circuits that can perform the *if-else* statement, which is also compatible with the comparator. In this scenario, a qubit is required to perform the *if* statement computation. The qubit for the *if* statement switches between 0 and 1 to disable or enable the computation of the target function blocks. The quantum circuit, for the high-level code *"If $a > b$: then Operation U_0; else: Operation U_1"*, is demonstrated in Figure 5e.

2) If-elif-else statement: The *if-elif-else* statement contains two conditional branches, which are *"if statement"* and *"else if (elif) statement"*. Since one *"if"* qubit can only control one conditional statement, another qubit is utilized to help with the computation on the second conditional statement. The *if-elif-else* statement with the comparator integrated is demonstrated in Figure 5f. As the figure shows, there are two comparators and three qubits for the conditional statements. To simplify the computation of the *else* statement, another qubit is utilized.



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

(c) Quantum Comparator based on the quantum subtractor.

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

(f) Conditional statements including if, else if, and else statements.

Fig. 5: Quantum conditional statement variants.

D. Iteration Loops

Since the states of qubits are unknown until they are measured, the quantum circuits containing a loop must be measured to decide whether the program should finish the loop. However, such an operation is inefficient because each iteration of the loop involves the measurement of the quantum circuit, which requires communication between the quantum processor and the classical processor. To this end, for the original C code, first, we will perform high-level optimizations to analyze the induction variables and unroll the loops. Next, the loop count will be translated to a quantum circuit representation to perform the computation so that the quantum circuit does not require measurement for each iteration.

E. Qubit Resource Determination

To automatically generate the quantum circuits using QHLS, a critical process is the qubit resource determination, which enables the program to arrange the quantum gates in the

```
unsigned short icrc1(unsigned short crc,
unsigned char onech)
{
    int i;
    unsigned short ans=(crc^onech << 8);

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

Fig. 6: The behavioral benchmark code "*icrc.c*".

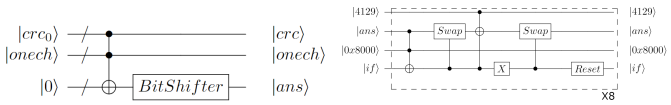
quantum circuit. For our proposed QHLS, three scenarios are required to determine new qubits. The first one is input variables. The second situation is during the calculation for each multiplication and division operation. For a multiplication circuit, the product of the multiplication operation requires empty qubits to store the product. The division circuit also requires empty qubits to store the quotient. The third situation involves logical operations, including conditional statement computation. Thus, the amount of qubit requirement for a particular high-level program can be obtained by counting the number of inputs and operations executed.

F. Example

Our proposed QHLS facilitates the design of quantum circuits corresponding to logical operations, arithmetic calculations, and conditional statements on a quantum computer. By combining these three types of circuits, we can generate a fairly large amount of quantum circuits to achieve various complex functionalities. In this section, we will demonstrate an example quantum circuit design using the proposed QHLS framework. Figure 6 shows a small benchmark program, called *ICRC*, written in C language.

First, the QHLS analyzes the original code by unrolling the loops. In this case, the loop is iterated eight times. The induction variable "i" is removed after determining the number of iterations. Next, the variable *"ans"* is computed using "logical XOR" and bit shifting operations. The bit shifting operation can be implemented using quantum SWAP gates. The QHLS resets the first 7 bits, and it swaps the first bit with the 8-th bit, the second bit with the 9-th bit, and so on until all bits are swapped. The quantum circuit to obtain the variable *"ans"* is shown in Figure 7a.

Next, for the *if* statement, we will use the proposed quantum if-else circuit, described in Section III-C1. For the logic operation *"ans & 0x8000"*, bitwise AND operation between ans and 0x8000 is utilized. Since only one bit is important to the result (only one bit in 0x8000 is one), while the other bits are zeros, we only need one AND operator for the computation. Such optimization reduces the quantum gate overhead drastically. Figure 7b shows the generated quantum circuit. For the generated circuit, it requires registers for all input variables, one bit for the *if* statement, and 32 bits for the "logic XOR" operations. After the computation is finished,



(a) Quantum circuit to obtain variable "ans"

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

Fig. 7: Quantum *ICRC* circuit.

the circuit only needs to measure the qubits corresponding to the variable *"ans"* to obtain the desired output.

IV. Experimental Results

A. Experimental Setup

In this section, we evaluate the proposed QHLS using simple HLS benchmark algorithms involving arithmetic and logical operations. These benchmarks have been used in traditional HLS frameworks like Bambu [13]. We constructed the following three benchmark programs applying QHLS: *ARF*, *GSM_NORM* and *GSM_DIV* (i.e., two subfunctions of the *GSM* benchmark), and *ICRC*. These high-level behavioral codes are utilized to evaluate the capability of QHLS and the resource requirement on a quantum computer. QHLS is the first HLS framework for quantum circuits, so comparison with prior research is impossible. However, we expect future researchers to build on the proposed QHLS and improve performance and efficiency.

It is difficult for a classical computer to simulate complex operations on a quantum computer. Indeed, the maximum number of qubits available for the Qiskit library is 32 qubits, while the ordinary integer representation requires 32 bits to express a single number. Hence, although our benchmarks are small for HLS, they are already too large for a quantum simulator. Thus, we reduce the bit length to 4 and simplify some parameters to simulate our benchmark programs. Quantum X gates are used for qubit initialization to a binary number.

B. Results

Our experimental results are presented in Table I. The first column of the table denotes the benchmark program name, the next six columns furnish the program statistics, and the last four columns present the resource requirements in terms of qubits and quantum gates. It should be noted that although our simulations were performed using four qubits integer size due to resource constraints, the values in this table correspond to the actual resource requirement when operated on 32-qubit data. The first column of the table provides the program name; the next six columns represent the program statistics, and the last four columns present the quantum resource requirement. For the *ARF* program, 11 addition operations and 17 multiplication operations are required for the computation. $16 \times 32 = 512$ qubits are required to initialize the 16 input variables. Moreover, 17 multiplication operations require extra qubits to store the product. Thus, the total qubits requirement for *ARF* is $512 + 17 \times 32 = 1056$ qubits. Furthermore, each quantum adder requires 166 CX gates and 62 CCX gates for 32-bit inputs, while the quantum multiplier requires 710 CX gates and 991 CCX gates to execute the program.

TABLE I: Statistics of benchmark programs and quantum circuit resource estimation.

Benchmark	Program Statistics						Quantum Resource Requirements				
	Variables	Loops	Iteration Count	If Statements	Arithmetic Operations	Logic Operations	Qubits	X	CX	CCX	SWAP
ARF	16	0	N/A	0	28	0	1056	0	13896	17529	0
GSM_NORM	11	0	N/A	5	5	7	484	175	857	534	48
GSM_DIV	8	1	15	16	30	46	272	960	7470	5642	930
ICRC	4	1	8	8	0	26	104	8	0	272	248

For *GSM*, we simulated two subfunctions: *GSM_NORM* and GSM_DIV. For *GSM_NORM*, there is one input variable "a" (used seven times), an integer -1073741824, four hexadecimal numbers 0xff000000, 0xff00, 0xFF, and 0xffff0000, and other integers 7, 15, 23, and 8, and four *if* statements. Thus, the quantum circuit requires $15 \times 32 = 480$ qubits to initialize the variables. The four *if* statements require 32 CCX gates and 4 qubits for the conditional operations. Furthermore, they require 128 X gates and $2 \times 127 = 254$ CCX gates and 1 CX gate to perform the logical computation. For the value initialization, logical operation, and calculation, we need 43 X gates, 760 CX gates, 248 CCX gates, and 48 SWAP gates to generate the quantum circuit, as shown in Table I. For *GSM_DIV*, QHLS performs a high-level optimization to pre-determine the number of loop iterations and unroll the loop accordingly. Two variable inputs require 32 bits each, and two longword format inputs require 64 bits each. One more internal variable, *"div"* requires 32 bits initialization. For each loop, the circuit needs two sets of bit shifting operations, one *if* statement with a comparator, a subtractor, and an adder. Thus, for *GSM_DIV*, the quantum circuit requires 7470 CX gates, 5642 CCX gates, and 930 SWAP gates to finish the computation.

For *ICRC*, the general design is demonstrated in Figure 7. The circuit applies XOR logic operations for the variable *"crc"* and *"onech"* and projects the result on the variable *"ans"*. Next, it performs a bit-shifter function block, composed of SWAP gates, to generate the variable *"ans"*. Each bit requires two CX gates and one CCX gate to perform a *"logic XOR"* operation. The loop includes 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 the statements are iterated eight times, which requires a total of 8 X gates, 32 CX gates, 272 CCX gates, 248 SWAP gates and 8 reset gates to finish the computation.

V. Conclusion

In this paper, we introduced a first Quantum High-level Synthesis (QHLS) framework that automatically translates high-level behavioral languages into quantum circuits, thus, reducing the onus of quantum computing. We proposed several quantum arithmetic/logical operations and conditional statement circuits, corresponding to software language constructs. We evaluated our framework on benchmark programs written in C. Our proposed QHLS framework was evaluated on HLS benchmark programs on a quantum simulator. Due to the limitation of qubits and noise levels on current quantum hardware and the inefficiency of classical computers to simulate large, complex quantum circuits, we limited our analysis to small circuits. In the future, we aim to further improve upon the proposed QHLS. For instance, we performed an analysis on the induction variable for unrolling the loop, which complicates the programming of the quantum circuit in the case of unbounded loops. Moreover, our current version of QHLS only supports integer operations. For our next step, we intend to develop provisions for floating-point operations. Furthermore, we plan on improving the efficiency of the proposed QHLS by using the entangled state of qubits to maximize the quantum advantage.

References

[1] F. Arute *et al.*, "Quantum Supremacy using a Programmable Superconducting Processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019.

[2] V. Bhatia *et al.*, "An efficient quantum computing technique for cracking rsa using shor's algorithm," in *2020 IEEE 5th ICCCA*, pp. 89–94.

[3] T. Monz *et al.*, "Realization of a Scalable Shor Algorithm," *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.

[4] J. M. Pino *et al.*, "Demonstration of the qccd trapped-ion quantum computer architecture," *arXiv preprint arXiv:2003.01293*, 2020.

[5] 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.

[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] A. Cross, "The ibm q experience and qiskit open-source quantum computing software," in *APS March meeting abstracts*, 2018.

[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] C. Lu, U. Banerjee, and K. Basu, "Design and analysis of a scalable and efficient quantum circuit for lwe matrix arithmetic," in *2022 IEEE ICCD*. IEEE, 2022, pp. 109–116.

[12] D. Pursley *et al.*, "High-level low-power system design optimization," in *2017 VLSI-DAT*. IEEE, 2017, pp. 1–4.

[13] F. Ferrandi *et al.*, "Bambu: an open-source research framework for the high-level synthesis of complex applications," in *2021 IEEE/ACM DAC*.

[14] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[15] M. A. Nielsen *et al.*, *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

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

[17] 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.

[18] H. Thapliyal *et al.*, "Quantum circuit designs of integer division optimizing t-count and t-depth," *IEEE TETC*, pp. 1045–1056, 2019.

[19] A. Pavlidis *et al.*, "Fast quantum modular exponentiation architecture for shor's factorization algorithm," *arXiv preprint arXiv:1207.0511*, 2012.

[20] C. Lu *et al.*, "Design and logic synthesis of a scalable, efficient quantum number theoretic transform," in *ACM/IEEE ISLPED*, 2022, pp. 1–6.

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

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

[23] C. Pilato *et al.*, "High-level synthesis of benevolent trojans," in *IEEE DATE*, 2019, pp. 1124–1129.

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