Quantum algorithms often apply classical operations, such as arithmetic or predicate checks, over a quantum superposition of classical data; these so-called oracles are often the largest components of a quantum program. To ease the construction of efficient, correct oracle functions, this paper presents vqo, a high-assurance framework implemented with the Coq proof assistant. The core of vqo is $\mathcal{O}_{qasm}$, the oracle quantum assembly language. $\mathcal{O}_{qasm}$ operations move qubits between two different bases via the quantum Fourier transform, thus admitting important optimizations, but without inducing entanglement and the exponential blowup that comes with it. $\mathcal{O}_{qasm}$‘s design enabled us to prove correct vqo’s compilers—from a simple imperative language called $\mathcal{O}_{qimp}$ to $\mathcal{O}_{qasm}$, and from $\mathcal{O}_{qasm}$ to $\mathcal{SQIR}$, a general-purpose quantum assembly language—and allowed us to efficiently test properties of $\mathcal{O}_{qasm}$ programs using the QuickChick property-based testing framework. We have used vqo to implement a variety of arithmetic and geometric operators that are building blocks for important oracles, including those used in Shor’s and Grover’s algorithms. We found that vqo’s QFT-based arithmetic oracles require fewer qubits, sometimes substantially fewer, than those constructed using “classical” gates; vqo’s versions of the latter were nevertheless on par with or better than (in terms of both qubit and gate counts) oracles produced by Quipper, a state-of-the-art but unverified quantum programming platform.

CCS Concepts:
• Software and its engineering → Formal language definitions;
• Hardware → Quantum computation; Hardware description languages and compilation;

Additional Key Words and Phrases: Quantum Oracle, Programming Language Design, Type System, Compiler Verification

1 INTRODUCTION

Quantum computers offer unique capabilities that can be used to program substantially faster algorithms compared to those written for classical computers. For example, Grover’s search algorithm [Grover 1996, 1997] can query unstructured data in sub-linear time (compared to linear time on a classical computer), and Shor’s algorithm [Shor 1994] can factorize a number in polynomial

‘Work completed prior to starting at Amazon.

Authors’ addresses: Liyi Li, University of Maryland, USA, liyili2@umd.edu; Finn Voichick, University of Maryland, USA, finn@umd.edu; Kesha Hietala, University of Maryland, USA, kesha@cs.umd.edu; Yuxiang Peng, University of Maryland, USA, ypeng15@umd.edu; Xiaodi Wu, University of Maryland, USA, xwu@cs.umd.edu; Michael Hicks, University of Maryland and Amazon, USA, mwh@cs.umd.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
time (compared to the sub-exponential time for the best known classical algorithm). An important source of speedups in these algorithms is the quantum computer’s ability to apply an oracle function coherently, i.e., to a superposition of classical queries, thus carrying out in one step a function that would potentially take many steps on a classical computer. For Grover’s, the oracle is a predicate function that determines when the searched-for data is found. For Shor’s, it is a classical modular exponentiation function; the algorithm finds the period of this function where the modulus is the number being factored.

While the classical oracle function is perhaps the least interesting part of a quantum algorithm, it contributes a significant fraction of the final program’s compiled quantum circuit. For example, Gidney and Ekerå [2021] estimated that Shor’s modular exponentiation function constitutes 90% of the final code. In our own experiments with Grover’s, our oracle makes up over 99% of the total gate count (the oracle has 3.3 million gates). Because quantum computers will be resource-limited for the foreseeable future [Somma 2020; Wilkins 2021], programmers and programming tools will be expected to heavily optimize their quantum circuits, especially the oracles. Such optimizations, including ones that involve approximation, risk bugs that can be hard to detect. This is because quantum programs are inherently difficult to simulate, test, and debug—qubits on real quantum computers are noisy and unreliable; observing a quantum program state mid-execution may change that state; and simulating a general quantum program on a classical computer is intractable because quantum states can require resources exponential in the number of qubits.

In this paper, we report on a framework we have been developing called vqo, the Verified Quantum Oracle framework, whose goal is to help programmers write quantum oracles that are correct and efficient. vqo is part of qvm, for Quantum Verified Machine, which has several elements, as shown in Figure 1.

- Using vqo, an oracle can be specified in a simple, high-level programming language we call \( OQIMP \), which has standard imperative features and can express arbitrary classical programs. It distinguishes quantum variables from classical parameters, allowing the latter to be partially evaluated [Jones et al. 1993], thereby saving qubits during compilation.
- The resulting \( OQIMP \) program is compiled to \( OQASM \) (pronounced “O-chasm”), the oracle quantum assembly language. \( OQASM \) was designed to be efficiently simulatable while nevertheless admitting important optimizations; it is our core technical contribution and we say more about it below. The generated \( OQASM \) code links against implementations of standard operators (addition, multiplication, sine, cosine, etc.) also written in \( OQASM \).
- The \( OQASM \) oracle is then translated to SQIR, the Simple Quantum Intermediate Representation, which is a circuit language embedded in the Coq proof assistant [Li et al. 2022]. SQIR has
been used to prove correct both quantum algorithms [Hietala et al. 2021a] and optimizations [Hietala et al. 2021b], the latter as part of vqoC, the Verified Optimizer for Quantum Circuits. After linking the oracle with the quantum program that uses it, the complete sqir program can be optimized and extracted to OpenQASM 2.0 [Cross et al. 2017] to run on a real quantum machine. Both vqo’s compilation from Oqimp to Oqasm and translation from Oqasm to sqir have been proved correct in Coq [Li et al. 2022].

vqo helps programmers ensure their oracles are correct by supporting both testing and verification, and ensures they are efficient by supporting several kinds of optimization. Both aspects are captured in the design of Oqasm, a quantum assembly language specifically designed for oracles.

Because oracles are classical functions, a reasonable approach would have been to design Oqasm to be a circuit language comprised of “classical” gates; e.g., prior work has targeted gates X (“not”), CNOT (“controlled not”), and CCNOT (“controlled controlled not,” aka Toffoli). Doing so would simplify proofs of correctness and support efficient testing by simulation because an oracle’s behavior could be completely characterized by its behavior on computational basis states (essentially, classical bitstrings). ReverC [Amy et al. 2017] and ReQWIRE [Rand et al. 2018] take this approach. However, doing so cannot support optimized oracle implementations that use fundamentally quantum functionality, e.g., as in quantum Fourier transform (QFT)-based arithmetic circuits [Beauregard 2003; Draper 2000]. These circuits employ quantum-native operations, like controlled-phase gates, in the QFT basis. Our key insight is that expressing such optimizations does not require expressing all quantum programs, as is possible in a language like sqir. Instead, Oqasm’s type system restricts programs to those that admit important optimizations while keeping simulation tractable. Oqasm also supports virtual qubits; its type system ensures that position shifting operations, commonly used when compiling arithmetic functions, require no extra SWAP gates when compiled to sqir, so there is no added runtime cost.

Leveraging Oqasm’s efficient simulatability, we implemented a property-based random testing (PBT) framework for Oqasm programs in QuickChick [Paraskevopoulou et al. 2015], a variant of Haskell’s QuickCheck [Claessen and Hughes 2000] for Coq programs. This framework affords two benefits. First, we can test that an Oqasm operator or Oqimp program is correct according to its specification. Formal proof in Coq can be labor-intensive, so PBT provides an easy-to-use confidence boost, especially prior to attempting formal proof. Second, we can use testing to assess the effect of approximations when developing oracles. For example, we might like to use approximate QFT, rather than full-precision QFT, in an arithmetic oracle in order to save gates. PBT can be used to test the effect of this approximation within the overall oracle by measuring the distance between the fully-precise result and the approximate one.

To assess vqo’s effectiveness we have used it to build several efficient oracles and oracle components, and have either tested or proved their correctness.

- Using Oqimp we implemented sine, cosine, and other geometric functions used in Hamiltonian simulation [Feynman 1982], leveraging the arithmetic circuits described below. Compared to a sine function implemented in Quipper [Green et al. 2013], a state-of-the-art quantum programming framework, vqo’s uses far fewer qubits thanks to Oqimp’s partial evaluation.

- We have implemented a variety of arithmetic operators in Oqasm, including QFT-, approximate QFT- and Toffoli-based multiplication, addition, modular multiplication, and modular division. Overall, circuit sizes are competitive with, and oftentimes better than, those produced by Quipper. Qubit counts for the final QFT-based circuits are always lower, sometimes significantly so (up to 53%), compared to the Toffoli-based circuits.
• We have proved correct both QFT and Toffoli-based adders, and QFT and Toffoli-based modular multipliers (which are used in Shor’s algorithm). These constitute the first proved-correct implementations of these functions, as far as we are aware.
• We used PBT to test the correctness of various \( \&QASM \) operators. Running 10,000 generated tests on 8- or 16-bit versions of the operators takes just a few seconds. Testing 60-bit versions of the adders and multipliers takes just a few minutes, whereas running a general quantum simulator on the final circuits fails. We found several interesting bugs in the process of doing PBT and proof, including in the original algorithmic description of the QFT-based modular multiplier [Beauregard 2003].
• We used PBT to analyze the precision difference between QFT and approximate QFT (AQFT) circuits, and the suitability of AQFT in different algorithms. We found that the AQFT adder (which uses AQFT in place of QFT) is not an accurate implementation of addition, but that it can be used as a subcomponent of division/modulo with no loss of precision, reducing gate count by 4.5–79.3%.
• Finally, to put all of the pieces together, we implemented the ChaCha20 stream cipher [Bernstein 2008] in \( \&QASM \) and used it as an oracle for Grover’s search, previously implemented and proved correct in SQIR [Hietala et al. 2021a]. We used PBT to test the oracle’s correctness. Combining its tested property with Grover’s correctness property, we demonstrate that Grover’s is able to invert the ChaCha20 function and find collisions.

The rest of the paper is organized as follows. We begin with some background on quantum computing (Section 2) and then present \( \&QASM \)’s syntax, typing, and semantics (Section 3). Then we discuss \( \&QOO \)’s implementation: \( \&QASM \)’s translator and property-based tester, and \( \&QIMP \) (Section 4). Finally, we present our results (Sections 5 to 7), compare against related work (Section 8), and conclude. All code presented in this paper is freely available at https://github.com/inQWIRE/VQO.

2 BACKGROUND

We begin with some background on quantum computing and quantum algorithms.

Quantum States. A quantum state consists of one or more quantum bits (qubits). A qubit can be expressed as a two dimensional vector \( \left( \begin{array}{c} \alpha \\ \beta \end{array} \right) \) where \( \alpha, \beta \) are complex numbers such that \( |\alpha|^2 + |\beta|^2 = 1 \). The \( \alpha \) and \( \beta \) are called amplitudes. We frequently write the qubit vector as \( \alpha |0 \rangle + \beta |1 \rangle \) where \( |0 \rangle = \left( \begin{array}{c} 1 \\ 0 \end{array} \right) \) and \( |1 \rangle = \left( \begin{array}{c} 0 \\ 1 \end{array} \right) \) are computational basis states. When both \( \alpha \) and \( \beta \) are non-zero, we can think of the qubit as being “both 0 and 1 at once,” a.k.a. a superposition. For example, \( \frac{1}{\sqrt{2}} (|0 \rangle + |1 \rangle) \) is an equal superposition of \( |0 \rangle \) and \( |1 \rangle \).

We can join multiple qubits together to form a larger quantum state with the tensor product \( (\otimes) \) from linear algebra. For example, the two-qubit state \( |0 \rangle \otimes |1 \rangle \) (also written as \( |01 \rangle \)) corresponds to vector \( [ 0 1 0 0 ]^T \). Sometimes a multi-qubit state cannot be expressed as the tensor of individual states; such states are called entangled. One example is the state \( \frac{1}{\sqrt{2}} (|00 \rangle + |11 \rangle) \), known as a Bell pair. Entangled states lead to exponential blowup: A general \( n \)-qubit state must be described with a \( 2^n \)-length vector, rather than \( n \) vectors of length two. The latter is possible for unentangled states like \( |0 \rangle \otimes |1 \rangle \); \( \&QASM \)’s type system guarantees that qubits remain unentangled.

Quantum Circuits. Quantum programs are commonly expressed as circuits, like those shown in Figure 2. In these circuits, each horizontal wire represents a qubit, and boxes on these wires indicate quantum operations, or gates. Gates may be controlled by a particular qubit, as indicated by a filled circle and connecting vertical line. The circuits in Figure 2 use four qubits and apply 10 (left) or 7 (right) gates: four Hadamard \( (H) \) gates and several controlled z-axis rotation (“phase”) gates. When programming, circuits are often built by meta-programs embedded in a host language.
Quantum Fourier Transform. The quantum Fourier transform (QFT) is the quantum analogue of the discrete Fourier transform. It is used in many quantum algorithms, including the phase estimation portion of Shor’s factoring algorithm [Shor 1994]. The standard implementation of a QFT circuit (for 4 qubits) is shown on the left of Figure 2; an approximate QFT (AQFT) circuit can be constructed by removing select controlled phase gates [Barenco et al. 1996; Hales and Hallgren 2000; Nam et al. 2020]. This produces a cheaper circuit that implements an operation mathematically similar to the QFT. The AQFT circuit we use in vqo (for 4 qubits) is shown on the right of Figure 2. When it is appropriate to use AQFT in place of QFT is an open research problem, and one that is partially addressed by our work on O_qasm, which allows efficient testing of the effect of AQFT inside of oracles.

Computational and QFT Bases. The computational basis \{|0\rangle, |1\rangle\} is just one possible basis for the underlying vector space. Another basis is the Hadamard basis, written as a tensor product of \{|+\rangle, |--\rangle\}, obtained by applying a Hadamard transform to elements of the computational basis, where |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) and |--\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). A third useful basis is the Fourier (or QFT) basis, obtained by applying a quantum Fourier transform (QFT) to elements of the computational basis.

Measurement. A measurement operation extracts classical information from a quantum state, typically when a computation completes. Measurement collapses the state to a basis states with a probability related to the state’s amplitudes. For example, measuring \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) in the computational basis will collapse the state to |0\rangle with probability \frac{1}{2} and likewise for |1\rangle, returning classical value 0 or 1, respectively. In all the programs discussed in this paper, we leave the final measurement operation implicit.

Quantum Algorithms and Oracles. Quantum algorithms manipulate input information encoded in "oracles," which are callable black box circuits. For example, Grover’s algorithm for unstructured quantum search [Grover 1996, 1997] is a general approach for searching a quantum "database," which is encoded in an oracle for a function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \). Grover’s finds an element \( x \in \{0, 1\}^n \) such that \( f(x) = 1 \) using \( O(2^n/2) \) queries, a quadratic speedup over the best possible classical algorithm, which requires \( \Omega(2^n) \) queries. An oracle can be constructed for an arbitrary function \( f \) simply by constructing a reversible classical logic circuit implementing \( f \) and then replacing classical logic gates with corresponding quantum gates, e.g., \( \text{X} \) for “not,” \( \text{CNOT} \) for “xor,” and \( \text{CCNOT} \) (aka Toffoli) for “and.” However, this approach does not always produce the most efficient circuits; for example, quantum circuits for arithmetic can be made more space-efficient using the quantum Fourier transform [Draper 2000].

Transforming an irreversible computation into a quantum circuit often requires introducing ancillary qubits, or ancillae, to store intermediate information [Nielsen and Chuang 2011, Chapter
Oracle algorithms typically assume that the oracle circuit is reversible, so any data in ancillae must be uncomputed by inverting the circuit that produced it. Failing to uncompute this information leaves it entangled with the rest of the state, potentially leading to incorrect program behavior. To make this uncomputation more efficient and less error-prone, recent programming languages such as Silq [Bichsel et al. 2020] have developed notions of implicit uncomputation. We have similar motivations in developing \texttt{vqo}: we aim to make it easier for programmers to write efficient quantum oracles, and to assure, through verification and randomized testing, that they are correct.

\section{\texttt{O_qasm}: An Assembly Language for Quantum Oracles}

We designed \texttt{O_qasm} to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct. \texttt{O_qasm} operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT). \texttt{O_qasm}'s type system tracks the bases of variables in \texttt{O_qasm} programs, forbidding operations that would introduce entanglement. \texttt{O_qasm} states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition, \texttt{O_qasm} uses virtual qubits to support position shifting operations, which support arithmetic operations without introducing extra gates during translation. All of these features are novel to quantum assembly languages.

This section presents \texttt{O_qasm} states and the language's syntax, semantics, typing, and soundness results. As a running example, we use the QFT adder [Beauregard 2003] shown in Figure 3. The Coq function \texttt{rz_adder'} generates an \texttt{O_qasm} program that adds two natural numbers \texttt{a} and \texttt{b}, each of length \texttt{n} qubits.

\subsection{\texttt{O_qasm} States}

An \texttt{O_qasm} program state is represented according to the grammar in Figure 4. A state \( \varphi \) of \( d \) qubits is a length-\( d \) tuple of qubit values \( q \); the state models the tensor product of those values. This means that the size of \( \varphi \) is \( O(d) \) where \( d \) is the number of qubits. A \( d \)-qubit state in a language like \texttt{sqir}
is represented as a length $2^d$ vector of complex numbers, which is $O(2^d)$ in the number of qubits. Our linear state representation is possible because applying any well-typed OQASM program on any well-formed OQASM state never causes qubits to be entangled.

A qubit value $q$ has one of two forms $\bar{b}$, scaled by a global phase $\alpha(r)$. The two forms depend on the basis $\tau$ that the qubit is in—it could be either Nor or Phi. A Nor qubit has form $|b\rangle$ (where $b \in \{0, 1\}$), which is a computational basis value. A Phi qubit has form $|\Phi(r)\rangle = \frac{1}{\sqrt{2}} (|0\rangle + \alpha(r)|1\rangle)$, which is a value of the (A)QFT basis. The number $n$ in Phi $n$ indicates the precision of the state $\varphi$. As shown by Beauléregard [2003], arithmetic on the computational basis can sometimes be more efficiently carried out on the QFT basis, which leads to the use of quantum operations (like QFT) when implementing circuits with classical input/output behavior.

### 3.2 OQASM Syntax, Typing, and Semantics

Figure 5 presents OQASM’s syntax. An OQASM program consists of a sequence of instructions $\iota$. Each instruction applies an operator to either a variable $x$, which represents a group of qubits, or a position $p$, which identifies a particular offset into a variable $x$.

The instructions in the first row correspond to simple single-qubit quantum gates—ID $p$, and $X p$—and instruction sequencing. The instructions in the next row apply to whole variables: QFT $n x$ applies the AQFT to variable $x$ with $n$-bit precision and $\text{QFT}^{-1} n x$ applies its inverse. If $n$ is equal to the size of $x$, then the AQFT operation is exact. SR$^{[-1]} n x$ applies a series of RZ gates (Figure 6). Operation $\text{CU} p \iota$ applies instruction $\iota$ controlled on qubit position $p$. All of the operations in this row—SR, QFT, and CU—will be translated to multiple SQIR gates. Function $\text{rz_adder}$ in Figure 3(b) uses many of these instructions; e.g., it uses QFT and $\text{QFT}^{-1}$ and applies $\text{CU}$ to the $m$th position of variable $a$ to control instruction $\text{SR} m b$.

In the last row of Figure 5, instructions Lshift $x$, Rshift $x$, and Rev $x$ are position shifting operations. Assuming that $x$ has $d$ qubits and $x_k$ represents the $k$-th qubit state in $x$, Lshift $x$ changes the $k$-th qubit state to $x_{(k+1)\%d}$, Rshift $x$ changes it to $x_{(k+d-1)\%d}$, and Rev changes it to $x_{d-1-k}$. In our implementation, shifting is virtual not physical. The OQASM translator maintains

$$
\text{Fig. 4. OQASM state syntax}
$$

$$
\text{Fig. 5. OQASM syntax. For an operator OP, OP}^{[-1]} \text{ indicates that the operator has a built-in inverse available.}
$$

$$
\text{Fig. 6. SR unfolds to a series of RZ instructions}
$$
a logical map of variables/positions to concrete qubits and ensures that shifting operations are no-ops, introducing no extra gates.

Other quantum operations could be added to $\mathcal{O}_{\text{QASM}}$ to allow reasoning about a larger class of quantum programs, while still guaranteeing a lack of entanglement. In [Li et al. 2021] Appendix A, we show how $\mathcal{O}_{\text{QASM}}$ can be extended to include the Hadamard gate $H$, $x$-axis rotations $RZ$, and a new basis $\text{H}$ad to reason directly about implementations of QFT and AQFT. However, this extension compromises the property of type reversibility (Theorem 3.5, Section 3.3), and we have not found it necessary in oracles we have developed.

**Typing.** In $\mathcal{O}_{\text{QASM}}$, typing is with respect to a *type environment* $\Omega$ and a *size environment* $\Sigma$, which map $\mathcal{O}_{\text{QASM}}$ variables to their basis and size (number of qubits), respectively. The typing judgment is written as $\Sigma; \Omega \vdash i : \Omega'$, which states that instruction $i$ is well-typed under environments $\Omega$ and $\Sigma$, and transforms variables’ bases as characterized by type environment $\Omega'$. For programs compiled to $\mathcal{O}_{\text{QASM}}$ (such as $\mathcal{O}_{\text{QMP}}$ in Section 4.4), the variable-to-size mapping $\Sigma$ can be generated by scanning variable declarations/initiation statements in the source program. Selected $\mathcal{O}_{\text{QASM}}$ type rules are given in Figure 7; the rules not mentioned (for ID, Rshift, Rev, and SR$^{-1}$) are similar.

The type system enforces three invariants. First, it enforces that instructions are well-formed, meaning that gates are applied to valid qubit positions (the second premise in rule X) and that any control qubit is distinct from the target(s) (the fresh premise in rule CU). This latter property enforces the quantum no-cloning rule. For example, we can apply the CU in rz_adder (Figure 3) because position $a, n$ is distinct from variable $b$.

Second, the type system enforces that instructions leave affected qubits in a proper basis (thereby avoiding entanglement). The rules implement the state machine shown in Figure 8. For example, QFT $n$ transforms a variable from Nor to Phi $n$ (rule QFT), while QFT$^{-1} n$ transforms it from Phi $n$ back to Nor (rule RQFT). Position shifting operations are disallowed on variables $x$ in the Phi basis because the qubits that make up $x$ are internally related (see Definition 3.1) and cannot be rearranged. Indeed, applying a Lshift and then a QFT$^{-1}$ on $x$ in Phi would entangle $x$’s qubits.

Third, the type system enforces that the effect of position shifting operations can be statically tracked. The neutral condition of CU requires that any shifting within $i$ is restored by the time it completes. For example, CU $p$ (Lshift $x$) ; $X (x, 0)$ is not well-typed, because knowing the final
physical position of qubit \((x, 0)\) would require statically knowing the value of \(p\). On the other hand, the program \(CU\ c\ (Lshift\ x; X\ (x, 0); Rshift\ x); X\ (x, 0)\) is well-typed because the effect of the Lshift is "undone" by an Rshift inside the body of the CU.

**Semantics.** We define the semantics of an \(\mathcal{O}\text{QASM}\) program as a partial function \(\llbracket\cdot\rrbracket\) from an instruction \(\i\) and input state \(\varphi\) to an output state \(\varphi'\), written \(\llbracket i \rrbracket \varphi = \varphi'\), shown in Figure 9.

Recall that a state \(\varphi\) is a tuple of \(d\) qubit values, modeling the tensor product \(q_1 \otimes \cdots \otimes q_d\). The rules implicitly map each variable \(x\) to a range of qubits in the state, e.g., \(\varphi(x)\) corresponds to some sub-state \(q_k \otimes \cdots \otimes q_{k+n-1}\) where \(\Sigma(x) = n\). Many of the rules in Figure 9 update a portion of a state. We write \(\varphi[i\ (x, i) \mapsto \gamma(x, i)]\) to update the \(i\)-th qubit of variable \(x\) to be the (single-qubit) state \(\gamma(x, i)\). To update variable \(x\) according to the qubit tuple \(\gamma(x, i)\) and \(\varphi[x \mapsto q_x]\) are similar, except that they also accumulate the previous global phase of \(\varphi(x, i)\) (or \(\varphi(x)\)). We use \(\downarrow\) to convert a qubit \(\alpha(b)\vec{q}\) to an unphased qubit \(\vec{q}\).

Function \(\text{xg}\) updates the state of a single qubit according to the rules for the standard quantum gate \(X\). \(\text{cu}\) is a conditional operation depending on the Nor-basis qubit \((x, i)\). \(\text{SR}\) (or \(\text{SR}^{-1}\)) applies an \(m + 1\) series of RZ (or RZ\(^{-1}\)) rotations where the \(i\)-th rotation applies a phase of \(\alpha(-\frac{1}{2m+i})\) (or \(\alpha(-\frac{1}{2m+i})\)). \(\text{qt}\) applies an approximate quantum Fourier transform; \(|y\rangle\) is an abbreviation of \(|b_1\rangle \otimes \cdots \otimes |b_1\rangle\) (assuming \(\Sigma(y) = i\) and \(n\) is the degree of approximation. If \(n = i\), then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to a range of qubits in the state. We write \(\text{cu}_{\alpha\beta}(\varphi)\) for the cumulative action of a sequence of \(\text{cu}\) instructions.

Figure 9. \(\mathcal{O}\text{QASM}\) semantics.

\[
\begin{align*}
\llbracket \text{id}\ p \rrbracket \varphi & = \varphi \\
\llbracket X\ (x, i) \rrbracket \varphi & = \varphi[(x, i) \mapsto \uparrow \text{xg}(\downarrow \varphi(x, i))] \quad \text{where xg}(0) = |1\rangle \text{xg}(1) = |0\rangle \\
\llbracket \text{cu}\ (x, i) \rrbracket \varphi & = \text{cu}_{\alpha}(\downarrow \varphi(x, i), i, \varphi) \quad \text{where cu}(0, i, \varphi) = \varphi \text{ cu}(1, i, \varphi) = \llbracket i \rrbracket \varphi \\
\llbracket \text{SR}\ m\ x \rrbracket \varphi & = \varphi[\forall i \leq m. (x, i) \mapsto \uparrow |\Phi(r_i + m)\rangle] \quad \text{when} \ \downarrow \varphi(x, i) = |\Phi(r_i)\rangle \\
\llbracket \text{SR}^{-1}\ m\ x \rrbracket \varphi & = \varphi[\forall i \leq m. (x, i) \mapsto \uparrow |\Phi(r_i - m)\rangle] \quad \text{when} \ \downarrow \varphi(x, i) = |\Phi(r_i)\rangle \\
\llbracket \text{QFT}\ n\ x \rrbracket \varphi & = \varphi[x \mapsto \uparrow \text{qt}(\Sigma(x), \downarrow \varphi(x), n)] \quad \text{where qt}(i, |y\rangle, n) = \bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{m-k}})\rangle) \\
\llbracket \text{QFT}^{-1}\ n\ x \rrbracket \varphi & = \varphi[x \mapsto \downarrow \text{qt}^{-1}(\Sigma(x), \uparrow \varphi(x), n)] \quad \text{where qt}(i, |y\rangle, n) = \bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{m-k}})\rangle) \\
\llbracket \text{Lshift}\ x \rrbracket \varphi & = \varphi[x \mapsto \text{pm}_{q_x}(\varphi(x))] \quad \text{where pm}_q(q_0 \otimes q_1 \otimes \cdots \otimes q_{n-1}) = q_{n-1} \otimes q_0 \otimes q_1 \otimes \cdots \\
\llbracket \text{Rshift}\ x \rrbracket \varphi & = \varphi[x \mapsto \text{pm}_r(\varphi(x))] \quad \text{where pm}_r(q_0 \otimes q_1 \otimes \cdots \otimes q_{n-1}) = q_1 \otimes \cdots \otimes q_{n-1} \otimes q_0 \\
\llbracket \text{Rev}\ x \rrbracket \varphi & = \varphi[x \mapsto \text{pm}_d(\varphi(x))] \quad \text{where pm}_d(q_0 \otimes q_1 \otimes \cdots \otimes q_{n-1}) = q_{n-1} \otimes \cdots \otimes q_0 \\
\llbracket t_1; t_2 \rrbracket \varphi & = \llbracket t_2 \rrbracket \llbracket t_1 \rrbracket \varphi \\
\downarrow \alpha(b)\vec{q} & = \vec{q} \quad \downarrow (q_1 \otimes \cdots \otimes q_n) = \downarrow q_1 \otimes \cdots \otimes \downarrow q_n \\
\varphi[(x, i) \mapsto \uparrow \text{xg}(\downarrow \varphi(x, i))] & = \varphi[(x, i) \mapsto \uparrow \text{qt}(\downarrow \varphi(x, i))] \quad \text{where} \ \varphi(x, i) = \alpha(b)\vec{q} \\
\varphi[(x, i) \mapsto \uparrow \alpha(b_1)\vec{q}] & = \varphi[(x, i) \mapsto \uparrow \alpha(b_1 + b_2)\vec{q}] \quad \text{where} \ \varphi(x, i) = \alpha(b_2)\vec{q} \\
\varphi[x \mapsto q_x] & = \varphi[\forall i < \Sigma(x). (x, i) \mapsto q_{(x,i)}] \\
\varphi[x \mapsto \uparrow q_x] & = \varphi[\forall i < \Sigma(x). (x, i) \mapsto \uparrow q_{(x,i)}]
\end{align*}
\]
3.3 QASM Metatheory

**Soundness.** We prove that well-typed QASM programs are well defined; i.e., the type system is sound with respect to the semantics. We begin by defining the well-formedness of an QASM state.

Definition 3.1 (Well-formed QASM state). A state $\varphi$ is well-formed, written $\Sigma; \Omega \vdash \varphi$, iff:

- For every $x \in \Omega$ such that $\Omega(x) = \text{Nor}$, for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r) |b\rangle$.
- For every $x \in \Omega$ such that $\Omega(x) = \Phi n$ and $n \leq \Sigma(x)$, there exists a value $v$ such that for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r) |\frac{v}{2^{n-k}}\rangle$.

Type soundness is stated as follows; the proof is by induction on $\Gamma$.

**Theorem 3.2.** [QASM type soundness] If $\Sigma; \Omega \vdash i \mapsto \Omega'$ and $\Sigma; \Omega \vdash \varphi$ then there exists $\varphi'$ such that $\Gamma \varphi = \varphi'$ and $\Sigma; \Omega' \vdash \varphi'$.

**Algebraic Structure.** A well-formed $d$-qubit QASM state for a given $\Omega$ can be interpreted as a vector of a 2-dimensional Hilbert space $\mathcal{H}^d$. The set of vectors interpreted from all well-formed QASM states is denoted as $\mathcal{S}^d \subseteq \mathcal{H}^d$.

The semantics $\Gamma [i]$ of instruction $i$ can then be interpreted as a 2-dimensional Hilbert space, and is standard when representing the semantics of programs without measurement [Hietala et al. 2021a]. Since $\mathcal{S}^d$ includes all the computational basis states, correctness properties extend by linearity from $\mathcal{S}^d$ to $\mathcal{H}^d$. An oracle that performs addition for classical Nor inputs will also perform addition over a superposition of Nor inputs. We have proved that $\mathcal{S}^d$ is closed under well-typed QASM programs.

Let $\{i\}$ be the set of QASM programs that are well-typed with respect to some $\Sigma$ and $\Omega$—i.e., $i \in \{i\}$ implies $\Sigma; \Omega \vdash i \mapsto \Omega'$ for some $\Omega'$. This set forms a groupoid $\{\{i\}, \Sigma, \Omega, \mathcal{S}^d\}$. Here, the set $\mathcal{S}^d$ corresponds to the well-formed states in Definition 3.1, i.e., there is a isomorphism $\Gamma [\_]$, such that $\Gamma [\varphi] \in \mathcal{S}^d$ iff $\Sigma; \Omega \vdash \varphi$. The structure $\{\{i\}, \Sigma, \Omega, \mathcal{S}^d\}$ defines a groupoid because $\mathcal{S}^d$ is closed under well-typed QASM programs $i \in \{i\}$.

We can extend this groupoid to another algebraic structure $\{\{i\}', \Sigma, \mathcal{H}^d\}$, where $\mathcal{H}^d$ is a general 2-dimensional Hilbert space and $\{i\}'$ is a universal set of quantum gate operations. Evidently, we have $\mathcal{S}^d \subseteq \mathcal{H}^d$ and $\{i\} \subseteq \{i\}'$, because sets $\mathcal{H}^d$ and $\{i\}'$ can be acquired by removing the well-formed ($\Sigma; \Omega \vdash \varphi$) and well-typed ($\Sigma; \Omega \vdash i \mapsto \Omega'$) definitions for $\mathcal{S}^d$ and $\{i\}$, respectively. $\{\{i\}', \Sigma, \mathcal{H}^d\}$ is a groupoid because every QASM operation is valid in a traditional quantum language like SQIR. We then have the following two theorems to connect QASM operations with operations in the general Hilbert space:

**Theorem 3.3.** $\{\{i\}, \Sigma, \Omega, \mathcal{S}^d\}$ is a subgroupoid of $\{\{i\}, \Sigma, \mathcal{H}^d\}$.

Theorem 3.3 suggests that QASM is a self-contained subset of general quantum operations. We then utilize the following theorem to connect the QASM oracle program semantics with general quantum programs and superposition states.

**Theorem 3.4.** Let $|y\rangle$ be an abbreviation of $\bigotimes_{m=0}^{d-1} \alpha(r_m) |b_m\rangle$ for $b_m \in \{0, 1\}$. If for every $i \in \{0, 2^d\}$, $\Gamma [i | y\rangle] = |y'\rangle$, then $\Gamma [\sum_{i=0}^{2^d-1} |y_i\rangle] = \sum_{i=0}^{2^d-1} |y'\rangle$.

We prove these theorems as corollaries of the compilation correctness theorem from QASM to SQIR (Theorem 4.1). Theorem 3.3 suggests that the space $\mathcal{S}^d$ is closed under the application of

1. Note that $\Phi(x) = \Phi(x + n)$, where the integer $n$ refers to phase $2\pi n$; so multiple choices of $n$ are possible.
2. A Hilbert space is a vector space with an inner product that is complete with respect to the norm defined by the inner product. $\mathcal{S}^d$ is a subset, not a subspace of $\mathcal{H}^d$ because $\mathcal{S}^d$ is not closed under addition: Adding two well-formed states can produce a state that is not well-formed.
3. Note that a superposition over classical states can describe any quantum state, including entangled states.
any well-typed \( \mathcal{QASM} \) operation. Theorem 3.4 says that \( \mathcal{QASM} \) oracles can be safely applied to superpositions over classical states.

**Type Reversibility.** \( \mathcal{QASM} \) programs are easily invertible, as shown by the rules in Figure 10. This inversion operation is useful for constructing quantum oracles; for example, the core logic in the QFT-based subtraction circuit is just the inverse of the core logic in the addition circuit (Figure 11). This allows us to reuse the proof of addition in the proof of subtraction. The inversion function satisfies the following properties:

**Theorem 3.5.** [Type reversibility] For any well-typed program \( \iota \), such that \( \Sigma; \Omega \vdash \iota \triangleright \Omega' \), its inverse \( \iota^{-1} \), where \( \iota \xrightarrow{\text{inv}} \iota^{-1} \), is also well-typed and we have \( \Sigma; \Omega' \vdash \iota^{-1} \triangleright \Omega \). Moreover, \( \| \iota; \iota^{-1} \| \varphi = \varphi \).

4 VQO QUANTUM ORACLE FRAMEWORK

This section presents \( \mathcal{VQO} \), our framework for specifying, compiling, testing, and verifying quantum oracles, whose architecture was given in Figure 1. We start by considering translation from \( \mathcal{QASM} \) to \( \mathcal{SQIR} \) and proof of its correctness. Next, we discuss \( \mathcal{VQO}' \)s property-based random testing framework for \( \mathcal{QASM} \) programs. Finally, we discuss \( \mathcal{QIMP} \), a simple imperative language for writing oracles, which compiles to \( \mathcal{QASM} \). We also present its proved-correct compiler and means to test the correctness of \( \mathcal{QIMP} \) oracles.

4.1 Translation from \( \mathcal{QASM} \) to \( \mathcal{SQIR} \)

\( \mathcal{VQO} \) translates \( \mathcal{QASM} \) to \( \mathcal{SQIR} \) by mapping \( \mathcal{QASM} \) positions to \( \mathcal{SQIR} \) concrete qubit indices and expanding \( \mathcal{QASM} \) instructions to sequences of \( \mathcal{SQIR} \) gates. Translation is expressed as the judgment \( \Sigma; \Omega \vdash \gamma \triangleright \epsilon \) where \( \Sigma \) maps \( \mathcal{QASM} \) variables to their sizes, \( \epsilon \) is the output \( \mathcal{SQIR} \) circuit, and \( \gamma \) maps an \( \mathcal{QASM} \) position \( p \) to a \( \mathcal{SQIR} \) concrete qubit index (i.e., offset into a global qubit register). At the start of translation, for every variable \( x \) and \( i < \Sigma(x) \), \( \gamma \) maps \( (x, i) \) to a unique concrete index chosen from 0 to \( \sum_{x} (\Sigma(x)) \).

Figure 12 depicts a selection of translation rules.\(^4\) The first rule shows how to translate \( X \) \( p \), which has a directly corresponding gate in \( \mathcal{SQIR} \). The second rule left-shifts the qubits of the target variable in the map \( \gamma \), and produces an identity gate (which will be removed in a subsequent optimization pass). For example, say we have variables \( x \) and \( y \) in the map \( \gamma \) and variable \( x \) has three qubits so \( \gamma \) is \( \{ (x, 0) \mapsto 0, (x, 1) \mapsto 1, (x, 2) \mapsto 2, (y, 0) \mapsto 3, \ldots \} \). Then after \( \text{Lshift} \ x \) the \( \gamma \)

\(^4\)Translation in fact threads through the typing judgment, but we elide that for simplicity.
map becomes \{ (x, 0) \mapsto 1, (x, 1) \mapsto 2, (x, 2) \mapsto 0, (y, 0) \mapsto 3, \ldots \}. The last two rules translate the CU and sequencing instructions. In the CU translation, the rule assumes that i’s translation does not affect the y position map. This requirement is assured for well-typed programs per rule CU in Figure 7. ctrl generates the controlled version of an arbitrary SQIR program using standard decompositions [Nielsen and Chuang 2011, Chapter 4.3].

We have proved \( O \text{QASM} \) to SQIR translation correct. To formally state the correctness property we relate d-qubit \( O \text{QASM} \) states to SQIR states, which are vectors of \( 2^d \) complex numbers, via a function \([ - ]^d\), where \( y \) is the virtual-to-physical qubit map. For example, say that our program uses two variables, \( x \) and \( y \), and both have two qubits. The qubit states are \( |0\rangle \) and \( |1\rangle \) (meaning that \( x \) has type Nor), and \( |\Phi(r_1)\rangle \) and \( |\Phi(r_2)\rangle \) (meaning that \( y \) has type Phi). Furthermore, say that \( y = \{(x, 0) \mapsto 0, (x, 1) \mapsto 1, (y, 0) \mapsto 2, (y, 1) \mapsto 3\} \). This \( O \text{QASM} \) program state will be mapped to the \( 2^4 \) -element vector \( |0\rangle \otimes |1\rangle \otimes (|0\rangle + e^{2\pi i r_1} |1\rangle) \otimes (|0\rangle + e^{2\pi i r_2} |1\rangle) \).

**Theorem 4.1.** [\( O \text{QASM} \) translation correctness] Suppose \( \Sigma; \Omega \vdash \iota \vdash \Omega' \) and \( \Sigma \vdash (y, i) \mapsto (y', e) \). Then for \( \Sigma; \Omega \vdash \varphi \) and \([ \varphi ]\varphi = \varphi' \), we have \([ \varphi ] \times [ \varphi ]^d_y \) \([ \varphi ] \times [ \varphi ]^d_y \) where \([ \varphi ]\) is the matrix interpretation of \( e \) per SQIR’s semantics.

The proof of translation correctness is by induction on the \( O \text{QASM} \) program \( i \). Most of the proof simply shows the correspondence of operations in \( i \) to their translated-to gates \( e \) in SQIR, except for shifting operations, which update the virtual-to-physical map.

Note that to link a complete, translated oracle \( i \) into a larger SQIR program may require that \( y = y' \), i.e., \( \text{neutral}(i) \), so that logical inputs match logical outputs. This requirement is naturally met for programs written to be reversible, as is the case for all arithmetic circuits in this paper, e.g., \( \text{rz_adder} \) from Figure 3.  

### 4.2 Using \( O \text{QASM} \) Types to Aid Circuit Verification

When verifying QFT-based oracle circuits, we can use knowledge of qubits’ types to simplify the proof. For example, we can verify a program that leaves qubits in the Nor basis by reasoning about classical bitstrings since every Nor qubit has the form \( \alpha(b) |b\rangle \) for \( b \in \{0, 1\} \). In this section, we discuss how \( O \text{QASM} \)’s Phi basis is useful for verifying the QFT-based adder from Figure 3.

The Coq function \( \text{rz_adder} \), defined in Figure 3(b), when applied to variables \( a \) and \( b \) produces a circuit that sums input wires \( a \) and \( b \), leaving the sum in output wire \( b \). The proof can be split into three stages by splitting the Figure 3(a) into three parts. Stage 1 is the first QFT gate application. Stage 2 is the series of SR gates that come next. Stage 3 is the QFT\(^{-1} \) gate application at the end. Initially, the states of variables \( a \) and \( b \) are in the Nor basis. Then, \( b \)’s state is transformed to the Phi basis by applying a QFT operation in the first stage. The proof is completed by the QFT gate semantics.

The second stage applies a series of control SR gates that conceptually add variable \( a \)’s bits, not to \( b \)’s qubit state bases, but to its phases. We wish to inductively show that \( b \)’s state properly records the step-by-step computation result. The difficulty is that quantum phase-state recording is not localized to individual qubits. In a Toffoli-based circuit, adding a number 2 flips the second least
significant bit and computes the carry-out on high bits; it never affects the least significant bit. However, adding a number 2 in a QFT-based circuit requires the action of “adding 2” to the phase value of possibly every qubit.\footnote{Recall that a single qubit phase is a complex number that is enough to hold the computation result of adding two numbers. Quantum phase additions can be viewed as rotations in a circle. “Adding 1” can be viewed as rotating a small fixed angle on the circle plane.}

To do this, we leverage an $\mathcal{O}_{\text{QASM}}$ property, based on Theorem 3.2 and Theorem 3.3, of an arbitrary qubit variable $x$’s state that is in $\Phi$-basis; that is, if $(x, 0)$ has state $\varphi(0)$, $(x, k)$’s state has a similar structure that can be written as $\varphi(k)$ regardless operations being applied. Thus, one can observe the state $(x, 0)$ to ascertain the other qubit states in $x$. We then utilize the property as a loop invariant, along with the transition semantics of $\text{SR}$ gates, to inductively prove in $\text{rz_adder}$ that we add bits of variable $a$ into $b$’s phase, one at a time, based solely on the phase changes in $(x, 0)$’s state.

The third stage is that after the phase manipulation step is done, the computation result is then transformed into qubit state bases by apply a $\text{QFT}^{-1}$ gate, which requires us to connect qubit state phases and bases. Fortunately, this step coincides with turning quantum states in $\Phi$-basis to $\text{Nor}$-basis, which is made evident in $\mathcal{O}_{\text{QASM}}$’s types. Here, we prove a lemma, which is independent of specific QFT-based oracle applications, to connect the application of an $\text{SR}$ gate on a $\Phi$-basis variable $x$ and the side-effect caused by such application after we turn $x$’s state from $\Phi$ to $\text{Nor}$-basis by applying a $\text{QFT}^{-1}$ gate. In verifying the $\text{rz_adder}$ circuit, for example, the series of $\text{SR}$ gate applications add variable $a$’s bits to $b$’s phase and the final $\text{QFT}^{-1}$ gate transforms the addition effect to variable $b$’s bases by turning it from $\Phi$ to $\text{Nor}$ basis. The lemma unveils that the transformation correctly conveys the bit addition information from $b$’s phase to bases. Thus, the verified inductive addition property in stage 2 is properly transformed to $b$’s bases in stage 3.

### 4.3 Property-Based Random Testing

$\mathcal{O}_{\text{QASM}}$’s type system ensures that states can be efficiently represented. We leverage this fact to implement a testing framework for $\mathcal{O}_{\text{QASM}}$ programs using QuickChick \cite{Paraskevopoulos:15}, which is a property-based testing (PBT) framework for Coq in the style of Haskell’s QuickCheck \cite{Claessen:H Hughes:00}. We use this framework for two purposes: To test correctness properties of $\mathcal{O}_{\text{QASM}}$ programs and to experiment with the consequences of approximation for efficiency and correctness.

**Implementation.** PBT randomly generates inputs using a hand-crafted generator, and confirms that a property holds for these inputs. Since oracle operations are defined over $\text{Nor}$-basis inputs, we wrote a generator for these inputs. A single test of an $\mathcal{O}_{\text{QASM}}$ program involves five steps: (1) generate (or specify) $n$, which is the number of qubits in the input; (2) for each input variable $x$, generate uniformly random bitstrings $b_0 b_1 \ldots b_{n-1}$ of length $n$, representing $x$’s initial qubit value $\bigotimes_{i=0}^{n-1} a(0) |b_i\rangle$; (3) prepare an $\mathcal{O}_{\text{QASM}}$ state $\varphi$ containing all input variables’ encoded bitstrings; (4) execute the $\mathcal{O}_{\text{QASM}}$ program with the prepared state; and (5) check that the resulting state satisfies the desired property.

We took several steps to improve testing performance. For example, we streamlined the representation of states: Per the semantics in Figure 9, in a state with $n$ qubits, the phase associated with each qubit can be written as $\varphi(\frac{v p}{2^n})$ for some natural number $v$. Qubit values in both bases are thus pairs of natural numbers: the global phase $v$ (in range $[0, 2^n)$) and $b$ (for $|b\rangle$) or $y$ (for $|\Phi(\frac{v p}{2^n})\rangle$). An $\mathcal{O}_{\text{QASM}}$ state $\varphi$ is a map from qubit positions $p$ to qubit values $q$; in our proofs, this map is implemented as a partial function, but for testing, we use an AVL tree implementation (proved equivalent to the functional map). To avoid excessive stack use, we implemented the $\mathcal{O}_{\text{QASM}}$
fixedp \sin(Q \ fixedp x/8, Q \ fixedp x_r, C \ nat n)\
\quad x_r = x/8; C \ fixedp n_y; Q \ fixedp x_2; Q \ fixedp x_1;\
C \ nat n_1; C \ nat n_2; C \ nat n_3; C \ nat n_4; C \ nat n_5;\
for (C \ nat i = 0; i < n; i++)\
\quad n_1 = i + 1; n_2 = 2 * n_1; n_3 = \text{pow}(8, n_2); n_4 = n_2 + 1;\
\quad n_5 = n_4!; n_y = n_3/n_5; x_2 = \text{pow}(x/8, n_4);\
\quad \text{if} \ (\text{even}(n_1)) \ {\{ x_1 = n_y * x_2; x_r += x_1; \}}\
\quad \text{else} \ {\{ x_1 = \text{pow}(x/8, n_4); x_1 \}}\
\quad \text{return} \ (8 * x_r);\
}

\begin{equation}
\sin x \approx 8 * \left( \frac{x}{\pi} - \frac{6}{\pi} \left( \frac{x}{\pi} \right)^3 + \frac{12}{\pi} \left( \frac{x}{\pi} \right)^5 - \frac{20}{\pi} \left( \frac{x}{\pi} \right)^7 + \ldots + (-1)^{n-1} \frac{8 \cdot (2n-2)!}{(2n-1)!} \left( \frac{x}{\pi} \right)^{2n-1} \right)
\end{equation}

Fig. 13. Implementing sine in \texttt{QIMP}

semantics function tail-recursively. To run the tests, QuickChick runs OCaml code that it \textit{extracts} from the Coq definitions; during extraction, we replace natural numbers and operations thereon with machine integers and operations. We present performance results in Section 5.

\textbf{Testing Correctness.} Full formal proof is the gold standard for correctness, but it is also laborious. It is especially deflating to be well through a proof only to discover that the intended property does not hold and, worse, that nontrivial changes to the program are necessary. Our PBT framework gives assurance that an \texttt{QASM} program property is correct by attempting to falsify it using thousands of randomly generated instances, with good coverage of the program’s input space. We have used PBT to test the correctness of a variety of operators useful in oracle programs, as presented in Section 5. When implementing a QFT-adder circuit, using PBT revealed that we had encoded the wrong endianness. We have also used PBT with \texttt{QIMP} programs by first compiling them to \texttt{QASM} and then testing their correctness at that level.

\textbf{Assessing the Effect of Approximation.} Because of the resource limitations of near-term machines, programmers may wish to \textit{approximate} the implementation of an operation to save qubits or gates, rather than implement it exactly. For example, a programmer may prefer to substitute QFT with an approximate QFT, which requires fewer gates. Of course, this substitution will affect the circuit’s semantics, and the programmer will want to understand the \textit{maximum distance} (similarity) between the approximate and exact implementations, to see if it is tolerable. To this end, we can test a relational property between the outputs of an exact and approximate circuit, on the same inputs, to see if the difference is acceptable. Section 5.4 presents experiments comparing the effect of approximation on circuits using QFT-based adders.

\subsection{\texttt{QIMP}: A High-Level Oracle Language}

It is not uncommon for programmers to write oracles as metaprograms in a quantum assembly’s host language, e.g., as we did for \texttt{rz_adder} in Figure 3. But this process can be tedious and error-prone, especially when trying to write optimized code. To make writing efficient arithmetic-based quantum oracles easier, we developed \texttt{QIMP}, a higher-level imperative language that compiles to \texttt{QASM}. Here we discuss \texttt{QIMP}’s basic features, describe how we optimize \texttt{QIMP} programs during compilation using partial evaluation, and provide correctness guarantees for \texttt{QIMP} programs. Using \texttt{QIMP}, we have defined operations for the ChaCha20 hash-function [Bernstein 2008], exponentiation, sine, arcsine, and cosine, and tested program correctness by running inputs through \texttt{QIMP}’s semantics. More details about \texttt{QIMP} are available in [Li et al. 2021] Appendix B.
Language Features. An $\text{OQIMP}$ program is a sequence of function definitions, with the last acting as the "main" function. Each function definition is a series of statements that concludes by returning a value $v$. $\text{OQIMP}$ statements contain variable declarations, assignments (e.g., $x_r = x/8$ in Figure 13), arithmetic computations ($n_1 = i + 1$), loops, conditionals, and function calls. Variables $x$ have types $\tau$, which are either primitive types $\omega^m$ or arrays thereof, of size $n$. A primitive type pairs a base type $\omega$ with a quantum mode $m$. There are three base types: type $\text{nat}$ indicates non-negative (natural) numbers; type $\text{fixdp}$ indicates fixed-precision real numbers in the range $(-1, 1)$; and type $\text{bool}$ represents booleans. The programmer specifies the number of qubits to use to represent $\text{nat}$ and $\text{fixdp}$ numbers when invoking the $\text{OQIMP}$ compiler. The mode $m \in \{C, Q\}$ on a primitive type indicates when a type’s value is expected to be known: $C$ indicates that the value is based on a classical parameter of the oracle, and should be known at compile time; $Q$ indicates that the value is a quantum input to the oracle, computed at runtime.

Figure 13 shows the $\text{OQIMP}$ implementation of the sine function, which is used in quantum applications such as Hamiltonian simulation [Childs 2009; Feynman 1982]. Because $\text{fixdp}$ types must be in the range $(-1, 1)$, the function takes $\frac{1}{8}$ times the input angle in variable $x/8$ (the input angle $x$ is in $[0, 2\pi]$). The result, stored in variable $x_r$, is computed by a Taylor expansion of $n$ terms. The standard formula for the Taylor expansion is $\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!}$; the loop in the algorithm computes an equivalent formula given input $\frac{1}{8}x$, as shown at the bottom of the figure.

Reversible Computation. Since programs that run on quantum computers must be reversible, $\text{OQIMP}$ compiles functions to reverse their effects upon returning. In Figure 13, after the main function returns, only the return value is copied and stored to a target variable. For other values, like $x/8$, the compiler will insert an inverse circuit to revert all side effects.

When variables are reused within a function, they must be uncomputed using $\text{OQIMP}$’s $\text{inv}$ $x$ operation. For example, in Figure 13, the second $\text{inv}$ operation returns $x_z$ to its state prior to the execution of $x_z = \text{pow}(x/8, n_4)$ so that $x_z$ can be reassigned in the next iteration. We plan to incorporate automatic uncomputation techniques to insert $\text{inv}$ $x$ calls automatically, but doing so requires care to avoid blowup in the generated circuit [Paradis et al. 2021].

The $\text{vqo}$ compiler imposes three restrictions on the use of $\text{inv}$ $x$, which aim to ensure that each use uncomputes just one assignment to $x$. First, since the semantics of an $\text{inv}$ operation reverses the most recent assignment, we require that every $\text{inv}$ operation have a definite predecessor. Example (1) in Figure 14 shows an $\text{inv}$ operation on a variable that does not have a predecessor; (2) shows a variable $z$ whose predecessor is not always executed. Both are invalid in $\text{OQIMP}$. Second, the statements between an $\text{inv}$ operation and its predecessor cannot write to any variables used in the body of the predecessor. Example (3) presents an invalid case where $x$ is used in the predecessor of $z$, and is assigned between the $\text{inv}$ and the predecessor. The third restriction is that, while sequenced $\text{inv}$ operations are allowed, the number of $\text{inv}$ operations must match the number of predecessors. Example (4) is invalid, while (5) is valid, because the first $\text{inv}$ in (5) matches the multiplication assignment and the second $\text{inv}$ matches the addition assignment.

To implement these well-formedness checks, $\text{vqo}$’s $\text{OQIMP}$ compiler maintains a stack of assignment statements. Once the compiler hits an $\text{inv}$ operation, it pops statements from the stack to find a match for the variable being uncomputed. It also checks that none of the popped statements contain an assignment of variables used in the predecessor statement.

Compilation from $\text{OQIMP}$ to $\text{QASM}$. The $\text{OQIMP}$ compiler performs partial evaluation [Jones et al. 1993] on the input program given classical parameters; the residual program is compiled to a quantum circuit. In particular, we compile an $\text{OQIMP}$ program by evaluating its $C$-mode components,
The oracle performs modular exponentiation on natural numbers via 
[Li et al. 2021](Appendix B).

We evaluate \( n \) when compiling the loop-body statement 
\( x = i + 1; \) into \( qasm \) code. For example, when compiling the
loop corresponds to an addition at
\( i \)'s value in the store for each iteration. When compiling the loop-body statement
\( n_1 = i + 1 \), variable \( n_1 \) will simply be updated in the store, and no code generated. When compiling statement
\( x_2 = \text{pow}(x, 8, n_4) \), the fact that \( x_2 \) has mode \( Q \) means that \( qasm \) code must be generated. Thus, each iteration will compile the
non \( C \)-mode components of the body, essentially inlining the loop. As an illustration, if we were to initialize \( n \)
to 3, the partially evaluated program would be equivalent to the equation in Figure 15 (in \( qasm \) rather than \( qimp \)).

We have verified that compilation from \( qimp \) to \( qasm \) is correct, in Coq, with a caveat: Proofs for assignment statements are parameterized by correctness statements about the involved operators. Each Coq operator function has a correctness statement associated with it; e.g., we state that the \( qasm \) code produced by invoking \( rz\_adder \) for addition corresponds to an addition at the \( qimp \) level. In the case of \( rz\_adder \) and a few others, we have a proof of this in Coq; for the rest, we use PBT to provide some assurance that the statement is true. Further details about \( qimp \) compilation and its correctness claims can be found in [Li et al. 2021] Appendix B.

## 5 EVALUATION: ARITHMETIC OPERATORS IN \( qasm \)

We evaluate \( vqo \) by (1) demonstrating how it can be used for validation, both by verification and random testing, and (2) showing that it gets good performance in terms of resource usage compared to Quipper, a state-of-the-art quantum programming framework [Green et al. 2013]. This section presents the arithmetic operators we have implemented in \( qasm \), while the next section discusses the geometric operators and expressions implemented in \( qimp \). The following section presents an end-to-end case study applying Grover’s search.

### 5.1 Implemented Operators

Figures 17 and 18 summarize the operators we have implemented in \( qasm \).

The addition and modular multiplication circuits (parts (a) and (d) of Figure 17) are components of the oracle used in Shor’s factoring algorithm [Shor 1994], which accounts for most of the algorithm’s cost [Gidney and Ekerå 2021]. The oracle performs modular exponentiation on natural numbers via modular multiplication, which takes a quantum variable \( x \) and two co-prime constants \( M, N \in \mathbb{N} \) and produces \((x \cdot M) \mod N\). We have implemented two modular multipliers—inspired by Beauregard [2003] and Markov and Saeedi [2012]—in \( qasm \). Both modular multipliers are constructed using...
controlled modular addition by a constant, which is implemented in terms of controlled addition and subtraction by a constant, as shown in Figure 16. The two implementations differ in their underlying adder and subtractor circuits: the first (QFT) uses a quantum Fourier transform-based circuit for addition and subtraction [Draper 2000], while the second (TOFF) uses a ripple-carry adder [Markov and Saeedi 2012], which uses classical controlled-controlled-not (Toffoli) gates.

Part (b) of Figure 17 shows results for \( \text{O}_{\text{QASM}} \) implementations of multiplication (without the modulo) and part (c) shows results for modular division by a constant, which is useful in Taylor series expansions used to implement operators like sine and cosine. Figure 18 lists additional operations we have implemented in \( \text{O}_{\text{QASM}} \) for arithmetic and Boolean comparison using natural and fixed-precision numbers.

### 5.2 Validating Operator Correctness

As shown in Figure 17, we have fully verified the adders and modular multipliers used in Shor’s algorithm. These constitute the first proved-correct implementations of these functions, as far as we are aware.

All other operations in the figure were tested with QuickChick. To ensure these tests were efficacious, we confirmed they could find hand-injected bugs; e.g., we reversed the input bitstrings for the QFT adder (Figure 3) and confirmed that testing found the endianness bug. The tables in Figure 17 give the running times for the QuickChick tests—the times include the cost of extracting the Coq code to OCaml, compiling it, and running it with 10,000 randomly generated inputs. We tested these operations both on 16-bit inputs (the number that’s relevant to the reported qubit and gate sizes) and 60-bit inputs. For the smaller sizes, tests complete in a few seconds; for the larger sizes, in a few minutes. For comparison, we translated the operators’ \( \text{O}_{\text{QASM}} \) programs to \( \text{sqir} \), converted the \( \text{sqir} \) programs to OpenQASM 2.0 [Cross et al. 2017], and then attempted to simulate the resulting circuits on test inputs using DDSIM [Burgholzer et al. 2021], a state-of-the-art quantum simulator. Unsurprisingly, the simulation of the 60-bit versions did not complete when running overnight.

We also verified and property-tested several other operations, as shown in Figure 18. We chose to verify the addition circuits because they are foundational components of other oracles. We additionally verified Toffoli/QFT-based modular multiplication and utility operations like subtraction and equality/inequality because they are critical components of Shor’s prime factoring algorithm, which we wanted to fully verify [Peng et al. 2022]. The remaining operations we randomly tested.
without proof. Some of these, such as Toffoli/QFT-based constant subtraction \((N - x)\) and multiplication \((N \times x)\), are similar to oracles that we verified already; e.g., the latter is just a loop over a (proved-correct) adder. Other operations, such as Toffoli/QFT-based division and modulo, would have been quite time-consuming to verify, especially since they use a variety of implementation techniques (e.g., three kinds of division/modulo algorithm in Figure 17 (c)). We felt that testing

---

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 146. Publication date: October 2022.
provided a good degree of assurance, and the time saved allowed us to better understand OQASM’s performance.

During development, we found two bugs in the original presentation of the QFT-based modular multiplier [Beauregard 2003]. The first issue was discovered via random testing and relates to assumptions about the endianness of stored integers. The binary number in Figure 6 of the paper uses a little-endian format whereas the rest of the circuit assumes big-endian. Quipper’s implementation of this algorithm solves the problem by using a function that reverses the order of qubits. In OQASM, we can use the Rev operation to correct the format of the input binary number. The second issue was discovered during verification. Beauregard [2003] indicates that the input $x$ should be less than $2^n$ where $n$ is the number of bits. However, to avoid failure the input must actually be less than $N$, where $N$ is the modulus defined in Shor’s algorithm. To complete the proof of correctness, we needed to insert a preprocessing step to change the input to $x \% N$. The original on-paper implementation of the ripple-carry-based modular multiplier [Markov and Saeedi 2012] has the same issue.

5.3 Operator Resource Usage

Figure 17 compares the resources used by OQASM operators with counterparts in Quipper. In both cases, we compiled the operators to OpenQASM 2.0 circuits, and then ran the circuits through the voqc optimizer [Hietala et al. 2021b] to ensure that the outputs account for inefficiencies in automatically-generated circuit programs (e.g., no-op gates inserted in the base case of a recursive function). voqc outputs the final result to use gates preferred by the Qiskit compiler [Cross 2018], which are the single-qubit gates $U_1, U_2, U_3$ and the two-qubit gate CNOT.

We also provide resource counts (computed by the same procedure) for our implementations of 8-bit modular multiplication. Quipper does not have a built-in operation for modular multiplication (which is different from multiplication followed by a modulo operator in the presence of overflow).

We define all of the arithmetic operations in Figure 17 for arbitrary input sizes; the limited sizes in our experiments (8 and 16 bits) are to account for inefficiencies in voqc. For the largest circuits (the modular multipliers), running voqc takes about 10 minutes.

Comparing QFT and Toffoli-based operators. The results show that the QFT-based implementations always use fewer qubits. This is because they do not need ancillae to implement reversibility. For both division/modulo and modular multiplication (used in Shor’s oracle), the savings are substantial because those operators are not easily reversible using Toffoli-based gates, and more ancillae are needed for uncomputation.

The QFT circuits also typically use fewer gates. This is partially due to algorithmic advantages of QFT-based arithmetic, partially due to voqc (voqc reduced QFT circuit gate counts by 57% and Toffoli circuit gate counts by 28%) and partially due to the optimized decompositions we use to convert many-qubit gates to the one- and two-qubit gates supported by voqc. We found during evaluation that gate counts are highly sensitive to the decompositions used: Using a more naïve decomposition of the controlled-Toffoli gate (which simply computes the controlled version of every gate in the standard Toffoli decomposition) increased the size of our Toffoli-based modular multiplication circuit by 1.9x, and a similarly naïve decomposition of the controlled-controlled-$R_z$

---

6 We converted the output Quipper files to OpenQASM 2.0 using a compiler produced at Dalhousie University [Bian 2020].
7 We use the decompositions for Toffoli and controlled-Toffoli at https://qiskit.org/documentation/_modules/qiskit/circuit/library/standard_gates/x.html; the decomposition for controlled-$R_z$ at https://qiskit.org/documentation/_modules/qiskit/circuit/library/standard_gates/u1.html; and the decomposition for controlled-controlled-$R_z$ at https://quantumcomputing.stackexchange.com/questions/11573/controlled-u-gate-on-ibmq. The decompositions we use are all proved correct in the sqir development. All of the decompositions are ancilla free.
gate increased the size of our QFT-based modular multiplication circuit by 4.4x. We also found that
gate counts (especially for the Toffoli-based circuits) are sensitive to choice of constant parameter:
The QFT-based constant multiplication circuits had between 1320 and 1412 gates, while the Toffoli-
based circuits had between 988 and 2264. Unlike gate counts, qubit counts are more difficult to
optimize because they require fundamentally changing the structure of the circuit; this makes
QFT’s qubit savings for modular multiplication even more impressive.

Overall, our results suggest that QFT-based arithmetic provides better and more consistent
performance, so when compiling OQIMP programs (like the sine function in Figure 13) to OQASM,
we should bias towards using the QFT-based operators.

Comparing to Quipper. Overall, Figure 17(a)-(c) shows that operator implementations in OQASM
consume resources comparable to those available in Quipper, often using fewer qubits and gates,
both for Toffoli- and QFT-based operations. In the case of the QFT adder, the difference in results
is because the Quipper-to-OpenQASM converter we use has a more expensive decomposition
of controlled-\(R_z\) gates.\(^8\) In the other cases (all Toffoli-based circuits), we made choices when
implementing the oracles that improved their resource usage. Nothing fundamental stopped the
Quipper developers from having made the same choices, but we note they did not have the benefit
of the OQASM type system and PBT framework. Quipper has recently begun to develop a random
testing framework based on QuickCheck [Claessen and Hughes 2000], but it only applies to Toffoli-
based (i.e., classical) gates.

5.4 Approximate Operators

OQASM’s efficiently-simulable semantics can be used to predict the effect of using approximate
components, which enables a new workflow for optimizing quantum circuits: Given an exact
circuit implementation, replace a subcomponent with an approximate implementation; use vqo’s
PBT framework to compare the outputs between the exact and approximate circuits; and finally
decide whether to accept or reject the approximation based on the results of these tests, iteratively
improving performance.

In this section, we use vqo’s PBT framework to study the effect of replacing QFT circuits with
AQFT circuits (Figure 3) in addition and division/modulo circuits.

Approximate Addition. Figure 19(a) shows the results of replacing QFT with AQFT in the QFT
adder from Figure 17(a). As expected, a decrease in precision leads to a decrease in gate count. On
the other hand, our testing framework demonstrates that this also increases error (measured as
absolute difference accounting for overflow, maximized over randomly-generated inputs). Random
testing over a wider range of inputs suggests that dropping \(b\) bits of precision from the exact
QFT adder always induces an error of at most \(|2^b - 1|\). This exponential error suggests that the
“approximate adder” is not particularly useful on its own, as it is effectively ignoring the least
significant bits in the computation. However, it computes the most significant bits correctly: if the
inputs are both multiples of \(2^b\) then an approximate adder that drops \(b\) bits of precision will always
produce the correct result.

Exact Division/Modulo using an Approximate Adder. Even though the approximate adder is
not particularly useful for addition, there are still cases where it can be useful as a subcomponent.
For example, the modulo/division circuit relies on an addition subcomponent, but does not need
every bit to be correctly added.

\(^8\)Bian [2020] decomposes a controlled-\(R_z\) gate into a circuit that uses two Toffoli gates, an \(R_z\) gate, and an ancilla qubit. In
voqc, each Toffoli gate is decomposed into 9 single-qubit gates and 6 two-qubit gates. In contrast, vqo’s decomposition for
controlled-\(R_z\) uses 3 single-qubit gates, 2 two-qubit gates, and no ancilla qubits.
Figure 20(a) shows one step of an N-bit QFT-based modulo circuit that computes $x \mod n$ for constant $n$. The algorithm runs for $I + 1$ iterations, where $2^{N-1} \leq 2^I n < 2^N$, with the iteration counter $i$ increasing from 0 to $I$ (inclusive). In each iteration, the circuit in Figure 20(a) computes $x - 2^I n$ and uses the result’s most significant bit (MSB) to check whether $x < 2^{N-1-i}$. If the MSB is 0, then $x \geq 2^{N-1-i}$ and the circuit continues to next iteration; otherwise, it adds $2^I n$ to the result and continues.

We can improve the resource usage of the circuit in Figure 20(a) by replacing the addition, subtraction, and QFT components with approximate versions, as shown in Figure 20(b). At the start of each iteration, $x < 2^{N-i}$, so it is safe to replace components with versions that will perform the intended operation on the lowest $(N - i)$ bits. The circuit in Figure 20(b) begins by subtracting the top $(N - i)$ bits, and then converts $x$ back to the $\text{Nor}$ basis using an $(N - i)$-bit precision QFT. It then swaps the MSB with an ancilla, guaranteeing that the MSB is 0. Next, it uses a $\text{Rshft}$ to move the cleaned MSB to become the lowest significant bit (effectively, multiplying $x$ by 2) and uses a $(N - i - 1)$-bit precision QFT to convert back to the $\Phi$ basis. Finally, it conditionally adds back the top $(N - i - 1)$ bit of the value $(2^I n \mod 2^{I-i})$, ignoring the original MSB.

The result is a division/modulo circuit that uses approximate components, but, as our testing assures, is exactly correct. Figure 19(b) shows the required resources for varying numbers of iterations. Compared to the QFT-based circuit, for a single iteration, the approximation provides a
4.5% savings. And the saving increases with more iterations. In the case of the maximum number of iterations (16 for \( n = 1 \)), the AQFT-based division/modulo circuit uses 61.1% fewer gates than the QFT-based implementation and 79.3% fewer gates than the Toffoli-based implementation.

6 EVALUATION: \( \texttt{OQIMP} \) ORACLES

The prior section considered arithmetic operators implemented in \( \texttt{OQASM} \), which are the building blocks for operators we have programmed using \( \texttt{OQIMP} \), including sine, arcsine, cosine, arccosine, and exponentiation on fixed-precision numbers. These operators are useful in near-term applications; for example, the arcsine and sine functions are used in the quantum walk algorithm [Childs 2009]. We used \( \texttt{OQIMP} \)'s source semantics to test each operator's correctness.

As discussed in Section 4.4, one of the key features of \( \texttt{OQIMP} \) is partial evaluation during compilation to \( \texttt{OQASM} \). A simple optimization similar to partial evaluation happens for a binary operation \( x = x \odot y \), where \( y \) is a constant value. Figure 17 hints at the power of partial evaluation for this case—all constant operations (marked "const") generate circuits with significantly fewer qubits and gates. Languages like Quipper take advantage of this by producing special circuits for operations that use classically-known constant parameters.

Partial evaluation takes this one step further, pre-evaluating as much of the circuit as possible. For example, consider the fixed precision operation \( x = \frac{xy}{M} \) where \( M \) is a constant natural number, and \( x \) and \( y \) are two fixed precision numbers that may be constants. This is a common pattern, appearing in many quantum oracles (recall the \( \frac{\sin x}{n!} \) in the Taylor series decomposition of sine). In Quipper, this is expression compiled to \( r_1 = \frac{x}{M}; r_2 = r_1 * y \). The \( \texttt{OQIMP} \) compiler produces different outputs depending on whether \( x \) and \( y \) are constants. If they both are constant, \( \texttt{OQIMP} \) simply assigns the result of computing \( \frac{xy}{M} \) to a quantum variable. If \( x \) is a constant, but \( y \) is not, \( \texttt{OQIMP} \) evaluates \( \frac{xy}{M} \) classically, assigns the value to \( r_1 \), and evaluates \( r_2 \) using a constant multiplication circuit. If they are both quantum variables, \( \texttt{OQIMP} \) generates a circuit to evaluate the division first and then the multiplication.

In Figure 21 (a) we show the size of the circuit generated for \( \frac{xy}{M} \) where zero, one, or both variables are classically known. As expected, more classical variables in a program lead to a more efficient output circuit. If \( x \) and \( y \) are both constants, then only a constant assignment circuit is needed, which is a series of \( X \) gates. Even if only one variable is constant, it may lead to substantial savings: In this example, if \( x \) is constant, the compiler can avoid the division circuit and use a constant multiplier instead of a general multiplier. These savings quickly add up: Figure 21 (b) shows the qubit size difference between our implementation of sine and Quippers'. Both the TOFF and QFT-based circuits use fewer than 7% of the qubits used by Quipper’s sine implementation.\(^9\)

\(^9\)\( \texttt{OQIMP} \) also benefits from its representation of fixed-precision numbers (Section 4.4), which is more restrictive than Quipper’s. Our representation of fixed-precision numbers reduces the qubit usage of the sine function by half, so about half of the qubit savings can be attributed to this.
Thus, the emergence of quantum computers may require lengthening hash functions. Grover’s search algorithm [Grover 1996, 1997], described in Section 2, has implications for cryptography, in part because it can be used to find collisions in cryptographic hash functions [Bernstein 2010]. Thus, the emergence of quantum computers may require lengthening hash function outputs.

We have used \texttt{QIMP} to implement the ChaCha20 stream cipher [Bernstein 2008] as an oracle for Grover’s search algorithm. This cipher computes a hash of a 256-bit key, a 64-bit message number, and a 64-bit block number, and it is actively being used in the TLS protocol [Langley et al. 2016; Rescorla 2018]. The procedure consists of twenty cipher rounds, most easily implemented when segmented into quarter-round and double-round subroutines. The only operations used are bitwise xor, bit rotations, and addition modulo $2^{32}$, all of which are included in \texttt{QIMP}; the implementation is given in Figure 22.

To test our oracle implementation, we wrote our specification as a Coq function on bitstrings. We then defined a correspondence between these bitstrings and program states in the \texttt{QASM} semantics and conjectured that, for any input, the semantics of our compiled oracle matches the outputs from our specification function. Using random testing (Section 4.3), we individually tested the quarter-round and double-round subroutines as well as the whole twenty-round cipher, performing a sort of unit testing. We also tested the oracle for the boolean-valued function that checks whether the ChaCha20 output matches a known bitstring. This oracle can be compiled to \texttt{SQIR} using our verified compiler, and then the compiled oracle can be used by Grover’s algorithm to invert the ChaCha20 function and find collisions. Grover’s algorithm was previously implemented and verified in \texttt{SQIR} [Hietala et al. 2021a], and we have modified this implementation and proof to allow for oracles with ancillae like the ones generated by our compiler; thus, our successful QuickChick tests combined with the previously proved theorems for Grover’s algorithm provide confidence that we can find ChaCha20 hash collisions with a certain probability using Grover’s algorithm.

---

Fig. 22. ChaCha20 implementation in \texttt{QIMP}

```python
Q nat[4] qr(Q nat x_1, Q nat x_2, Q nat x_3, Q nat x_4) {
  x_1 += x_2; x_4 ⊕= x_1; x_4 <<< 16;
  x_3 += x_4; x_2 ⊕= x_3; x_4 <<< 12;
  x_1 += x_2; x_2 ⊕= x_1; x_4 <<< 8;
  x_3 += x_4; x_2 ⊕= x_3; x_4 <<< 7
  return [x_1, x_2, x_3, x_4];
}

void chacha20(Q nat[16], x) {
  for(C nat i = 20; i > 0; i -= 2) {
    [x[0], x[4], x[8], x[12]] = qr(x[0], x[4], x[8], x[12]);
    [x[1], x[5], x[9], x[13]] = qr(x[1], x[5], x[9], x[13]);
    [x[2], x[6], x[10], x[14]] = qr(x[2], x[6], x[10], x[14]);
    [x[3], x[7], x[11], x[15]] = qr(x[3], x[7], x[11], x[15]);
    [x[10], x[5], x[10], x[15]] =qr(x[0], x[5], x[10], x[15]);
    [x[11], x[6], x[11], x[12]] = qr(x[11], x[6], x[11], x[12]);
    [x[2], x[7], x[8], x[13]] = qr(x[2], x[7], x[8], x[13]);
    [x[3], x[4], x[9], x[14]] = qr(x[3], x[4], x[9], x[14]);
  }
}
```

---

7 CASE STUDY: GROVER’S SEARCH

Here we present a case study of integrating an oracle implemented with \texttt{QOO} into a full quantum algorithm, Grover’s search algorithm, implemented and verified in \texttt{SQIR}.

Grover’s search algorithm [Grover 1996, 1997], described in Section 2, has implications for cryptography, in part because it can be used to find collisions in cryptographic hash functions [Bernstein 2010]. Thus, the emergence of quantum computers may require lengthening hash function outputs.

To test our oracle implementation, we wrote our specification as a Coq function on bitstrings. We then defined a correspondence between these bitstrings and program states in the \texttt{QASM} semantics and conjectured that, for any input, the semantics of our compiled oracle matches the outputs from our specification function. Using random testing (Section 4.3), we individually tested the quarter-round and double-round subroutines as well as the whole twenty-round cipher, performing a sort of unit testing. We also tested the oracle for the boolean-valued function that checks whether the ChaCha20 output matches a known bitstring. This oracle can be compiled to \texttt{SQIR} using our verified compiler, and then the compiled oracle can be used by Grover’s algorithm to invert the ChaCha20 function and find collisions. Grover’s algorithm was previously implemented and verified in \texttt{SQIR} [Hietala et al. 2021a], and we have modified this implementation and proof to allow for oracles with ancillae like the ones generated by our compiler; thus, our successful QuickChick tests combined with the previously proved theorems for Grover’s algorithm provide confidence that we can find ChaCha20 hash collisions with a certain probability using Grover’s algorithm.
8 RELATED WORK

**Oracles in Quantum Languages.** Quantum programming languages have proliferated in recent years. Many of these languages (e.g., Quil [Rigetti Computing 2019], OpenQASM 2.0 [Cross et al. 2017], sqir [Hietala et al. 2021b]) describe low-level circuit programs and provide no abstractions for describing quantum oracles. Higher-level languages may provide library functions for performing common oracle operations (e.g., Q# [Microsoft 2017], Scaffold [Abhari et al. 2012; Litteken et al. 2020]) or support compiling from classical programs to quantum circuits (e.g., Quipper [Green et al. 2013]), but still leave some important details (like uncomputation of ancilla qubits) to the programmer. vqo is most similar to Quipper in terms of programmer experience: an oracle can be either programmed as a circuit in $O_{qasm}$, or programmed as a classical expression in $O_{qimp}$ and then compiled to a circuit. $O_{qimp}$ also sometimes requires uncomputation annotations but is C-like rather than functional (Quipper uses Haskell). One added benefit of vqo is its support for randomized testing, which allows programmers to troubleshoot their applications and debug their programs more quickly, especially when quantum techniques like QFT are used. vqo also offers more flexible compilation, discussed below.

There has been some work on type systems to enforce that uncomputation happens correctly (e.g. Silq [Bichsel et al. 2020]), and on automated insertion of uncomputation circuits (e.g. Quipper [Green et al. 2013], Unqomp [Paradis et al. 2021]), but while these approaches provide useful automation, they also lead to inefficiencies in compiled circuits. In particular, all of these tools force compilation into the classical gate set $X$, CNOT, and CCNOT (or “Toffoli”), which precludes the use of QFT-based arithmetic, which uses fewer qubits than Toffoli-based approaches. Of course, programmers are not obligated to use automation for constructing oracles—they can do it by hand for greater efficiency—but this risks mistakes. vqo allows programmers to produce oracles automatically from $O_{qimp}$ using $\text{inv}$ to uncompute, or to manually implement oracle functions in $O_{qasm}$, in both cases supporting formal verification and testing.

**Verified Quantum Programming.** Recent work on formally verifying quantum programs includes $\text{Qwire}$ [Rand 2018], sqir [Hietala et al. 2021a], and $\text{Qbricks}$ [Chareton et al. 2021]. These tools have been used to verify a range of quantum algorithms, from Grover’s search to quantum phase estimation. Like these tools, properties of $O_{qasm}$ programs are expressed and verified in a proof assistant. But, unlike these tools, we focus on a quantum sub-language that, while not able to express any quantum program, is efficiently simulatable. This allows us to reuse existing infrastructure (like QuickChick [Paraskevopoulou et al. 2015]) for testing Coq properties.

**Verified Compilation of Quantum Programs.** Recent work has looked at verified optimization of quantum circuits (e.g., voqc [Hietala et al. 2021b], CertiQ [Shi et al. 2019]), but the problem of verified compilation from high-level languages to quantum circuits has received less attention. The only examples of verified compilers for quantum circuits are ReVerC [Amy et al. 2017] and ReQWIRE [Rand et al. 2018]. Both of these tools support verified translation from a low-level Boolean expression language to circuits consisting of $X$, CNOT, and CCNOT gates. Compared to these tools, vqo supports both a higher-level classical source language ($O_{qimp}$) and a more interesting quantum target language ($O_{qasm}$).

9 CONCLUSION

We presented vqo, a framework for expressing, testing, and verifying quantum oracles. The key component of vqo is $O_{qasm}$, the oracle quantum assembly language, which can express a restricted class of quantum programs that are efficiently simulatable (and hence testable) and are useful for implementing quantum oracles. We have verified the translator from $O_{qasm}$ to sqir and
have verified (or randomly tested) many arithmetic circuits written in $\mathcal{O}_{\text{QASM}}$. We also presented $\mathcal{O}_{\text{QIMP}}$, a high-level imperative language, and compiler from $\mathcal{O}_{\text{QIMP}}$ to $\mathcal{O}_{\text{QASM}}$ (framework verified and arithmetic operations randomly tested). We have used vqo to implement oracles and oracle components useful in quantum programming, like modular multiplication and sine, and showed that our performance is comparable to the state-of-the-art (unverified) framework Quipper. We also demonstrated the benefit of partial evaluation in $\mathcal{O}_{\text{QIMP}}$, showing that partial evaluation results in our implementation of sine using just 7% of the qubits used in Quipper’s implementation.

ACKNOWLEDGMENTS

We thank Leonidas Lampropoulos for helping us with effective use of QuickChick, and Aaron Green and Robert Rand for helpful comments and contributions during the development of this work. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, and the Air Force Office of Scientific Research under Grant No. FA95502110051 and FA95502110209.

REFERENCES

Ali Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Frederic Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. Scaffold: Quantum Programming Language. Technical Report. Princeton University.

Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In Computer Aided Verification, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–21.

Adriano Barenco, Artur Ekert, Kalle-Antti Suominen, and Päivi Törmä. 1996. Approximate quantum Fourier transform and decoherence. Physical Review A 54, 1 (Jul 1996), 139–146. https://doi.org/10.1103/physreva.54.139

Stephane Beauregard. 2003. Circuit for Shor’s Algorithm Using 2n+3 Qubits. Quantum Info. Comput. 3, 2 (March 2003), 175–185. https://doi.org/10.26421/QIC3.2-8

Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20 (The State of the Art of Stream Ciphers). ECRYPT Network of Excellence in Cryptology, 273–278. https://cr yp.to/papers.html#chacha

Daniel J. Bernstein. 2010. Grover vs. McEliece. In Post-Quantum Cryptography, Nicolas Sendrier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–80. https://doi.org/10.1007/978-3-642-12929-2_6

Xiaoning Bian. 2020. Compile Quipper quantum circuit to OpenQasm 2.0 program. https://www.mathstat.dal.ca/~xbian/QasmTrans/ [Online; accessed 8-July-2021].

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. 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 (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007

Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman Simulation of Quantum Circuits With Decision Diagrams. https://doi.org/10.48550/ARXIV.2105.07045 arXiv:2105.07045 [quant-ph]

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648), Nobuko Yoshida (Ed.). Springer, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

Andrew M. Childs. 2009. On the Relationship Between Continuous- and Discrete-Time Quantum Walk. Communications in Mathematical Physics 294, 2 (Oct 2009), 581–603. https://doi.org/10.1007/s00220-009-0930-1

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00). Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Andrew Cross. 2018. The IBM Q experience and Qiskit open-source quantum computing software. In APS Meeting Abstracts. Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. arXiv e-prints (Jul 2017). https://doi.org/10.48550/ARXIV.1707.03429 arXiv:1707.03429 [quant-ph]

Thomas G. Draper. 2000. Addition on a Quantum Computer. arXiv e-prints, Article quant-ph/0008033 (Aug, 2000), quant-ph/0008033 pages. https://doi.org/10.48550/ARXIV.QUANT-PH/0008033 arXiv:quant-ph/0008033 [quant-ph]
Richard P. Feynman. 1982. Simulating physics with computers. *International journal of theoretical physics* 21, 6/7 (1982), 467–488. https://doi.org/10.1007/BF02650179

Craig Gidney and Martin Ekerä. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (April 2021), 433. https://doi.org/10.22331/q-2021-04-15-433

Google Quantum AI. 2019. Cirq: An Open Source Framework for Programming Quantum Computers. https://quantumai.google/cirq

Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI ’13). 333–342. https://doi.org/10.1145/2491956.2462177

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) (STOC ’96). Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866 arXiv:quant-ph/9605043

Lov K. Grover. 1997. Quantum Mechanics Helps in Searching for a Needle in a Haystack. *Phys. Rev. Lett.* 79 (July 1997), 325–328. Issue 2. https://doi.org/10.1103/PhysRevLett.79.325 arXiv:quant-ph/9706033

L. Hales and S. Hallgren. 2000. An improved quantum Fourier transform algorithm and applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 515–525. https://doi.org/10.1109/SFCS.2000.892139

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021a. Proving Quantum Programs Correct. In *12th International Conference on Interactive Theorem Proving* (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs)). Vol. 193. Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:19. https://doi.org/10.4230/LIPIcs.ITP.2021.21

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan 2021), 29 pages. https://doi.org/10.1145/3434318

N.D. Jones, C.K. Gomard, and P. Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall. https://books.google.com/books?id=7rPScYo8w8C

A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. 2016. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905. https://doi.org/10.17487/RFC7905

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2021. Verified Compilation of Quantum Oracles. https://doi.org/10.48550/ARXIV.2112.06700

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. VQO: Verified Compilation of Quantum Oracles. https://doi.org/10.5281/zenodo.7068972

Andrew Litteken, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T Chong. 2020. An updated LLVM-based quantum research compiler with further OpenQASM support. *Quantum Science and Technology* 5, 3 (may 2020), 034013. https://doi.org/10.1088/2058-9565/ab82c2

Igor L. Markov and Mehdi Saeedi. 2012. Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation. *Quantum Inf. Comput.* 12, 5–6 (May 2012), 361–394.

Microsoft. 2017. The Q# Programming Language. https://docs.microsoft.com

Yunseong Nam, Yuan Su, and Dmitri Maslov. 2020. Approximate quantum Fourier transform with O(n log(n)) T gates. *npj Quantum Information* 6, 1 (2020). https://doi.org/10.1038/s41534-020-0257-5

Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information* (10th anniversary ed.). Cambridge University Press, USA.

Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) (PLDI ’21). Association for Computing Machinery, New York, NY, USA, 222?236. https://doi.org/10.1145/3453483.3454040

Zoe Paraskevopoulou, Cátálin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving*, Christian Urban and Xinyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. 2022. A Formally Certified End-to-End Implementation of Shor’s Factorization Algorithm. https://doi.org/10.48550/ARXIV.2204.07112

Robert Rand. 2018. *Formally verified quantum programming*. Ph. D. Dissertation. University of Pennsylvania.

Robert Rand, Jennifer Paykin, Dong-Ho Lee, and S. Zdancewic. 2018. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *QPL*. https://doi.org/10.4204/eptcs.287.17

E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. https://doi.org/10.17487/RFC8446

Rigetti Computing. 2019. The @rigetti optimizing Quil compiler. https://github.com/rigetti/quilc

Rigetti Computing. 2021. PyQuil: Quantum computing in Python. https://pyquil-docs.rigetti.com

Yunong Shi, Xupeng Li, Runzhou Tao, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2019. Contract-based verification of a realistic quantum compiler. *arXiv e-prints* (Aug 2019). arXiv:1908.08963 [quant-ph]
P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In Proceedings 35th Annual Symposium on Foundations of Computer Science. 124–134. https://doi.org/10.1109/SFCS.1994.365700

Rolando Somma. 2020. Are We Ready for Quantum Computers? Scientific American (2020). https://blogs.scientificamerican.com/observations/are-we-ready-for-quantum-computers

Angela Wilkins. 2021. The Way I See It: The State of Quantum Computing. Rice University (2021). https://news.rice.edu/news/2021/way-i-see-it-state-quantum-computing