Verified Compilation of Quantum Oracles

ANONYMOUS AUTHOR(S)

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 algorithm. 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 ℚASM, the *oracle quantum assembly language*. 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. ℚASM's design enabled us to prove correct VQO's compilers—from a simple imperative language called ℚIMP to ℚASM, and from ℚASM to SQIR, a general-purpose quantum assembly language—and allowed us to efficiently test properties of ℚASM programs using the QuickChick property-based testing framework. We have used VQO to implement oracles used in Shor's and Grover's algorithms, as well as several common arithmetic operators. VQO's oracles have performance comparable to those produced by Quipper, a state-of-the-art but unverified quantum programming platform. By using VQO, we can design correct and efficient quantum oracle circuits, especially, VQO enables the infrastructure to define correct and efficient QFT and approximate QFT based oracles.

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 time (compared to the sub-exponential time for the best known classical algorithm). An important source of speedups in these algorithms are 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 ℚIMP, which has standard imperative features and can express arbitrary classical programs.

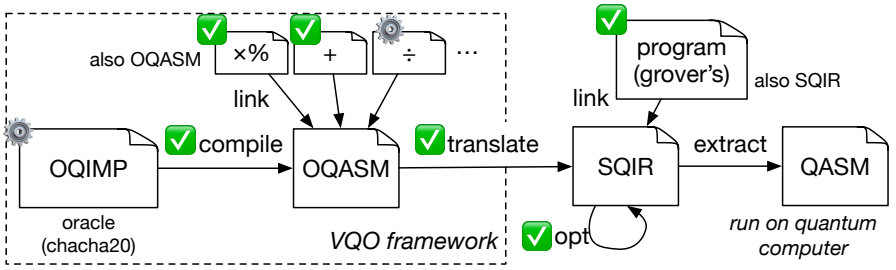


Fig. 1. The QVM high-assurance compiler stack. Checkbox means verified; gear means property-tested.

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 ℚIMP program is compiled to ℚASM (pronounced "O-chasm"), the *oracle quantum assembly language*. ℚASM 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 ℚASM code links against implementations of standard operators (addition, multiplication, sine, cosine, etc.) also written in ℚASM.
- The ℚASM oracle is then translated to SQIR, the *Simple Quantum Intermediate Representation*, which is a circuit language embedded in the Coq proof assistant. 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 VOQC, 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 ℚIMP to ℚASM and translation from ℚASM to SQIR have been proved correct in Coq.

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 ℚASM, a quantum assembly language specifically designed for oracles.

Because oracles are classical functions, a reasonable approach would have been to design ℚASM 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 (e.g., controlled-phase operations) 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, ℚASM's type system restricts programs to those that admit important optimizations while keeping simulation tractable. ℚASM 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 run-time cost.

Leveraging ℚASM's efficient simulatability, we implemented a *property-based random testing* (PBT) framework for ℚASM 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 ⟦QASM operator or ⟦QIMP 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 ⟦QIMP 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 ⟦QIMP's partial evaluation.
- We have implemented a variety of arithmetic operators in ⟦QASM, 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 ⟦QIMP 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 VQO'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 (URL redacted).

2 BACKGROUND

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

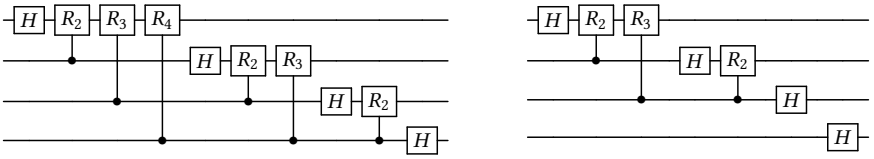


Fig. 2. Example quantum circuits: QFT over 4 qubits (left) and approximate QFT with 3 qubit precision (right). R_m is a z-axis rotation by $2\pi/2^m$.

Quantum States. A quantum state consists of one or more quantum bits (*qubits*). A qubit can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ where α, β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write the qubit vector as $\alpha\,|0\rangle + \beta\,|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *computational basis states*. When both α and β 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 by means of 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, e.g., Python (for Qiskit [Cross 2018], Cirq [Google Quantum AI 2019], PyQuil [Rigetti Computing 2021], and others), Haskell (for Quipper [Green et al. 2013]), or Coq (for SQIR and our work).

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 ℚASM, which allows efficient testing of the effect of AQFT inside of oracles.

Computational and QFT Bases. The computational basis 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 special, non-unitary *measurement* operation is used to extract 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 $f : \{0,1\}^n \to \{0,1\}$. 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., `X` for "not", `CNOT` for "xor", and `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 3.2]. 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 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.

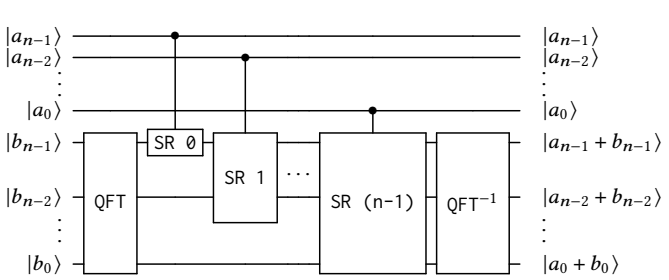
3 ⒪QASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

We designed ⒪QASM to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct. ⒪QASM operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT). ⒪QASM's type system tracks the bases of variables in ⒪QASM programs, forbidding operations that would introduce entanglement. ⒪QASM states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition, ⒪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 ⒪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 `rz_adder` generates an ⒪QASM program that adds two natural numbers `a` and `b`, each of length `n` qubits.

3.1 ⒪QASM States

An ⒪QASM program state is represented according to the grammar in Figure 4. A state φ 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 φ is $O(d)$ where d is the number of qubits. A d-qubit state in a language like SQIR is represented as a length 2^d vector of complex numbers, which is $O(2^d)$ in the number of qubits.



(a) Quantum circuit

```
Fixpoint rz_adder' (a b:var) (n:ℕ)
  := match n with
     | 0 ⇒ ID (a,0)
     | S m ⇒ CU (a,m) (SR m b);
              rz_adder' a b m
     end.
Definition rz_adder (a b:var) (n:ℕ)
  := Rev a ; Rev b ; QFT b ;
     rz_adder' a b n;
     QFT⁻¹ b; Rev b ; Rev a.
```

(b) 𝒪QASM metaprogram (in Coq)

Fig. 3. Example 𝒪QASM program: QFT-based adder

Bit	b	::=	$0 \mid 1$
Natural number	n	$\in$	$\mathbb{N}$
Real	r	$\in$	$\mathbb{R}$
Phase	$\alpha(r)$	::=	$e^{2\pi i r}$
Basis	τ	::=	Nor | Phi n
Unphased qubit	$\overline{q}$	::=	$\|b\rangle \mid \|\Phi(r)\rangle$
Qubit	q	::=	$\alpha(r)\overline{q}$
State (length d)	φ	::=	$q_1 \otimes q_2 \otimes \ldots \otimes q_d$

Fig. 4. 𝒪QASM state syntax

Position	p	::=	(x, n) Nat. Num n Variable x
Instruction	ι	::=	ID p | X p | ι ; ι
		|	$\mathsf{SR}^{[-1]}$ n x | $\mathsf{QFT}^{[-1]}$ n x | CU p ι
		|	Lshift x | Rshift x | Rev x

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

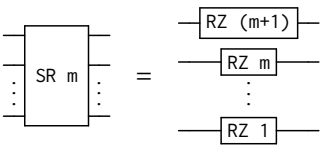


Fig. 6. SR unfolds to a series of RZ instructions

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 $\overline{q}$, scaled by a global phase $\alpha(r)$. The two forms depend on the *basis* τ 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 φ. As shown in Beauregard [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 ι. 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.

$$\text{X}\ \frac{\Omega(x) = \texttt{Nor} \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \texttt{X}\ (x, n) \triangleright \Omega} \qquad \text{SR}\ \frac{\Omega(x) = \texttt{Phi}\ n \qquad m < n}{\Sigma; \Omega \vdash \texttt{SR}\ m\ x \triangleright \Omega}$$

$$\text{QFT}\ \frac{\Omega(x) = \texttt{Nor} \qquad n \leq \Sigma(x)}{\Sigma; \Omega \vdash \texttt{QFT}\ n\ x \triangleright \Omega[x \mapsto \texttt{Phi}\ n]} \qquad \text{RQFT}\ \frac{\Omega(x) = \texttt{Phi}\ n \qquad n \leq \Sigma(x)}{\Sigma; \Omega \vdash \texttt{QFT}^{-1}\ n\ x \triangleright \Omega[x \mapsto \texttt{Nor}]}$$

$$\text{CU}\ \frac{\Omega(x) = \texttt{Nor} \qquad \texttt{fresh}\ (x, n)\ \iota \qquad \Sigma; \Omega \vdash \iota \triangleright \Omega \qquad \texttt{neutral}(\iota)}{\Sigma; \Omega \vdash \texttt{CU}\ (x, n)\ \iota \triangleright \Omega} \qquad \text{LSH}\ \frac{\Omega(x) = \texttt{Nor}}{\Sigma; \Omega \vdash \texttt{Lshift}\ x \triangleright \Omega}$$

$$\text{SEQ}\ \frac{\Sigma; \Omega \vdash \iota_1 \triangleright \Omega' \qquad \Sigma; \Omega' \vdash \iota_2 \triangleright \Omega''}{\Sigma; \Omega \vdash \iota_1\ ;\ \iota_2 \triangleright \Omega''}$$

Fig. 7. Select 𝔔ASM typing rules

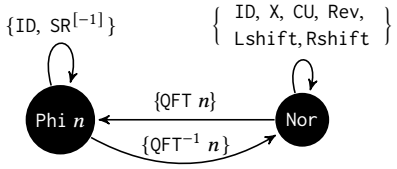


Fig. 8. Type rules' state machine

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 $\texttt{QFT}^{-1}$ n x applies its inverse. If n is equal to the size of x, then the AQFT operation is exact. $\texttt{SR}^{[-1]}$ n x applies a series of RZ gates (Figure 6). Operation CU p ι applies instruction ι *controlled* on qubit position p. All of the operations in this row—SR, QFT, and CU—will be translated to multiple SQIR gates. Function `rz_adder` in Figure 3(b) uses many of these instructions; e.g., it uses QFT and $\texttt{QFT}^{-1}$ and applies CU to the mth position of variable a to control instruction 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 𝔔ASM translator maintains a logical map of variables/positions to concrete qubits and ensures that shifting operations are no-ops, introducing no extra gates.

There are other quantum operations that could be added to 𝔔ASM to allow reasoning about a larger class of quantum programs, while still guaranteeing a lack of entanglement. In Appendix A, we show how 𝔔ASM can be extended to include the Hadamard gate H, z-axis rotations Rz, and a new basis Had 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 𝔔ASM, typing is with respect to a *type environment* Ω and a *size environment* Σ, which map 𝔔ASM variables to their basis and size (number of qubits), respectively. The typing judgment is written $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ which states that ι is well-typed under Ω and Σ, and transforms the variables' bases to be as in Ω' (Σ is unchanged). Select type rules are given in Figure 7; the rules not shown (for ID, Rshift, Rev, and $\texttt{SR}^{-1}$) 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 X) and that any control qubit is distinct from the target(s) (the fresh premise in CU). This latter property enforces

$$
\begin{array}{lll}
[\![\texttt{ID}\ p]\!]\varphi & = \varphi & \\
[\![\texttt{X}\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{xg}(\downarrow \varphi(x,i))] & \text{where } \texttt{xg}(|0\rangle) = |1\rangle \quad \texttt{xg}(|1\rangle) = |0\rangle \\
[\![\texttt{CU}\ (x,i)\ \iota]\!]\varphi & = \texttt{cu}(\downarrow \varphi(x,i), \iota, \varphi) & \text{where } \texttt{cu}(|0\rangle, \iota, \varphi) = \varphi \quad \texttt{cu}(|1\rangle, \iota, \varphi) = [\![\iota]\!]\varphi \\
[\![\texttt{SR}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i + \frac{1}{2^{m-i+1}})\rangle] & \text{when } \downarrow \varphi(x,i) = |\Phi(r_i)\rangle \\
[\![\texttt{SR}^{-1}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i - \frac{1}{2^{m-i+1}})\rangle] & \text{when } \downarrow \varphi(x,i) = |\Phi(r_i)\rangle \\
[\![\texttt{QFT}\ n\ x]\!]\varphi & = \varphi[x \mapsto \uparrow \texttt{qt}(\Sigma(x), \downarrow \varphi(x), n)] & \text{where } \texttt{qt}(i, |y\rangle, n) = \bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle) \\
[\![\texttt{QFT}^{-1}\ n\ x]\!]\varphi & = \varphi[x \mapsto \uparrow \texttt{qt}^{-1}(\Sigma(x), \downarrow \varphi(x), n)] & \\
[\![\texttt{Lshift}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_l(\varphi(x))] & \text{where } \texttt{pm}_l(q_0 \otimes q_1 \otimes \ldots \otimes q_{n-1}) = q_{n-1} \otimes q_0 \otimes q_1 \otimes \ldots \\
[\![\texttt{Rshift}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_r(\varphi(x))] & \text{where } \texttt{pm}_r(q_0 \otimes q_1 \otimes \ldots \otimes q_{n-1}) = q_1 \otimes \ldots \otimes q_{n-1} \otimes q_0 \\
[\![\texttt{Rev}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_a(\varphi(x))] & \text{where } \texttt{pm}_a(q_0 \otimes \ldots \otimes q_{n-1}) = q_{n-1} \otimes \ldots \otimes q_0 \\
[\![\iota_1; \iota_2]\!]\varphi & = [\![\iota_2]\!]([\![\iota_1]\!]\varphi) &
\end{array}
$$

$$
\begin{array}{l}
\downarrow \alpha(b)\overline{q} = \overline{q} \qquad \downarrow (q_1 \otimes \ldots \otimes q_n) = \downarrow q_1 \otimes \ldots \otimes \downarrow q_n \\
\varphi[(x,i) \mapsto \uparrow \overline{q}] = \varphi[(x,i) \mapsto \alpha(b)\overline{q}] \quad \text{where } \varphi(x,i) = \alpha(b)\overline{q_i} \\
\varphi[(x,i) \mapsto \uparrow \alpha(b_1)\overline{q}] = \varphi[(x,i) \mapsto \alpha(b_1 + b_2)\overline{q}] \quad \text{where } \varphi(x,i) = \alpha(b_2)\overline{q_i} \\
\varphi[x \mapsto q_x] = \varphi[\forall i < \Sigma(x).\ (x,i) \mapsto q_{(x,i)}] \\
\varphi[x \mapsto \uparrow q_x] = \varphi[\forall i < \Sigma(x).\ (x,i) \mapsto \uparrow q_{(x,i)}]
\end{array}
$$

Fig. 9. ⟦QASM semantics

the quantum *no-cloning rule*. For example, we can apply the CU in `rz_adder'` (Figure 3) because position `a,m` 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 $\texttt{QFT}^{-1}$ 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 $\texttt{QFT}^{-1}$ 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 ι 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 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 ⟦QASM program as a partial function $[\![\,]\!]$ from an instruction ι and input state φ to an output state φ', written $[\![\iota]\!]\varphi = \varphi'$, shown in Figure 9.

Recall that a state φ is a tuple of d qubit values, modeling the tensor product $q_1 \otimes \ldots \otimes q_d$. The rules implicitly map each variable x to a range of qubits in the state, e.g., $\varphi(x)$ corresponds to some sub-state $q_k \otimes \ldots \otimes q_{k+n-1}$ where $\Sigma(x) = n$. Many of the rules in Figure 9 update a *portion* of a state. We write $\varphi[(x,i) \mapsto q_{(x,i)}]$ to update the i-th qubit of variable x to be the (single-qubit) state $q_{(x,i)}$, and $\varphi[x \mapsto q_x]$ to update variable x according to the qubit *tuple* q_x. $\varphi[(x,i) \mapsto \uparrow q_{(x,i)}]$ and $\varphi[x \mapsto \uparrow q_x]$ 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 $\alpha(b)\overline{q}$ to an unphased qubit $\overline{q}$.

Function xg updates the state of a single qubit according to the rules for the standard quantum gate X. cu is a conditional operation depending on the Nor-basis qubit (x, i). SR (or $\texttt{SR}^{-1}$) applies an $m + 1$ series of RZ (or $\texttt{RZ}^{-1}$) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$

(or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes ... \otimes |b_i\rangle$ (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 $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \leq k$ (since $\alpha(n) = 1$ for any natural number n). $\mathtt{qt}^{-1}$ is the inverse function of qt. Note that the input state to $\mathtt{qt}^{-1}$ is guaranteed to have the form $\bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type Phi n. $\mathtt{pm}_l$, $\mathtt{pm}_r$, and $\mathtt{pm}_a$ are the semantics for Lshift, Rshift, and Rev, respectively.

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 φ is *well-formed*, written $\Sigma; \Omega \vdash \varphi$, iff:

- For every $x \in \Omega$ such that $\Omega(x) = \mathtt{Nor}$, for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|b\rangle$.
- For every $x \in \Omega$ such that $\Omega(x) = \mathtt{Phi}\ n$ and $n \leq \Sigma(x)$, there exists a value v such that for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|\Phi(\frac{v}{2^{n-k}})\rangle$.[1]

Type soundness is stated as follows; the proof is by induction on ι, and is mechanized in Coq.

Theorem 3.2. [Ⓞqasm type soundness] If $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma; \Omega \vdash \varphi$ then there exists φ' such that $[\![\iota]\!]\varphi = \varphi'$ and $\Sigma; \Omega' \vdash \varphi'$.

Algebra. Mathematically, the set of well-formed d-qubit Ⓞqasm states for a given Ω can be interpreted as a subset $\mathcal{S}^d$ of a 2^d-dimensional Hilbert space $\mathcal{H}^d$,[2] and the semantics function $[\![\,]\!]$ can be interpreted as a $2^d \times 2^d$ unitary matrix, as is standard when representing the semantics of programs without measurement [Hietala et al. 2021a]. Because Ⓞqasm's semantics can be viewed as a unitary matrix, 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.

Given a qubit size map Σ and type environment Ω, the set of Ⓞqasm programs that are well-typed with respect to Σ and Ω (i.e., $\Sigma; \Omega \vdash \iota \triangleright \Omega'$) form a groupoid $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d)$, where $\mathcal{S}^d$ is the set of d-qubit states that are well-formed ($\Omega \vdash \varphi$) according to Definition 3.1.

We can extend the groupoid to $(\{\iota\}, \Sigma, \mathcal{H}^d)$ by defining a general 2^d dimensional Hilbert space $\mathcal{H}^d$, such that $\mathcal{S}^d \subseteq \mathcal{H}^d$, and removing the typing requirements on $\{\iota\}$. Clearly, $(\{\iota\}, \Sigma, \mathcal{H}^d)$ is still a groupoid because every Ⓞqasm operation is valid in a traditional quantum language like sqir. We then have the following two two theorems to connect Ⓞqasm operations with operations in the general Hilbert space:

Theorem 3.3. $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d) \subseteq (\{\iota\}, \Sigma, \mathcal{H}^d)$ is a subgroupoid.

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 $i \in [0, 2^d)$, $[\![\iota]\!]\,|y_i\rangle = |y_i'\rangle$, then $[\![\iota]\!](\sum_{i=0}^{2^d-1} |y_i\rangle) = \sum_{i=0}^{2^d-1} |y_i'\rangle$.

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 any well-typed Ⓞqasm operation. Theorem 3.4 says that Ⓞqasm oracles can be safely applied to superpositions over classical states.[3]

[1] Note that $\Phi(x) = \Phi(x + n)$, where the integer n refers to phase $2\pi n$; so multiple choices of v 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 sub*set*, not a sub*space* 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.

$$\texttt{X}\ (x,n) \xrightarrow{\text{inv}} \texttt{X}\ (x,n) \qquad \texttt{SR}\ m\ x \xrightarrow{\text{inv}} \texttt{SR}^{-1}\ m\ x \qquad \texttt{QFT}\ n\ x \xrightarrow{\text{inv}} \texttt{QFT}^{-1}\ n\ x \qquad \texttt{Lshift}\ x \xrightarrow{\text{inv}} \texttt{Rshift}\ x$$

$$\frac{\iota \xrightarrow{\text{inv}} \iota'}{\texttt{CU}\ (x,n)\ \iota \xrightarrow{\text{inv}} \texttt{CU}\ (x,n)\ \iota'} \qquad \frac{\iota_1 \xrightarrow{\text{inv}} \iota_1' \qquad \iota_2 \xrightarrow{\text{inv}} \iota_2'}{\iota_1\ ;\ \iota_2 \xrightarrow{\text{inv}} \iota_2'\ ;\ \iota_1'}$$

Fig. 10. Select ⟦QASM inversion rules

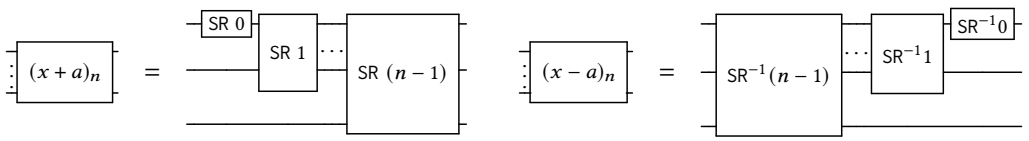


Fig. 11. Addition/subtraction circuits are inverses

$$\frac{}{\Sigma \vdash (\gamma, \texttt{X}\ p) \longrightarrow (\gamma, \texttt{X}\ \gamma(p))} \qquad \frac{\gamma' = \gamma[\forall i.\ i < \Sigma(x) \Rightarrow (x,i) \mapsto \gamma(x, (i+1)\%\Sigma(x))]}{\Sigma \vdash (\gamma, \texttt{Lshift}\ x) \longrightarrow (\gamma', \texttt{ID}\ (\gamma'(x,0)))}$$

$$\frac{\Sigma \vdash (\gamma,\iota) \longrightarrow (\gamma,\epsilon) \qquad \epsilon' = \texttt{ctrl}(\gamma(p),\epsilon)}{\Sigma \vdash (\gamma, \texttt{CU}\ p\ \iota) \longrightarrow (\gamma,\epsilon')} \qquad \frac{\Sigma \vdash (\gamma,\iota_1) \longrightarrow (\gamma',\epsilon_1) \qquad \Sigma \vdash (\gamma',\iota_2) \longrightarrow (\gamma'',\epsilon_2)}{\Sigma \vdash (\gamma, \iota_1\ ;\ \iota_2) \longrightarrow (\gamma'', \epsilon_1\ ;\ \epsilon_2)}$$

Fig. 12. Select ⟦QASM to SQIR translation rules (SQIR circuits are marked blue)

⟦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 10). 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 ι, such that $\Sigma; \Omega \vdash \iota \triangleright \Omega'$, its inverse ι', where $\iota \xrightarrow{\text{inv}} \iota'$, is also well-typed and we have $\Sigma; \Omega' \vdash \iota' \triangleright \Omega$. Moreover, $[\![\iota; \iota']\!]\varphi = \varphi$.

4 VQO QUANTUM ORACLE FRAMEWORK

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

4.1 Translation from ⟦QASM to SQIR

VQO translates ⟦QASM to SQIR by mapping ⟦QASM positions to SQIR concrete qubit indices and expanding ⟦QASM instructions to sequences of SQIR gates. Translation is expressed as the judgment $\Sigma \vdash (\gamma, \iota) \longrightarrow (\gamma', \epsilon)$ where Σ maps ⟦QASM variables to their sizes, ϵ is the output SQIR circuit, and γ maps an ⟦QASM position p to a SQIR concrete qubit index (i.e., offset into a global qubit register). At the start of translation, for every variable x and $i < \Sigma(x)$, γ 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 SQIR. The second rule left-shifts the qubits of the target variable in the map γ, 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 γ and variable x has three qubits so γ is $\{(x,0) \mapsto 0, (x,1) \mapsto 1, (x,2) \mapsto 2, (y,0) \mapsto 3, ...\}$. Then after `Lshift` x the γ map becomes $\{(x,0) \mapsto 1, (x,1) \mapsto 2, (x,2) \mapsto 0, (y,0) \mapsto 3, ...\}$. The last two rules translate the `CU` and sequencing instructions. In the `CU` translation, the rule assumes that ι's translation does not affect the γ 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 ⟦QASM-to-SQIR translation correct. To formally state the correctness property we relate d-qubit ⟦QASM states to SQIR states, which are vectors of 2^d complex numbers, via a function $[\![-]\!]^d_\gamma$, where γ 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 $\gamma = \{(x,0) \mapsto 0, (x,1) \mapsto 1, (y,0) \mapsto 2, (y,1) \mapsto 3\}$. This ⟦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. [⟦QASM translation correctness] Suppose $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma \vdash (\gamma, \iota) \longrightarrow (\gamma', \epsilon)$. Then for $\Sigma; \Omega \vdash \varphi$, $[\![\iota]\!]\varphi = \varphi'$, and we have $[\![\epsilon]\!] \times [\![\varphi]\!]^d_\gamma = [\![\varphi']\!]^d_{\gamma'}$ where $[\![\epsilon]\!]$ is the matrix interpretation of ϵ per SQIR's semantics.

The proof of translation correctness is by induction on the ⟦QASM program ι. Most of the proof simply shows the correspondence of operations in ι to their translated-to gates ϵ in SQIR, except for shifting operations, which update the virtual-to-physical map.

Note that to link a complete, translated oracle ι into a larger SQIR program may require that $\gamma = \gamma'$, i.e., `neutral`(ι), 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., `rz_adder` from Figure 3.

4.2 Property-based Random Testing

⟦QASM's type system ensures that states can be efficiently represented. We leverage this fact to implement a testing framework for ⟦QASM programs using QuickChick [Paraskevopoulou et al. 2015], which is a property-based testing (PBT) framework for Coq in the style of Haskell's QuickCheck [Claessen and Hughes 2000]. We use this framework for two purposes: To test correctness properties of ⟦QASM programs and to experiment with the impact of approximation on efficiency and correctness.

Implementation. PBT randomly generates inputs using a hand-crafted *generator*, and confirms that a property holds for the given inputs. Since arithmetic/oracle operations are defined over `Nor`-basis inputs, we wrote a generator for these inputs. A single test of an ⟦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 $b_0 b_1 ... b_{n-1}$ of length n, representing x's initial qubit value $\bigotimes_{i=0}^{n-1} \alpha(0) |b_i\rangle$; (3) prepare an ⟦QASM state φ containing all input variables' encoded bitstrings; (4) execute the ⟦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. First, we streamlined the representation of states: Per the semantics in Figure 9, in a state with n qubits, the phase associated with each qubit

[4] Translation in actual fact threads through the typing judgment, but we elide that for simplicity.

```
fixedp sin(Q fixedp x/8, Q fixedp x_r, C nat n){
 x_r = x/8; C fixedp n_y; Q fixedp x_z; 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++){
   n_1 = i + 1; n_2 = 2 * n_1; n_3 = pow(8, n_2); n_4 = n_2 + 1;
   n_5 = n_4!; n_y = n_3/n_5; x_z = pow(x/8, n_4);
   if (even(n_1)) {x_1 = n_y * x_z; x_r += x_1;}
   else {x_1 = n_y * x_z; x_r -= x_1;};
   inv(x_1); inv(x_z);}
 return (8 * x_r);
}
```

$$\sin x \approx 8 * (\frac{x}{8} - \frac{8^2}{3!}(\frac{x}{8})^3 + \frac{8^4}{5!}(\frac{x}{8})^5 - \frac{8^6}{7!}(\frac{x}{8})^7 + \ldots + (-1)^{n-1}\frac{8^{2n-2}}{(2n-1)!}(\frac{x}{8})^{2n-1})$$

Fig. 13. Implementing sine in ℚIMP

can be written as $\alpha(\frac{v}{2^n})$ 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 $|\Phi(\frac{y}{2^n})\rangle$). An ℚASM state φ 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 ℚASM semantics function tail-recursively. To run the tests, QuickChick runs OCaml code that it *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.

Testing Correctness. A 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, necessitates nontrivial changes to the program. Our PBT framework gives assurance that an 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 ℚIMP programs by first compiling them to ℚASM and then testing their correctness at that level.

Assessing the Effect of Approximation. Because of the resource limitations of near-term machines, programmers may wish to *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 *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.

4.3 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 `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 ℚIMP, a higher-level imperative language that compiles to

ⓆASM. Here we discuss ⓆIMP's basic features, describe how we optimize ⓆIMP programs during compilation using partial evaluation, and provide correctness guarantees for ⓆIMP programs. Using ⓆIMP, we have defined operations for the ChaCha20 hash-function [Bernstein 2008], exponentiation, sine, arcsine, and cosine, and tested program correctness by running inputs through ⓆIMP's semantics. More details about ⓆIMP are available in Appendix B.

Language Features. An ⓆIMP 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. ⓆIMP statements contain variable declarations, assignments (e.g., x_r `=` $x_{/8}$), arithmetic computations (n_1 `=` $i+1$), loops, conditionals, and function calls. Variables x have types τ, which are either primitive types ω^m or arrays thereof, of size n. A primitive type pairs a base type ω with a *quantum mode* m. There are three base types: type `nat` indicates non-negative (natural) numbers; type `fixedp` indicates fixed-precision real numbers in the range $(-1, 1)$; and type `bool` represents booleans. The programmer specifies the number of qubits to use to represent `nat` and `fixedp` numbers when invoking the ⓆIMP 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 run-time.

Figure 13 shows the ⓆIMP implementation of the sine function, which is used in quantum applications such as Hamiltonian simulation [Childs 2009; Feynman 1982]. Because `fixedp` 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*, ⓆIMP 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 ⓆIMP's `inv` x operation. For example, in Figure 13, the second `inv` operation returns x_z to its state prior to the execution of x_z=`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 `inv` x calls automatically, but doing so requires care in order to avoid blowup in the generated circuit [Paradis et al. 2021].

The VQO compiler imposes three restrictions on the use of `inv` x, which aim to ensure that each use uncomputes just one assignment to x. First, since the semantics of an `inv` operation reverses the most recent assignment, we require that every `inv` operation have a definite predecessor. Example `(1)` in Figure 14 shows an `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 ⓆIMP. Second, the statements between an `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 `inv` and the predecessor. The third restriction is that, while sequenced `inv` operations are allowed, the number of `inv` operations must match the number of predecessors. Example `(4)` is invalid, while `(5)` is valid, because the first `inv` in `(5)` matches the multiplication assignment and the second `inv` matches the addition assignment.

To implement these well-formedness checks, VQO's ⓆIMP compiler maintains a stack of assignment statements. Once the compiler hits an `inv` operation, it pops statements from the stack to

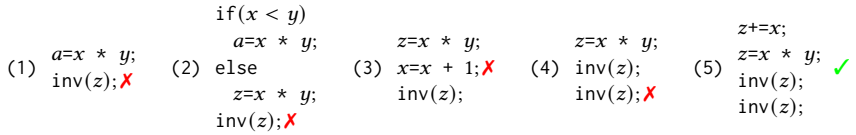


Fig. 14. Example (in)valid uses of `inv`

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* ℚIMP *to* ℚASM.** The ℚIMP compiler performs *partial evaluation* [Jones et al. 1993] on the input program with respect to classical parameters; the residual program is compiled to a quantum circuit. In particular, we compile an ℚIMP program by evaluating its C-mode components, storing the results in a store σ, and then using these results while translating its Q-mode components into ℚASM code. For example, when compiling the `for` loop in Figure 13, the compiler will look up the value of loop-bound variable n in the store and update 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_z=pow($x_{/8}$, n_4), the fact that x_z 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 following (in ℚASM rather than ℚIMP).

$$\begin{aligned} x_r \mathrel{+}= x_{/8}; x_z &= \mathsf{pow}(x_{/8}, 3); x_1 = \tfrac{8^2}{3!} * x_z;\ x_r \mathrel{-}= x_1; \mathsf{inv}\, x_1;\ \mathsf{inv}\, x_z; \\ x_z &= \mathsf{pow}(x_{/8}, 5); x_1 = \tfrac{8^4}{5!} * x_z;\ x_r \mathrel{+}= x_1; \mathsf{inv}\, x_1;\ \mathsf{inv}\, x_z; \\ x_z &= \mathsf{pow}(x_{/8}, 7); x_1 = \tfrac{8^6}{7!} * x_z;\ x_r \mathrel{-}= x_1; \mathsf{inv}\, x_1;\ \mathsf{inv}\, x_z; \end{aligned}$$

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 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) by 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

Figure 16 and Figure 17 tabulate the arithmetic operators we have implemented in ℚASM.

The addition and modular multiplication circuits (parts (a) and (d) of Figure 16) are components of the oracle used in Shor's factoring algorithm [Shor 1994], which accounts for the vast majority 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

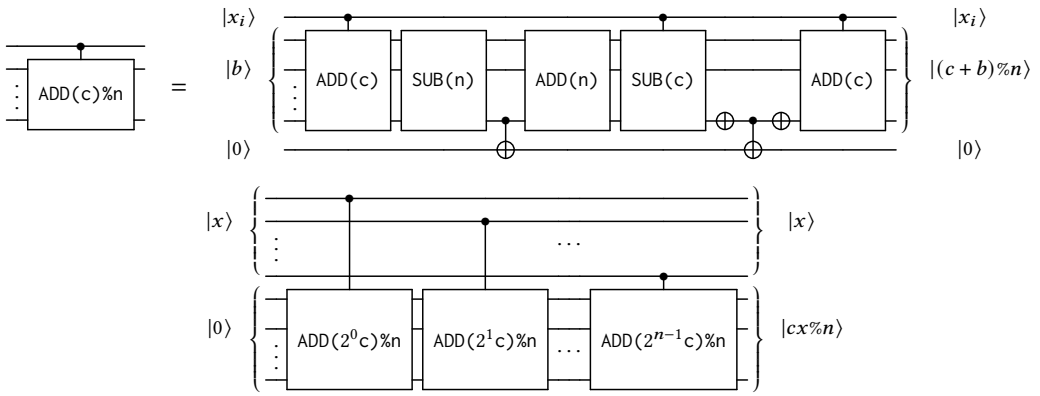


Fig. 15. Structure of modular multiplication circuits

constants $M, N \in \mathbb{N}$ and produces $(x * M)\%N$. 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 15. 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 makes use of classical controlled-controlled-not (Toffoli) gates.

Part (b) of Figure 16 shows results for 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 17 lists additional operations we have implemented in ℚASM for arithmetic and Boolean comparison using natural and fixed-precision numbers.

5.2 Validating Operator Correctness

As shown in Figure 16, 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 make sure 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 16 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' ℚASM programs to SQIR, converted the SQIR programs to OpenQASM 2.0 [Cross et al. 2017], and then attempted to simulate the resulting circuits on test inputs using the 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 17.

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

	# qubits	# gates	Verified
ⓠQASM TOFF	33	423	✓
ⓠQASM QFT	32	1206	✓
ⓠQASM QFT (const)	16	756 ± 42	✓
Quipper TOFF	47	768	
Quipper QFT	33	6868	
Quipper TOFF (const)	31	365 ± 11	

(a) Addition circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
ⓠQASM TOFF	49	11265	6 / 74
ⓠQASM TOFF (const)	33	1739 ± 367	3 / 31
ⓠQASM QFT	48	4339	4 / 138
ⓠQASM QFT (const)	32	1372 ± 26	4 / 158
Quipper TOFF	63	8060	
Quipper TOFF (const)	41	2870 ± 594	

(b) Multiplication circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
ⓠQASM TOFF (const)	49	28768	16 / 397
ⓠQASM QFT (const)	34	15288	5 / 412
ⓠQASM AQFT (const)	34	5948	4 / 323
Quipper TOFF	98	37737	

(c) Division/modulo circuits (16 bits)

	# qubits	# gates	Verified
ⓠQASM TOFF (const)	41	56160	✓
ⓠQASM QFT (const)	19	18503	✓

(d) Modular multiplication circuits (8 bits)

Fig. 16. Comparison of ⓠQASM and Quipper arithmetic operations. In the "const" case, one argument is a classically-known constant parameter. For (a)-(b) we present the average (± standard deviation) over 20 randomly selected constants c with $0 < c < 2^{16}$. For the division/modulo circuits x mod n, we only consider the gate counts for the maximum iteration case when $n = 1$; the Quipper version assumes n is a variable, but they use the same algorithm as us by guessing a maximum iteration number for n. In (d), we use the constant 255 ($= 2^8 - 1$) for the modulus and set the other constant to 173 (which is invertible mod 255). Quipper supports no QFT-based circuits aside from an adder. "QC time" is the time (in seconds) for QuickChick to run 10,000 tests.

type	Verified	Randomly Tested
Nat /Bool	$[x{-}N]_q$ $[N{-}x]_q$ $[x{-}y]_{q,t}$ $[x{=}N]_{q,t}$ $[x{<}N]_{q,t}$ $[x{=}y]_{q,t}$ $[x{<}y]_{q,t}$	$[x{+}N]_a$ $[x{+}y]_a$ $[x{-}N]_t$ $[N{-}x]_t$ $[x{-}y]_q$ $[x\%N]_{a,q,t}$ $[x/N]_{a,q,t}$
FixedP	$[x{+}N]_q$ $[x{+}y]_t$ $[x{-}N]_q$ $[N{-}x]_q$ $[x{-}y]_t$ $[x{=}N]_{q,t}$ $[x{<}N]_{q,t}$ $[x{=}y]_{q,t}$ $[x{<}y]_{q,t}$	$[x{+}N]_t$ $[x{+}y]_q$ $[x{-}N]_t$ $[N{-}x]_t$ $[x{-}y]_q$ $[x{*}N]_{q,t}$ $[x{*}y]_{q,t}$ $[x/N]_{q,t}$

x, y = variables, N = constant,
$[\,]_{a,q,t}$ = AQFT-based (a), QFT-based (q), or Toffoli-based (t)

All testing is done with 16-bit/60-bit circuits.

Fig. 17. Other verified & tested operations

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 creating a function in their Haskell compiler to reverse the order of qubits. In ⓠQASM, we can use the Rev operation (which does not insert SWAPs) 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 16 compares the resources used by ©QASM operators with counterparts in Quipper. In both cases, we compiled the operators to OpenQASM 2.0 circuits,[5] 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 16 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.[6] 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-Rz gate increased the size of our QFT-based modular multiplication circuit by 4.4x. 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.

In addition, when we test different constant inputs for different arithmetic circuits, we find that the gate counts for Toffoli-based circuits tend to vibrate more than the QFT-based ones. This means that Toffoli-based circuits are more sensitive towards input constant changes. For example, with different constant inputs, the gate counts of the QFT-based Multiplication circuits are in the range

[5] We converted the output Quipper files to OpenQASM 2.0 using a compiler produced at Dalhousie University [Bian 2020].
[6] 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-Rz at https://qiskit.org/documentation/_modules/qiskit/circuit/library/standard_gates/u1.html; and the decomposition for controlled-controlled-Rz 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 these decompositions are ancilla free.

1372 ± 26, while the gate counts of our Toffoli-based circuits are in the range 1739 ± 367, and the Quipper ones are in the range 2870 ± 594.

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

Comparing to Quipper. Overall, Figure 16(a)-(c) shows that operator implementations in ⓠQASM 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 is that the Quipper-to-OpenQASM converter we use has a more expensive decomposition of controlled-*Rz* gates.[7] In the other cases (all Toffoli-based circuits), we made choices when implementing the oracles that improved their resource usage. There is nothing fundamental that stopped the Quipper developers from having made the same choices, but we note they did not have the benefit of the ⓠQASM 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., effectively classical) gates.

5.4 Approximate Operators

ⓠQASM'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. The results of replacing QFT with AQFT in the QFT adder from Figure 16(a) are shown in Figure 18(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 $\pm 2^b - 1$. 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.

Figure 19(a) shows one step of an N-bit QFT-based modulo circuit that computes $x \bmod n$ 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 19(a) computes $x - 2^{I-i}n$ and uses the result's most significant bit (MSB) to check whether $x < 2^{N-1-i}$. If the MSB

[7] The decomposition in Bian [2020] transforms a controlled-*Rz* gate into a circuit that uses two Toffoli gates, an *Rz* 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-*Rz* uses 3 single-qubit gates, 2 two-qubit gates, and no ancilla qubits.

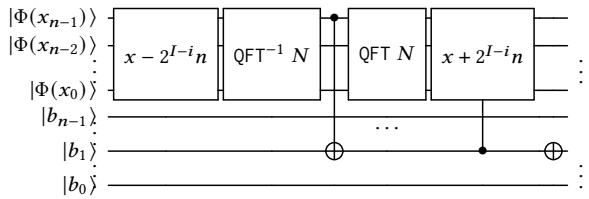
Precision	# gates	Error
16 bits (full)	1206	± 0
15 bits	1063	± 1
14 bits	929	± 3

(a) Varying the precision in a 16-bit adder

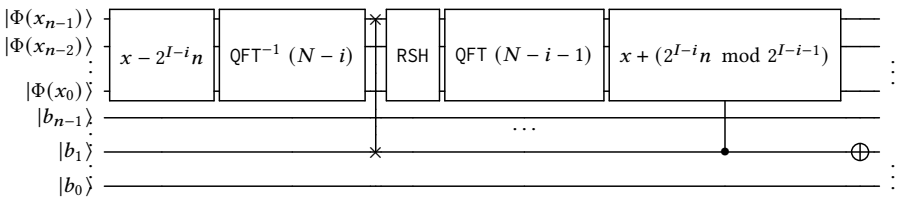
# iters. ($I+1$)	TOFF	QFT	AQFT	% savings
1	1798	1794	1717	4.5 / 4.5
4	7192	4432	3488	48.5 / 21.2
8	14384	8017	4994	65.2 / 37.7
12	21576	11637	5684	73.6 / 51.1
16	28768	15288	5948	79.3 / 61.1

(b) Gate counts for TOFF vs. QFT vs. AQFT division/modulo circuits, the left saving numbers in the savings column are comparing TOFF vs. AQFT, and the right one are QFT vs. AQFT

Fig. 18. Effects of approximation



(a) QFT-based



(b) AQFT-based (addition and subtraction are approximate)

Fig. 19. One step of the QFT/AQFT division/modulo circuit

is 0, then $x \geq 2^{N-1-i}$ and the circuit continues to next iteration; otherwise, it adds $2^{I-i}n$ to the result and continues.

We can improve the resource usage of the circuit in Figure 19(a) by replacing the addition, subtraction, and QFT components with approximate versions, as shown in Figure 19(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 19(b) begins by subtracting the top $(N-i)$ bits, and then converts x back to the `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 `Rshift` 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 $(2^{I-i}n \mod 2^{I-i-1})$, ignoring the original MSB.

The result is a division/modulo circuit that uses approximate components, but, as our testing assures, is exactly correct. The resources require are shown in Figure 18(b) for varying numbers of iterations. Compared to QFT-based circuit, for a single iteration, the approximation provides a 4.5% savings, and the saving increases with more iterations. For $n = 1$, we need 16 iterations. In this case, the AQFT-based division/modulo circuit uses 61.1% fewer gates than the QFT-based implementation. If we compare the AQFT-based division/modulo circuit with the Toffoli-based one,

	# qubits	# gates
OQIMP (x, y const)	16	16
OQIMP TOFF (x const)	33	1739 ± 376
OQIMP QFT (x const)	16	1372 ± 26
OQIMP TOFF	33	61470
OQIMP QFT	32	25609

(a) Fixed-precision circuits for $\frac{x*y}{M}$ with $M = 5$ (16 bits)

	# qubits
OQIMP TOFF	418
OQIMP QFT	384
Quipper	6142

(b) Sine circuits (64 bits)

Fig. 20. Effects of partial evaluation

the result is more significant. For 16 iterations, the AQFT-based division/modulo circuit uses 79.3% fewer gates than the Toffoli-based implementation.

6 EVALUATION: ℚIMP ORACLES AND PARTIAL EVALUATION

The prior section considered arithmetic operators implemented in ℚASM. They are building blocks for operators we have programmed using ℚIMP, which include sine, arcsine, cosine, arccosine, and exponentiation on fixed-precision numbers. We used ℚIMP's source semantics to test each operator's correctness. These operators are useful in near-term applications; e.g., the arcsine and sine functions are sub-components to repair the phase values in constructing Hamiltonian simulations [Feynman 1982] by the quantum walk algorithm [Childs 2009].

As discussed in Section 4.3, one of the key features of ℚIMP is *partial evaluation* during compilation to ℚASM. The simplest optimization similar to partial evaluation happens for a binary operation $x := x \odot y$, where y is a constant value. Figure 16 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 $\frac{x*y}{M}$ where M is constant and a 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 $\frac{8^n*x}{n!}$ in the Taylor series decomposition of sine). In Quipper, this is expression compiled to $r_1 = \frac{x}{M}; r_2 = r1 * y$. The ℚIMP compiler produces different outputs depending on whether x and y are constants. If they both are constant, ℚIMP simply assigns the result of computing $\frac{x*y}{M}$ to a quantum variable. If x is a constant, but y is not, ℚIMP evaluates $\frac{x}{M}$ classically, assigns the value to r_1, and evaluates r_2 using a constant multiplication circuit. If they are both quantum variables, ℚIMP generates a circuit to evaluate the division first and then the multiplication.

In Figure 20 (a) we show the size of the circuit generated for $\frac{x*y}{M}$ where zero, one, or both variables are classically known. It is clear that 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 20 (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. [8]

[8] ℚIMP also benefits from its representation of fixed-precision numbers (Section 4.3), 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.

7 CASE STUDY: GROVER'S SEARCH

Here we present a case study of integrating an oracle implemented with VQO into a full quantum algorithm, Grover's search algorithm, implemented and verified in 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 necessitate lengthening hash function outputs.

We have used 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 ℚIMP; the implementation is given in Figure 21.

To test our oracle implementation, we wrote our specification as a Coq function on bitstrings. We then defined correspondence between these bitstrings and program states in ℚASM semantics and conjectured that for any inputs, the semantics of our compiled oracle matches the corresponding outputs from our specification function. Using random testing (Section 4.2), 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 rather than producing the output directly. This oracle can be compiled to 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 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's hash collisions in a certain probability through 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.

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. For example, 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 ℚIMP

```
Q nat[4] qr(Q nat x1, Q nat x2, Q nat x3, Q nat x4) {
  x1 += x2; x4 ⊕= x1; x4 <<<= 16;
  x3 += x4; x2 ⊕= x3; x4 <<<= 12;
  x1 += x2; x2 ⊕= x1; x4 <<<= 8;
  x3 += x4; x2 ⊕= x3; x4 <<<= 7
  return [x1, x2, x3, x4];
}

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[0], x[5], x[10], x[15]] = qr(x[0], x[5], x[10], x[15]);
    [x[1], x[6], x[11], x[12]] = qr(x[1], 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]);
  }
}
```

Fig. 21. ChaCha20 implementation in ℚIMP

using `inv` to perform uncomputation, or to manually implement oracle functions in ℚASM, in both cases supporting formal verification and testing.

Verified Quantum Programming. Recent work on formally verifying quantum programs includes ℚWIRE [Rand 2018], SQIR [Hietala et al. 2021a], and ℚBRICKS [Chareton et al. 2020]. 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 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 (ℚIMP) and a more interesting quantum target language (ℚASM).

9 CONCLUSION

We present VQO, a framework for expressing, testing, and verifying quantum oracles. The key component of VQO is 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 ℚASM to SQIR and have verified (or randomly tested) many arithmetic circuits written in ℚASM. We also present ℚIMP, a high-level imperative language, and compiler from ℚIMP to 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 ⟦QIMP⟧, showing that partial evaluation results in our implementation of sine using just 7% of the qubits used in Quipper's implementation.

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.

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. arXiv:2105.07045 [quant-ph]

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. 2020. Toward Certified Quantum Programming. *arXiv e-prints* (2020). arXiv:2003.05841 [cs.PL]

Andrew M. Childs. 2009. On the Relationship Between Continuous- and Discrete-Time Quantum Walk. *Communications in Mathematical Physics* 294, 2 (Oct 2009), 581–603.

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). 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. arXiv:quant-ph/0008033 [quant-ph]

Thomas G. Draper. 2000. Addition on a Quantum Computer. *arXiv: Quantum Physics* (2000).

Richard P Feynman. 1982. Simulating physics with computers. *International journal of theoretical physics* 21, 6/7 (1982), 467–488.

Craig Gidney and Martin Ekerå. 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 2013).* 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 *Proceedings of the Conference on Interative Theorem Proving (ITP)*.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA.

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

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/ab8c2c

Thomas Loke. 2017. *Quantum circuit design for quantum walks*. Ph.D. Dissertation. the University of Western Australia.

Igor L. Markov and Mehdi Saeedi. 2012. Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation. *Quantum Info. 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 (Mar 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 2021)*. Association for Computing Machinery, New York, NY, USA, 222?236. https://doi.org/10.1145/3453483.3454040

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

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*.

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 programming 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