A Practical Quantum Instruction Set Architecture

Robert S. Smith, Michael J. Curtis, William J. Zeng
Rigetti Computing
775 Heinz Ave.
Berkeley, California 94710
Email: {robert, spike, will}@rigetti.com

Abstract*—We introduce an abstract machine architecture for classical/quantum computations—including compilation—along with a quantum instruction language called Quil for explicitly writing these computations. With this formalism, we discuss concrete implementations of the machine and non-trivial algorithms targeting them. The introduction of this machine dovetails with ongoing development of quantum computing technology, and makes possible portable descriptions of recent classical/quantum algorithms. *Keywords*—*quantum computing, software architecture

Contents

I. Introduction

The underlying hardware for quantum computing has advanced rapidly in recent years. Superconducting chips with 4–9 qubits have been demonstrated with the performance required to run quantum simulation algorithms [1, 2, 3], quantum machine learning [4], and quantum error correction benchmarks [5, 6, 7].

Hybrid classical/quantum algorithms—including variational quantum eigensolvers [8, 9, 10], correlated material simulations [11], and approximate optimization [12]—have much promise in reducing the overhead required

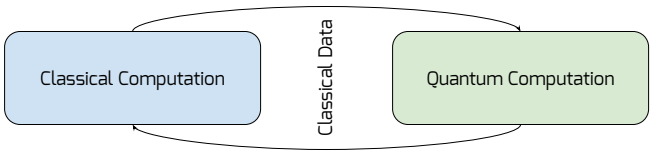

Fig. 1. A classical/quantum feedback loop.

for valuable applications. In machine learning and quantum simulation, particularly for catalysts [13] and high-temperature superconductivity [9], scalable quantum computers promise performance unrivaled by classical supercomputers.

The promise of hardware and applications must be matched with advances in programming architectures. The demands of practical algorithm design, as well as the shift to hybrid classical/quantum algorithms, necessitate an update to the *quantum Turing machine* model [14]. We need new frameworks for quantum program compilation [15, 16, 17, 18, 19] and emulation [20, 21]. For more details on prior work and its relationship to the topics introduced here, we refer the reader to Appendix B.

These classical/quantum algorithms require a classical computer and quantum computer to communicate and work cooperatively, as in Figure 1. Within the presented framework, classical information is fed back via a defined memory model, which can be implemented efficiently in both hardware and software.

In this paper, we describe an abstract machine which serves as a model for hybrid classical/quantum computation. Alongside this, we describe an instruction language for this machine called Quil and its suitability for program analysis and compilation. Together these form a **quantum instruction set architecture** (ISA). Lastly, we give various examples of algorithms in Quil, and discuss an executable implementation.

II. The Quantum Abstract Machine

Turing machines provide a vehicle for studying important concepts in computer science such as complexity and computational equivalence. While theoretically important, they do not provide a foundation for the construction of practical computing machines. Instead, specialized abstract machines are designed to accomplish real-world tasks, like arithmetic, efficiently while maintaining Turing completeness. These machines are often specified in the form of an instruction set architecture. Quantum Turing machines lie in the same vein as its classical counterpart, and we follow a similar approach in the creation of a practical quantum analog.

The **Quantum Abstract Machine** (QAM) is an abstract representation of a general-purpose quantum computing device. It includes support for manipulating both classical and quantum state. The QAM executes programs represented in a quantum instruction language called **Quil**, which has well-defined semantics in the context of the QAM.

The state of the QAM is specified by the following elements:

- A fixed but arbitrary number of qubits N_q indexed from 0 to $N_q - 1$. The k^{th} qubit is written Q_k. The state of these qubits is written $|\Psi\rangle$ and is initialized to $|00\ldots0\rangle$. The semantics of the qubits are described in Section II-A.
- A classical memory C of N_c bits, initialized to zero and indexed from 0 to $N_c - 1$. The k^{th} bit is written $C[k]$.
- A fixed but arbitrary list of static gates G, and a fixed but arbitrary list of parametric gates G'. These terms are defined in Section III-C.
- A fixed but arbitrary sequence of Quil instructions P. These instructions will be described in Section III.
- An integer program counter $0 \leq \kappa \leq |P|$ indicating position of the next instruction to execute when $\kappa \neq |P|$ or a halted program when $\kappa = |P|$.

The 6-tuple $(|\Psi\rangle, C, G, G', P, \kappa)$ summarizes the state of the QAM.

The QAM may be implemented either classically or on quantum hardware. A classical implementation is called a **Quantum Virtual Machine** (QVM). We describe one such implementation in Section V-F. An implementation on quantum hardware is called a **Quantum Processing Unit** (QPU).

The semantics of the quantum state and operations on it are described in the language of tensor products of Hilbert spaces and linear maps between them. The following subsections give these semantics in meticulous detail. Readers with intuition about these topics are encouraged to skip to Section III for a description of Quil.

A. Qubit Semantics

A finite-dimensional Hilbert space over the complex numbers $\mathbb{C}$ is denoted by $\mathscr{H}$. The state space of a qubit is a two-dimensional Hilbert space over $\mathbb{C}$ and is denoted by $\mathscr{B}$. Each of these Hilbert spaces is equipped with a chosen orthonormal basis and indexing map on that basis. An **indexing map** is a bijective function that maps elements of a finite set Σ to the set of non-negative integers below $|\Sigma|$, denoted $[|\Sigma|]$. For a Hilbert space spanned by $\{|u\rangle, |v\rangle\}$ with an indexing map defined by $|u\rangle \mapsto 0$ and $|v\rangle \mapsto 1$, we write $|0\rangle \coloneqq |u\rangle$ and $|1\rangle \coloneqq |v\rangle$.

In the context of the QAM, each qubit Q_k in isolation has a state space $\mathscr{B}_k$ spanned by an orthonormal basis $\{|0\rangle_k, |1\rangle_k\}$ called the **computational basis**. Since qubits can entangle, the state space of the system of all qubits is not a direct product of each constituent space, but rather a rightward tensor product

$$|\Psi\rangle \in \mathscr{H} \coloneqq \bigotimes_{k=0}^{N_q-1} \mathscr{B}_{N_q-k-1}. \tag{1}$$

The meaning of the tensor product is as follows. Let $|p\rangle_i$ be the p^{th} basis vector of $\mathscr{H}_i$ according to its indexing map.

The tensor product of two Hilbert spaces is then

$$\mathscr{H}_i \otimes \mathscr{H}_j \coloneqq \Big\{ \sum_{\substack{p\in[\dim \mathscr{H}_i]\\ q\in[\dim \mathscr{H}_j]}} \mathsf{C}_{p,q} \underbrace{|p\rangle_i \otimes |q\rangle_j}_{\text{basis element}} : \mathsf{C} \in \mathbb{C}^{\dim \mathscr{H}_i \times \dim \mathscr{H}_j} \Big\}. \tag{2}$$

The resulting basis elements are ordered by way of the **lexicographic indexing map**

$$|p\rangle_i \otimes |q\rangle_j \mapsto q + p \dim \mathscr{H}_j. \tag{3}$$

Having the basis elements of the Hilbert space $\mathscr{H}_i$ "dominate" the indexing map is a convention due to the standard definition of the Kronecker product, in which for matrices A and B, $A \otimes B$ is a block matrix $(A \otimes B)_{i,j} = A_{i,j}B$. This convention, while standard, somewhat muddles the semantics below with busy-looking variable indexes which count down, not up.

A basis element of $\mathscr{H}$

$$|b_{N_q-1}\rangle_{N_q-1} \otimes \cdots \otimes |b_1\rangle_1 \otimes |b_0\rangle_0$$

can be written shorthand in **bit string notation**

$$|b_{N_q-1} \dots b_1 b_0\rangle .$$

This has the particularly useful property that the bit string corresponds to the binary representation of the index of that basis element. For clarity, however, we will not use this notation elsewhere in this paper.

Example 1. *A two-qubit system in the Hilbert space $\mathscr{B}_2 \otimes \mathscr{B}_1$ has the lexicographic indexing map defined by*

$$|0\rangle_2 \otimes |0\rangle_1 \mapsto 0, \qquad |0\rangle_2 \otimes |1\rangle_1 \mapsto 1,$$
$$|1\rangle_2 \otimes |0\rangle_1 \mapsto 2, \qquad |1\rangle_2 \otimes |1\rangle_1 \mapsto 3.$$

The standard Bell state in this system is represented by the element in the tensor space with the matrix

$$\mathsf{C} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Eqs. (1) and (2) imply that $\dim \mathscr{H} = 2^{N_q}$, and as such, $|\Psi\rangle$ can be represented as a complex vector of that length.

B. Quantum Gate Semantics

The quantum state of the system evolves by applying a sequence of operators called **quantum gates**. Most generally, these are complex unitary matrices of size $2^{N_q} \times 2^{N_q}$, written succinctly as $\mathrm{U}(2^{N_q})$. However, quantum gates are typically abbreviated as one- or two-qubit unitary matrices, and go through a process of "tensoring up" before application to the quantum state.

Example 2. *The **Hadamard gate** on Q_k is defined as*

$$\mathsf{H} \coloneqq \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} : \mathscr{B}_k \to \mathscr{B}_k.$$

This is unitary because $\mathsf{H}\mathsf{H}^\dagger = \mathsf{I}_k$ is the identity map on $\mathscr{B}_k$.

*The **controlled-**X or **controlled-not gate** with* control qubit Q_j *and* target qubit Q_k *is defined as*

$$\mathsf{CNOT} \coloneqq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} : \mathscr{B}_j \otimes \mathscr{B}_k \to \mathscr{B}_j \otimes \mathscr{B}_k.$$

This gate is common for constructing entanglement *in a quantum system.*

It turns out that many different sets of these one- and two-qubit gates are sufficient for universal quantum computation, i.e., a discrete set of gates can be used to approximate any unitary matrix to arbitrary accuracy [22, 23]. In fact almost any two-qubit quantum gate can be shown to be universal [24]. Delving into different universal gate sets is beyond the scope of this work and we refer the reader to [23] as a general reference.

We now wish to provide a constructive method for interpreting operators on a portion of a Hilbert space as operators on the Hilbert space itself. In the simplest case, we have a one-qubit gate U acting on qubit Q_k, which induces an operator $\tilde{U}$ on $\mathscr{H}$ by tensor-multiplying with the identity map a total of $N_q - 1$ times:

$$\tilde{U} = \mathsf{I}_{N_q-1} \otimes \cdots \otimes \underbrace{U}_{(N_q-k-1)^{\text{th}} \text{ position (zero-based)}} \otimes \cdots \otimes \mathsf{I}_1 \otimes \mathsf{I}_0. \tag{4}$$

We refer to this process as **lifting** and reserve the tilde over the operator name to indicate such.

Example 3. *Consider a system of four qubits and consider a Hadamard gate acting on Q_2. Lifting this operator according to* (4) *gives*

$$\tilde{\mathsf{H}} = \mathsf{I}_3 \otimes \mathsf{H} \otimes \mathsf{I}_1 \otimes \mathsf{I}_0.$$

A two-qubit gate acting on adjacent Hilbert spaces is just as simple. An operator $U : \mathscr{B}_k \otimes \mathscr{B}_{k-1} \to \mathscr{B}_k \otimes \mathscr{B}_{k-1}$ is lifted as

$$\tilde{U} = \underbrace{\mathsf{I}_{N_q-1} \otimes \cdots \otimes U \otimes \cdots \otimes \mathsf{I}_1 \otimes \mathsf{I}_0}_{N_q-1 \text{ factors, } U \text{ at position } N_q-k-1} . \tag{5}$$

However, when the Hilbert spaces are not adjacent, lifting involves a bit more bookkeeping because there is no obvious place to tensor-multiply the identity maps of the Hilbert spaces indexed between j and k. We can resolve this by suitably rearranging the space. We need two tools: a method to "reorganize" an operator's action on a tensor product of Hilbert spaces and isomorphisms between these Hilbert spaces. The general principle to be employed is to find some operator $\pi : \mathscr{H} \to \mathscr{H}$ which acts as a permutation operator on $\mathscr{H}$, and to compute $\pi^{-1}U'\pi$, where U' is isomorphic to U. Simply speaking, π is a temporary re-indexing of basis vectors and U' is just a trivial reinterpretation of U.

To reorganize, we use the fact that any permutation can be decomposed into adjacent transpositions. For swapping

two qubits, consider the gate

$$\mathsf{SWAP}_{j,k} \coloneqq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} : \mathscr{B}_j \otimes \mathscr{B}_k \to \mathscr{B}_j \otimes \mathscr{B}_k. \quad (6)$$

This is a permutation matrix which maps the basis elements according to

$$|0\rangle_j \otimes |1\rangle_k \mapsto |1\rangle_j \otimes |0\rangle_k$$
$$|1\rangle_j \otimes |0\rangle_k \mapsto |0\rangle_j \otimes |1\rangle_k ,$$

and mapping the others identically. For adjacent transpositions in the full Hilbert space, this can be trivially lifted:

$$\tau_i \coloneqq \mathsf{SWAP}_{i,i+1} \text{ lifted by way of (5).} \quad (7)$$

We will use a sequence of these operators to arrange for $\mathscr{B}_j$ and $\mathscr{B}_k$ to be adjacent by moving one of them next to the other. There are two cases we need to be concerned about: $j > k$ and $j < k$.

For the $j > k$ case, we want to map the state of $\mathscr{B}_k$ to $\mathscr{B}_{j-1}$. This is accomplished with the operator[1]

$$\pi_{j,k} \coloneqq \prod_{i=k}^{j-2} \tau_{j+k-i-2},$$

where the product right-multiplies and is empty when $k \geq j-1$.

For the $j < k$ case, we want to map the state of $\mathscr{B}_j$ to $\mathscr{B}_{k-1}$, and then swap $\mathscr{B}_{k-1}$ with $\mathscr{B}_k$. We can do this with the π operator succinctly:

$$\pi'_{j,k} \coloneqq \tau_{k-1}\pi_{k,j}.$$

Note the order of j and k in the π factor.

Lastly, for the purpose of correctness, we need to construct a **trivial isomorphism**

$$f : \mathscr{B}_j \otimes \mathscr{B}_k \to \mathscr{B}_j \otimes \mathscr{B}_{j-1}, \quad (8)$$

which is defined as the bijection between basis vectors with the same index.

Now we may consider two-qubit gates in their full generality. Let $U : \mathscr{B}_j \otimes \mathscr{B}_k \to \mathscr{B}_j \otimes \mathscr{B}_k$ be the gate under consideration. Perform a "change of coordinates" on U to define

$$V \coloneqq fUf^{-1} : \mathscr{B}_j \otimes \mathscr{B}_{j-1} \to \mathscr{B}_j \otimes \mathscr{B}_{j-1},$$

and lift V to $\tilde{V} : \mathscr{H} \to \mathscr{H}$ by way of (5). Then the lifted U can be constructed as follows:

$$\tilde{U} = \begin{cases} \pi_{j,k}^{-1}\tilde{V}\pi_{j,k} & \text{if } j > k, \text{ and} \\ (\pi'_{j,k})^{-1}\tilde{V}\pi'_{j,k} & \text{if } j < k. \end{cases} \quad (9)$$

Since the π operators are essentially compositions of involutive SWAP gates, their inverses are just reversals.

With care, the essence of this method generalizes accordingly for gates acting on an arbitrary number of qubits.

[1]This is an example of the effects of following the Kronecker convention. This product is really just "τ_i in reverse."

We end this section with an example of a universal QAM.

Example 4. *Define all possible liftings of U within $\mathscr{H}$ as*

$$L(U) \coloneqq \{U \textit{ lifted for all qubit permutations}\}.$$

Define the gates $\mathsf{S} \coloneqq \left(\begin{smallmatrix} 1 & 0 \\ 0 & i \end{smallmatrix}\right)$ *and* $\mathsf{T} \coloneqq \left(\begin{smallmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{smallmatrix}\right)$. *A QAM with*[2]

$$G = L(\mathsf{H}) \cup L(\mathsf{S}) \cup L(\mathsf{T}) \cup L(\mathsf{CNOT}) \textit{ and } G' = \{\}$$

can compute to arbitrary accuracy the action of any N_q-qubit gate, possibly with P exponential in length. See [23, §4.5] for details.

C. Measurement Semantics

Measurement is a surjective-only operation and is nondeterministic. In the space of a single qubit Q_k, there are two outcomes to a measurement. The outcomes are determined by lifting and applying—up to a scalar factor—either of the **measurement operators**[3]

$$M_0^k \coloneqq |0\rangle_k \langle 0|_k \qquad \text{and} \qquad M_1^k \coloneqq |1\rangle_k \langle 1|_k$$

to the quantum state. These can be interpreted as projections onto either of the basis elements of the Hilbert space. More generally, in any finite Hilbert space $\mathscr{H}$, we have the set of measurement operators

$$M(\mathscr{H}) \coloneqq \left\{ |v\rangle \langle v| : |v\rangle \in \text{basis}\, \mathscr{H} \right\}.$$

In the QAM, when a qubit is measured, a particular measurement operator μ is selected and applied according to the probability

$$P(\mu) \coloneqq P(\tilde{\mu} \text{ is applied during meas.}) = \langle\Psi| \tilde{\mu}^\dagger \tilde{\mu} |\Psi\rangle . \quad (10)$$

Upon selection of an operator, the quantum state transforms according to

$$|\Psi\rangle \leftarrow \frac{1}{P(\mu)} \tilde{\mu} |\Psi\rangle . \quad (11)$$

This irreversible operation is called **collapse of the wavefunction**.

In quantum mechanics, measurement is much more general than the description given above. In fact, any Hermitian operator can correspond to measurement. Such operators are called **observables**, and the quantum state of a system can be seen as a superposition of vectors in the observable's eigenbasis. The eigenvalues of the observable are the outcomes of the measurement, and the corresponding eigenvectors are what the quantum state collapses to. For additional details on measurement and its quantum mechanical interpretation, we refer the interested reader to [23] and [25].

[2]When the context is clear, G is sometimes abbreviated to just be the set of unlifted gates, but it should be understood that it's actually every lifted combination. See Section III-C.

[3]In Dirac notation, $\langle v|$ lies in the dual space of $|v\rangle$.

III. Quil: a Quantum Instruction Language

Quil is an instruction-based language for representing quantum computations with classical feedback and control. In its textual format—as presented below—it is line-based and assembly-like. It can be written directly for the purpose of quantum programming, used as an intermediate format within classical programs, or used as a compilation target for quantum programming languages. Most importantly, however, Quil defines what can be done with the QAM. Quil has support for:

- Applying arbitrary quantum gates,
- Defining quantum gates as optionally parameterized complex matrices,
- Defining quantum circuits as sequences of other gates and circuits, which can be bit- and qubit-parameterized,
- Expanding quantum circuits,
- Measuring qubits and recording the measurements into classical memory,
- Synchronizing execution of classical and quantum algorithms,
- Branching unconditionally or conditionally on the value of bits in classical memory, and
- Including files as modular Quil libraries such as the standard gate library (see Appendix A).

By virtue of being instruction-based, Quil code effects transitions of the QAM as a state machine. In the next subsections, we will describe the various elements of Quil, using the syntax and conventions outlined in Section II.

A. Classical Addresses and Qubits

The central atomic objects on which various operations act are qubits, classical bits designated by an address, and classical memory segments.

Qubit A qubit is referred to by its integer index. For example, Q_5 is referred to by `5`.

Classical memory address A **classical memory address** is referred to by an integer index in square brackets. For example, the address 7 pointing to the bit $C[7]$ is referred to as `[7]`.

Classical memory segment A **classical memory segment** is a contiguous range of addresses from a to b inclusive with $a \leq b$. These are written in square brackets as well, with a hyphen separating the range's endpoints. For example, the bits between 0 and 63 are addressed by `[0-63]` and represent the concatenation of bits

$$C[63]C[62]\dots C[1]C[0],$$

written in the usual MSB-to-LSB order.

B. Numerical Interpretation of Classical Memory Segments

Classical memory segments can be interpreted as a numerical value for the purpose of controlling parametric gates. In particular, a 64-bit classical memory segment refers to an IEEE-754 double-precision floating point number [26]. A 128-bit classical memory segment `[`x`-(`$x+127$`)]` refers to a double-precision complex number $a + ib$ where a is the 64-bit interpretation of `[`x`-(`$x + 63$`)]`, the first half of the segment, and b is the 64-bit interpretation of `[(`$x+64$`)-(`$x+127$`)]`, the second half of the segment. The use of these numbers can be found in Section III-C and some practical consequences of their use can be found in Section IV-B2.

C. Static and Parametric Gates

There are two gate-related concepts in the QAM: static and parametric gates. A **static gate** is an operator in $\mathrm{U}(2^{N_q})$, and a **parametric gate** is a function[4] $\mathbb{C}^n \to \mathrm{U}(2^{N_q})$, where the n complex numbers are called *parameters*. The implication is that *operators in G and G' are always lifted to the Hilbert space of the QAM*. This is a formalism, however, and Quil abstracts away the necessity to be mindful of lifting gates.

In Quil, every gate is defined separately from its invocation. Each unlifted gate is identified by a symbolic name[5], and is invoked with a fixed number of qubit arguments. The invocation of a static (resp. parametric) gate whose lifting is not a part of the QAM's G (resp. G') is undefined.

A static two-qubit gate named `NAME` acting on Q_2 and Q_5, which is an operator lifted from $\mathscr{B}_2 \otimes \mathscr{B}_5$, is written in Quil as the name of the gate followed by the qubit indexes it acts on, as in

```
NAME 2 5
```

Example 5. *The Bell state on qubits Q_0 and Q_1 can be constructed with the following Quil code:*

```
H 0
CNOT 0 1
```

A parametric three-qubit gate named `PNAME` with a single parameter $e^{-i\pi/7}$ acting on Q_1, Q_0, and Q_4, which is an operator lifted from $\mathscr{B}_1 \otimes \mathscr{B}_0 \otimes \mathscr{B}_4$, is written in Quil as

```
PNAME(0.9009688679-0.4338837391i) 1 0 4
```

When a parametric gate is provided with a constant parameter, one could either consider the *resulting* gate on the specified qubits to be a part of G, or the parametric gate itself on said qubits to be a part of G'.

Parametric gates can take a "dynamic parameter", as specified by a classical memory segment. Suppose a parameter is stored in `[8-71]`. Then we can invoke the aforementioned gate with that parameter via

```
PNAME([8-71]) 1 0 4
```

In some cases, using dynamic parameters can be expensive or infeasible, as discussed in Section IV-B2. Gates which use dynamic parameters are elements of G'.

[4] Calling a parametric gate a "gate" is somewhat of a misnomer. The quantum gate is actually the *image* of a point in $\mathbb{C}^n$.

[5] To be precise, the symbolic name actually represents the equivalence class of operators under all trivial isomorphisms, as in (8).

D. Gate Definitions

Static gates are defined by their real or complex matrix entries in the basis described in Section II-A. Matrix entries can be written literally with scientific `E`-notation (e.g., real `-1.2e2` or complex `0.3-4.1e-4i` $= 0.3 - 0.00041i$), or built up from constant arithmetic expressions. These are:

- Simple arithmetic: addition `+`, subtraction/negation `-`, multiplication `*`, division `/`, exponentiation `^`,
- Constants: `pi` ($= pi$), `i` ($=$ `1.0i`), and
- Functions: `sin`, `cos`, `sqrt`, `exp`, `cis`[6].

The gate is declared using the `DEFGATE` directive followed by comma-separated lists of matrix entries indented by exactly four spaces. Matrices that are not unitary (up to noise or precision) have undefined[7] execution semantics.

Example 6. *The Hadamard gate can be defined by*

```
DEFGATE HADAMARD:
    1/sqrt(2), 1/sqrt(2)
    1/sqrt(2), -1/sqrt(2)
```

This gate is included in the collection of standard gates, but under the name `H`.

Parametric gates are the same, except for the allowance of *formal parameters*, which are names prepended with a '`%`' symbol. Comma-separated formal parameters are listed in parentheses following the gate name, as is usual.

Example 7. *The rotation gate* R_x *can be defined by*

```
DEFGATE RX(%theta):
    cos(%theta/2), -i*sin(%theta/2)
    -i*sin(%theta/2), cos(%theta/2)
```

This gate is also included in the collection of standard gates.

Defining several gates or circuits with the same name is undefined.

E. Circuits

Sometimes it is convenient to name and parameterize a particular sequence of Quil instructions for use as a subroutine to other quantum programs. This can be done with the `DEFCIRCUIT` directive. Similar to the `DEFGATE` directive, the body of a circuit definition must be indented exactly four spaces. Critically, it specifies a list of *formal arguments* which can be substituted with either classical addresses or qubits.

Example 8. *In example 5, we constructed a Bell state on* Q_0 *and* Q_1*. We can generalize this for arbitrary qubits* Q_m *and* Q_n *by defining a circuit.*

```
DEFCIRCUIT BELL Qm Qn:
    H Qm
    CNOT Qm Qn
```

With this, Example 5 is replicated by just a single line:

```
BELL 0 1
```

Similar to parametric gates, `DEFCIRCUIT` can optionally specify a list of parameters, specified as a comma-separated list in parentheses following the circuit name, as the following example shows.

Example 9. *Using the x-y-z convention, an extrinsic Euler rotation by* (α, β, γ) *of the state of qubit* q *on the Bloch sphere is codified by the following circuit:*

```
DEFCIRCUIT EULER(%alpha, %beta, %gamma) q:
    RX(%alpha) q
    RY(%beta)  q
    RZ(%gamma) q
```

Within circuits, labels are renamed uniquely per expansion. As a consequence, it is possible to expand the same circuit multiple times, but it is not possible to jump into a circuit.

Circuits are intended to be used more as macros than as specifications for general quantum circuits. Indeed, `DEFCIRCUIT` is very limited in its expressiveness, only performing argument and parameter substitution. It is included mainly to help with the debugging and human readability of Quil code. More advanced constructs are intended to be written on top of Quil, as in Section V-B.

F. Measurement

Measurement provides the "side effects" of quantum programming, and is an essential part of most practical quantum algorithms (e.g., phase estimation and teleportation). Quil provides two forms of measurement: measurement-for-effect, and measurement-for-record.

Measurement-for-effect is a measurement performed on a single qubit used solely for changing the state of the quantum system. This is done with a `MEASURE` instruction of a single argument. Performing a measurement on Q_5 is written as

```
MEASURE 5
```

More useful, however, is measurement-for-record. Measurement-for-record is a measurement performed and recorded in classical memory. This form of the `MEASURE` instruction takes two arguments, the qubit and the classical memory address. To measure Q_7 and deposit the result at address 8 is written

```
MEASURE 7 [8]
```

The semantics of the measurement operation are described in Section II-C.

Example 10. *Producing a random number between* 0 *and* 3 *inclusive can be accomplished with the following program:*

```
H 0
H 1
```

[6] $\operatorname{cis}\theta := \cos\theta + i\sin\theta = \exp i\theta$

[7] Software processing Quil is encouraged to warn or error on such matrices.

```
MEASURE 0 [0]
MEASURE 1 [1]
```

The memory segment `[0-1]` *is now representative of the number in binary.*

G. Program Control

Program control is determined by the state of the program counter. The program counter κ determines if the program has halted, and if not, it determines the location of the next instruction to be executed. Every instruction, except for the following, has the effect of incrementing κ. The exceptions are:

- Conditional and unconditional jumps.
- The halt instruction `HALT` which terminates execution and assigns $\kappa \leftarrow |P|$.
- The last instruction in the program, which—after its execution—implicitly terminates execution as if by `HALT`.

Locations within the instruction sequence are denoted by **labels**, which are names that are prepended with an '`@`' symbol, like `@start`. The declaration of a new label within the instruction sequence is called a **jump target**, and is written with the `LABEL` directive.

Unconditional jumps are executed by the `JUMP` instruction which sets κ to the index of a given jump target.

Conditional jumps are executed by the `JUMP-WHEN` (resp. `JUMP-UNLESS`) instruction, which set κ to the index of a given jump target if the bit at a classical memory address is 1 (resp. 0), and to $\kappa + 1$ otherwise. This is a critical and differentiating element of Quil; it allows fast classical feedback.

Example 11. *Consider the following C-like pseudocode of an* `if`*-statement branching on the bit contained at address* `x`*:*

```
if (*x) {
    // instrA...
} else {
    // instrB...
}
```

This can be translated into Quil in the following way:

```
JUMP-WHEN @THEN [x]
# instrB...
JUMP @END
LABEL @THEN
# instrA...
LABEL @END
```

Lines starting with the `#` *character are comments and are ignored.*

Labels that are declared within the body of a `DEFCIRCUIT` are unique to that circuit. While it is possible to jump out of a `DEFCIRCUIT` to a globally declared label, it is not possible to jump inside of one.

Example 12. *Consider the following two* `DEFCIRCUIT` *declarations and their instantiations. Note the comments on correct and incorrect usages of* `JUMP`*.*

```
DEFCIRCUIT FOO:
    LABEL @FOO_A
    JUMP @GLOBAL   # valid, global label
    JUMP @FOO_A    # valid, local to FOO
    JUMP @BAR_A    # invalid

DEFCIRCUIT BAR:
    LABEL @BAR_A
    JUMP @FOO_A    # invalid

LABEL @GLOBAL
FOO
BAR
JUMP @FOO_A        # invalid
JUMP @BAR_A        # invalid
```

H. Zeroing the Quantum State

The quantum state of the QAM can be reset to the zero state with the `RESET` instruction. This has the effect of setting

$$|\Psi\rangle \leftarrow \bigotimes_{k=0}^{N_q-1} |0\rangle_{N_q-k-1}\,.$$

There are no provisions to clear the state of a single qubit, but we can do this by taking advantage of projective measurement.

Example 13. *We can clear a qubit using a single bit of classical scratch space.*

```
DEFCIRCUIT CLEAR q scratch_bit:
    MEASURE q scratch_bit
    JUMP-UNLESS @end scratch_bit
    X q
    LABEL @end
```

I. Classical/Quantum Synchronization

Some classical/quantum programs can be constructed in a way such that at a certain point of a quantum program, execution must be suspended until some classical computations are performed and corresponding classical state is modified. This is accomplished using the `WAIT` instruction, a synchronization primitive which signals to the classical computer that computation will not continue until some condition is satisfied. `WAIT` takes no arguments.

The mechanism by which `WAIT` works is deliberately unspecified. Some example mechanisms include monitors and interrupts, depending on the QAM implementation. An example use of `WAIT` can be found in Section IV-B2.

J. Classical Instructions

Quil is intended to be a language to manipulate quantum state using quantum operations and classical control. Classical computation on the classical state should be

done as much as possible with a classical computer, and using Quil's classical/quantum synchronization to mediate the hand-off of classical data between the classical and quantum processors. However, a few instructions for manipulating the classical state are provided for convenience, with emphasis on making control flow easier.

1) Classical Unary Instructions: The classical unary instructions are instructions that take a single classical address as an argument and modify the bit at that address accordingly. In each of the following, let a be the address provided as the sole argument. The three instructions are:

Constant False `FALSE`, which has the effect $C[a] \leftarrow 0$;
Constant True `TRUE`, which has the effect $C[a] \leftarrow 1$; and
Negation `NOT`, which has the effect $C[a] \leftarrow 1 - C[a]$.

2) Classical Binary Instructions: The classical binary instructions are instructions that take two classical addresses as arguments, and modify the bits at those addresses accordingly. In all of the following, let a be the first address and b be the second address provided as arguments. The four instructions are:

Conjunction `AND`, which has the effect $C[b] \leftarrow C[a]C[b]$;
Disjunction `OR`, which has the effect

$$C[b] \leftarrow 1 - (1 - C[a])(1 - C[b]);$$

Copy `MOVE`, which has the effect $C[b] \leftarrow C[a]$; and
Exchange `EXCHANGE`, which has the effect of swapping the bits at a and b: $C[a] \leftrightarrow C[b]$.

Example 14. *Exclusive disjunction $r \leftarrow a + b \mod 2$ can be implemented with the following circuit:*

```
DEFCIRCUIT XOR a b r:
    # Uses (a | b) & (~a | ~b)
    MOVE b r
    OR a r              # r = a | b
    JUMP-UNLESS @end r  # short-circuit
    MOVE b r
    NOT a
    NOT r
    OR a r              # r = ~a | ~b
    NOT a               # undo change to a
    LABEL @end
```

Note that `r` *has to be distinct from* `a` *and* `b`.

K. The No-Operation Instruction

The **no-operation**, **no-op**, or `NOP` instruction does not affect the state of the QAM except in the way described in Section III-G, i.e., by incrementing the program counter. This instruction may appear useless, especially on a machine with no notion of alignment or dynamic jumps. However, it has purpose when the QAM is used as the foundation for hardware emulation. For example, consider a QAM with some associated gate noise model. If one were to use an identity gate in place of a no-op, then the identity gate would be interpreted as noisy while the no-op would not. Moreover, the no-op has no qubit dependencies, which would otherwise affect program analysis. Rigetti Computing has used the no-op instruction as a way to force a break in instruction parallelization, described in Section V-E.

L. File Inclusion Semantics

File inclusion is done via the `INCLUDE` directive. For example, the library of standard gates—described in Appendix A—can be included with

```
INCLUDE "stdgates.quil"
```

File inclusion is *not* simple token substitution as it is in languages like the C preprocessor. Instead, the included file is parsed into a set of circuit definitions, gate definitions, and instruction code. Instruction code is substituted verbatim, but definitions will be recorded as if they were originally placed at the top of the file.

Generally, best practice is to include files containing *only* contain gate or circuit definitions (in which case the file is called a *library*), or *only* executable code, and not both. However, this is not enforced.

M. Pragma Support

Programs that process Quil code may want to take advantage of extra information provided by the programmer. This is especially true when targeting QPUs where additional information about the machine's characteristics affect how the program will be processed. Quil supports a `PRAGMA` directive to include extra information in a program which does not otherwise affect execution semantics. The syntax of `PRAGMA` is as follows:

```
PRAGMA <identifier>+ <string>?
```

where `+` indicates one or more instances and `?` indicates zero or one instance.

Example 15. *The QAM does not have any notion of instruction parallelism, but programs are generally parallelized before execution on a QPU. (See Section V-E.) Programs processing Quil may wish to enforce boundaries across which parallelization may not occur. An implementation may opt to support a parallelization barrier pragma. Despite the fact that the* `X`*-gates below are commuting, an implementation may opt to treat the instructions sequentially.*

```
X 0
PRAGMA parallelization_barrier
X 1
```

Note that this does not change the semantics of Quil vis-à-vis the QAM.

Example 16. *On modern superconducting qubit architectures, applications of different gates take different times. This can affect how instructions get scheduled for execution. Programs processing Quil may wish to allow the physical time it takes to perform a gate to be defined with a pragma, like so:*

```
PRAGMA gate_time H "50 ns"
```

```
PRAGMA gate_time CNOT "150 ns"
H 0
CNOT 0 1
```

N. The Standard Gates

Quil provides a collection of standard one-, two-, and three-qubit gates, which are fully defined in Appendix A. The standard gates can be used by including `stdgates.quil`.

IV. Quil Examples

A. Quantum Fourier Transform

In the context of the QAM's quantum state $|\Psi\rangle \in \mathscr{H}$, the **quantum Fourier transform** (QFT) [23, §5.1] is a unitary operator $\mathsf{F} : \mathscr{H} \to \mathscr{H}$ defined by the matrix

$$\mathsf{F}_{j,k} \coloneqq \frac{1}{\sqrt{2^{N_q}}}\omega^{jk}, \qquad 0 \le j,k < \dim \mathscr{H} = 2^{N_q} \tag{12}$$

with $\omega \coloneqq \exp(2\pi i/2^{N_q})$ the complex primitive root of unity. It can be shown that this operator acts on the basis vectors (up to permutation) via the map

$$\bigotimes_{\substack{k=0 \\ k' \coloneqq N_q-k-1}}^{N_q-1} |b_{k'}\rangle_{k'} \mapsto \frac{1}{\sqrt{2^{N_q}}} \bigotimes_{k=0}^{N_q-1} \left[|0\rangle_k + \phi(k)\,|1\rangle_k\right], \tag{13}$$

$$\text{where } \phi(k) = \prod_{j=0}^{k} \exp\left(2\pi i b_{N_q-j-1}/2^{j+1}\right).$$

Here, b is the bit string representation of the basis vectors, as in Section II-A.

The first thing to notice is that the basis elements get reversed in this factorization. This is easily fixed via a product of SWAP gates. In the context of classical fast Fourier transforms [27], this is called **bit reversal**.

The second and more important thing to notice is that each factor of the $\bigotimes$ can be seen as an operation on the qubit Q_k. The factors of $\phi(k)$ indicate the operations are two-qubit *controlled-phase* or CPHASE gates with Q_k as the target, and each previous qubit as the control. In the degenerate one-qubit case, this is a Hadamard gate. Further details on this algorithm can be found in [23].

The core QFT logic can be implemented as a straightforward recursive function `QFT'`. We write it as one which transforms qubits Q_k for $l \le k < r$. The base case is the action on a single qubit—a Hadamard. In the general case, we do a sequence of CPHASE gates targeted on the current qubit Q_l and controlled by all qubits before it, topped off by a Hadamard. In the Python-like pseudocode below, we generate Quil code for this algorithm. We prepend lines of Quil code to be generated with two colons '`::`'.

```
def QFT'(l, r):
    n = r - l          # num qubits
    if n == 1:         # base case
        :: H l
    else:              # general case
        QFT'(l + 1, r)
        for i in range(1, n):
            q = l + n - i
            alpha = pi / 2 ** (n - i)
            :: CPHASE(alpha) l q
        :: H l
```

The bit reversal routine can be implemented straightforwardly as exactly $\lfloor N_q/2 \rfloor$ SWAP gates.

```
def revbin(Nq):
    for i in range(Nq / 2):
        :: SWAP i (Nq - i - 1)
```

All of this is put together in a final `QFT` routine.

```
def QFT(Nq):
    QFT'(0, Nq)
    revbin(Nq)
```

B. Variational Quantum Eigensolver

The variational quantum eigensolver (VQE) [8, 9, 10] is a classical/quantum algorithm for finding eigenvalues of a Hamiltonian H variationally. The quantum subroutine has two fundamental steps.

1) Prepare the quantum state $|\Psi(\vec{\theta})\rangle$, often called the *ansatz*.
2) Measure the expectation value $\langle\, \Psi(\vec{\theta}) \,|\, H \,|\, \Psi(\vec{\theta}) \,\rangle$.

The classical portion of the computation is an optimization loop.

1) Use a classical non-linear optimizer to minimize the expectation value by varying ansatz parameters $\vec{\theta}$.
2) Iterate until convergence.

We refer to given references for details on the algorithm.

Practically, the quantum subroutine of VQE amounts to preparing a state based off of a set of parameters $\vec{\theta}$ and performing a series of measurements in the appropriate basis. Using the QAM, these measurements will end up in classical memory. Doing this iteratively followed by a small amount of postprocessing, one may compute a real expectation value for the classical optimizer to use.

This algorithm can be implemented in at least two distinct styles which impose different requirements on a quantum computer. These two styles can serve as prototypical examples for programming a QAM with Quil.

1) Static Implementation: One simple implementation of VQE is to generate a new Quil listing for every iteration of $\vec{\theta}$. Before calling out to the quantum subroutine, a new Quil program is generated for the next desired $\vec{\theta}$ and loaded onto the quantum computer for immediate execution. For a parameter $\theta = 0.00724\ldots$, one such program might look like

```
# State prep
...
CNOT 5 6
CNOT 6 7
```

```
RZ(0.00724195969993) 7
CNOT 6 7
...
# Measurements
MEASURE 0 [0]
MEASURE 1 [1]
...
```

This technique poses no issue from the perspective of coherence time, but adds a time penalty to each iteration of the classical optimizer.

A static implementation of VQE was written using Rigetti Computing's pyQuil library and the Rigetti QVM, both of which are mentioned in Sections V-B and V-F respectively.

2) Dynamic Implementation: Perhaps the most encapsulated implementation of VQE would be to use dynamic parameters. Without loss of generality, let's assume a single parameter θ. We can define a circuit which takes our single θ parameter and prepares $|\Psi\rangle$.

```
DEFCIRCUIT PREP_STATE(%theta):
    ...
    H 3
    H 5
    ...
    CNOT 3 5
    CNOT 5 6
    RZ(%theta) 6
    ...
```

Next, we define a memory layout for classical/quantum communication:

- `[0]`: Flag to indicate convergence completion.
- `[1-64]`: Dynamic parameter θ.
- `[100]`, `[101]`, ...: Measurements corresponding to $Q_0, Q_1, \ldots$.

Finally, we can define our VQE circuit.

```
DEFCIRCUIT VQE:
    LABEL @REDO
    RESET
    PREP_STATE([1-64]) # Dynamic Parameter
    MEASURE 0 [100]
    MEASURE 1 [101]
    ...
    WAIT
    JUMP-UNLESS @REDO [0]
```

This program has the advantage that the quantum portion of the algorithm is completely encapsulated. It is not necessary to dynamically reload Quil code for each newly varied parameter.

The main disadvantage of this approach is its implementation difficulty in hardware. This is because of the diminished potential for program analysis to occur before execution. The actual gates that get applied will not be known until the runtime of the algorithm (hence the "dynamic" name). This may limit opportunities for optimization and poses particular issues for current quantum computing architectures which have limited natural gate sets and limited high-speed dynamic tune-up of new gates or their approximations.

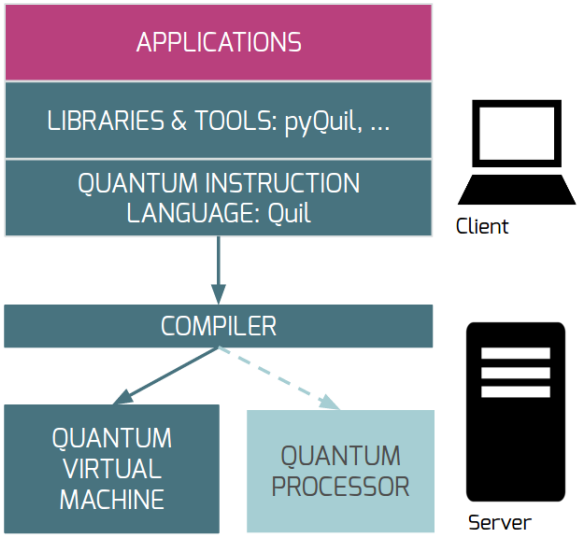

Fig. 2. Outline of Forest, Rigetti Computing's quantum programming toolkit, described in Section V.

V. Forest: A Quantum Programming Toolkit

A. Overview

Quantum computers, and specifically QAM-conforming QPUs, are not yet widely available. Despite this, software can make use of the the QAM and Quil to (a) study practical quantum algorithmic performance in a spirit similar to MMIX [28], (b) prepare a suite of software which will be able to be run on QPUs, and (c) provide a uniform interface to physical and virtualized QAMs. Rigetti Computing has built a toolkit called *Forest* for accomplishing these tasks.

The hierarchy of software roughly falls into four layers, as in figure 2. Descending in abstraction, we have

Applications & Tools Software written to accomplish real-world tasks (e.g., study electronic structure problems), and software to assist quantum programming.

Quil The language described in this document and associated software for processing Quil. It acts as an intermediate representation of general quantum programs.

Compiler Software to convert arbitrary Quil to simpler Quil (or some other representation) for execution on a QPU or QVM.

Execution Units A QPU, a QVM, or a hardware emulator. Software written at this level will typically incorporate noise models intrinsic to a particular QPU.

We will briefly describe each of these components of the toolkit in the following sections.

B. Applications and Tools

Quil is an assembly-like language that is intended to be both human readable and writable. However, more expressive power comes from being able to manipulate Quil programs programmatically. Rigetti Computing has developed a Python library called *pyQuil* [29] which allows the construction of Quil programs by treating them as first-class objects. Using pyQuil along with scientific libraries such SciPy [30], Rigetti Computing has implemented non-trivial algorithms such as VQE using the abstractions of the QAM.

C. Quil Manipulation

Quil, as a language in its own right, is amenable to processing and computation independent of any particular (real or virtual) machine for execution. Rigetti Computing has written a reusable static analyzer and parser application for Quil, that allows Quil to easily be interchanged between programs. For example, Rigetti Computing's `quil-json` program converts Quil into a structured JSON [31] format:

```
$ cat bell.quil
H 0
CNOT 0 1
$ quil-json bell.quil
{
  "type": "parsed_program",
  "executable_program": [
    {
      "type": "unresolved_application",
      "operator": "H",
      "arguments": [["qubit", 0]],
      "parameters": null
    },
    {
       "type": "unresolved_application",
       "operator": "CNOT",
       "arguments": [["qubit", 0],
                     ["qubit", 1]],
       "parameters": null
        }
    ]
}
```

Note the two instances of "`unresolved_application`". These were generated because of a simple static analysis determining that these gates were not otherwise defined in the Quil file. (This could be ameliorated by including `stdgates.quil`.)

D. Compilation

In the context of quantum computation, **compilation** is the process of converting a sequence of gates into an approximately equivalent sequence of gates executable on a quantum computer with a particular qubit topology. This requires two separate kinds of processing: gate approximation and routing.

Since Quil is specified as a symbolic language, it is amenable to symbolic analysis. Quil programs can be decomposed into *control flow graphs* (CFGs) [32] whose nodes are *basic blocks* of straight-line Quil instructions, as is typical in compiler theory. Arrows between basic blocks indicate transfers of control. For example, consider the following Quil program:

```
LABEL @START
H 0
MEASURE 0 [0]
JUMP-WHEN @END [0]
H 0
H 1
CNOT 1 0
JUMP @START
LABEL @END
Y 0
MEASURE 0 [0]
MEASURE 1 [1]
```

Roughly speaking, each segment of straight-line control makes up a basic block. Figure 3 depicts the control flow graph of this program, with jump instructions elided.

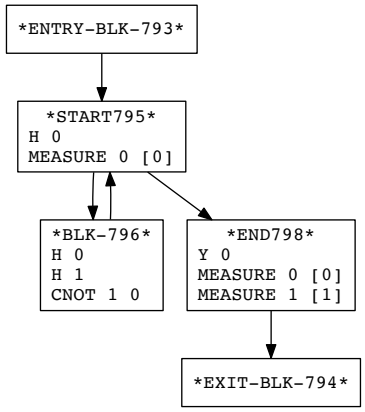

Fig. 3. The control flow graph of a Quil program.

Many of these basic blocks will be composed of gates and measurements, which themselves can be symbolically and algebraically manipulated. Gates can go through **approximation** to reduce a general set of gates to a sequence of gates native to the particular architecture, and then **routing** so that these simpler gates are arranged to act on neighboring qubits in a physical architecture. Another example of a transformation on basic blocks is parallelization, talked about in the next section.

Both approximation and routing can be formalized as transformations between QAMs. In this first example, we show how we can formally describe a transformation between a QAM to another one with a smaller but computationally equivalent gate set.

Example 17 (Compiling)**.** *Let* $\mathfrak{M} = (|\Psi\rangle, C, G_{\mathfrak{M}}, G', P, \kappa)$ *be a one-qubit QAM with*

$$G_{\mathfrak{M}} = \{\mathsf{H}, \mathsf{R}_x(\theta), \mathsf{R}_z(\theta)\}$$

for some fixed $\theta \in \mathbb{R}$, and let $\mathfrak{M}'$ be a QAM with

$$G_{\mathfrak{M}'} = \{\mathsf{H}, \mathsf{R}_z(\theta)\}.$$

Because $\mathsf{R}_x = \mathsf{H}\mathsf{R}_z\mathsf{H}$, we can define a compilation function $\mathfrak{M} \mapsto \mathfrak{M}'$ specifically transforming[8] *$\iota \in P$ according to*

$$f(\iota) = \begin{cases} (\mathsf{H}, \mathsf{R}_z(\theta), \mathsf{H}) & \text{if } \iota = \mathsf{R}_x(\theta), \\ (\iota) & \text{otherwise.} \end{cases}$$

In this next example, we show how qubit connectivity can be encoded in a QAM, and how one can route instructions to give the illusion of a fully connected topology.

Example 18 (Routing)**.** *Let $\mathfrak{M} = (|\Psi\rangle, C, G_{\mathfrak{M}}, G', P, \kappa)$ be a three-qubit QAM with*

$$G_{\mathfrak{M}} = L(\mathsf{H}) \cup L(\mathsf{CNOT}) \cup L(\mathsf{SWAP}),$$

where L was defined in Example 4. Consider a three-qubit QPU with the qubits arranged in a line

$$Q_0 — Q_1 — Q_2$$

so that two-qubit gates can only be applied on adjacent qubits. Then this QPU can be modeled by another three-qubit QAM $\mathfrak{M}'$ with the lifted gates

$$G_{\mathfrak{M}'} = \left\{ \begin{array}{c} \mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2, \\ \mathsf{CNOT}_{01}, \mathsf{CNOT}_{10}, \mathsf{CNOT}_{12}, \mathsf{CNOT}_{21}, \\ \mathsf{SWAP}_{01}, \mathsf{SWAP}_{12} \end{array} \right\}.$$

Because of the qubit topology, there are no gates that act on $\mathscr{B}_2 \otimes \mathscr{B}_0$. However, we can reason about transforming between $\mathfrak{M}$ and $\mathfrak{M}'$ in either direction. We can transform from $\mathfrak{M}$ to $\mathfrak{M}'$ by way of a transformation f similar to that in the last example, namely

$$f(\iota) = \begin{cases} (\mathsf{SWAP}_{01}, \mathsf{SWAP}_{12}, \mathsf{SWAP}_{01}) & \text{if } \iota = \mathsf{SWAP}_{02}, \\ (\mathsf{SWAP}_{01}, \mathsf{CNOT}_{12}, \mathsf{SWAP}_{01}) & \text{if } \iota = \mathsf{CNOT}_{02}, \\ (\mathsf{SWAP}_{01}, \mathsf{CNOT}_{21}, \mathsf{SWAP}_{01}) & \text{if } \iota = \mathsf{CNOT}_{20}, \\ (\iota) & \text{otherwise.} \end{cases}$$

Similarly, we can transform from $\mathfrak{M}'$ to $\mathfrak{M}$ by adding three additional gates to $G_{\mathfrak{M}'}$, namely those implied by f.

Many other classes of useful transformations on QAMs exist, such as G-preserving algebraic simplifications of P, an analog of peephole optimization in compiler theory [33].

E. Instruction Parallelism

Instruction parallelism, the ability to apply commuting operations to a quantum state in parallel, is one of the many benefits of quantum computation. Quil code as written is linear and serial, but can be interpreted as an instruction-parallel program. In particular, many subsequences of Quil instructions may be executed in parallel. Such sequences include:

- Commuting gate applications and measurements, and
- Measurements with non-overlapping memory addresses.

In general, parallelization cannot occur over jumps, resets, waits, and measurements and dynamic gate applications with overlapping address ranges. We suggest that `NOP` is used as a way to force a parallelization break.

If a control flow graph is constructed as in Section V-D, then parallelization can be done over each basic block. A parallelized basic block is called a **parallelization schedule**. See Figure 4 for an example of quantum parallelization within a CFG.

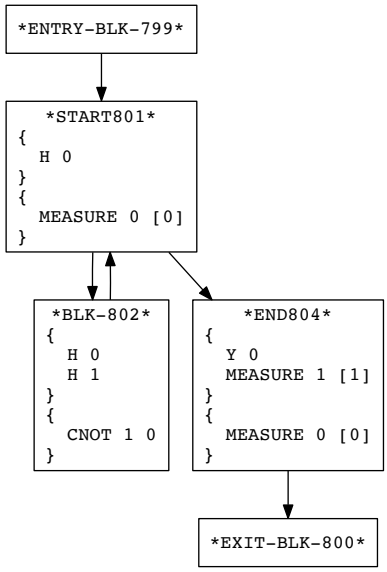

Fig. 4. The parallelized version of Figure 3. Instruction sequences which can be executed in parallel are surrounded in curly braces '{}'.

F. Rigetti Quantum Virtual Machine

Rigetti Computing has implemented a QVM in ANSI Common Lisp [34] called the *Rigetti QVM*. It is a high-performance[9], programmable simulator for the QAM and emulator for noisy quantum computers. The Rigetti QVM exposes two interfaces for executing quantum programs: execution of Quil files directly with POSIX-style shared memory ("local execution"), and execution of Quil from HTTP server requests ("remote execution").

Local execution is useful for high-speed testing of small-to-medium sized instances of classical/quantum algorithms. It also provides convenient ways of debugging quantum programs, such as allowing direct inspection of the quantum state at any point in the execution process, as well as limited quantum hardware emulation with tunable noise models.

Remote execution is used for distributed, cloud access to QVM instances. HTTP libraries exist in nearly all modern programming languages, and allow programs in

[8] In the parlance of functional programming, f is applied to P via a *concatmap* operation.

[9] The Rigetti QVM has optimized vectorized and parallelized numerics, and has no theoretical limit for the number of qubits it can simulate. It has been demonstrated to simulate 36 qubits.

these languages to make connections. Rigetti Computing has built in to pyQuil the ability to send the first-class Quil program objects to a local or secured remote Rigetti QVM instance using the Forest API.

VI. Conclusion

We have introduced a practical abstract machine for reasoning about and executing quantum programs. Furthermore, we have described a notation for such programs on this machine, which is amenable to analysis, compilation, and execution. Finally, we have described a pragmatic toolkit for quantum program construction and execution built atop these ideas.

VII. Acknowledgements

The authors would like to thank their colleagues at Rigetti Computing for their support, especially Nick Rubin. We are also grateful to Erik Davis, Jarrod McClean, Bill Millar, and Eric Peterson for their helpful discussions and valuable suggestions provided throughout the development of this work.

Appendix

A. The Standard Gate Set

The following static and parametric gates are defined in `stdgates.quil`. Many of these gates are standard gates used in theoretical quantum computation [23], and some of them find their origin in the theory of superconducting qubits [35].

Pauli Gates

$$\mathtt{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathtt{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \mathtt{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \mathtt{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Hadamard Gate

$$\mathtt{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Phase Gates

$$\mathtt{PHASE}(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix} \quad \mathtt{S} = \mathtt{PHASE}(\pi/2) \quad \mathtt{T} = \mathtt{PHASE}(\pi/4)$$

Controlled-Phase Gates

$$\begin{aligned} \mathtt{CPHASE00}(\theta) &= \mathrm{diag}(e^{i\theta}, 1, 1, 1) \\ \mathtt{CPHASE01}(\theta) &= \mathrm{diag}(1, e^{i\theta}, 1, 1) \\ \mathtt{CPHASE10}(\theta) &= \mathrm{diag}(1, 1, e^{i\theta}, 1) \\ \mathtt{CPHASE}(\theta) &= \mathrm{diag}(1, 1, 1, e^{i\theta}) \end{aligned}$$

Cartesian Rotation Gates

$$\begin{aligned} \mathtt{RX}(\theta) &= \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \\ \mathtt{RY}(\theta) &= \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \\ \mathtt{RZ}(\theta) &= \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \end{aligned}$$

Controlled-X Gates

$$\mathtt{CNOT} = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix} \qquad \mathtt{CCNOT} = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \end{pmatrix}$$

Swap Gates

$$\begin{aligned} \mathtt{PSWAP}(\theta) &= \begin{pmatrix} 1&0&0&0 \\ 0&0&e^{i\theta}&0 \\ 0&e^{i\theta}&0&0 \\ 0&0&0&1 \end{pmatrix} \\ \mathtt{SWAP} &= \mathtt{PSWAP}(0) \\ \mathtt{ISWAP} &= \mathtt{PSWAP}(\pi/2) \end{aligned}$$

$$\mathtt{CSWAP} = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \end{pmatrix}$$

B. Prior Work

There exists much literature on abstract models, syntaxes, and semantics of quantum computing. Most of them are in the form of quantum programming languages (QPLs) and simulators, which achieve various levels of expressiveness. Languages roughly fall into three categories:

- embedded domain-specific languages,
- high-level quantum programming languages, and
- low-level quantum intermediate representations.

In addition, work has been done on designing larger tool chains for quantum programming [18, 19, 36].

In the following subsections, we provide a non-exhaustive account of some previous work within the above categories, and describe how they relate to the design of our quantum ISA.

1) Embedded Domain-Specific Languages: An **embedded domain-specific language** (EDSL) is a language to express concepts and problems from within another programming language. Within the context of quantum programming, two prominent examples are Quipper [16], which is embedded in Haskell [37], and LIQ$Ui|\rangle$ [15], which is embedded in F# [38]. Representation of quantum programs (and, in particular, the subclass of quantum programs called *quantum circuits*) is expressed with data structures in the host language. Feedback of classical information is achieved through an interface with the host language.

Quantum programs written in an EDSL are not directly executable on a quantum computer, due to the requirement of being present in the host language's runtime. However, since quantum programs are represented as first-class objects, these objects are amenable to processing and compilation to a quantum computer's natively executable instruction set. Quil is one such target for compilation.

2) High-Level QPLs: High-level QPLs are the quantum analog of languages like C [39] in the imperative case or ML [40] in the functional case. They provide a plethora of classical and quantum data types, as well as control flow constructs. In the case of functional quantum languages, the lambda calculus for quantum computation with classical control by Selinger and Valiron can act as a theoretical basis for their semantics of such a language[10] [42].

[10] This is similar to how the classical untyped lambda calculus formed the basis for LISP in 1958 [41].

One prominent example of a high-level QPL is Bernhard Ömer's QCL [43]. It is a C-like language with classical control and quantum data. Among the many that exist, two important—and indeed somewhat dual—data types are `int` and `qureg`. The following example shows a Hadamard initialization on eight qubits, and measuring the first four of those qubits into an integer variable.

```
// Allocate classical and quantum data
qureg q[8];
int m;
// Hadamard initialize
H(q);
// Measure four qubits into m
measure q[0..3], m;
// Print m, which will be anywhere from 0 to 15.
print m;
```

Ömer defines the semantics of this language in detail in his PhD thesis, and presents an implementation of a QCL interpreter written in C++ [44]. However, a compilation or execution strategy on quantum hardware is not presented. Similar to EDSLs, QPLs such as QCL can be compiled into a lower-level quantum instruction set such as Quil.

3) Low-Level Quantum Intermediate Representations: In the context of compiler theory, an **intermediate representation** (IR) is some representation of a language which is amenable to further processing. IR can be higher level (as with abstract syntax trees), or lower level (as with linear bytecodes). Low-level IRs often act as compilation targets for classical programming languages, and are used for analysis and program optimization. For example, LLVM IR [45] is used as an intermediate compilation target for the Clang C compiler [46].

The most well-known example of a low-level quantum IR is QASM [47]. This was originally a language to describe the quantum circuits for LaTeX output in [23], and hence, *not* an IR in that form. However, the syntax was adapted for executable use in [48]. QASM, however, does not have any notion of classical control within the language, acting solely as a quantum circuit description language.

Quil is considered a low-level quantum IR with classical control.

References

[1] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White, P. V. Coveney, P. J. Love, H. Neven, A. Aspuru-Guzik, and J. M. Martinis, "Scalable quantum simulation of molecular energies," *Phys. Rev. X*, vol. 6, p. 031007, Jul 2016. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevX.6.031007

[2] M. R. Geller, J. M. Martinis, A. T. Sornborger, P. C. Stancil, E. J. Pritchett, H. You, and A. Galiautdinov, "Universal quantum simulation with prethreshold superconducting qubits: Single-excitation subspace method," *Physical Review A*, vol. 91, no. 6, p. 062309, 2015.

[3] R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. Las Heras, R. Babbush, A. Fowler, B. Campbell, Y. Chen *et al.*, "Digitized adiabatic quantum computing with a superconducting circuit," *Nature*, vol. 534, no. 7606, pp. 222–226, 2016.

[4] D. Riste, M. P. da Silva, C. A. Ryan, A. W. Cross, J. A. Smolin, J. M. Gambetta, J. M. Chow, and B. R. Johnson, "Demonstration of quantum advantage in machine learning," *arXiv preprint arXiv:1512.06069*, 2015.

[5] J. M. Chow, S. J. Srinivasan, E. Magesan, A. Córcoles, D. W. Abraham, J. M. Gambetta, and M. Steffen, "Characterizing a four-qubit planar lattice for arbitrary error detection," in *SPIE Sensing Technology+ Applications*. International Society for Optics and Photonics, 2015, pp. 95 001G–95 001G.

[6] J. Kelly, R. Barends, A. Fowler, A. Megrant, E. Jeffrey, T. White, D. Sank, J. Mutus, B. Campbell, Y. Chen *et al.*, "State preservation by repetitive error detection in a superconducting quantum circuit," *Nature*, vol. 519, no. 7541, pp. 66–69, 2015.

[7] D. Ristè, S. Poletto, M.-Z. Huang, A. Bruno, V. Vesterinen, O.-P. Saira, and L. DiCarlo, "Detecting bit-flip errors in a logical qubit using stabilizer measurements," *Nature communications*, vol. 6, 2015.

[8] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature communications*, vol. 5, 2014.

[9] D. Wecker, M. B. Hastings, and M. Troyer, "Progress towards practical quantum variational algorithms," *Physical Review A*, vol. 92, no. 4, p. 042303, 2015.

[10] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms," *New Journal of Physics*, vol. 18, no. 2, p. 023023, 2016.

[11] B. Bauer, D. Wecker, A. J. Millis, M. B. Hastings, and M. Troyer, "Hybrid quantum-classical approach to correlated materials," *arXiv preprint arXiv:1510.03859*, 2015.

[12] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[13] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, "Elucidating reaction mechanisms on quantum computers," *arXiv preprint arXiv:1605.03590*, 2016.

[14] D. Deutsch, "Quantum Theory, the Church–Turing Principle and the Universal Quantum Computer," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.

[15] D. Wecker and K. M. Svore, "LIQU$i|\rangle$: A Software Design Architecture and Domain-Specific Language for Quantum Computing," 2014. [Online]. Available: arXiv:1402.4467v1

[16] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger,

and B. Valiron, "Quipper: A scalable quantum programming language," *SIGPLAN Not.*, vol. 48, no. 6, pp. 333–342, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462177

[17] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *arXiv preprint arXiv:1604.01401*, 2016.

[18] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *IEEE Computer*, vol. 39, no. 1, pp. 74–83, 2006.

[19] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[20] T. Häner, D. S. Steiger, M. Smelyanskiy, and M. Troyer, "High performance emulation of quantum circuits," *arXiv preprint arXiv:1604.06460*, 2016.

[21] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, "qhipster: the quantum high performance software testing environment," *arXiv preprint arXiv:1601.07195*, 2016.

[22] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Physical review A*, vol. 52, no. 5, p. 3457, 1995.

[23] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2010.

[24] D. Deutsch, A. Barenco, and A. Ekert, "Universality in quantum computation," in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 449, no. 1937. The Royal Society, 1995, pp. 669–677.

[25] S. Aaronson, *Quantum computing since Democritus*. Cambridge University Press, 2013.

[26] *IEEE standard for binary floating-point arithmetic*, New York, 1985, note: Standard 754–1985.

[27] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[28] D. E. Knuth, *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium (Art of Computer Programming)*. Addison-Wesley Professional, 2005.

[29] Rigetti Computing, "pyQuil," 2016, accessed: 2016-08-10. [Online]. Available: https://github.com/rigetticomputing/pyQuil

[30] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–2016, accessed: 2016-06-21. [Online]. Available: http://www.scipy.org/

[31] *The JSON Data Interchange Format*, 1st ed., October 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[32] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[33] W. M. McKeeman, "Peephole optimization," *Commun. ACM*, vol. 8, no. 7, pp. 443–444, Jul. 1965. [Online]. Available: http://doi.acm.org/10.1145/364995.365000

[34] American National Standards Institute and Information Technology Industry Council, *American National Standard for Information Technology: programming language — Common LISP*, American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996, approved December 8, 1994.

[35] J. M. Chow, "Quantum information processing with superconducting qubits," Ph.D. dissertation, Yale University, 2010.

[36] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, "Scaffold: Quantum programming language," DTIC Document, Tech. Rep., 2012.

[37] S. Marlow, "Haskell 2010 Language Report," https://www.haskell.org/definition/haskell2010.pdf, 2010, accessed: 2016-06-21.

[38] D. Syme *et al.*, "The F# 4.0 Language Specification," http://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-latest.pdf, 2016, accessed: 2016-06-21.

[39] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[40] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[41] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[42] P. Selinger and B. Valiron, "A lambda calculus for quantum computation with classical control," *Mathematical Structures in Computer Science*, vol. 16, no. 03, pp. 527–552, 2006.

[43] B. Ömer, "Structured quantum programming," Ph.D. dissertation, Technical University of Vienna, 2003. [Online]. Available: http://tph.tuwien.ac.at/~oemer/doc/structquprog.pdf

[44] ——, "QCL - A Programming Language for Quantum Computers," http://tph.tuwien.ac.at/~oemer/qcl.html, 2014, accessed: 2016-06-30.

[45] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[46] Clang developers, "clang: a C language family frontend for LLVM," http://clang.llvm.org/, accessed: 2016-06-30.

[47] MIT Quanta Group, "Quantum circuit viewer: qasm2circ," http://www.media.mit.edu/quanta/qasm2circ/, accessed: 2016-06-30.

[48] ——, "qasm-tools," http://www.media.mit.edu/quanta/quanta-web/projects/qasm-tools/, accessed: 2016-06-30.