Quantum Control Machine

The Limits of Control Flow in Quantum Programming

CHARLES YUAN, MIT CSAIL, USA
AGNES VILLANYI, MIT CSAIL, USA
MICHAEL CARBIN, MIT CSAIL, USA

Quantum algorithms for tasks such as factorization, search, and simulation rely on control flow such as branching and iteration that depends on the value of data in superposition. High-level programming abstractions for control flow, such as switches, loops, higher-order functions, and continuations, are ubiquitous in classical languages. By contrast, many quantum languages do not provide high-level abstractions for control flow in superposition, and instead require the use of hardware-level logic gates to implement such control flow.

The reason for this gap is that whereas a classical computer supports control flow abstractions using a program counter that can depend on data, the typical architecture of a quantum computer does not analogously provide a program counter that can depend on data in superposition. As a result, the complete set of control flow abstractions that can be correctly realized on a quantum computer has not yet been established.

In this work, we provide a complete characterization of the properties of control flow abstractions that are correctly realizable on a quantum computer. First, we prove that even on a quantum computer whose program counter exists in superposition, one cannot correctly realize control flow in quantum algorithms by lifting the classical conditional jump instruction to work in superposition. This theorem denies the ability to directly lift general abstractions for control flow such as the λ-calculus from classical to quantum programming.

In response, we present the necessary and sufficient conditions for control flow to be correctly realizable on a quantum computer. We introduce the quantum control machine, an instruction set architecture featuring a conditional jump that is restricted to satisfy these conditions. We show how this design enables a developer to correctly express control flow in quantum algorithms using a program counter in place of logic gates.

CCS Concepts: • **Software and its engineering** → *Control structures*; • **Theory of computation** → Abstract machines; *Control primitives*; • **Computer systems organization** → **Quantum computing**.

Additional Key Words and Phrases: quantum programming languages, quantum instruction set architectures

ACM Reference Format:
Charles Yuan, Agnes Villanyi, and Michael Carbin. 2024. Quantum Control Machine: The Limits of Control Flow in Quantum Programming. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 94 (April 2024), 28 pages. https://doi.org/10.1145/3649811

1 INTRODUCTION

Quantum algorithms promise computational advantage in areas ranging from factorization [Shor 1997] and search [Grover 1996] to data analysis [Harrow et al. 2009; Lloyd et al. 2014; Wiebe et al. 2012] and simulation [Abrams and Lloyd 1997; Babbush et al. 2018; Childs et al. 2018].

The power of a quantum algorithm arises from its ability to manipulate quantum data, which exists in a weighted sum over many classical states known as a *superposition*. The basic unit of

Authors' addresses: Charles Yuan, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, chenhuiy@csail.mit.edu; Agnes Villanyi, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, agivilla@mit.edu; Michael Carbin, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, mcarbin@csail.mit.edu.

quantum information is the *qubit* — a superposition of the bits 0 and 1. A quantum computer can transform the values and weights within a superposition by performing quantum logic gates, formally known as *unitary operators*. It can also *measure* quantum data, which collapses a superposition to a classical state with a probability that is determined by its weight in the superposition.

One example of data in superposition is a 2-qubit quantum integer k that takes on the values 0, 1, 2, and 3 at the same time. In the notation of quantum algorithms, one such k is written as $\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle - \frac{1}{2}|2\rangle + \frac{1}{2}|3\rangle$ where $|2\rangle$ or $|10_2\rangle$ in binary denotes a qubit in the 1 state and a qubit in the 0 state, and $-\frac{1}{2}$ denotes the weight of that state, whose sign component is called a *phase*.

Unlike classical probabilities that are non-negative real numbers, the weights in a superposition are complex numbers whose values can combine or cancel when added, in a phenomenon known as *interference*. In turn, interference enables quantum advantage by amplifying the probability that a quantum algorithm produces a correct result to be larger than that of any classical algorithm.

A widely adopted representation of a quantum computation is as a *quantum circuit*, a sequence of quantum logic gates operating over qubits that is the quantum analogue of a Boolean logic circuit. To assist in manipulating the complex quantum circuits that arise when implementing quantum algorithms, researchers have developed *quantum programming languages* [Altenkirch and Grattage 2005; Bichsel et al. 2020; Green et al. 2013; Paykin et al. 2017; Svore et al. 2018; Voichick et al. 2023].

1.1 The Challenge of Control Flow in Superposition

A programming abstraction that is integral to classical algorithms is control flow such as branching and iteration that depends on the value of data. An analogous concept is also integral to quantum algorithms, in which control flow depends on the value of data in quantum superposition.

Example 1.1 (Branching). Where a classical computation takes a value k and executes the kth branch of a switch statement on the value x, a quantum computation takes a superposition of k and executes a superposition of the corresponding branches to produce a superposition of x.

In the formal notation of quantum algorithms, given a set of functions $\{U_i\}$ representing the branches, branching transforms the data from $\sum_k |k\rangle |x\rangle \mapsto \sum_k |k\rangle (U_k |x\rangle)$. This operation is used in algorithms for physical simulation [Babbush et al. 2018; Childs and Wiebe 2012; Low and Chuang 2019], in which each U_i encodes a component of the description of the target system.

Example 1.2 (Iteration). Where a classical computation takes a value k and repeats an operation for k iterations on the value x, a quantum computation takes a superposition of k and repeats the operation for a superposition of numbers of iterations to produce a superposition of x.

In formal notation, given a function U whose ith power is U^i, iteration transforms the data from $\sum_k |k\rangle |x\rangle \mapsto \sum_k |k\rangle (U^k |x\rangle)$. Iteration is a special case of branching that is used in algorithms for factoring [Shor 1997], where U maps $|x\rangle \mapsto |ax \bmod N\rangle$, and for phase estimation [Kitaev 1995] as found in simulation [Abrams and Lloyd 1997] and linear algebra [Harrow et al. 2009].

Programming abstractions for control flow are natively supported by the typical architecture of a classical computer. In imperative programming, the `if`-statement for branching and `for`-loop for iteration compile to a *program counter* that determines the current instruction and a *conditional jump* instruction that updates the program counter using a condition on a data register. Control flow is also straightforward to realize in functional programming, in which abstractions for branching and iteration emerge from the Church encoding [Church 1941] of data types in the λ-calculus.

By contrast, programming abstractions for control flow in superposition are not natively supported by the typical architecture of a quantum computer. Whereas a classical computer provides a program counter that can depend on data, the typical architecture of a quantum computer does not provide a program counter that can depend on data in superposition, nor a representation of

λ-terms in superposition. Instead, it requires a program to be represented as a quantum circuit, a fixed sequence of bit-level logic gates whose structure cannot depend on data in superposition.

In turn, the lack of a program counter requires alternatives to the typical strategy of compiling abstractions for control flow to manipulations of the program counter. For example, some quantum languages [Green et al. 2013; Paykin et al. 2017; Qiskit Developers 2021; Svore et al. 2018] enable the developer to implement Examples 1.1 and 1.2 by explicitly building a circuit of bit-controlled logic gates. Alternatively, emerging languages [Bichsel et al. 2020; Yuan and Carbin 2022] provide certain abstractions for control flow, such as a quantum `if`-statement that branches over a qubit in superposition, that are guaranteed to be physically realizable and can be compiled into circuits.

Prior work, however, leaves unknown whether it is possible to program a quantum computer using other general forms of control flow, such as higher-order functions, that underpin expressive classical languages. Answering this question requires identifying the complete set of control flow abstractions that a quantum computer can support – that is, the necessary and sufficient conditions for control flow in superposition to be correctly realizable – which has not been done to date.

1.2 Theoretical Limits of Control Flow in Superposition

In this work, we provide a complete characterization of the properties of control flow abstractions that are correctly realizable on a quantum computer, showing that many general forms of control flow from classical programming fail to work correctly over data in quantum superposition.

First, we prove that even given a quantum computer endowed with a representation of a program counter in superposition, one cannot correctly realize control flow within a quantum algorithm by directly lifting the classical conditional jump instruction to work on data in superposition. In turn, the typical strategy from classical programming of compiling arbitrary control flow abstractions to conditional jumps can lead a quantum computer to incorrectly execute quantum algorithms.

Landauer Embedding. To explain why the classical conditional jump does not work on a quantum computer, we first show that the standard technique researchers use to lift a classical computation into a quantum one gives incorrect results for control flow abstractions such as conditional jump.

Fundamentally, the hardware primitives that can be used to realize a quantum computer that supports control flow in superposition are quantum logic gates, which operate on quantum data without measuring it and thereby inappropriately collapsing its superposition. The reason that it is critical for quantum data to not collapse from superposition is to ensure that the data correctly exhibits interference. Without interference, an algorithm such as Shor [1997] would produce a correct output with exponentially smaller probability and thus lose its quantum advantage.

The challenge is that the mathematical semantics of each quantum logic gate is a unitary operator, which is an invertible and therefore injective function. By contrast, the state transition function of the conventional conditional jump instruction is not injective. When two distinct instructions jump to the same point, then the identity of the machine state before the jump is lost after the jump.

One reason to hope that this problem may be solvable is a technique, developed by researchers in quantum algorithms and complexity theory, to lift a non-injective classical computation to an injective quantum computation. Specifically, one can convert any function $f(x)$ into an injective function $g(x) = (x, f(x))$ that returns a copy of its input. When iterated over an entire computation, this process yields the computation's output alongside a *history* of its intermediate states.

This standard technique is known as *Landauer embedding* [Landauer 1961], so named as it embeds a computation into a larger state space containing the history. Prior work [Lagana et al. 2009] has proposed to use Landauer embedding to implement the non-injective semantics of conditional jump as a unitary operator. In this scheme, the machine stores a history of previous values of the program counter, and appends a value to the history upon executing each jump instruction.

Disruptive Entanglement. In this work, however, we demonstrate that a quantum computer that uses Landauer embedding to implement conditional jump produces incorrect outputs, which lead a quantum algorithm to not correctly produce interference and to lose its quantum advantage.

The cause is quantum *entanglement*, the phenomenon in which a superposition state cannot be represented as the product of independent components, and in which discarding one component necessarily collapses the superposition of the other. For a program whose control flow depends on data, the machine produces a state in which the history of the program counter is entangled with the output data. Here, being entangled with the history means that the output data fails to correctly produce interference, and the quantum algorithm produces incorrect results.

No-Embedding Theorem. More generally, we prove that a quantum computer cannot correctly support any form of control flow in superposition, including conditional jump, whose state transition semantics is not injective. Formally, we generalize the Landauer embedding to the concept of a *quantum embedding* that defines the most general way to lift a classical computation to a quantum one by storing auxiliary information. Then, we prove a *no-embedding theorem* stating that using any such technique to realize control flow causes the computer to produce incorrect outputs.

As a corollary, we prove that a λ-calculus featuring superpositions of λ-terms cannot be used to program a quantum computer, as conjectured by van Tonder [2004]. The reason is that β-reduction is not injective — both $\lambda x.x$ and $(\lambda x.x)(\lambda x.x)$ reduce to $\lambda x.x$. This result precludes the Church encoding of quantum information into λ-terms, in that it prevents the superposition of two bits $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ from being represented as a superposition of λ-terms $\frac{1}{\sqrt{2}}(|\lambda x.\lambda y.x\rangle + |\lambda x.\lambda y.y\rangle)$.

Implications. Our theorem provides a unifying explanation of why numerous classical control flow abstractions, ranging from closures to continuations, remain challenging to adapt to quantum programming since their classical basis in conditional jumps or the λ-calculus cannot be correctly realized in superposition. New abstractions must take their place in quantum programming.

This result raises caution for proposals from the hardware community [Meier et al. 2024; Wang 2022] for a quantum equivalent of the von Neumann architecture in the form of a reprogrammable quantum computer that stores instructions alongside data in superposition. Though experimentalists have attempted to physically realize such a design [Kjaergaard et al. 2020], our theorem implies that it necessarily supports limited forms of control flow compared to classical computers.

To our knowledge, prior designs of quantum von Neumann architectures do not acknowledge or account for the fundamental limitations to control flow that are formalized by our no-embedding theorem. For instance, to implement a conditional jump instruction, the proposal of Lagana et al. [2009] uses the approach of histories and hence does not correctly execute quantum algorithms.

1.3 Specification for Sound Control Flow in Superposition

The no-embedding theorem implies that one cannot use arbitrary classical abstractions for control flow to correctly program a quantum computer. To codify the properties of forms of control flow that are correct to use on a quantum computer, we next present the necessary and sufficient conditions for control flow in superposition to be correctly realizable as part of a quantum program.

The first correctness condition, *injectivity*, specifies the control flow abstractions that may be correctly realized on a quantum computer. The second, *synchronization*, specifies the valid programs that may be constructed using these abstractions to correctly implement quantum algorithms.

Injectivity. As stated by the no-embedding theorem, a quantum computer can correctly realize a programming abstraction for control flow in superposition only if its state transition semantics is inherently injective. The semantics cannot use an embedding — that is to say, it cannot accumulate a history or any other auxiliary information not integral to the control state of the machine.

Synchronization. Injectivity is strong enough to guarantee that an abstraction is realizable, but not that all programs constructed from it produce correct outputs. Given a computation in which the program counter exists in superposition, it is possible for the program counter to become entangled with the data registers and remain entangled in the final machine state of the computation. If so, the problem of disruptive entanglement arises once again, meaning the output is incorrect.

To address this issue, the second necessary condition of *synchronization* states that control flow must eventually become separable from, i.e. not entangled with, the data. More precisely, a program is synchronized when at the end of execution, the value of the program counter is identical across each classical state within the machine state superposition. Hence, the data registers and program counter are not entangled, making the output data correct for use by the quantum algorithm.

Implications. Critically, the fact that control flow must be synchronized to be correctly realizable implies that a quantum computer based on the concept of a program counter can support a `while`-loop whose number of iterations is a data-dependent value in superposition only if that number is bounded by a classical value. Only then can the loop be synchronized and thus correctly realizable.

1.4 Instruction Set Architecture for Control Flow in Superposition

To implement the specification above, we present the *quantum control machine*, an instruction set architecture for quantum programming with control flow in superposition. This architecture is physically realizable via quantum logic gates and is the first to provide both a representation of the program counter in superposition and a sound means of manipulating the program counter.

Instruction Set. Instead of the conventional conditional jump, the machine uses other control flow instructions whose state transition semantics are inherently injective. These instructions, originally introduced by architectures for classical reversible computers [Axelsen et al. 2007; Thomsen et al. 2012], use reversible arithmetic to manipulate a *branch control* register whose value tracks how much the program counter advances and is added to the program counter after each cycle.

On top of the control flow instructions from prior work, the quantum control machine adds the ability to execute unitary operators that create and manipulate superpositions of data such as the Hadamard gate, and provides a representation of a program counter in superposition.

Case Studies. The quantum control machine unifies several forms of control flow that can be implemented in existing quantum programming languages. In a case study that implements core components of quantum algorithms for phase estimation, quantum walk, and physical simulation, we illustrate how a developer can realize the imperative control flow abstractions of branching (`switch`) and bounded iteration (`for`) as synchronized programs. The case study demonstrates how existing control flow patterns that appear in quantum algorithms can be represented in a uniform way using the abstraction of a program counter in superposition and its correct manipulation.

Implications. Rather than as a target for near-term hardware realization, which is likely to be challenging, we view the quantum control machine as a theoretical model for expressing algorithms that reduces reliance on hardware-level logic gates, and as an intermediate compilation target for proposed quantum programming languages with control flow abstractions such as recursion [Ying 2014; Ying et al. 2012] that have to date not been realized directly in terms of circuits.

A problem that remains open is whether an injective analogue of the λ-calculus could be similarly developed, which would enable functional programming abstractions for control flow in superposition. The challenge is that in general, function application is not injective, as there is no general way to turn the result of a function application back into a pair of the function and its argument. Moreover, any substitution-based model of computation in which an expression reduces to a final value, and that value also reduces to itself, is fundamentally not injective.

1.5 Contributions

In this work, we present the following contributions:

- (Section 3) We identify that the Landauer embedding, a standard approach to lift classical to quantum computation, does not correctly realize a conditional jump instruction in superposition.
- (Section 4.1) We prove that programming abstractions with non-injective transition semantics, such as the conventional conditional jump or the β-reduction of λ-calculus, cannot correctly realize control flow in superposition, thereby proving conjecture of van Tonder [2004].
- (Section 4.2) We define the necessary and sufficient conditions for control flow in superposition to be correctly realizable as part of a quantum program. First, each programming abstraction must have injective state transition semantics. Second, a program must not entangle the states of data and control flow in its final output, a condition we term synchronization.
- (Sections 5 and 6) We introduce the quantum control machine, an instruction set architecture that correctly supports imperative abstractions for branching and bounded iteration in superposition, and present a case study that uses the machine to implement a range of quantum algorithms.

Summary. In this work, we reveal limits on our ability to lift foundational programming abstractions such as the conditional jump and the λ-calculus to work with data in quantum superposition, a stark contrast to the historical development of control flow abstractions in classical programming. Faced with these limits, we propose sound principles for using control flow in quantum programs. These principles underpin a new instruction set architecture that paves way to more convenient theoretical models for quantum algorithms and more expressive quantum programming languages.

2 BACKGROUND ON QUANTUM COMPUTATION

This section overviews key concepts in quantum computation relevant to this work. For a comprehensive reference in quantum computation, please see Nielsen and Chuang [2010].

Superposition. A *qubit* exists in a *superposition* of the classical states 0 and 1 — a linear combination $\gamma_0 |0\rangle + \gamma_1 |1\rangle$ where $\gamma_0, \gamma_1 \in \mathbb{C}$ are complex *amplitudes* satisfying $|\gamma_0|^2 + |\gamma_1|^2 = 1$. Examples of qubits are classical $|0\rangle$ and $|1\rangle$, as well as the states $\frac{1}{\sqrt{2}}(|0\rangle + e^{i\varphi} |1\rangle)$ where $\varphi \in [0, 2\pi)$ is known as a *phase*. More precisely, φ is the phase of $|1\rangle$ *relative* to $|0\rangle$. By contrast, two states that differ only by a *global* phase, such as $|0\rangle$ and $e^{i\varphi} |0\rangle$, are physically indistinguishable and considered equivalent.

Quantum State. More generally, a *quantum state* $|\psi\rangle$ of dimension 2^n is a superposition over n-bit strings. For example, $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is a quantum state over two qubits.

The set of 2^n-dimensional quantum states constitutes the Hilbert space, i.e. the formal vector space, $\mathbb{C}^{2^n}$. A frequently used basis for this space, known as the *computational basis*, is the subset of states $\{|x\rangle \mid x \text{ is an } n\text{-bit string}\}$ in which one classical state has the entire amplitude 1.

Tensor Product. Formally, multiple component states form a composite state by the *tensor product* operator $\otimes$. For example, the state $|01\rangle$ is equal to $|0\rangle \otimes |1\rangle$. As is standard in quantum computation, we interchangeably use the notations $|0\rangle |1\rangle$, $|01\rangle$, and $|0, 1\rangle$ to represent $|0\rangle \otimes |1\rangle$.

Physical Operations. The *norm* of a quantum state $|\psi\rangle = \sum_i \gamma_i |\psi_i\rangle$ is defined as $\||\psi\rangle\| = \sum_i |\gamma_i|^2$. A quantum state is physically realizable only if its norm is 1. An operator, i.e. function O over states is *norm-preserving* if $\||\psi\rangle\| = \|O |\psi\rangle\|$ for any $|\psi\rangle$, and *linear* if $O(\gamma_1 |\psi_1\rangle + \gamma_2 |\psi_2\rangle) = \gamma_1(O |\psi_1\rangle) + \gamma_2(O |\psi_2\rangle)$ for any γ_i and $|\psi_i\rangle$. There are exactly two types of operations over quantum states that are physically realizable on a quantum computer — *unitary operators* and *measurement*.

Unitary Operator. A *unitary operator* U is a linear and norm-preserving operator over quantum states. Any unitary operator U satisfies the property that its inverse U^{-1} is equal to its Hermitian adjoint $U^\dagger$. Formally, a unitary operator may be constructed as a circuit of *quantum gates*. For example, the quantum gates that operate over a single qubit include:

- Bit flip (X or NOT), which maps $|x\rangle \mapsto |1-x\rangle$ for $x \in \{0,1\}$;
- Phase flip (Z), which maps $|x\rangle \mapsto (-1)^x |x\rangle$;
- Hadamard (H), which maps $|x\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x |1\rangle)$.

A gate may be *controlled* by one or more qubits, forming a larger unitary operator. For example, the two-qubit CNOT gate maps $|0\rangle |x\rangle \mapsto |0\rangle |x\rangle$ and $|1\rangle |x\rangle \mapsto |1\rangle \, \mathtt{NOT} \, |x\rangle = |1\rangle |1-x\rangle$.

Measurement. Measuring a quantum state probabilistically collapses its superposition into a classical outcome.[1] Measuring a qubit $\gamma_0 |0\rangle + \gamma_1 |1\rangle$ yields 0 with probability $|\gamma_0|^2$ and 1 with probability $|\gamma_1|^2$. Selectively measuring a state yields a partial outcome and an unmeasured remainder. For example, measuring the first qubit in the state $|\psi\rangle = \gamma_0 |0\rangle |\psi_0\rangle + \gamma_1 |1\rangle |\psi_1\rangle$ yields the outcome 0 and remainder $|\psi_0\rangle$ with probability $|\gamma_0|^2$, and outcome 1 and remainder $|\psi_1\rangle$ with probability $|\gamma_1|^2$.

Copying and Discarding. The *no-cloning* theorem [Wootters and Zurek 1982] says that no physical process can transform $|\psi\rangle \mapsto |\psi\rangle \otimes |\psi\rangle$ for arbitrary $|\psi\rangle$. Quantum data can be copied only if its basis is fixed — that is, classical information in computational basis can be copied, whereas arbitrary superpositions cannot. The *no-deleting* theorem [Pati and Braunstein 2000] states that no unitary operator realizes the converse process $|\psi_1\rangle \otimes |\psi_2\rangle \mapsto |\psi_1\rangle$ to delete an arbitrary $|\psi_2\rangle$. In fact, the principle of *implicit measurement* [Nielsen and Chuang 2010, Section 4.4] dictates that discarding a quantum state, i.e. throwing it away permanently, is indistinguishable from measuring it.

Entanglement. Given a product $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$, measuring $|\psi_1\rangle$ leaves behind the remainder $|\psi_2\rangle$, and we call such a state $|\psi\rangle$ *separable*. The opposite of a separable state is an *entangled* state that cannot be written as a tensor product of two components. Given an entangled state, measuring one component causes the superposition of the other to also collapse. For example, the *Bell state* [Bell 1964] $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled as it cannot be written as a product of two independent qubits. In this state, measuring either of the qubits causes both qubits to collapse from superposition to equal outcomes: either $|0\rangle$ and $|0\rangle$ or $|1\rangle$ and $|1\rangle$ with probability $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$ each.

Interference. A superposition state fundamentally differs from the distribution of outcomes to which it collapses — only the former exhibits quantum *interference*, the phenomenon in which the complex amplitudes of a state combine and cancel, as needed by quantum algorithms.

For example, applying the Hadamard gate to the qubit $|0\rangle$ yields the new state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ for that qubit. Next, applying the Hadamard gate to that qubit a second time yields:

$$\begin{aligned} &\tfrac{1}{\sqrt{2}}\left(\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle) + \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \\ = {}& \tfrac{1}{2}(|0\rangle \cancel{+|1\rangle} + |0\rangle \cancel{-|1\rangle}) = |0\rangle \end{aligned}$$

which when measured always yields 0. Here, interference is the phenomenon that the branches $|1\rangle$ and $-|1\rangle$ with opposite phase mathematically cancel, leaving only the branch $|0\rangle$.

In a quantum algorithm such as integer factorization [Shor 1997], interference is essential to efficiently pruning down a large search space and achieving computational advantage.

Disruptive Measurement. Given the qubit $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ after performing the first Hadamard gate, suppose we were to measure it before performing the second Hadamard gate.

[1] For ease of understanding, the definition of measurement given here is for a projective measurement in the computational basis. Nevertheless, all results in this work hold equally on the more general definitions of measurement, which can be realized using only unitary operations and projective measurements in the computational basis.

Upon measurement, the qubit collapses from superposition to an equal probabilistic mixture of $|0\rangle$ and $|1\rangle$. When we then apply the second Hadamard gate on this mixture, we obtain not $|0\rangle$ but rather a mixture of $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$ and $\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$. Now, no interference occurs — if we measure this new mixture, we observe an outcome of 0 or 1 with equal probability rather than 0 with certainty.

Disruptive Entanglement. A key building block for the results in this paper is the fact that the presence of entanglement between the primary state of a computation and an auxiliary or temporary value can cause quantum interference to not occur and quantum advantage to be lost.

To demonstrate, let us start with the same qubit $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ from above, and then execute the two-qubit CNOT gate on this qubit alongside a new second qubit initialized to $|0\rangle$:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle \xmapsto{\texttt{CNOT}} \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

The result is the entangled Bell state. If we were to discard the second qubit, which is equivalent to measuring it, then the first qubit would also collapse from superposition and fail to exhibit interference, as above. A key fact is that even if we refuse to discard or measure the second qubit, and simply apply the Hadamard gate again to the first qubit, interference still fails to occur:[2]

$$\begin{aligned}&\tfrac{1}{\sqrt{2}}\left(\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle + \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes |1\rangle\right)\\ &= \tfrac{1}{2}(|00\rangle + |10\rangle + |01\rangle - |11\rangle) \neq |0\rangle \otimes |\psi\rangle \text{ for any } |\psi\rangle\end{aligned}$$

Unlike $|1\rangle$ and $-|1\rangle$, the branches $|10\rangle$ and $-|11\rangle$ do not interfere, meaning that measuring the first qubit of the final state yields 0 or 1 with equal probability rather than 0 with certainty.

A quantum computation subject to disruptive entanglement degrades to a classical probabilistic computation, which is commonly understood to result in a loss of quantum advantage [Nielsen and Chuang 2010, Section 3.2.5]. In an algorithm such as Shor [1997], disrupting interference via entanglement as above causes a wrong answer to be produced or quantum advantage to be lost.

In the example, a correct way to recover the qubit $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ with its superposition intact from the entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is to execute the inverse of the CNOT gate, eliminating the undesired entanglement so as to recover the separable state of $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$ once again.

3 FAILURE OF CONDITIONAL JUMP IN SUPERPOSITION

In this section, we illustrate the challenges in programming with control flow that depends on data in quantum superposition, and reveal that standard techniques for lifting classical to quantum computation produce incorrect outputs when applied to abstractions for control flow.

Running Example. Suppose we must implement a program P that, given two machine integer variables x and y, updates their values according to the following transformation:

$$|\mathsf{x}, \mathsf{y}\rangle \overset{P}{\mapsto} \begin{cases} |\mathsf{x}, \mathsf{y} + 1\rangle & \text{if } \mathsf{x} = 0 \\ |\mathsf{x} + 1, \mathsf{y}\rangle & \text{if } \mathsf{x} \neq 0 \end{cases} \tag{1}$$

where $|\mathsf{x}:0, \mathsf{y}:3\rangle$ denotes a state in which x is 0 and y is 3. For example, given input $|\mathsf{x}:0, \mathsf{y}:3\rangle$, the program should yield output $|\mathsf{x}:0, \mathsf{y}:4\rangle$, and given $|\mathsf{x}:3, \mathsf{y}:0\rangle$, it should yield $|\mathsf{x}:4, \mathsf{y}:0\rangle$. We assume that arithmetic operations do not overflow in this example and address overflow in Appendix A.

Basic operations interleaving arithmetic and control flow such as this example are essential to algorithms for simulation [Babbush et al. 2018] and factoring [Proos and Zalka 2003; Shor 1997] and more generally illustrate the use of control flow in the algorithms we will present in Section 6.

[2]Familiar readers may see this case as the deferred measurement principle [Nielsen and Chuang 2010].

3.1 Classical Implementation with Conditional Jumps

On a classical computer, it would be straightforward to realize Equation (1) as a transformation of classical data. In Figure 1, we depict a typical implementation of this specification as a classical assembly program that relies on conditional jump instructions.

```
1      jnz l1 x  ; if x != 0, goto l1
2      add y $1  ; add 1 to y
3      jmp l2    ; goto l2
4  l1: add x $1  ; add 1 to x
5      jmp l2    ; goto l2
6  l2: nop       ; no-op
```

Fig. 1. Classical assembly implementing Equation (1).

In this assembly program, the machine state contains three registers: x, y, and the program counter pc. Given the two example initial states of x and y from above, the machine executes the program in Figure 1 by evolving the state according to the following two execution traces:

$$|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:1\rangle \xmapsto{\texttt{jnz l1 x}} |\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:2\rangle \xmapsto{\texttt{add y \$1}} |\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:3\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:6\rangle \quad (2)$$

$$|\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:1\rangle \xmapsto{\texttt{jnz l1 x}} |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:4\rangle \xmapsto{\texttt{add x \$1}} |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:5\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:6\rangle \quad (3)$$

where the first trace is the $\mathsf{x} = 0$ branch of Equation (1), and the second is the $\mathsf{x} \neq 0$ branch.

3.2 Superposition of Program Executions

On a quantum computer, we require a program P that manipulates x and y as data that exist in quantum superposition. Given a superposition of the two input states of Equations (2) and (3), the program must produce the corresponding superposition of their output states:

$$\tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3\rangle - |\mathsf{x}:3,\mathsf{y}:0\rangle) \overset{P}{\mapsto} \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4\rangle - |\mathsf{x}:4,\mathsf{y}:0\rangle) \quad (4)$$

As a contrast, it would be incorrect for the machine to simply measure x and branch on the outcome. Doing so collapses superposition,[3] meaning the program produces the output:

$$\begin{cases} |\mathsf{x}:0,\mathsf{y}:4\rangle & \text{with probability } \tfrac{1}{2} \\ |\mathsf{x}:4,\mathsf{y}:0\rangle & \text{with probability } \tfrac{1}{2} \end{cases} \quad (5)$$

If conditional branching collapsed the state as above, it would prevent a quantum algorithm such as Ambainis [2004]; Babbush et al. [2018]; Shor [1997] from leveraging interference (Section 2) in order to obtain computational advantage. Specifically, interference cannot possibly occur after the phase stored by the minus sign in Equation (4) is lost upon collapse.

Program Superposition. As an alternative to measuring the data, we consider the possibility of a quantum instruction set architecture analogous to the classical one in Section 3.1 in which the control flow of the program, as embodied by the program counter, may depend on the value of data.

Such a machine is specified as follows. Its state contains quantum registers x, y, and pc, and its state transition function lifts the machine in Section 3.1 to superposition, taking:

$$\textstyle\sum_i \gamma_i\, |\mathsf{x}_i,\mathsf{y}_i,\mathsf{pc}_i\rangle \mapsto \sum_i \gamma_i\, |\mathsf{x}'_i,\mathsf{y}'_i,\mathsf{pc}'_i\rangle$$

whenever the classical machine of Section 3.1 would step each constituent $|\mathsf{x}_i,\mathsf{y}_i,\mathsf{pc}_i\rangle \mapsto |\mathsf{x}'_i,\mathsf{y}'_i,\mathsf{pc}'_i\rangle$. Mathematically, this operator is *linear* over the quantum state of the machine.

[3]As a technical note, measuring x in this example also collapses y because x and y are entangled (Section 2), but that fact is not crucial, as the output would be incorrect even if x alone collapsed.

Running Example. One may envision that on this machine, we could directly execute the program in Figure 1 to manipulate x and y while preserving their superposition.

Given the superposition input from Equation (4), this machine sets the initial pc to 1. It then evolves the state according to a superposition of the traces in Equations (2) and (3):

$$\begin{aligned}
& \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:1\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:1\rangle) \\
\xmapsto{\texttt{jnz l1 x} + \texttt{jnz l1 x}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:2\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:4\rangle) \\
\xmapsto{\texttt{add y \$1} + \texttt{add x \$1}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:3\rangle - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:5\rangle) \\
\xmapsto{\texttt{jmp l2} + \texttt{jmp l2}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:6\rangle - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:6\rangle) \\
=\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4\rangle - |\mathsf{x}:4,\mathsf{y}:0\rangle) \otimes \cancel{|\mathsf{pc}:6\rangle}
\end{aligned} \tag{6}$$

In this trace, each $\mapsto$ represents the execution of a superposition of instructions. For example, the first instance of $\mapsto$ executes a superposition of **jnz** l1 x and itself, the second executes a superposition of **add** y $1 and **add** x $1, and the third a superposition of **jmp** l2 and itself.

At the end, the machine may discard the value of pc, leaving behind only the desired output $\frac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4\rangle - |\mathsf{x}:4,\mathsf{y}:0\rangle)$ as specified by the right-hand side of Equation (4).

Physical Realizability. However, the ideal semantics in Equation (6) is not physically realizable on a quantum computer. Physical principles dictate that a transformation over quantum states must take any physically realizable input state to a physically realizable output state. Formally, it must *preserve norms* of states. By contrast, given certain physically realizable states over x, y, and pc, the state transition function described above can produce a physically unrealizable state:

$$\begin{aligned}
& \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:3\rangle - |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:5\rangle) \\
\xmapsto{\texttt{jmp l2} + \texttt{jmp l2}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6\rangle - |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6\rangle) = 0
\end{aligned}$$

Here, the output has norm 0, a physically impossible outcome. The reason is that the state transition in Section 3.1 is not injective — it maps two distinct inputs to the same output:

$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:3\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6\rangle \tag{7}$$

$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:5\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6\rangle \tag{8}$$

By contrast, for an operator over quantum states to be both linear and norm-preserving, it must be unitary (Section 2), meaning it has an inverse and is injective by definition.

3.3 Landauer Embedding and Disruptive Entanglement

There exists a standard technique to compute a non-injective function inside a quantum computation, known as Landauer embedding [Landauer 1961]. We show, however, that using this technique to implement control flow leads a quantum algorithm to produce incorrect outputs.

Definition 3.1 (Landauer Embedding). Given a non-injective function f, one may *embed* it into an injective function $F(s) = (s, f(s))$ that also returns a copy of the input. From $F(s)$, one extracts the embedded value of $f(s)$ by discarding s. This process can be iterated as necessary.

History. One may attempt to apply Landauer embedding to the semantics of conditional jump by maintaining a *history* of program counters in memory, into which each executed step is written. We denote the current program counter by pc_0 and the history after t time steps have elapsed by $\mathsf{pc}_1, \mathsf{pc}_2, \ldots, \mathsf{pc}_t$, such that pc_1 is the value from the immediately previous time step, and so on.

Physical Realizability. This construction is realizable as a unitary operator. One can see that under Landauer embedding, the analogues of the states from Equations (7) and (8) evolve differently:

$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}_0:3,\ldots,\mathsf{pc}_t:1\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}_0:6,\mathsf{pc}_1:3,\ldots,\mathsf{pc}_{t+1}:1\rangle$$
$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}_0:5,\ldots,\mathsf{pc}_t:1\rangle \xmapsto{\texttt{jmp l2}} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}_0:6,\mathsf{pc}_1:5,\ldots,\mathsf{pc}_{t+1}:1\rangle$$

Disruptive Entanglement. Landauer embedding, however, introduces a new problem that causes the computation to produce incorrect results. Consider the new execution trace of the program in Figure 1, which updates the old trace in Equation (6) to add a history of program counters:

$$\begin{aligned}
& \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}_0:1\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}_0:1\rangle) \\
\xmapsto{\texttt{jnz l1 x} + \texttt{jnz l1 x}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}_0:2,\mathsf{pc}_1:1\rangle \\
& - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}_0:4,\mathsf{pc}_1:1\rangle) \\
\xmapsto{\texttt{add y \$1} + \texttt{add x \$1}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}_0:3,\mathsf{pc}_1:2,\mathsf{pc}_2:1\rangle \\
& - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}_0:5,\mathsf{pc}_1:4,\mathsf{pc}_2:1\rangle) \\
\xmapsto{\texttt{jmp l2} + \texttt{jmp l2}}\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}_0:6,\mathsf{pc}_1:3,\mathsf{pc}_2:2,\mathsf{pc}_3:1\rangle \\
& - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}_0:6,\mathsf{pc}_1:5,\mathsf{pc}_2:4,\mathsf{pc}_3:1\rangle) \\
\neq\; & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4\rangle - |\mathsf{x}:4,\mathsf{y}:0\rangle) \otimes |\psi\rangle \text{ for any } |\psi\rangle
\end{aligned} \tag{9}$$

Like in Equation (6), each $\mapsto$ represents the execution of a superposition of instructions. The difference is that the final state is now entangled, meaning that discarding or, equivalently, measuring one component collapses the superposition of the other (Section 2). Thus, the machine cannot discard the history without destroying the superposition of x and y. Discarding pc_1 yields:

$$\begin{cases} |\mathsf{x}:0,\mathsf{y}:4,\cancel{\mathsf{pc}_0:6,\mathsf{pc}_2:2,\mathsf{pc}_3:1}\rangle & \text{w.p. } \tfrac{1}{2} \\ |\mathsf{x}:4,\mathsf{y}:0,\cancel{\mathsf{pc}_0:6,\mathsf{pc}_2:4,\mathsf{pc}_3:1}\rangle & \text{w.p. } \tfrac{1}{2} \end{cases}$$

which is the same incorrect outcome as having measured x to begin with, as in Equation (5).

Moreover, as established in Section 2, simply refusing to measure or discard the history is not an admissible workaround — using part of an entangled state in place of the right side of Equation (4) still leads a quantum algorithm to produce a wrong answer or lose computational advantage.

3.4 No Recovery from Disruptive Entanglement

From this point, one conceivable avenue to recover from entanglement is to use *uncomputation* [Bennett 1973], the standard technique to erase temporary data in quantum computation.

Definition 3.2 (Uncomputation). Suppose we have non-injective f and g, and seek an injective H such that $H(s) = (s, g(f(s)))$. Denote by F and G the Landauer embeddings of f and g respectively.

Composing F and G produces $H(s)$ alongside a temporary value $f(s)$, which we seek to erase. Since F is injective, we execute its partial inverse, denoted $F^\dagger$, thereby *uncomputing* $f(s)$:

$$s \overset{F}{\mapsto} (s, f(s)) \xmapsto{\mathrm{id}\otimes G} (s, \cancel{f(s)}, g(f(s))) \xmapsto{F^\dagger\otimes\mathrm{id}} (s, g(f(s)))$$

Iterating this process in a computation enables all values, except for the initial s, to be erased from the machine state. Importantly, the initial s necessarily persists in the state.

Pitfall. One may hope to use uncomputation to erase all but the initial program counter from the history in Equation (9). The problem is that uncomputing $f(s)$ both requires and leaves behind the value s, and when f is the machine state transition function, the value of s must store not only the value of pc but also that of all data registers on which a jump may depend.

```
l0: jnz l2 x   ; if x != 0, goto l2
    add y $1   ; add 1 to y
l1: jmp l3     ; jump to l3
l2: rjmp l0    ; come from l0
    add x $1   ; add 1 to x
l3: rjz l1 x   ; if x = 0, come from l1
    nop        ; no-op
```

Fig. 2. Assembly program for the quantum control machine implementing Equation (1) using reverse jumps.

$$
\begin{aligned}
& |\mathsf{x}:3, \mathsf{y}:0, \mathsf{pc}:1, \mathsf{br}:1\rangle \\
\xrightarrow{\texttt{jnz l2 x}}\ & |\mathsf{x}:3, \mathsf{y}:0, \mathsf{pc}:4, \mathsf{br}:3\rangle \\
\xrightarrow{\texttt{rjmp l0}}\ & |\mathsf{x}:3, \mathsf{y}:0, \mathsf{pc}:5, \mathsf{br}:1\rangle \\
\xrightarrow{\texttt{add x \$1}}\ & |\mathsf{x}:4, \mathsf{y}:0, \mathsf{pc}:6, \mathsf{br}:1\rangle \\
\xrightarrow{\texttt{rjz l1 x}}\ & |\mathsf{x}:4, \mathsf{y}:0, \mathsf{pc}:7, \mathsf{br}:1\rangle
\end{aligned}
$$

Fig. 3. Execution trace of the program in Figure 2, given the input state $|\mathsf{x}:3, \mathsf{y}:0\rangle$ from Equation (3).

One may suggest modifying the history to store copies of all data registers, but even when doing so is possible,[4] it still does not resolve the problem of entanglement. The initial values of data registers x and y, denoted x_{in} and y_{in} respectively, are now stored in the history and persist in the state that is left behind after all possible uncomputation, which remains entangled:

$$
\begin{aligned}
&\tfrac{1}{\sqrt{2}}(|\mathsf{x}:0, \mathsf{y}:4, \mathsf{x}_{\text{in}}:0, \mathsf{y}_{\text{in}}:3\rangle - |\mathsf{x}:4, \mathsf{y}:0, \mathsf{x}_{\text{in}}:3, \mathsf{y}_{\text{in}}:0\rangle) \\
\neq\ &\tfrac{1}{\sqrt{2}}(|\mathsf{x}:0, \mathsf{y}:4\rangle - |\mathsf{x}:4, \mathsf{y}:0\rangle) \otimes |\psi\rangle \text{ for any } |\psi\rangle
\end{aligned}
$$

No-Embedding Theorem. One may hope that the presence of entanglement is caused by the specific encoding of the history in Landauer embedding, and can be avoided by storing less information.

In Section 4.1, we prove otherwise. By generalizing the arguments above, we show that any attempt to implement an instruction set with a non-injective state transition semantics on a quantum computer necessarily suffers from the problem of disruptive entanglement. This theorem implies that there is fundamentally no way to lift the conventional conditional jump to superposition and guarantee that the output of the program is correct for use by a quantum algorithm.

3.5 Quantum Control Machine

Faced with the impossibility of lifting conventional conditional jumps to superposition, we present a new instruction set architecture called the *quantum control machine*. The key idea of the approach is to start over with alternative control flow primitives, originally introduced by classical reversible architectures [Axelsen et al. 2007; Thomsen et al. 2012], whose transition semantics are inherently injective without the need for Landauer embedding.

Branch Control. Instead of a history, this machine tracks the difference in pc from the previous instruction in a new *branch control* register [Axelsen et al. 2007; Thomsen et al. 2012] denoted br. The design of the machine redefines the semantics of each jump instruction to manipulate br, and adds its value to pc after each instruction. Specifically, a **jmp** updates br, and the value of br persists until it is changed again. When br is 1, the program executes step by step, and when br is greater than 1, it continually jumps forward. To resume single-step execution, the program invokes designated instructions, known as *reverse jumps*, that reset br back to 1.

Running Example. In Figure 2, we present an implementation of Equation (1) on the quantum control machine. The notable differences from Figure 1 are the new reverse jump instructions **rjmp** and **rjz**. To illustrate how these instructions work, in Figure 3 we depict the execution trace of the program in Figure 2 given the input state $|\mathsf{x}:3, \mathsf{y}:0\rangle$ from Equation (3).

First, **jnz** `l2 x` increases br from 1 to 3. The value of pc advances by 3 from 1 to 4, and br remains 3. If nothing else is done, on the next iteration pc would jump again from 4 to 7.

[4]As a technical note, copying data registers is not possible in general under the no-cloning theorem (Section 2).

However, the next instruction is a reverse jump, **rjmp** `l0`. The effect of this instruction is precisely the opposite of the jump from `l0` — it decreases br from 3 to 1. The rest of the program then executes step by step, where **rjz** `l1 x` has no effect when x is not 0.

Physical Realizability. This transition function is injective and unitary, even without employing Landauer embedding. The reason is that jumps from different lines to the same line must leave behind distinct values of br, as can be seen using the examples from Equations (7) and (8):

$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:3,\mathsf{br}:1\rangle \xmapsto{\text{jump by }+3} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6,\mathsf{br}:3\rangle$$
$$|\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:5,\mathsf{br}:1\rangle \xmapsto{\text{jump by }+1} |\mathsf{x}:0,\mathsf{y}:0,\mathsf{pc}:6,\mathsf{br}:1\rangle$$

Furthermore, no matter what instruction comes on the next line, it is impossible for the two states above to both transition to exactly the same state:

- an unconditional **jmp** (or **rjmp**) adds the same value to br in both states, meaning the values of br cannot both become equal when they are initially different, and
- a conditional **jnz** (or **rjnz** or jump with any other condition) sees the same values for data registers x and y in both states, meaning it always modifies br the same way in both states.

Disruptive Entanglement Avoided. We now demonstrate how the program in Figure 2 avoids the entanglement problem and produces output data that is correct for use by a quantum algorithm. When given the superposition input of Equation (4), this program executes as follows:

$$\begin{aligned}
& \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:1,\mathsf{br}:1\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:1,\mathsf{br}:1\rangle)\\
\xmapsto{\textbf{jnz}\ \mathtt{l2\ x}\ +\ \textbf{jnz}\ \mathtt{l2\ x}}\ & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:3,\mathsf{pc}:2,\mathsf{br}:1\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:4,\mathsf{br}:3\rangle)\\
\xmapsto{\textbf{add}\ \mathtt{y\ \$1}\ +\ \textbf{rjmp}\ \mathtt{l0}}\ & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:3,\mathsf{br}:1\rangle - |\mathsf{x}:3,\mathsf{y}:0,\mathsf{pc}:5,\mathsf{br}:1\rangle)\\
\xmapsto{\textbf{jmp}\ \mathtt{l3}\ +\ \textbf{add}\ \mathtt{x\ \$1}}\ & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:6,\mathsf{br}:3\rangle - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:6,\mathsf{br}:1\rangle)\\
\xmapsto{\textbf{rjz}\ \mathtt{l1\ x}\ +\ \textbf{rjz}\ \mathtt{l1\ x}}\ & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:7,\mathsf{br}:1\rangle - |\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:7,\mathsf{br}:1\rangle)\\
=\ & \tfrac{1}{\sqrt{2}}(|\mathsf{x}:0,\mathsf{y}:4\rangle - |\mathsf{x}:4,\mathsf{y}:0\rangle) \otimes \cancel{|\mathsf{pc}:7,\mathsf{br}:1\rangle}
\end{aligned}$$

Just as in Equation (6), each $\mapsto$ in the trace denotes the execution of a superposition of instructions. In particular, the final $\mapsto$ executes a superposition of **rjz** `l1 x` and itself. On both branches of the superposition, this instruction causes pc to become 7 and br to become 1, but by different means. On the branch where x is 0, br decreases from 3 to 1, and on the branch where x is 3, br starts as 1 and does not change. Note that the transition function is injective thanks to the presence of distinct values of x and y, as $|\mathsf{x}:0,\mathsf{y}:4,\mathsf{pc}:7,\mathsf{br}:1\rangle$ and $|\mathsf{x}:4,\mathsf{y}:0,\mathsf{pc}:7,\mathsf{br}:1\rangle$ are distinct states.

At the end of this execution, the final machine state is separable, i.e. not entangled, and discarding pc and br leaves behind the state of x and y with superposition intact. Thus, the program produces the output that is specified by Equation (4) and correct for use by a quantum algorithm.

Synchronization. The example illustrates how a program avoids the entanglement problem by ensuring that at the end of execution, pc and br are equal again across all branches of the state superposition. To do so, the program must possess an appropriate reverse jump instruction at the target of each forward jump instruction according to the structure of its control flow.

In Section 4.2, we define the class of *synchronized* programs, which include Figure 2, that satisfy this condition and thereby guarantee that their output is correct for use by a quantum algorithm. Moreover, whereas the analysis above depicts detailed formal reasoning to show that a program is synchronized, in Section 6 we demonstrate how in practice a developer may leverage the structure of control flow within a program to more easily identify the program as synchronized.

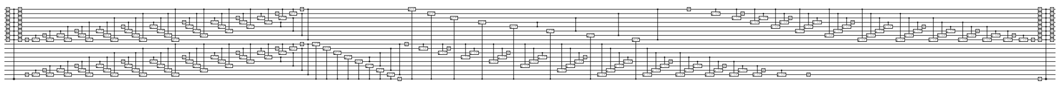

Fig. 4. Quantum circuit for Equation (1), restricted to 8-bit integers, as implemented by a Q# program.

Existing Methods. In principle, Equation (1) may be implemented using an existing quantum programming language such as Quipper [Green et al. 2013], QWire [Paykin et al. 2017], or Q# [Svore et al. 2018]. These languages expose the abstractions of qubits and bit-controlled logic gates that enable the developer to build a representation of the program as a quantum circuit.

It can be challenging, however, to represent control flow in superposition directly via a quantum circuit of logic gates. When expressed as an explicit circuit, a conditional branch over a variable `x` corresponds to a sequence of logic gates controlled on individual qubits that realize the comparison with `x`. We illustrate this complexity in Appendix B, in which we present a 15-line Q# program that implements Equation (1) by instantiating the 229-gate circuit in Figure 4.

To alleviate the complexity of building circuits, researchers have developed higher-level languages such as Silq [Bichsel et al. 2020], which provides a quantum `if`-statement that branches on the value of a qubit in superposition. In Appendix B, we show that the Silq implementation of Equation (1) using quantum `if` also uses the `forget` statement, an unsafe operation whose correctness must be shown using reasoning outside the immediate automated capabilities of Silq's type system. The condition that `forget` must use for the Silq program to be correct is analogous to the condition that the reverse jump instruction must use for the program in Figure 2 to be synchronized.

Thus, by formally characterizing the properties of sound and realizable control flow on a quantum computer, our work generalizes the reasoning that type systems of existing quantum programming languages perform to enforce injectivity and synchronization.

Summary. In this section, we presented examples of programs that are important for quantum algorithms and easy to implement on a classical computer, yet are hard to implement on a quantum computer without the abstraction of control flow that can depend on data in superposition.

We illustrated why one cannot program a quantum computer using the conventional conditional jump instruction, and presented an alternative – reversible jumps and synchronization in the quantum control machine – that enables correct programming with control flow in superposition.

4 THEORETICAL LIMITS OF CONTROL FLOW IN SUPERPOSITION

In this section, we present the *no-embedding theorem*, a no-go theorem for the design of quantum programming languages that states that no programming abstraction with non-injective transition semantics, such as the conventional conditional jump and the λ-calculus, can be correctly realized in superposition. More generally, we present the necessary and sufficient conditions for the correct realizability of control flow in superposition. Alongside injectivity, we introduce *synchronization*, the property that the state of data is separable from the state of control flow in the final output of a computation, so that the output data can be correctly used by a quantum algorithm.

4.1 No-Embedding Theorem and Injectivity

We begin by defining a *transition system* [Hines 2008], a formalization of any classical model of computation as a set of computation states S and a partial transition function $T : S \rightharpoonup S$. For example, the Turing machine has S as its state-head-tape configurations and T as its state transition function, while the λ-calculus has S as the set of λ-terms and T as β-reduction.

Though S may be a countably infinite set in principle, any physical computer has a bounded state space in reality. In this section, we assume that all sets are finite and bounded by a function of the length of the longest computation that is physically feasible on the computer.

Superposition of Data. We lift the definition of transition system to operate over data in quantum superposition. Formally, we generalize the definitions of Section 2 from concrete bit strings to abstract states. Given a finite set of classical states X, we denote the corresponding Hilbert space of quantum states as $\mathcal{H}_X$, spanned by the basis $\{|x\rangle \mid x \in X\}$. Then, a transition system acts on the Hilbert space $\mathcal{H}_S$ via the linear mapping $\mathcal{T}$ that takes $|s\rangle \mapsto |s'\rangle$ whenever $T(s) = s'$ and takes $|s\rangle$ to an unspecified value if $T(s)$ is not defined.

Physical Realizability. Physically realizing a transition system that operates over quantum information requires constructing it as a unitary operator over an appropriate Hilbert space. When T is an injective function, $\mathcal{T}$ corresponds to a unitary operator over the Hilbert space $\mathcal{H}_S$.

By contrast, as established in Section 3, the transition T of the conventional conditional jump is not injective, meaning that it cannot be realized as a unitary operator over $\mathcal{H}_S$. In the non-injective case, the only alternative possibility is to embed T into a Hilbert space of higher dimension.

Example 4.1 (Landauer Embedding). Let the set L be of *histories* — lists of elements of S. Then, the larger Hilbert space $\mathcal{H}_S \otimes \mathcal{H}_L$ contains a unitary U that embeds the classical behavior of T on S.

Specifically, whenever $T(s) = s'$, we may define $U|s, \ell\rangle = |s', \ell'\rangle$ where $\ell' = \mathsf{append}(\ell, s)$. In the Landauer embedding, computation starts with the empty history $\ell_0 = []$, and discarding, i.e. measuring the final ℓ' enables a classical output $s' \in S$ to be extracted from the system.

More generally, the use of an auxiliary space makes it possible to embed the classical behavior of a transition system. The Landauer embedding is a specific instance of the following concept:

Definition 4.2 (Classical Embedding). A *classical embedding* for (S, T) is a triple $(\mathcal{H}_L, U, |\eta_0\rangle)$ where $\mathcal{H}_L$ is an auxiliary Hilbert space, $|\eta_0\rangle$ is a quantum state with norm 1 that is fixed in $\mathcal{H}_L$, and U is a unitary operator over $\mathcal{H}_S \otimes \mathcal{H}_L$ such that for any $s, s' \in S$ where $T(s) = s'$, we have $U|s, \eta_0\rangle = |s', \eta'\rangle$ for some $|\eta'\rangle \in \mathcal{H}_L$ with norm 1.

In the above definition, the fact that $|\eta_0\rangle$ must be fixed in $\mathcal{H}_L$ is essential to avoid the possibility of cheating by, for example, setting $|\eta_0\rangle = |s'\rangle$ given advanced knowledge of s.

Next, we generalize the above definition to account for not only how a transition system takes classical inputs to classical outputs, but also how in a quantum computation it should accept a superposition of input states and produce a superposition of corresponding output states:

Definition 4.3 (Quantum Embedding). A *quantum embedding* for (S, T) is a triple $(\mathcal{H}_L, U, |\eta_0\rangle)$ where $\mathcal{H}_L$ is an auxiliary Hilbert space, $|\eta_0\rangle$ is a quantum state with norm 1 that is fixed in $\mathcal{H}_L$, and U is a unitary over $\mathcal{H}_S \otimes \mathcal{H}_L$ such that for any $|\psi\rangle, |\psi'\rangle \in \mathcal{H}_S$ where $\mathcal{T}|\psi\rangle = |\psi'\rangle$, we have $U(|\psi\rangle \otimes |\eta_0\rangle) = |\psi'\rangle \otimes |\eta'\rangle$ for some $|\eta'\rangle \in \mathcal{H}_L$ with norm 1.

Critically, this definition ensures that the output state $|\psi'\rangle \otimes |\eta'\rangle$ is separable, and subsumes the classical embedding, in which the output state is classical and trivially separable. As established in Section 3.3, separability is necessary to guarantee that the output data exhibits correct quantum interference so that the quantum algorithm being implemented produces correct results.

However, while this separability requirement is essential for correctness of algorithms, it in fact precludes the existence of a quantum embedding for any non-injective transition system:

Theorem 4.4 (No-Embedding). *If the state transition function T is not injective, then no quantum embedding exists for the transition system (S, T).*

Proof. Assume that a quantum embedding $(\mathcal{H}_L, U, |\eta_0\rangle)$ exists for the transition system (S, T), where T is not injective. Let s_1 and s_2 in S with $s_1 \neq s_2$ be such that $T(s_1) = T(s_2) = s'$. Let $|\psi\rangle = \frac{1}{\sqrt{2}}(|s_1\rangle - |s_2\rangle)$. By linearity of $\mathcal{T}$, we have $\mathcal{T}|\psi\rangle = \frac{1}{\sqrt{2}}(\mathcal{T}|s_1\rangle - \mathcal{T}|s_2\rangle) = \frac{1}{\sqrt{2}}(|s'\rangle - |s'\rangle) = 0 =: |\psi'\rangle$. By Definition 4.3, we therefore have $U(|\psi\rangle \otimes |\eta_0\rangle) = |\psi'\rangle \otimes |\eta'\rangle = 0 \otimes |\eta'\rangle = 0$ for some $|\eta'\rangle \in \mathcal{H}_L$. This is a contradiction to the assumption that U is unitary and thus norm-preserving. □

In other words, even though one can embed the purely classical component of a non-injective transition system into a unitary operator, one cannot successfully realize the desired semantics of such a transition system over superpositions of data. There exists no general scheme that correctly lifts a programming abstraction with non-injective transition semantics into superposition.

An immediate corollary of the theorem is that a quantum λ-calculus featuring superpositions of λ-terms is not physically realizable, because β-reduction is not injective — the distinct terms $\lambda x.x$ and $(\lambda x.x)(\lambda x.x)$ reduce to the same term $\lambda x.x$. This result formalizes an informal claim of van Tonder [2004], and we give a detailed instantiation of the theorem to this case in Appendix C.

4.2 Synchronization

We next present *synchronization*, the property that the state of data is separable from the state of control flow in the final output of a computation. Together, injectivity and synchronization form a complete specification of the forms of control flow that are correctly realizable in superposition.

We consider transition systems in which T is injective and $S = C \times D$, where C and D denote the control state and data state of the model of computation respectively. By control state, we refer to the component of the machine state that is not an explicit output of the algorithm being implemented, and will be discarded at the end of computation to leave behind the data.

We define a transition system with $T : C \times D \to C \times D$ as having *control flow dependent on data* when there exist $c, c'_1, c'_2 \in C$ and $d_1, d'_1, d_2, d'_2 \in D$ such that $T(c, d_1) = (c'_1, d'_1)$ and $T(c, d_2) = (c'_2, d'_2)$ and $d_1 \neq d_2$ and $c'_1 \neq c'_2$. In other words, the evolution of the control state may depend on the value of data, and the function T does not factor into two independent transitions $C \to C$ and $C \times D \to D$.

Example 4.5. A classical register machine has a program counter pc and a set R of data registers, in which the semantics of an instruction is a function $\{\mathsf{pc}\} \times R \to \{\mathsf{pc}\} \times R$. Here, C is the state of pc and D is the state of R. Control flow dependent on data manifests in the conditional jump instruction, whose semantics updates pc based on the value of R.

Given a termination time and initial quantum states of control and data, a transition system executes a computation to produce a corresponding final quantum state over control and data:

Definition 4.6 (Final State). Given an initial control state $|\kappa_0\rangle \in \mathcal{H}_C$, a termination time $t \in \mathbb{N}$, and an input data state $|\delta_0\rangle \in \mathcal{H}_D$, the transition system (S, T) performs the unitary $\mathcal{T}$ (defined in Section 4.1) for t iterations on the state $|\kappa_0\rangle \otimes |\delta_0\rangle$ to produce the *final state* $|\psi\rangle \in \mathcal{H}_C \otimes \mathcal{H}_D$.

When control flow can depend on data, the final $|\psi\rangle$ can be an entangled state in general. However, as established in Section 3.3, the presence of entanglement would make the output data incorrect for use by a quantum algorithm. To prevent this issue, the property of synchronization specifies that in the final output, the state of data is always separable from the state of control flow:

Definition 4.7 (Synchronization). The transition system (S, T) is *synchronized* at initial control state $|\kappa_0\rangle$ and termination time t if there exists some final control state $|\kappa'\rangle \in \mathcal{H}_C$ such that for any input $|\delta_0\rangle \in \mathcal{H}_D$, there exists some output $|\delta'\rangle \in \mathcal{H}_D$ such that the final state of the machine after executing t instructions, as defined in Definition 4.6, is $|\psi\rangle = |\kappa'\rangle \otimes |\delta'\rangle$.

To show that a transition system is synchronized, by linearity it suffices to show that the final value of $|\kappa'\rangle$ is fixed across all values of $|\delta_0\rangle$ in computational basis, i.e. classical input data alone.

Terminologically, we refer to a program as synchronized when the transition system and initial control state corresponding to the program are synchronized. This term is agnostic to whether the model of computation hardcodes its program in the transition function T, as does the standard Turing machine, or whether it accepts its program in $|\kappa_0\rangle$, as does the universal Turing machine.

A synchronized transition system expresses a computation over the data state that is a unitary operator corresponding to a quantum circuit without measurement. The converse also holds — if a transition system is not synchronized, then it does not express a unitary computation over the data.

Theorem 4.8 (Soundness). *If a transition system is injective and synchronized, then given any input data $|\delta_0\rangle$, it produces a final state after t instructions in which the output data $|\delta'\rangle$ is separable from the control state. Furthermore, its mapping from input data to output data is a unitary operator.*

Proof. If (S, T) is synchronized, then given any $|\delta_0\rangle = \sum_i \gamma_i |\delta_i\rangle \in \mathcal{H}_D$ such that $\mathcal{T}^t(|\kappa_0\rangle \otimes |\delta_i\rangle) = |\kappa'\rangle \otimes |\delta'_i\rangle$, we have $|\psi\rangle = \mathcal{T}^t(|\kappa_0\rangle \otimes \sum_i \gamma_i |\delta_i\rangle) = \sum_i \gamma_i(|\kappa'\rangle \otimes |\delta'_i\rangle) = |\kappa'\rangle \otimes |\delta'\rangle$ where $|\delta'\rangle = \sum_i \gamma_i |\delta'_i\rangle$. The mapping $|\delta_0\rangle = \sum_i \gamma_i |\delta_i\rangle \mapsto |\delta'\rangle = \sum_i \gamma_i |\delta'_i\rangle$ is linear and norm-preserving as $\mathcal{T}$ is unitary. □

Theorem 4.9 (Completeness). *If a transition system is not synchronized, then either there exists some input $|\delta_0\rangle$ for which it produces a final state $|\psi\rangle$ after t instructions in which data and control are entangled, or its mapping from input data to output data is not injective and hence not unitary.*

Proof. If (S, T) is not synchronized, then there exist $|\delta_A\rangle, |\delta_B\rangle \in \mathcal{H}_D$ with unit norm such that $|\delta_A\rangle \neq |\delta_B\rangle$, $\mathcal{T}^t(|\kappa_0\rangle \otimes |\delta_A\rangle) = |\kappa_A\rangle \otimes |\delta'_A\rangle$, $\mathcal{T}^t(|\kappa_0\rangle \otimes |\delta_B\rangle) = |\kappa_B\rangle \otimes |\delta'_B\rangle$, and $|\kappa_A\rangle \neq |\kappa_B\rangle$. Then, $\mathcal{T}^t(|\kappa_0\rangle \otimes \frac{1}{\sqrt{2}}(|\delta_A\rangle + |\delta_B\rangle)) = \frac{1}{\sqrt{2}}(|\kappa_A\rangle \otimes |\delta'_A\rangle + |\kappa_B\rangle \otimes |\delta'_B\rangle)$, which is entangled unless $|\delta'_A\rangle = |\delta'_B\rangle$. □

Together, injectivity and synchronization provide a complete specification for the forms of control flow in superposition that are correctly realizable on a quantum computer — the programming abstractions must have injective semantics, and the program must be synchronized.

5 QUANTUM CONTROL MACHINE

In this section, we present the *quantum control machine*, an instruction set architecture for quantum programming with control flow in superposition. This architecture provides for the first time both a program counter in superposition and a sound means of manipulating the program counter.

To satisfy the specification for correctly realizable control flow in superposition, the machine uses variants of conditional jump introduced by classical reversible architectures [Axelsen et al. 2007; Thomsen et al. 2012] that possess inherently injective semantics and can be used to build synchronized programs. On top of that concept, the quantum control machine adds the ability to execute unitary operators that create and manipulate superpositions of data such as the Hadamard gate, and provides a sound representation of a program counter that exists in superposition.

5.1 Architectural Overview

The quantum control machine operates over word-sized quantum registers. Let k, the system word size, be large enough for a word to represent a machine integer or an encoding of an instruction.

Program Encoding. A *program* is a sequence of instructions of length ℓ where $0 < \log_2 \ell < k$. We denote by ι_i the word-size encoding of the ith instruction of the program.

As is the case for a classical Turing machine, a specific instance of the quantum control machine executes a specific fixed program. The machine obtains access to the instruction sequence through a unitary operator P, defined such that given integer $1 \leq i \leq \ell$, we have $\mathsf{P}|i, 0\rangle = |i, \iota_i\rangle$.

In principle, the operator P and its inverse may be physically realized via any technology to make classical data available for read-only use in superposition, such as those in the works of Babbush et al. [2018]; Berry et al. [2019]; Giovannetti et al. [2008]; Low et al. [2018].

Table 1. Core instruction set for the quantum control machine. The notation x_i denotes the ith bit of x and $x_{\backslash i}$ denotes x with ith bit set to zero. The notation p denotes a signed immediate offset and the notation U denotes a built-in unitary operator. The expression $[y = 0]$ evaluates to 1 if $y = 0$ or 0 otherwise.

	Instruction	Semantics
No-op	`nop`	(identity)
Unitary	`u` U `ra`	$\lvert x\rangle_{ra} \mapsto (U\lvert x\rangle)_{ra} \quad U \in \{\mathsf{H}, \mathsf{NOT}\}$
Swap	`swap` `ra` `rb`	$\lvert x\rangle_{ra}\lvert y\rangle_{rb} \mapsto \lvert y\rangle_{ra}\lvert x\rangle_{rb}$
Get Bit	`get` `ra` `rb` `rc`	$\lvert i\rangle_{ra}\lvert 0\rangle_{rb}\lvert x\rangle_{rc} \mapsto \lvert i\rangle_{ra}\lvert x_i\rangle_{rb}\lvert x_{\backslash i}\rangle_{rc}$
Add	`add` `ra` `rb`	$\lvert x\rangle_{ra}\lvert y\rangle_{rb} \mapsto \lvert x+y\rangle_{ra}\lvert y\rangle_{rb}$
	`add` `ra` `ra`	$\lvert x\rangle_{ra} \mapsto \lvert x\times 2\rangle_{ra}$
Multiply	`mul` `ra` `rb`	$\lvert x\rangle_{ra}\lvert y\rangle_{rb} \mapsto \lvert x\times y\rangle_{ra}\lvert y\rangle_{rb} \quad y \neq 0$
	`mul` `ra` `ra`	$\lvert x\rangle_{ra} \mapsto \lvert x^2\rangle_{ra}$
Jump	`jmp` p	$\lvert x\rangle_{\mathsf{br}} \mapsto \lvert x+p\rangle_{\mathsf{br}}$
Conditional Jump	`jz` p `ra`	$\lvert x\rangle_{\mathsf{br}}\lvert y\rangle_{ra} \mapsto \lvert x+p\times[y=0]\rangle_{\mathsf{br}}\lvert y\rangle_{ra}$
Indirect Jump	`jmp*` `ra`	$\lvert x\rangle_{\mathsf{br}}\lvert y\rangle_{ra} \mapsto \lvert x+y\rangle_{\mathsf{br}}\lvert y\rangle_{ra}$

Data Registers. The machine provides a finite number of quantum registers – binary-encoded qubit arrays of word size k – that are indexed and addressable by classical names. The n data registers are named `r1`, `r2`, ..., `r`n and are initialized to 0.

Control Unit. The machine contains three control registers: the program counter `pc`, initially 0; the branch control register `br`, signed and initially 1; and the instruction register `in`, initially 0.

5.2 Machine Execution

Execution occurs in cycles consisting of *fetch*, *execute*, and *retire* stages. Each of the three stages has as its mathematical semantics a unitary operator over the Hilbert space of machine states:

- The fetch stage loads the next instruction to execute. First, it adds the value of `br` to `pc`, using a unitary circuit construction for arithmetic [Cheng and Tseng 2002; Cho et al. 2020; Draper 2000; Islam et al. 2009; Rines and Chuang 2018]. It then executes the operator P defined above on $\lvert\mathsf{pc}, \mathsf{in}\rangle$ to load ι_{pc}, the instruction at the new `pc`, into register `in`.
  As is the case for a classical architecture, the behavior is implementation-defined when the program counter does not point to a valid instruction, i.e. when $\mathsf{pc} = 0$ or $\mathsf{pc} > \ell$.
- The execute stage updates the registers `r1` through `r`n and `br` according to the semantics of the instruction in the `in` register, which is specified in Section 5.3.
- The retire stage executes the inverse of P on $\lvert\mathsf{pc}, \mathsf{in}\rangle$, so that after a cycle, `in` has value 0.

5.3 Instruction Set

In Table 1, we present the core instruction set and the semantics of each instruction, which updates the data registers `r1`, ..., `r`n and the control register `br`. The control registers `pc` and `in` are updated only by the fetch and retire stages, and not explicitly by any instruction.

No-op. In Table 1, the first instruction `nop` is a no-op that simply causes the machine to advance to the next cycle, and its semantics is the identity operator over the machine state.

Unitary Gates. The next instruction `u` executes a unitary quantum logic gate, specified by its encoded name U from a fixed set of built-in primitive gates, on the specified register `ra`.

Any unitary operator can be approximated to any degree of precision using single-qubit gates that can be controlled by arbitrarily many qubits [Kitaev 1997]. Among the gates that are sufficient for this purpose, the choice of gate set is arbitrary in principle.[5] In this work, we use the single-qubit gate set consisting of {`H`, `NOT`} — Hadamard and bit-flip gates on the zeroth qubit of the register, whose controlled variants are sufficient to approximate any unitary operator [Shi 2003].

We note that in the quantum control machine, because the choice of instruction may be controlled on quantum data by the program counter, multi-qubit gates such as `CNOT` need not be primitive, and can instead be implemented through control flow instructions.

Data and Arithmetic. The `swap` instruction swaps the contents of registers `ra` and `rb`. The `get` instruction extracts the ith qubit of `rc` into `rb` and leaves 0 in place of that qubit in `rc`, where i is a quantum integer stored in the register `ra`. In principle, the `get` instruction may be realized in hardware via the quantum analogue of random-access memory, as presented by Arunachalam et al. [2015]; Giovannetti et al. [2008]; Matteo et al. [2020]; Paler et al. [2020]. For a detailed description of the arithmetic instructions `add` and `mul` for addition and multiplication and the behavior of these operations under integer overflow, please see Appendix A.

Control Flow. The jump instructions of the machine are from classical reversible architectures [Axelsen et al. 2007; Thomsen et al. 2012]. The unconditional jump instruction `jmp` adds the signed immediate value p to the branch control register `br`, producing a relative jump. The magnitude of the jump stored by `br` persists across cycles until reset by another jump instruction.

The conditional jump instruction `jz` adds p to `br` if the value of `ra` is 0, and has no effect on `br` otherwise. The indirect jump instruction `jmp*` adds the value of `ra` to `br`.

Derived Instructions. Each instruction in the table also has a corresponding reverse instruction prefixed by the character `r`, whose semantics inverts input and output. For example, the instruction `ru` executes the operator $U^\dagger$ on `ra`. The exceptions are `nop` and `swap`, which are self-inverse.

In a quantum computation, reverse instructions are used to uncompute (Section 3.4) a register after it is no longer useful, by reversing the sequence of operations that produced its value and restoring it to 0. Reverse instructions must be used for this purpose as no instruction can in general erase a register from an arbitrary value to 0, which is not mathematically a unitary operator.

The core instruction set is readily extended with other derived instructions. For example, we use `jnz` to denote a jump-if-not-zero instruction, the opposite of `jz`. In this work, we also permit immediates in place of registers and named labels in place of offsets. Named labels may be translated to a signed offset relative to a `jmp` instruction that is negated for a `rjmp` instruction.

5.4 Termination and Measurement

Following Section 4, we represent the machine state as the product of the control state C containing `pc`, `br`, and `in`, and the data state D containing all of the data registers. We denote the unitary operator corresponding to one execution cycle by $\mathsf{E} : \mathcal{H}_C \otimes \mathcal{H}_D \to \mathcal{H}_C \otimes \mathcal{H}_D$. The machine repeats for a total of $t \in \mathbb{N}$ cycles, so that E^t describes the overall evolution of the machine state. After t cycles, the machine discards, or equivalently measures, the registers `pc`, `br`, and `in`. If desired, an external process then has the opportunity to measure the states of the data registers.

We follow established convention [Bernstein and Vazirani 1997; Deutsch 1985] in performing all measurement at the end of computation, so that the machine evolution is realizable via unitary logic gates alone. The principle of deferred measurement states that one final measurement is sufficient to express any quantum computation [Nielsen and Chuang 2010]. Though mid-computation measurement could be added as an extension of the design, it is not strictly necessary to express any computation and does not in general produce output data with superposition intact.

[5]As a technical note, since it is not possible to construct a controlled-U gate given only the ability to apply an unknown gate U [Araújo et al. 2014; Nielsen and Chuang 1997], the set of gates of the machine must at a minimum be fixed and known.

5.5 Synchronization

As shown in Section 3.5, the quantum control machine enables the construction of synchronized programs. We now instantiate the definition of synchronization (Section 4.2) for this machine.

Inputs and Outputs. We identify a program with two sets Z_{in} and Z_{out} of data registers that are assumed to be 0 at its start and end respectively. Let $\mathcal{I}$ be the subspace of $\mathcal{H}_D$ where all registers in Z_{in} are 0 — the *input* states of C. Similarly, let $\mathcal{O}$ be where those in Z_{out} are 0 — its *output* states.

Final State. Instantiating Definition 4.6, we say that given an input state $|\delta_0\rangle \in \mathcal{I}$, and a termination time $t \in \mathbb{N}$, the quantum control machine performs the unitary E^t (defined in Section 5.4) on the state $|\texttt{pc}:0, \texttt{br}:1, \texttt{in}:0\rangle \otimes |\delta_0\rangle$ to produce the final state $|\psi\rangle \in \mathcal{H}_C \otimes \mathcal{O}$.

Synchronization. Instantiating Definition 4.7, we say that the machine is synchronized at time t if there exists x so that for any $|\delta_0\rangle \in \mathcal{I}$, there exists $|\delta'\rangle \in \mathcal{O}$ such that $|\psi\rangle = |\texttt{pc}:x, \texttt{br}:1, \texttt{in}:0\rangle \otimes |\delta'\rangle$.

Expressiveness. In principle, one can express any unitary quantum computation as a synchronized program for the quantum control machine given some appropriate set of primitive unitary gates. Any unitary operator can be approximated to arbitrary precision as a polynomial-length circuit in our gate set of Hadamard and arbitrarily controllable `NOT` gates [Shi 2003], the latter of which can be implemented by a synchronized program using conditional jumps.

In practice, a developer can verify that a program built from structured branching or iteration constructs is synchronized without use of detailed mathematical reasoning, through two insights:

- To verify that all conditional branches are synchronized, one needs only to check that the target of each conditional jump instruction in the program is a reverse jump that points back to the original jump and has the same semantic condition as the original jump.
- To verify that all bounded loops are synchronized, one needs only to check that the execution time of the overall program is independent of each quantum variable in the program, which does not require any specific information about the values of the quantum variables.

In the next section, we illustrate how to use the above principles to build synchronized programs for a variety of high-level control flow constructs as found in quantum algorithms.

6 CASE STUDIES

In this section, we illustrate how a developer uses the abstractions for control flow in superposition provided by the quantum control machine to express quantum algorithms. Specifically, we show how a developer can implement imperative abstractions for control flow – analogues of classical `for`, `if`, and `switch` – as synchronized programs. The case study demonstrates how a developer can represent control flow patterns from existing quantum algorithms and programming languages in a uniform way by correctly manipulating a program counter in superposition.

We have implemented a simulator for the quantum control machine, which accepts a program, input, and runtime t, and outputs the machine state after t steps. Implementations of all case study examples as executable programs are packaged with the simulator in the artifact of this paper.

6.1 Iteration and Phase Estimation

The quantum control machine enables a program to execute a loop for a quantum number of iterations bounded by a classical value, which is integral to the algorithmic building block of quantum phase estimation [Kitaev 1995] as used in algorithms for factoring [Shor 1997], simulation [Abrams and Lloyd 1997], and linear algebra [Abrams and Lloyd 1999; Harrow et al. 2009].

```
1      add res $1    ; copy 1 into res
2      add r1 y      ; copy y into r1
3  l1: jz l2 r1      ; if r1 == 0, break
4      mul res x     ; multiply res by x
5      radd r1 $1    ; decrement r1
6      jmp l1        ; goto loop start
7  l2: nop           ; end of program
```

Fig. 5. Classical program for exponentiation.

```
1      add res $1      ; copy 1 into res
2      add r1 y        ; copy y into r1
3  l1: rjne l3 r1 y   ; if r1 != y, come from l3
4  l2: jz l4 r1       ; if r1 == 0, break
5      mul res x       ; multiply res by x
6      radd r1 $1      ; decrement r1
7  l3: jmp l1         ; goto loop start
8  l4: rjmp l2        ; come from l2
```

Fig. 6. Exponentiation with reverse jumps.

Exponentiation. In Figure 5, we present a classical assembly program for exponentiation. Given x and y, it stores $\mathtt{x}^{\mathtt{y}}$ into the register res. To do so, it repeatedly decrements a copy of y, multiplying res by x on each iteration using the quantum analogue of a for-loop (Example 1.2).

For ease of understanding, in this example we store the output in an auxiliary register rather than in place and use repeated multiplication rather than squaring as is typical, which would be more efficient but also more difficult to understand as an example program.

Adapted Program. Adapting this program to the quantum control machine is done by 1) adapting its control flow to use reversible jumps, and then 2) ensuring that the program is synchronized.

The first step is not the main challenge. By leveraging prior work [Axelsen 2011; Yokoyama et al. 2008] that presents reversible variants of structured if and while constructs, we may straightforwardly insert the appropriate reverse jumps. In Figure 6, we present the adapted program in which we insert a corresponding reverse jump as the target of every conditional or backward jump.

The main challenge is the second step of ensuring that the resulting program is synchronized, which in the example means that the final values of pc and br are independent of x and y. We can see this challenge by executing the program in Figure 6. On the input $|\mathtt{x}:2, \mathtt{y}:1\rangle$, it produces the final state $|\mathtt{x}:2, \mathtt{y}:1, \mathtt{res}:2, \mathtt{pc}:8, \mathtt{br}:1\rangle$. Likewise, input $|\mathtt{x}:2, \mathtt{y}:2\rangle$ results in $|\mathtt{x}:2, \mathtt{y}:2, \mathtt{res}:4, \mathtt{pc}:8, \mathtt{br}:1\rangle$. At first glance, the program seems synchronized — pc is always 8 and br is 1.

Problem: Tortoise and Hare. However, the above values of pc are not for the same time step t. The loop from lines 4 to 7 executes once when y is 1, but twice when y is 2. At $t = 10$, the first input has a pc of 8, having exited the loop, but the second input has a pc of 5, starting the second iteration.

One could continue the slower execution until it also reaches line 8, but by that time, the faster execution will have advanced further again. In a reversible machine semantics, there can exist no concept of a barrier at which the faster execution stops and waits for the slower one. If execution momentarily halts at an instruction that decrements br to 0, then on the next cycle, br would decrement again, meaning pc starts moving again. In general, if the tortoise never catches up to the hare, then there is no point in time at which execution may be safely terminated.

Solution: Padding. In Figure 7, we present a synchronized program that avoids the problem. This program executes the loop a fixed, rather than data-dependent, number of times. Its loop body multiplies res by x for only y iterations, and afterward, it executes padding nops with no effect.

The new argument max, which we require to be classical, upper-bounds the possible values of y. After max iterations of the loop, each branch stores the correct res, and the values of pc and br are equal across all branches, so the program is synchronized. Here, padding is not the only possible approach, but it is simple to use and verify as it guarantees that the program is synchronized.

This example demonstrates how the fundamental property of synchronization restricts the space of valid programs on any quantum computer supporting control flow in superposition. In particular, loops without classical upper bounds cannot be synchronized, which is consistent with prior impossibility results in quantum Turing machines — for more details, see Appendix F.

```
    add res $1      ; copy 1 into res
    add r1 max      ; copy max into r1
l1: rjne l3 r1 max ; if r1 != max, come from l3
l2: jz l4 r1        ; if r1 == 0, break
l5: jg l7 r1 y      ; if r1 > y, goto l7
    mul res x       ; multiply res by x
l6: jmp l8          ; break
l7: rjmp l5         ; come from l5
    nop             ; no-op padding
l8: rjle l6 r1 y    ; if r1 <= y, come from l6
    radd r1 $1      ; decrement r1
l3: jmp l1          ; goto start of loop
l4: rjmp l2         ; come from l2
```

Fig. 7. Synchronized exponentiation.

```
    add r1 i      ; copy i into r1
l1: rjne l3 r1 i ; if r1 != i, come from l3
l2: jz l4 r1      ; if r1 == 0, break
    u H c         ; apply H gate to c
l5: jz l7 c       ; if c == 0, goto l7
    add x $1      ; add 1 to x
l6: jmp l8        ; break
l7: rjmp l5       ; come from l5
    radd x $1     ; subtract 1 from x
l8: rjnz l6 c     ; if c != 0, come from l6
    radd r1 $1    ; subtract 1 from r1
l3: jmp l1        ; goto start of loop
l4: rjmp l2       ; come from l2
```

Fig. 8. Program implementing a Hadamard walk.

6.2 Branch Interference and Quantum Walk

The quantum control machine distinguishes itself from any classical computer by its ability to exhibit quantum interference across control flow paths of the computation. Such interference is essential to the advantage of quantum walk algorithms such as Aharonov et al. [2001]; Ambainis [2004]; Ambainis et al. [2010]; Childs et al. [2007]; Shenvi et al. [2003].

In Figure 8, we present a program that implements a Hadamard walk [Ambainis et al. 2001], adapted from Ying [2014]. This program loops over i iterations, where the algorithm specifies i to be classical. Each round, the program executes an H gate over a qubit c and then adds or subtracts 1 from x based on c, using the quantum analogue of an if-statement (Example 1.1).

$$
\begin{aligned}
& |\mathrm{x}:3,\mathrm{c}:0,\mathrm{pc}:5\rangle \\
\mapsto\ & \tfrac{1}{\sqrt{2}}(|\mathrm{x}:3,\mathrm{c}:0,\mathrm{pc}:9\rangle + |\mathrm{x}:3,\mathrm{c}:1,\mathrm{pc}:6\rangle) \\
\mapsto\ & \tfrac{1}{\sqrt{2}}(|\mathrm{x}:2,\mathrm{c}:0,\mathrm{pc}:5\rangle + |\mathrm{x}:4,\mathrm{c}:1,\mathrm{pc}:5\rangle) \\
\mapsto\ & \tfrac{1}{2}(|\mathrm{x}:2,\mathrm{c}:0,\mathrm{pc}:9\rangle + |\mathrm{x}:2,\mathrm{c}:1,\mathrm{pc}:6\rangle \\
& \quad + |\mathrm{x}:4,\mathrm{c}:0,\mathrm{pc}:9\rangle - |\mathrm{x}:4,\mathrm{c}:1,\mathrm{pc}:6\rangle) \\
\mapsto\ & \tfrac{1}{2}(|\mathrm{x}:1,\mathrm{c}:0,\mathrm{pc}:5\rangle + |\mathrm{x}:3,\mathrm{c}:1,\mathrm{pc}:5\rangle \\
& \quad + |\mathrm{x}:3,\mathrm{c}:0,\mathrm{pc}:5\rangle - |\mathrm{x}:5,\mathrm{c}:1,\mathrm{pc}:5\rangle) \\
\mapsto\ & \tfrac{1}{2\sqrt{2}}(|\mathrm{x}:1,\mathrm{c}:0,\mathrm{pc}:9\rangle + |\mathrm{x}:1,\mathrm{c}:1,\mathrm{pc}:6\rangle \\
& \quad + |\mathrm{x}:3,\mathrm{c}:0,\mathrm{pc}:9\rangle \cancel{- |\mathrm{x}:3,\mathrm{c}:1,\mathrm{pc}:6\rangle} \\
& \quad + |\mathrm{x}:3,\mathrm{c}:0,\mathrm{pc}:9\rangle \cancel{+ |\mathrm{x}:3,\mathrm{c}:1,\mathrm{pc}:6\rangle} \\
& \quad - |\mathrm{x}:5,\mathrm{c}:0,\mathrm{pc}:9\rangle + |\mathrm{x}:5,\mathrm{c}:1,\mathrm{pc}:6\rangle) \\
\mapsto\ & \tfrac{1}{2\sqrt{2}}(|\mathrm{x}:0,\mathrm{c}:0,\mathrm{pc}:14\rangle + |\mathrm{x}:2,\mathrm{c}:1,\mathrm{pc}:14\rangle \\
& \quad + |\mathrm{x}:2,\mathrm{c}:0,\mathrm{pc}:14\rangle + |\mathrm{x}:2,\mathrm{c}:0,\mathrm{pc}:14\rangle \\
& \quad - |\mathrm{x}:4,\mathrm{c}:0,\mathrm{pc}:14\rangle + |\mathrm{x}:6,\mathrm{c}:1,\mathrm{pc}:14\rangle)
\end{aligned}
$$

Fig. 9. Partial execution trace of Figure 8, showing only steps where $\mathrm{br} = 1$. In each state, i and r1 are uniform across the superposition, and they are not shown.

Branch Interference. The way in which this program demonstrates quantum interference is that measuring the final x value it produces yields a substantially different distribution from a classical random walk that on every round moves x in a uniformly random direction.

In particular, letting the initial x and i be 3, the program executes as in Figure 9. At the end of the program, measuring the value of x yields outcome 4 with probability only $1/6 \approx 17\%$, as compared to $3/8 \approx 38\%$ for the classical random walk. The reason is that quantum interference cancels two execution paths, corresponding to outcome 4, that have opposite phase.

The correctness of the program relies on the use of injective abstractions for control flow as opposed to writing down a history of the program counter as in Section 3.3. For contrast, we depict in Appendix D the incorrect execution that would result from writing down such a history. The difference is that the analogues of the identical states that cancel in Figure 9 are instead distinct and do not interfere to cancel. In the final measurement outcome of the incorrect execution, the outcome of 4 occurs with probability 3/8, the same result as on a classical computer.

6.3 Indexed Branching and Quantum Simulation

The quantum control machine enables branching operations to be indexed by data in superposition, in a way analogous to classical array indexing or `switch`-statements (Example 1.1).

In Figure 10, we present a program that, given two quantum registers `i` and `x`, applies a Majorana fermion operator $|\mathtt{i}\rangle\,|\mathtt{x}\rangle \mapsto |\mathtt{i}\rangle\,\mathtt{Y}_{\mathtt{i}} \cdot \mathtt{Z}_{\mathtt{i}-1} \cdots \mathtt{Z}_0\,|\mathtt{x}\rangle$ to them, which is useful to algorithms for simulation of fermionic systems [Babbush et al. 2018]. In this operator, `Y` is a single-qubit Pauli-Y gate as defined in Nielsen and Chuang [2010], and we assume that the `Y` and `Z` (Section 2) gates are supported as primitive unitary gates.

The program operates as follows. First, lines 1 to 3 apply the `Y` gate on the `i`th qubit of the `x` register. The following loop then performs the `Z` gate on each of the qubits $\mathtt{i} - 1$ through 0 of the `x` register. For clarity, the loop has not yet been subject to padding as in Section 6.1, which must still be done if `i` is in superposition.

Though this program has not been optimized for practical concerns such as qubit and gate usage, it exemplifies a new programming model in which one can work with quantum data via abstractions similar to classical arrays.

```
 1      get i r1 x     ; put bit i of x into r1
 2      u Y r1         ; apply Y gate to r1
 3      rget i r1 x    ; put r1 into bit i of x
 4      add r1 i       ; copy i into r1
 5  l1: rjne l3 r1 i   ; if r1 != i, come from l3
 6  l2: jz l4 r1       ; if r1 == 0, break
 7      radd r1 $1     ; subtract 1 from r1
 8      get r1 r2 x    ; put bit r1 of x into r2
 9      u Z r2         ; apply Z gate to r2
10      rget r1 r2 x   ; put r2 into bit r1 of x
11  l3: jmp l1         ; goto start of loop
12  l4: rjmp l2        ; come from l2
```

Fig. 10. Program for a Majorana fermion operator.

7 IMPLICATIONS AND DIRECTIONS FORWARD

In this section, we discuss implications of this work to research in quantum programming languages, computer architecture, and theory of computation. For a detailed discussion of the practical costs to realizing control flow in superposition in terms of hardware support and program verification, see Appendix E. For other related work studying designs for quantum λ-calculi, reversible computation, oblivious computation, and unbounded-time quantum computation, see Appendix F.

7.1 Quantum Programming Languages

Abstractions for control flow in superposition such as quantum `if`-statements and `for`-loops have become a value proposition in emerging quantum programming languages [Bichsel et al. 2020; Pal and Ghosh 2022; Voichick et al. 2023; Yuan and Carbin 2022]. The no-embedding theorem provides a unifying explanation for why these abstractions, and others such as recursion and continuations, cannot be adapted to superposition by directly lifting the classical conditional jump.

As an example, language designers have repeatedly and independently confronted the fact that a quantum `if`-statement is not realizable in general if its branches can be arbitrary statements. Proposed solutions have included the dynamic enforcement of an orthogonality judgment [Altenkirch and Grattage 2005] and the static enforcement of conditions such as preventing the condition of the `if` from being modified under its branches [Bichsel et al. 2020; Yuan and Carbin 2022]. This work presents a correctness condition for control flow in superposition that unifies and generalizes prior approaches, which is that the semantics of each abstraction must be injective and the program must be synchronized. In principle, this condition can be enforced at the level of the `if`-statement or at the more primitive level of assembly, as in the quantum control machine.

An advantage of sound primitives at the assembly level is that they in turn empower generalized reasoning about the space of realizable abstractions and can act as an intermediate compilation target for emerging abstractions. For instance, though researchers have proposed quantum analogues of recursion and closures [Díaz-Caro et al. 2019; Ying 2014; Ying et al. 2012], to date we are not aware of their realization via a compiler or equivalent. We hope that the quantum control machine may act as a compilation target for such proposals and create new opportunities in language design.

7.2 Quantum Computer Architecture

Researchers have proposed stored-program quantum architectures, commonly referred to as quantum von Neumann architectures, with suggested benefits for the efficiency [Kjaergaard et al. 2020], realizability [Meier et al. 2024], and security [Wang 2022] of the resulting system.

For example, Kjaergaard et al. [2020] experimentally realize a device that uses one set of qubits to parameterize a rotation gate over another set of qubits. That work describes the “use of quantum instructions to implement a quantum program” and “instructions derived from the present quantum state of the processor” as potentially advantageous in quantum algorithms for semi-definite programming, simulation, and principal component analysis [Kjaergaard et al. 2020].

However, to our best knowledge, these prior designs have not acknowledged the challenges that will be ultimately encountered when making instructions such as conditional jump operate in superposition. For instance, the no-embedding theorem implies that the machine proposed by Lagana et al. [2009], which attempts to provide conditional jump via a history of program counters, does not correctly execute quantum algorithms. While the designs of Meier et al. [2024]; Wang [2022] lack an operational specification for the control unit as an instruction set, these designs would face the same fundamental limitations on expressible control flow when fully formalized.

7.3 Theory of Quantum Computation

A common, and true, maxim in quantum computation is that any classical computation is also realizable on a quantum computer [Nielsen and Chuang 2010, Section 1.4.1]. By contrast, in this work, we show that a stronger assumption – any classical programming abstraction is also correctly realizable on a quantum computer – is false, as seen in the conventional conditional jump.

An implication is that designers of algorithms would benefit from explicitly specifying control flow as part of the state of a quantum computation rather than leaving it as an implementation detail. The realization of a control flow abstraction requires careful reasoning from the language and potentially the programmer to produce a correct output and preserve quantum advantage.

The scope of this implication is over algorithms that transform quantum data and then leverage interference on the output data, which include Shor [1997] and the other examples in Section 6. We note that it may be possible in limited cases for the design of algorithms to preemptively avoid this concern — for example, the classical oracle component of Grover [1996] produces an output that is promptly uncomputed, and the algorithm instead leverages interference on the input data.

8 CONCLUSION

Researchers have long studied designs for quantum computers to learn how to realize the design in hardware or analyze its theoretical power. This work advocates for a new dimension of study — how to correctly and intuitively program the computer to implement quantum algorithms.

Studying a quantum computer through the lens of a programmer reveals the danger that trying to implement a quantum algorithm using classical control flow abstractions such as conditional jump can cause the algorithm to produce incorrect results. Put plainly, programming a quantum computer in the same way as a classical one can in fact turn the quantum computer into a classical computer. If so, the computer’s quantum advantage and the return on its investment are lost.

Despite these challenges, we believe that control flow in superposition will remain an indispensible abstraction for expressing quantum algorithms. This work makes it possible for the first time to correctly program a quantum computer using the abstraction of a program counter, bringing the vision of making quantum programs as easy to write as classical programs closer to reach.

DATA AVAILABILITY STATEMENT

The software that supports Section 6 is available on Zenodo [Yuan et al. 2024].

ACKNOWLEDGEMENTS

We thank Ellie Cheng, Tian Jin, Jesse Michel, Patrick Rall, and Logan Weber for helpful feedback on this work, and also Scott Aaronson, Soonwon Choi, Isaac Chuang, Aram Harrow, Stacey Jeffrey, Bobak Toussi Kiani, and Yuval Sanders for providing references to related work. This work was supported in part by the National Science Foundation (CCF-1751011) and the Sloan Foundation.

REFERENCES

Daniel S. Abrams and Seth Lloyd. 1997. Simulation of Many-Body Fermi Systems on a Universal Quantum Computer. *Phys. Rev. Letters* 79 (Sep 1997). Issue 13. https://doi.org/10.1103/PhysRevLett.79.2586

Daniel S. Abrams and Seth Lloyd. 1999. Quantum Algorithm Providing Exponential Speed Increase for Finding Eigenvalues and Eigenvectors. *Phys. Rev. Letters* 83, 24 (Dec 1999). https://doi.org/10.1103/PhysRevLett.83.5162

Dorit Aharonov, Andris Ambainis, Julia Kempe, and Umesh Vazirani. 2001. Quantum Walks on Graphs. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/380752.380758

Thorsten Altenkirch and J. Grattage. 2005. A Functional Quantum Programming Language. In *IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1109/LICS.2005.1

Andris Ambainis. 2004. Quantum walk algorithm for element distinctness. In *IEEE Symposium on Foundations of Computer Science*. https://doi.org/10.1109/FOCS.2004.54

Andris Ambainis, Eric Bach, Ashwin Nayak, Ashvin Vishwanath, and John Watrous. 2001. One-Dimensional Quantum Walks. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/380752.380757

Andris Ambainis, A. M. Childs, B. W. Reichardt, R. Špalek, and S. Zhang. 2010. Any AND-OR Formula of Size N Can Be Evaluated in Time $N^{1/2+o(1)}$ on a Quantum Computer. *SIAM J. Comput.* 39, 6 (2010). https://doi.org/10.1137/080712167

Pablo Andrés-Martínez and Chris Heunen. 2022. Weakly measured while loops: peeking at quantum states. *Quantum Science and Technology* 7, 2 (Feb 2022). https://doi.org/10.1088/2058-9565/ac47f1

Mateus Araújo, Adrien Feix, Fabio Costa, and Časlav Brukner. 2014. Quantum circuits cannot control unknown operations. *New Journal of Physics* 16, 9 (Sep 2014). https://doi.org/10.1088/1367-2630/16/9/093026

Pablo Arrighi, Alejandro Díaz-Caro, and Benoît Valiron. 2017. The vectorial λ-calculus. *Information and Computation* 254 (2017). https://doi.org/10.1016/j.ic.2017.04.001

Srinivasan Arunachalam, Vlad Gheorghiu, Tomas Jochym-O'Connor, Michele Mosca, and Priyaa Varshinee Srinivasan. 2015. On the robustness of bucket brigade quantum RAM. *New Journal of Physics* 17, 12 (Dec 2015). https://doi.org/10.1088/1367-2630/17/12/123010

Holger Bock Axelsen. 2011. Clean Translation of an Imperative Reversible Programming Language. In *International Conference on Compiler Construction*. https://doi.org/10.1007/978-3-642-19861-8_9

Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. 2007. Reversible Machine Code and Its Abstract Processor Architecture. In *International Conference on Computer Science: Theory and Applications*. https://doi.org/10.1007/978-3-540-74510-5_9

Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. 2018. Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity. *Phys. Rev. X* 8, 4 (Oct 2018). https://doi.org/10.1103/PhysRevX.8.041015

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal Verification of a Constant-Time Preserving C Compiler. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3371075

J. S. Bell. 1964. On the Einstein Podolsky Rosen paradox. *Physics* 1 (Nov 1964). Issue 3. https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195

Charles H. Bennett. 1973. Logical Reversibility of Computation. *IBM Journal of Research and Development* 17, 6 (1973). https://doi.org/10.1147/rd.176.0525

Ethan Bernstein and Umesh Vazirani. 1997. Quantum Complexity Theory. *SIAM J. Comput.* 26, 5 (1997). https://doi.org/10.1137/S0097539796300921

Dominic W. Berry, Craig Gidney, Mario Motta, Jarrod R. McClean, and Ryan Babbush. 2019. Qubitization of Arbitrary Basis Quantum Chemistry Leveraging Sparsity and Low Rank Factorization. *Quantum* 3 (Dec 2019). https://doi.org/10.22331/q-2019-12-02-208

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *ACM SIGPLAN Conference on Programming Language Design and*

Implementation. https://doi.org/10.1145/3385412.3386007

Kai-Wen Cheng and Chien-Cheng Tseng. 2002. Quantum full adder and subtractor. *Electronics Letters* 38 (2002). https://doi.org/10.1049/el:20020949

Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (Sep 2018). https://doi.org/10.1073/pnas.1801723115

Andrew M. Childs, Ben W. Reichardt, Robert Spalek, and Shengyu Zhang. 2007. Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer. https://doi.org/10.48550/ARXIV.QUANT-PH/0703015 arXiv:0703015 [quant-ph]

Andrew M. Childs and Nathan Wiebe. 2012. Hamiltonian Simulation Using Linear Combinations of Unitary Operations. *Quantum Information and Computation* 12, 11&12 (Nov 2012). https://doi.org/10.26421/qic12.11-12

Giulio Chiribella, Giacomo Mauro D'Ariano, Paolo Perinotti, and Benoît Valiron. 2013. Quantum computations without definite causal structure. *Phys. Rev. A* 88, 2 (Aug 2013). https://doi.org/10.1103/PhysRevA.88.022318

Seong-Min Cho, Aeyoung Kim, Dooho Choi, Byung-Soo Choi, and Seung-Hyun Seo. 2020. Quantum Modular Multiplication. *IEEE Access* 8 (2020). https://doi.org/10.1109/ACCESS.2020.3039167

Alonzo Church. 1941. *The Calculi of Lambda Conversion.* https://doi.org/10.1515/9781400881932

Pierre Clairambault and Marc de Visme. 2019. Full Abstraction for the Quantum Lambda-Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3371131

Pierre Clairambault, Marc De Visme, and Glynn Winskel. 2019. Game Semantics for Quantum Programming. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3290345

David Deutsch. 1985. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society A* 400, 1818 (1985). https://doi.org/10.1098/rspa.1985.0070

Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2006. On Reversible Combinatory Logic. In *International Workshop on Developments in Computational Models.* https://doi.org/10.1016/j.entcs.2005.09.018

Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoît Valiron. 2019. Realizability in the Unitary Sphere. In *ACM/IEEE Symposium on Logic in Computer Science.* https://doi.org/10.1109/LICS.2019.8785834

Thomas G. Draper. 2000. Addition on a Quantum Computer. arXiv:quant-ph/0008033 [quant-ph]

Michael Frank. 1999. *Reversibility for Efficient Computing.* Ph. D. Dissertation. Massachusetts Institute of Technology. https://doi.org/1721.1/9464

Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Quantum Random Access Memory. *Phys. Rev. Letters* 100, 16 (Apr 2008). https://doi.org/10.1103/PhysRevLett.100.160501

Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996). https://doi.org/10.1145/233551.233553

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/2491956.2462177

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *ACM Symposium on Theory of Computing.* https://doi.org/10.1145/237814.237866

Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Phys. Rev. Letters* 103, 15 (Oct 2009). https://doi.org/10.1103/PhysRevLett.103.150502

Ichiro Hasuo and Naohiko Hoshino. 2017. Semantics of higher-order quantum computation via geometry of interaction. In *Games for Logic and Programming Languages Workshop.* https://doi.org/10.1016/j.apal.2016.10.010

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3434318

Peter Hines. 2008. Machine semantics. *Theoretical Computer Science* 409, 1 (2008). https://doi.org/10.1016/j.tcs.2008.07.015

Peter Hines. 2011. Can a Quantum Computer Run the von Neumann Architecture? In *New Structures for Physics.* https://doi.org/10.1007/978-3-642-12821-9_14

Md Saiful Islam, Muhammad Mahbubur Rahman, Zerina Begum, and Mohd Z Hafiz. 2009. Low cost quantum realization of reversible multiplier circuit. *Information Technology Journal* 8, 2 (2009). https://doi.org/10.3923/itj.2009.208.213

Tien D. Kieu and Michael Danos. 1998. The halting problem for universal quantum computers. https://doi.org/10.48550/ARXIV.QUANT-PH/9811001 arXiv:9811001 [quant-ph]

A. Yu. Kitaev. 1995. Quantum measurements and the Abelian Stabilizer Problem. https://doi.org/10.48550/arXiv.quant-ph/9511026 arXiv:quant-ph/9511026 [quant-ph]

A. Yu. Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (Dec 1997). https://doi.org/10.1070/RM1997V052N06ABEH002155

Morten Kjaergaard, Mollie E. Schwartz, Ami Greene, Gabriel O. Samach, Andreas Bengtsson, Michael O'Keeffe, Christopher M. McNally, Jochen Braumüller, David K. Kim, Philip Krantz, Milad Marvian, Alexander Melville, Bethany M. Niedzielski, Youngkyu Sung, Roni Winik, Jonilyn Yoder, Danna Rosenberg, Kevin Obenland, Seth Lloyd, Terry P. Orlando, Iman Marvian, Simon Gustavsson, and William D. Oliver. 2020. Programming a quantum computer with quantum instructions. https://doi.org/10.48550/ARXIV.2001.08838 arXiv:2001.08838 [quant-ph]

Andre Kornell, Bert Lindenhovius, and Michael Mislove. 2021. Quantum CPOs. *Electronic Proceedings in Theoretical Computer Science* 340 (Sep 2021). https://doi.org/10.4204/eptcs.340.9

Antonio A. Lagana, M. A. Lohe, and Lorenz von Smekal. 2009. Construction of a universal quantum computer. *Phys. Rev. A* 79, 5 (May 2009). https://doi.org/10.1103/PhysRevA.79.052322

R. Landauer. 1961. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development* 5, 3 (1961). https://doi.org/10.1147/rd.53.0183

Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3428218

Noah Linden and Sandu Popescu. 1998. The Halting Problem for Quantum Computers. https://doi.org/10.48550/ARXIV.QUANT-PH/9806054 arXiv:9806054 [quant-ph]

Seth Lloyd, Silvano Garnerone, and Paolo Zanardi. 2014. Quantum algorithms for topological and geometric analysis of big data. *Nature Communications* 7 (Aug 2014). https://doi.org/10.1038/ncomms10138

Guang Hao Low and Isaac L. Chuang. 2019. Hamiltonian Simulation by Qubitization. *Quantum* 3 (Jul 2019). https://doi.org/10.22331/q-2019-07-12-163

Guang Hao Low, Vadym Kliuchnikov, and Luke Schaeffer. 2018. Trading T-gates for dirty qubits in state preparation and unitary synthesis. https://doi.org/10.48550/arXiv.1812.00954 arXiv:1812.00954 [quant-ph]

Octavio Malherbe, Philip Scott, and Peter Selinger. 2013. Presheaf Models of Quantum Computation: An Outline. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*. https://doi.org/10.1007/978-3-642-38164-5_13

Olivia Di Matteo, Vlad Gheorghiu, and Michele Mosca. 2020. Fault-Tolerant Resource Estimation of Quantum Random-Access Memories. *IEEE Transactions on Quantum Engineering* 1 (2020). https://doi.org/10.1109/tqe.2020.2965803

Florian Meier, Marcus Huber, Paul Erker, and Jake Xuereb. 2024. Autonomous Quantum Processing Unit: What does it take to construct a self-contained model for quantum computation? https://doi.org/10.48550/arXiv.2402.00111 arXiv:2402.00111 [quant-ph]

John M. Myers. 1997. Can a Universal Quantum Computer Be Fully Quantum? *Phys. Rev. Letters* 78, 9 (Mar 1997). https://doi.org/10.1103/PhysRevLett.78.1823

Moni Naor and Vanessa Teague. 2001. Anti-persistence: History independent data structures. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/380752.380844

Michael A. Nielsen and Isaac L. Chuang. 1997. Programmable Quantum Gate Arrays. *Phys. Rev. Letters* 79, 2 (Jul 1997). https://doi.org/10.1103/PhysRevLett.79.321

Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* https://doi.org/10.1017/CBO9780511976667

Michele Pagani, Peter Selinger, and Benoît Valiron. 2014. Applying Quantitative Semantics to Higher-Order Quantum Computing. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2535838.2535879

Abhinandan Pal and Anubhab Ghosh. 2022. Qiwi: A Beginner Friendly Quantum Language. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity.* https://doi.org/10.1145/3563768.3563959

Alexandru Paler, Oumarou Oumarou, and Robert Basmadjian. 2020. Parallelizing the queries in a bucket-brigade quantum random access memory. *Phys. Rev. A* 102, 3 (Sep 2020). https://doi.org/10.1103/PhysRevA.102.032608

A. Pati and S. Braunstein. 2000. Impossibility of deleting an unknown quantum state. *Nature* 404 (2000). https://doi.org/10.1038/404130b0

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3009837.3009894

Nicholas Pippenger and Michael J. Fischer. 1979. Relations Among Complexity Measures. *J. ACM* 26, 2 (Apr 1979). https://doi.org/10.1145/322123.322138

John Proos and Christof Zalka. 2003. Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves. *Quantum Information and Computation* 3, 4 (Jul 2003).

Qiskit Developers. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

Mathys Rennela and Sam Staton. 2018. Classical Control and Quantum Circuits in Enriched Category Theory. In *Conference on the Mathematical Foundations of Programming Semantics*. https://doi.org/10.1016/j.entcs.2018.03.027

Mathys Rennela, Sam Staton, and Robert Furber. 2017. Infinite-Dimensionality in Quantum Foundations: W*-algebras as Presheaves over Matrix Algebras. *Electronic Proceedings in Theoretical Computer Science* 236 (Jan 2017). https://doi.org/10.4204/eptcs.236.11

Rich Rines and Isaac Chuang. 2018. High Performance Quantum Modular Multipliers. https://doi.org/10.48550/arXiv.1801.01081 arXiv:1801.01081 [quant-ph]

Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *International Conference on Foundations of Software Science and Computation Structures.* https://doi.org/10.1007/978-3-319-89366-2_19

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14 (Aug 2004). https://doi.org/10.1017/S0960129504004256

Sanjit A. Seshia and Jonathan Kotker. 2011. GameTime: A Toolkit for Timing Analysis of Software. In *Tools and Algorithms for the Construction and Analysis of Systems.* https://doi.org/10.1007/978-3-642-19835-9_34

Neil Shenvi, Julia Kempe, and K. Birgitta Whaley. 2003. Quantum random-walk search algorithm. *Phys. Rev. A* 67, 5 (May 2003). https://doi.org/10.1103/PhysRevA.67.052307

Yaoyun Shi. 2003. Both Toffoli and Controlled-NOT need little help to do universal quantum computing. *Quantum Information and Computation* 3, 1 (Jan 2003). https://doi.org/10.26421/QIC3.1-7

Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct 1997). https://doi.org/10.1137/S0097539795293172

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/2908080.2908092

Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Real World Domain Specific Languages Workshop.* https://doi.org/10.1145/3183895.3183901

Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück. 2012. A Reversible Processor Architecture and Its Reversible Logic Design. In *Conference on Reversible Computation.* https://doi.org/10.1007/978-3-642-29517-1_3

Dominique Unruh. 2019. Quantum Relational Hoare Logic. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3290346

André van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (2004). https://doi.org/10.1137/S0097539703432165

Carlin Vieri, M. Ammer, Michael Frank, Norman Margolus, and Tom Knight. 1998. A Fully Reversible Asymptotically Zero Energy Microprocessor. In *Power-Driven Microarchitecture Workshop.*

Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3571225

Dong-Sheng Wang. 2022. A prototype of quantum von Neumann architecture. *Communications in Theoretical Physics* 74, 9 (Aug 2022). https://doi.org/10.1088/1572-9494/ac68d8

Nathan Wiebe, Daniel Braun, and Seth Lloyd. 2012. Quantum Algorithm for Data Fitting. *Phys. Rev. Letters* 109, 5 (Aug 2012). https://doi.org/10.1103/PhysRevLett.109.050505

W. Wootters and W. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299 (1982). https://doi.org/10.1038/299802a0

Mingsheng Ying. 2014. Quantum Recursion and Second Quantisation. https://doi.org/10.48550/ARXIV.1405.4443 arXiv:1405.4443 [quant-ph]

Mingsheng Ying and Yuan Feng. 2010. Quantum Loop Programs. *Acta Informatica* 6 (2010). https://doi.org/10.1007/s00236-010-0117-4

Mingsheng Ying, Nengkun Yu, and Yuan Feng. 2012. Defining Quantum Control Flow. https://doi.org/10.48550/ARXIV.1209.4379 arXiv:1209.4379 [quant-ph]

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Reversible Flowchart Languages and the Structured Reversible Program Theorem. In *Automata, Languages and Programming.* https://doi.org/10.1007/978-3-540-70583-3_22

Nengkun Yu and Jens Palsberg. 2021. Quantum Abstract Interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/3410291

Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.* https://doi.org/10.1145/3563297

Charles Yuan, Christopher McNally, and Michael Carbin. 2022. Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3498691

Charles Yuan, Agnes Villanyi, and Michael Carbin. 2024. *Quantum Control Machine: The Limits of Control Flow in Quantum Programming.* https://doi.org/10.5281/zenodo.10452601

Received 16-OCT-2023; accepted 2024-02-24