*Proc. QPL 2004, pp. 145–162*

# Toward a software architecture for quantum computing design tools

K. Svore[*] Columbia   A. Cross MIT   A. Aho Columbia   I. Chuang MIT   I. Markov U. of Mich.

**Abstract**

Compilers and computer-aided design tools will be essential for quantum computing. We present a computer-aided design flow that transforms a high-level language program representing a quantum computing algorithm into a technology-specific implementation. We trace the significant steps in this flow and illustrate the transformations to the representation of the quantum program. The focus of this paper is on the languages and transformations needed to represent and optimize a quantum algorithm along the design flow. Our software architecture provides significant benefits to algorithm designers, tool builders, and experimentalists. Of particular interest are the trade-offs in performance and accuracy that can be obtained by weighing different optimization and error-correction procedures at given levels in the design hierarchy.

## 1 Introduction

Just as they are for classical computing, compilers and design tools are going to be indispensable for quantum computing. In this paper, we present a layered software architecture for a four-phase computer-aided design flow mapping a high-level language source program representing a quantum algorithm to a technology-specific implementation. We trace the series of steps in this flow and show the corresponding transformations in the intermediate representations of the quantum algorithm. We use results from our work on optimization and simulation tools to illustrate the overall design flow and individual design transformations.

We discuss the languages and notational representations needed to support this design flow. Of particular importance is the high-level programming language used to specify quantum algorithms. We also need optimizing compilers that will map the quantum programming language programs into efficient and robust low-level programs that can be executed on a quantum device or simulated on classical computers using high-performance tools.

In the next section we discuss the toolsuite and the benefits of the layered software architecture and the design flow. We then describe properties needed in a quantum programming language, a proposed quantum computer compiler, a quantum assembly language, a

[*] contact:kmsvore@cs.columbia.edu

quantum physical operations language, and a simulator with layout tools. Sample design flows for ion-trap computers illustrate what is practical today. In conclusion, we discuss challenges in building software architectures for quantum design automation and point out areas where software support may soon be required.

# 2 Quantum Design Tools

## 2.1 Toolsuite

We envision a layered hierarchy of notations and tools that includes programming languages, compilers, optimizers, simulators, and layout tools. The programming languages and compilers at the top level of our toolsuite need to support the abstractions used to specify quantum algorithms and need to accommodate changes in technology-independent and dependent optimizations as our understanding of new quantum technologies matures. The simulation and layout tools at the bottom end need to incorporate details of the emerging quantum technologies that would ultimately implement the algorithms described in the high-level programming language. The tools need to balance tradeoffs involving performance, minimization of qubits, and fault-tolerant implementations. For these reasons, we propose a layered hierarchy of design tools with simple interfaces between each layer.

## 2.2 Benefits of an Open Software Architecture

There are several benefits to having a layered software architecture with well-defined interfaces for quantum computing. On the technical front, a layered architecture facilitates tool interoperability and makes it easier to add new tools to an existing toolsuite. It also makes it easier to maintain and add improvements to existing tools. At this time our knowledge of how best to optimize quantum circuits for noise avoidance is limited as is our ability to minimize errors and maximize speed. A layered architecture would allow researchers to experiment with new algorithms and through simulation determine their benefits with given quantum technologies before constructing actual physical device components. On the economic side, no single group can afford the huge software development cost required to develop all the tools needed to make quantum devices.

All stakeholders in quantum computing would benefit from the open software architecture, which allows wider community participation in the creation and use of needed tools. Quantum algorithm designers can explore new algorithms in more realistic settings involving actual noise and physical resource constraints. Researchers developing quantum circuit optimizations can evaluate tradeoffs taking into account quantum noise and physical parameters. Experimentalists and device designers can do simulations of important quantum algorithms on proposed new technologies before doing expensive lab experiments. Tools designers can experiment with new algorithms and can evaluate their overall impact on the design process. Researchers can also develop more refined noise models for tailored-correction procedures.

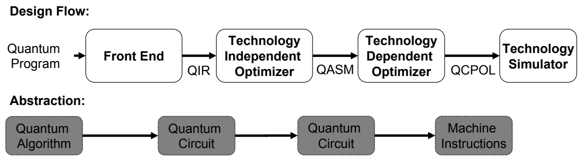


Figure 1: PROPOSED DESIGN FLOW.

## 2.3 The Design Flow

The design flow (Figure 1) is a four-phase process that maps a high-level program representing a quantum algorithm into a technology-specific implementation or simulation. The first three phases of the design flow are part of the quantum computer compiler (QCC). The last phase is a simulation or an implementation of the quantum algorithm on a quantum device.

The representations of the quantum algorithm between the phases are the key to an interoperable tools hierarchy. The first phase, the front-end of the compiler, maps a high-level specification of a quantum algorithm into a quantum intermediate representation (QIR). The most popular choice for the quantum intermediate representation is an encoding of quantum circuits where the gates are drawn from some universal basis set.

The second phase is a technology-independent optimizer, which maps the QIR representation of the quantum algorithm into an equivalent lower-level circuit representation of single-qubit and CNOT gates. We call the lower-level notation QASM, for quantum assembly language. The second phase attempts to produce as good a QASM representation as possible, where the metric of goodness can be varied, such as circuit size, circuit depth, accuracy or fault tolerance.

The third phase performs optimizations suited to the quantum computing technology used to simulate or implement the quantum algorithm. It outputs a physical-language representation with technology-specific parameters called QCPOL, for quantum computing physical operations language. The third phase is divided into several subphases. The first subphase maps the QASM representation of single-qubit and CNOT gates into a QASM representation using a fault-tolerant discrete universal set of gates. In mapping the gates to a discrete set, we must approximate the original gate set using the Solovay-Kitaev method [17]. In the second subphase, these gates are mapped into a QCPOL representation containing the physical instructions for the fault-tolerant operations, including the required movements. Knowledge of the physical layout enters at this step.

The final phase includes technology-dependent tools such as circuit or physical operations simulators and layout modules, as well as execution on an actual quantum device.

Note that it is likely that any real design will be iterative in nature, requiring several passes through one or more of these phases. Also, a fault-tolerant physical implementation of a quantum algorithm is crucial. At present, it is not known where in the design flow is the best place to introduce error-correction circuitry but, as described in Section 8, this design flow can accommodate it at any level in the hierarchy. Particularly, it is interesting to determine if the compiler can automatically introduce error-correction and produce a fault-tolerant QASM or QCPOL representation. The remainder of this paper describes these phases in detail, focusing on the representations, algorithms, and design issues relevant to each phase. Section 8 of the paper gives some example design flows.

## 3 Quantum Programming Source Languages

Designing a quantum programming language is a difficult task since there is currently a limited repertoire of quantum algorithms. Moreover, at this point, we do not know whether a quantum computer will be a special-purpose ASIC or a general processor. However, we assume the communication between the quantum device and the classical processor follows the QRAM model described in [5, 15].

The high-level quantum programming language must offer both experimental physicists and algorithm designers the necessary abstractions to perform any quantum computation. Many programming languages have been proposed for representing quantum algorithms [5, 18, 19, 20, 23, 25, 38]. Most of these languages offer abstractions based either on the quantum circuit model or on Dirac notation [17].

A quantum programming environment should possess several key characteristics [5]. First, the environment needs a quantum programming language that is complete in the sense that every current and future quantum algorithm can be written and expressed in the language. Basically, the language should support complex numbers and be capable of expressing and composing unitary transforms and measurements, as well as describing the classical pre- and post-processing calculations. The environment should also have reusable subroutines or gate libraries that can be called by the programmer to implement a quantum algorithm. The exact modularization of a quantum programming environment, however, is an open research question.

Second, it is advantageous if the language and environment are based on familiar concepts and constructs. This way, learning how to write, debug and run a quantum program will be easier than if a totally new environment were used.

Third, the environment should allow easy separation of classical computations from quantum computations. Since a quantum computer has noise and limited coherence time, the separation can help determine and limit the length of time needed to implement the quantum computations on the quantum device. The compiler for a quantum programming language should be able to translate a source program into an efficient and robust quantum circuit or physical implementation; it should be easy to translate into different gate sets or optimize with respect to a desired cost function using algorithms that are simple, efficient and effective.

Fourth, the high-level programming language should be hardware independent since the source program should be able to run on different quantum technologies. However, the language and environment should allow the inclusion of technology-specific modules.

Lastly, it would be advantageous to have a language that supports high-level abstractions that would allow the easy development of new algorithms using quantum-mechanical phenomena such as entanglement and superposition. Although there are several existing quantum programming languages, such as QCL [18, 19], Q [5], and others [20, 23, 25, 38], the abstractions captured in these languages do not take full advantage of the quantum-mechanical principles used in quantum computations. For example, QCL [18, 19] is an interpreted language that contains many usual programming constructs such as arrays and subroutines. It allows the implementation of new procedures by the programmer and provides a universal set of gates. However, the descriptions of the operators and states are based on matrices and vectors, and thus provides us with no further insight beyond the usual Dirac notation. Q [5] is an extension of C++ that is designed with a clear separation of quantum and classical procedures. It builds a quantum operator as a data object rather than as a function to allow for run-time optimizations and easy compositions. But again, this language is based on Dirac notation and does not provide us with further insights.

By incorporating appropriate abstractions into a language and environment for quantum computing, we will hopefully develop an environment that makes both the design and programming of quantum algorithms an easier task. In addition, we seek a language from which robust, optimized target programs can be easily created.

# 4 Quantum Computer Compiler (QCC)

We now investigate the compilation steps of our quantum computer compiler (Figure 1). A generic compiler for a classical language on a classical machine consists of a sequence of phases, each of which transforms the source program from one representation into another [3]. The lexical analyzer tokenizes the source program into logical sequences of characters. The tokens are passed to the syntax analyzer to translate into a syntax tree. This representation is passed to the semantic analyzer, which further analyzes the representation for semantic errors and compatible operations on types. The output is then sent to the intermediate code generator to create an intermediate representation for the program being compiled. The intermediate code optimization phase searches for possible machine independent optimizations to improve the efficiency of the code. After these optimizations, the code generation phase produces the target code for the machine to execute. Before outputting a target program, machine dependent optimizations are performed.

This partitioning of the compilation process into a sequence of phases has led to the development of efficient algorithms and tools for building components for each of the phases [3]. Since the front-end processes for quantum compilers are similar to those of classical compilers, we can use the algorithms and tools for building lexical, syntactic and semantic analyzers for QCCs. However, the intermediate representations and the optimization and code generation phases of QCCs are very different from classical compilers and require novel approaches, such as a way to insert error-correction operations into the target language program. This section describes the current state of the art concerning intermediate representations, code generation, and optimization.

## 4.1 Front End and the Quantum Intermediate Representation

In the first phase of our QCC, a high-level specification of a quantum algorithm is mapped into a quantum intermediate representation (QIR) that is based on the quantum circuit model [17]. Since other representations of quantum computation, such as adiabatic quantum computing, can be converted to the quantum circuit model, this is an appropriate formalism. In particular, by choosing the circuit formalism, we can consider fault-tolerant constructions at various phases in our design flow (see Section 8). Also, by using the quantum circuit model, we can incorporate circuit synthesis and optimization techniques in both the technology-independent and technology-dependent phases of our design flow.

## 4.2 Circuit Synthesis and Optimization

The second and third phases of our QCC synthesize and optimize a QASM representation of a quantum circuit. In this section, we discuss possible optimization and synthesis procedures that can be applied in these two phases, by analogy with classical chip design techniques. Algorithms for classical logic circuit synthesis [13] map a Boolean function into a circuit that implements the function using gates from a given gate library. These algorithms are typically applied after high-level synthesis or in conjunction with compilers in traditional chip design methodologies. Similarly, we can talk about quantum circuit synthesis, where a quantum circuit is created that "computes" a given unitary matrix, up to a relative phase or up to a prescribed quantum measurement.

Given the truth table of a Boolean function, a two-level circuit, linear in the size of the truth table, can be constructed immediately. Yet, the optimization of the circuit structure is nontrivial. In contrast, given a $2^n \times 2^n$ unitary matrix, it is not even easy to find a good quantum circuit to implement it. Only very recently have constructive algorithms been developed that yield an asymptotically optimal circuit with $4^n$ gates for a $2^n \times 2^n$ unitary matrix [30]. However, the constant numerical factor between the lower and upper bounds remains high, except for special cases, such as two-qubit circuits [26] and for diagonal circuits [6]. An arbitrary two-qubit operator requires up to three CNOT gates, and either six additional generic (basic) single-qubit gates or fifteen additional elementary single-qubit gates.

The difference between basic and elementary gates deserves particular attention. Basic gates can be decomposed, up to phase, into a product of one-parameter rotations according to the Euler-angles formula [17]. Therefore we view only one-parameter rotations as elementary. Some results in terms of such elementary gates can be reformulated in terms of coarser gates, but coarser gates do not always correspond to realistic costs of physical implementations. It is thus necessary in the third phase to map the representation into a universal set of gates depending on the choice of a particular technology. For example, single-qubit basic gates appear equally cheap in many NMR implementations [17]. However, when working with ion traps, $R_z$ gates are significantly easier to implement than $R_x$ and $R_y$ gates [35].

When developing reusable software for automating the design of quantum circuits, it is desirable, to some extent, to avoid such fundamental dependence on technology. Indeed, this problem is not new. Recall the standard choice of elementary logic gates in classical computing (AND-OR-NOT) was suggested in the $19^{th}$ century by Boole for abstract reasons rather than based on specific technologies. Today the NAND gate is easier to implement than the AND gate in CMOS-based integrated circuits. This fact is addressed by commercial circuit synthesis tools by decoupling *libraryless logic synthesis* from *technology-mapping* [13]. The former uses an abstract gate library, such as AND-OR-NOT, and emphasizes the scalability of synthesis algorithms that capture the global structure of the given computation. The latter step maps logic circuits to a technology-specific gate library, often supplied by a semiconductor manufacturer, and is based on local optimizations. Technology-specific libraries may contain composite multi-input gates with optimized layouts such as the `AOI` gate (`AND/OR/INVERTER`). From a theoretical point of view, re-expression of circuits in terms of a different set of universal gates may increase circuit sizes by at most a constant, under certain reasonable assumptions about the gate libraries involved.

We expect the distinction between technology-independent circuit synthesis and technology mapping to carry over to quantum circuits as well. To this end, the work in [26] shows that basic-gate circuits can be simplified by temporarily decomposing basic gates into elementary gates, so as to apply convenient circuit identities. This is precisely our reason, in the second phase, for mapping the quantum algorithm into a QASM representation consisting of single-qubit and CNOT gates. Indeed, all lower bounds for two-qubit circuits from [26] and also their lower bound for the number of CNOTS in an $n$-qubit circuit ($\lceil(4^n-3n-1)/4\rceil$) rely on such circuit identities. Additionally, temporary decompositions into elementary gates may help optimizing pulse sequences in physical implementations. In terms of technology mapping in the third phase, the work in [26] shows how to map a CNOT gate into a specific implementation technology by appropriately timing a given two-qubit Hamiltonian and applying additional single-qubit operators. Circuits resulting after such substitutions may potentially be optimized further.

Ongoing work in quantum circuit synthesis and optimization involves automatically instantiating error-correction, a potentially key feature for scalable quantum computing. Additionally, circuit synthesis and optimization with discrete gate libraries, required for the technology-dependent optimization phase, remains largely unexplored. To this end, we point out that the Gottesman-Knill theorem [11] suggests a mapping between stabilizer circuits, key to quantum error-correction, and classical reversible circuits consisting of CNOT and Toffoli gates. Optimizing the latter [27] could help optimizing stabilizer circuits. An alternative approach to optimizing stabilizer circuits was suggested by Aaronson [1] and entails partitioning them into eight groups of Phase, Hadamard and CNOT gates (H-P-C-P-H-P-C-P). Given that the Phase and Hadamard are single-qubit gates, the respective groups cannot have more than linear size (in the number of qubits) after cancellations. Thus, all the complexity is concentrated in two CNOT groups, to which the asymptotically optimal algorithm from [21] can be applied to find circuits with at most $cn^2/\log_2(n)$ gates. Another interesting observation is that the work in [27] proposes to partition classical reversible circuits into four groups Toffoli, CNOT, NOT, Toffoli (T-C-N-T). The study of such universal circuit partitions may lead to canonical forms, potentially useful in optimizing compilers. Additionally, circuit partitions suggest circuit layouts and can lead to new architectures for reprogrammable circuits. Methods such as those discussed above will be needed in the second and third phases of our QCC to produce good circuits for implementation on a quantum device.

## 5 A Quantum Assembly Language (QASM)

The technology-independent phase of our QCC maps a representation of the quantum algorithm into an equivalent set of quantum assembly language (QASM) instructions. QASM is an extension of a RISC assembly language with the classical RISC instruction sets and a set of quantum instructions based on the quantum circuit model of quantum computation. Just as the quantum circuit model contains qubits and classical bits (cbits), QASM uses qubits and cbits as the basic units of information. To ensure the usability of the classical RISC instruction set, each cbit is stored in a separate classical register. However, there are no quantum registers in QASM. Each qubit and classical register used in a QASM program is a static resource and must be declared at the beginning of the program.

Quantum operations consist solely of unitary operations and measurements. In QASM, each unitary operator is expressed in terms of an equivalent sequence of single-qubit and CNOT gates. We have chosen these gates as the universal gate set since it can express any quantum circuit exactly. The single-qubit operations are stored as a triple of rationals, since single-qubit operations are specified by three Euler-angles. We assume that each rational number in the triple is implicitly multiplied by $\pi$. The non-reversible measurement operation on a single-qubit copies the measurement result to a classical register by first zeroing out the classical register and then copying the measurement result into the classical register. The qubit is also set to the resulting value of the measurement. In QASM, classical control of a qubit is performed by conditioning off of the classical register with an OR of the bit values. Thus, in order to classically control with the result of multiple measurements, the bits must be masked and shifted into a single register.

## 6 A Quantum Physical Operations Language (QCPOL)

The quantum physical operations language (QCPOL) describes precisely how a given quantum circuit should be executed on a particular physical system. The instruction set of a physical operations language is specific to the target system and contains enough low-level information to execute the quantum circuit unambiguously. We now describe the general properties of a QCPOL and proceed to a specific example of a physical language for the trapped-ion quantum processor.

### 6.1 General Properties of a QCPOL

We classify physical operations as initialization, computation, communication, classical control, and system-specific instructions. This classification admits devices that have realized quantum information processing to date and is general enough to admit future devices.

Initialization instructions specify how to prepare the initial state of the system. These include operations to load qubits into the system and put those qubits into a valid, known starting state. In practice this requires manipulating the physical qubit carriers and controlling the carrier's degrees of freedom that might affect an internal qubit or the ability to control an internal qubit.

Computation instructions include both gates and measurements. For most physical systems gates correspond to controlled electromagnetic pulses. Gate types and speeds depend strongly on the interaction that couples qubits, so typical systems permit only a limited set of gates. Measurement methods rely on coupling to a measurement apparatus and will be limited to particular operators in practice as well.

Movement instructions control the relative distance between qubits, bringing groups of qubits together to participate in local gates. Some systems have stationary qubit carriers and will spend a majority of their time performing swap gates. Other systems have mobile qubit carriers, or perhaps a mixture of mobile and immobile carriers. These systems will have machine-specific movement instructions.

Quantum information processors will contain at least a subset of classical logic operations. In the simplest case, quantum processors will be controlled by external computers and have access to a complete classical instruction set. Future quantum processors, however, may have integrated classical logic with specific low-level functions and interfaces.

Finally, a physical operations language includes system-specific instructions that may not fall into general categories. These instructions are likely to control other degrees of freedom of the qubit carriers or nondestructively detect the presence or absence of carriers.

QCPOL instructions are organized into a coherent program by specifying the starting time and duration of each instruction. If the machine can operate in parallel, instructions can be organized into streams or groups that execute simultaneously. Essentially, physical constraints yield further semantic constraints.

## 6.2 QCPOL Example: Trapped-ion QCPOL

Physical operations languages may vary considerably over different physical systems. This section describes one example of a physical operations language for the operation of a trapped-ion quantum computer system.

Trapped-ion devices use charged, electromagnetically trapped atoms as qubit carriers. Each qubit is represented by internal electronic and nuclear states of a single ion. Laser pulses of specific frequencies addressing one or more ions apply single and multi-qubit quantum gates. Laser pulses, appropriately tuned, can also perform measurement, by causing ions to fluoresce when they are in the $|0\rangle$ state. Two or more ions can be contained in a single trap, where they couple to each other through Coulomb repulsion, thus providing a qubit-qubit interaction through their collective vibrational modes. These modes can serve as a “bus” qubit, as long as ion temperatures are kept low, and vibrational states controlled. We say that ion-qubits are *chained* if they are close enough to interact using the bus qubit. This bus qubit is also manipulated optically using *sideband* laser pulses.

Trapped-ion systems have shown considerable potential as a future technology for quantum information processing. Several groups have demonstrated a universal set of gates and measurements for trapped-ion quantum information processing [16, 24], including basic multi-qubit quantum algorithms, and recently, quantum teleportation. Further, experiments have demonstrated that static voltages can move ion-qubits between traps [22]. Together these experiments offer a route toward a scalable system, possibly configured in a large microarray akin to charge-coupled-devices [14].

We have designed an instance of QCPOL targeted to ion traps, consisting of initialization, computation, movement, classical control, and system-specific instructions.

*Initialization* of an ion trap processor has two stages: loading of multiple ions into a special loading region, and laser cooling to reduce ion temperatures. Measurement is then performed, followed by conditional rotations, to put all qubits in the $|0\rangle$ state.

*Computation* with quantum gates is naturally described in terms of single-qubit rotations in the $\hat{x} - \hat{y}$ plane, achieved using pulsed laser excitation, and a controlled-phase gate between ions in the same trap, implemented using three sideband pulses. Chained ions may also participate in a multiply-controlled phase gate, useful for creating large entangled states [28]. Qubit readout with a readout laser pulse is described by a projective measurement.

*Movement* of ions is accomplished into and out of traps (and chains) using electrostatic fields. We assume a set of movement instructions sufficient for a planar rectangular trap configuration with “T” and “X” junctions. Additional splitting and joining instructions separate and rejoin chains.

*Classical control* of ions is assumed to be universal, and implemented by a fast, external classical processor. In practice, this can either be a remote control PC, or a local microprocessor chip integrated nearby the trap.

*System-specific instructions* for trapped ions are necessary to deal with the heating and decoherence of ion-qubits and bus qubits caused throughout a computation, in the movement, splitting, and joining operations. In the worst case, high temperatures may eject ions from the trap. Thus, the instruction set includes a system-specific method to reinitialize the bus qubit, using recooling pulses. These are also sideband pulses like those used in multi-qubit gates, but they are applied differently and must be treated specially by the design tools.

# 7 High-performance Simulation of Quantum Circuits

The intrinsic computational difficulty of simulating quantum computation on classical computers was pointed out by Richard Feynman in the 1980s [8]. Moreover, this led him to suggest the use of quantum-mechanical effects to speed up classical computing. Even though such speed-ups have been theoretically identified by Shor and Grover, numerical simulation of quantum computers on classical computers remains attractive for engineering reasons. Similarly, in classical Electronic Design Automation, chip designers always test independent modules and complete systems by simulating them on test vectors, before costly manufacturing. Numerical simulations can also help to evaluate “quantum heuristics” that defy formal worst-case analysis or only work well for a fraction of inputs.

For the numerical simulation phase, the quantum circuit formalism seems most suitable for reasons discussed in Section 4.1. Mathematical models of quantum states and circuits involve linear algebra [17]: $n$-qubit quantum states can be represented by $2^n$-dimensional vectors, and gates by square matrices of various sizes. The parallel composition of gates corresponds to the tensor (Kronecker) product, and serial composition to the ordinary matrix product. A quantum circuit can be simulated naively by a sequence of $2^n \times 2^n$ matrices that are applied sequentially to a state vector. This reduces quantum simulation to standard linear algebraic operations with exponentially sized matrices and vectors. Quantum measurement is also simulated via linear algebra. A key insight to efficient simulation is to use structure in matrices and vectors that may arise from quantum circuits. To this end, polynomial-time simulation techniques were proposed for circuits with restricted gate types [11] and for “slightly entangled” quantum computation [33]. The Gottesman-Knill technique [11] targets circuits used for quantum error-correction and states that can be produced with such circuits (stabilizer states). In the next sections, we describe two possible simulation tools for use in the fourth phase of our design flow.

## 7.1 QuIDDPro: A Generic Graph-based Simulator

Viamontes et al. [31] proposed a generic simulation technique based on data compression using the QuIDD data structure. Its worst-case performance is no better than what can be achieved with basic linear algebra, but it can dramatically compress structured vectors and matrices, including all basis states, small gates and some tensor products. A QuIDD is a directed acyclic graph with one source and multiple sinks, where each sink is labeled with a complex number. Matrix and vector elements are modeled by directed paths in the graph; any given vector or matrix can be encoded as a QuIDD, and vice versa (subject to memory constraints). Surprisingly, all linear-algebraic operations can then be implemented as graph algorithms in terms of compressed data representations.

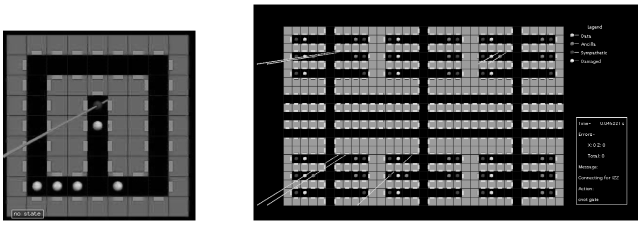
QuIDDs simulate a useful class of quantum circuits using time and memory that scale polynomially with the number of qubits [31]. For example, all the components of Grover's algorithm, except for the application-dependent oracle, fall into this class. Furthermore, QuIDD-based simulation of Grover's algorithm requires time and memory resources that are polynomial in the size of the oracle $p(\cdot)$ function represented as a QuIDD [31]. Thus, if a particular $p(\cdot)$ for some search problem can be represented as a QuIDD using polynomial time and memory resources (including conversion of an original specification into a QuIDD), then classical simulation of Grover's algorithm performs the search nearly as fast as an ideal quantum circuit. That is because the complexity of Grover's algorithm is dominated by the number of iterations.

Unlike other simulation techniques proposed in the physics literature and tied to the state-vector representation, QuIDDs are a formal data structure for compressed linear algebra operations used in quantum computing. Extending QuIDDs to simulate density matrices only requires implementing several additional operations, such as trace-overs, in terms of graph traversals. Such extensions have been described in [32], along with empirical performance results on several types of circuits up to 24 qubits. For a comparison, straightforward modelling of any 16-qubit density matrix would require 64TB of memory. For a reversible 16-qubit adder circuit that uses CNOT and Toffoli gates, the QuIDDPro package requires less than 2MB of memory. This package is currently available from the authors with an ASCII front end that supports input language similar to Matlab.

## 7.2 Layout Tools, Scheduling Tools, and the Trapped-Ion Simulator

This section discusses initial tools that we have implemented for studying trapped-ion systems. Layout, scheduling, and simulation tools can be used to aid in design and testing of large-scale devices or device components. Layout tools use the structure of a quantum network to infer both an initial geometric arrangement of qubits and the valid locations that qubits can be moved to during computation. Scheduling tools determine movement patterns during a computation and insert machine-specific operations related to movement. Finally, simulation tools evaluate the layout and movement sequence.

A general layout tool that we have implemented maps an arbitrary quantum circuit onto a Turing machine with a single head and a circular tape (Figure 2(a)). The simple structure of the single-headed Turing machine makes scheduling operations particularly simple. Qubits are moved into the head before each multi-qubit gate and returned to the tape after the gate.



Time- 0.045221 s
Errors-
X: 0 Z: 0
Total: 0
Message:
Connecting for IZZ
Action:
cnot gate

(a) (b)

Figure 2: (a) Turing machine layout for a trapped ion processor produced by an automatic layout tool. (b) Snapshot of the simulator graphical display showing an H-tree layout. Qubits are ions represented by spheres, and gates are applied using laser pulses, represented by lines. The qubits can move within the black regions of the figure and are prohibited from moving into the substrate which is drawn using light squares. In the right window the simulator displays feedback regarding the current operations, noise induced failures, and estimated execution time.

We have also implemented a more specialized layout tool that maps circuits constructed from concatenated quantum codes onto an *H-tree*. An H-tree is constructed recursively in the same way as a Koch curve or other fractal. Concatenated quantum codes have a self-similar structure, so fewer movement operations are required per gate because qubits for the inner codes are kept near one another. We currently schedule operations onto the H-tree by specifying paths for basic operations, but expect that this procedure can be automated by use of more sophisticated techniques.

Both of these layout and scheduling methods lead to physical operation sequences that can be simulated to verify correctness, reliability, and total running time. For large-scale trapped ion systems, we have developed a simulation tool that implements the model of a trapped-ion quantum computer described in Section 6. The simulator accepts a layout and a QCPOL program implementing a quantum circuit, each generated by scheduling and layout tools. Simulator output includes the final quantum state, single-shot measurement and failure histories, total execution time, and overall circuit reliability. In addition, the simulator can graphically display the QCPOL instructions as they are simulated. An example of this output is shown in Figure 2(b).

We believe that the performance of quantum-circuit simulators and layout tools can be significantly improved in the near future. These improvements will stem from better algorithms and from deeper understanding of structure present in useful quantum circuits. A particularly interesting approach suggested recently by Aaronson [1] is to automatically restructure a given quantum circuit so that the new circuit produces the same output, but is easier to simulate (e.g., has fewer gates). We are currently exploring another connection between simulation and synthesis where optimized simulation primitives directly support only a small universal set of quantum gates. Simulating more complicated gates requires decomposing them into low-level primitives [4, 26]. Such decompositions also appear central from the physics perspective, where a given Hamiltonian can be numerically simulated after being decomposed into a quantum circuit.

# 8 Design Flow for a Fault-tolerant Ion-trap Architecture

In this section, we introduce the concept of fault-tolerance and detail a process that inserts fault-tolerant components. The process can be applied manually by a system architect in a hardware-driven design flow, or it can be applied automatically by a compiler in a software-driven design flow. Both of these design flows are being implemented specifically for trapped-ion systems, though the principles may extend to other devices. The central goal of both design approaches is to guarantee that the final stream of physical operations will execute fault-tolerantly. These examples demonstrate the utility and generality of the high-level design flow for organized study of quantum device design approaches.

## 8.1 Classical Fault-Tolerant Components

We begin by reviewing triple modular redundancy (TMR) as the canonical method for implementing fault-tolerant computation in modern digital computers [34, 36]. If the basic gates in digital computers were not naturally error-rejecting, this construction would be a standard element of computer architectures. To construct a fault-tolerant gate, the inputs are encoded using the TMR code and fed into three basic gates. The output lines of each logic gate fan-out into three majority voting gates. The majority gates output the encoded computation result.

By applying this fault-tolerant procedure recursively $k$ times, as illustrated in Figure 3, fault-tolerant components can be made to fail with probability $(cp)^{2^k}/c$ for a constant $c$ determined by the number of ways basic gates in the component can fail. For the NAND gate, $c = 6$ because there are 6 ways for two basic gate failures to cause a component failure. If each basic gate fails with probability $p < 1/6$ then the fault-tolerant NAND can be made arbitrarily reliable in principle. We say that this construction exhibits a *fault-tolerance threshold* $p_{th} = 1/6$. The fault-tolerant NAND component must then be placed onto a layout in such a way that wire delays are not too long and cross-talk does not introduce too much additional noise.

## 8.2 Fault-Tolerant Quantum Components

Fault-tolerant quantum components are constructed using similar procedures to those used for classical fault-tolerance. Quantum information can be encoded using *quantum computation codes* [2] that allow fault-tolerant computation using a discrete universal set of gates. The CSS codes [7, 29] are one family of quantum computation codes with the useful property that CNOT gates can be applied *transversally* [9]. Transversal gates are always fault tolerant since they are implemented in a bitwise fashion. Figure 4(a) illustrates transversal Hadamard and CNOT gates.

Nontrivial fault-tolerant quantum gates such as the Toffoli gate (Figure 4(b)) can be constructed using a general method based on quantum teleportation [37]. Fault-tolerant gates constructed in this manner consist entirely of Clifford group gates and measurements, both of which can be applied transversally. These gates are applied to a tensor product of input qubit states and a specially-prepared ancilla state.

Performing these gates in practice requires fault-tolerant preparation of several kinds of ancilla. First, a specially-prepared ancilla state must be prepared for each nontrivial

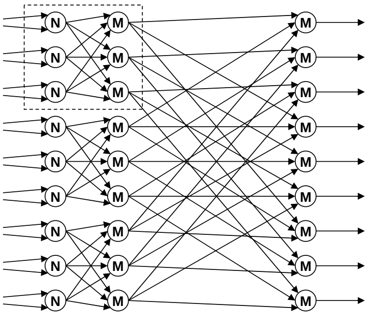


Figure 3: A TMR FAULT-TOLERANT NAND GATE AT THE SECOND LEVEL OF RECURSION, CONSTRUCTED FROM THREE FAULT-TOLERANT NAND GATES. N AND M DENOTE NAND AND MAJORITY GATES. ALL GATES ARE ASSUMED TO FAIL WITH PROBABILITY $p$, SUCH THAT THE BOXED TMR NAND GATE (UPPER LEFT) FAILS WITH PROBABILITY $< 6p^2$. THE ENTIRE CIRCUIT SHOWN IN THIS FIGURE FAILS WITH PROBABILITY $< 6^3p^4$. IF $p < 1/6$ THEN THIS CIRCUIT IS MORE RELIABLE THAN A BASIC GATE.

gate, like the Toffoli gate or the $\pi/8$ gate. These states can be prepared fault-tolerantly through measurement. A recovery operation on each qubit has to follow each nontrivial gate. The generic structure of a recovery operation is shown in Figure 5(a). Recovery operations consume a syndrome extraction ancilla for every syndrome bit they acquire. This ancilla must also be prepared fault-tolerantly and be available in great supply. Figure 5(b) also shows the generic structure of a single syndrome bit extraction. The final ancilla, a cold verification qubit, must be available for every extraction ancilla in order to check for critical errors.

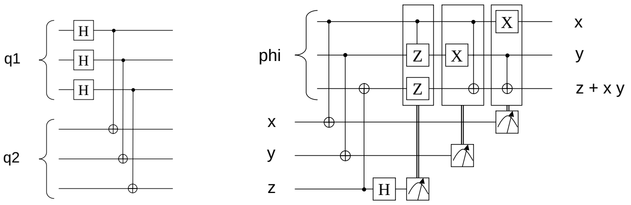
All of these operations must remain fault tolerant when qubits can only interact locally. If the target machine permits a geometry that allows frequent *intermediate error-correction*, then movement errors can be corrected on each level of code concatenation before becoming too large [12]. This can be true for a device with an engineered layout, such as the “quantum CCD” proposal for a trapped ion quantum device [14].

## 8.3 Design flows

We now describe the software and hardware driven design flows in more detail. Both approaches fundamentally apply the same replacement rules described in this section by taking advantage of the conceptual separation of the logical and physical machine.

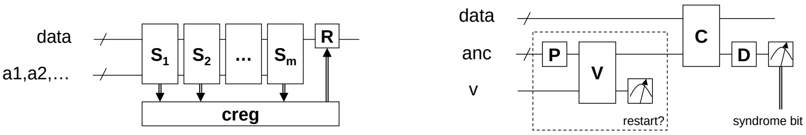
A software-driven design flow applies the replacement rules we have described to insert fault-tolerance before technology-dependent code is generated. Scheduling algorithms and layout tools like those in Section 7.2 both minimize movement and insert machine-specific instructions to preserve fault-tolerance. The software approach relies on fine-grained replacements and transformations at the lowest levels of the design flow. These tools may operate in several stages but must ultimately generate physical operations in the QCPOL language because moving qubits may make frequent use of device dependent details.

In a hardware-driven design flow, a system architect creates universal, fault-tolerant,



(a) (b)

Figure 4: (a) Transversal Hadamard and CNOT gates acting on two logical qubits represented by 3 physical qubits. Transversal CNOT is a valid operation for a quantum code iff that code is a CSS code [9]. Transversal Hadamard is a valid operation for a doubly-even self-dual CSS code. (b) Fault-tolerant Toffoli gate constructed using quantum teleportation [37]. Each gate in this circuit can be applied transversally when using a punctured, self-dual, doubly-even CSS code such as the $[\![7,1,3]\!]$ code [10]. The gate requires an ancilla $|\phi\rangle$ that can be prepared fault-tolerantly as well by applying Clifford group operators and fault-tolerant measurements of Clifford group operators.



(a) (b)

Figure 5: The left network (a) is a *recovery network*. A recovery operation interacts fault-tolerantly with the data via syndrome bit extraction networks $S_i$. Each syndrome bit is measured, possibly several times, and stored to a classical register. A classical computer processes the register and applies the appropriate error correction $R$ to the data. Recovery operations must follow every fault-tolerant gate to correct errors introduced by that gate. The right network (b) is a *syndrome extraction network*. Extracting a single syndrome bit fault-tolerantly first requires an ancilla state $|anc\rangle$. The ancilla is prepared and verified by the network in the outlined box. A verification qubit indicates if the verification failed. Once an ancilla has been successfully prepared, the $C$ network interacts with the data fault-tolerantly to collect a syndrome bit. This bit is decoded by $D$ and measured. Some classical post processing may take the place of the quantum network $D$.

technology-specific components through a combination of replacement rules and heuristic methods. The set of components is published together with design rules for connecting them. The lowest levels of the design flow then target the machine architecture rather than the physical device. The hardware approach abstracts the technological details of fault-tolerance into a machine architecture and supplies coarse component placement rules.

## 9 Conclusions and Important Challenges

This paper has presented a design flow in which a high-level language representing a quantum algorithm is mapped into a quantum device or quantum device simulator. The paper has focused on the languages and notations needed along the design flow and open problems that need to be solved to make this design flow a reality. We conclude by listing the most important challenges.

1. Design a high-level programming language for creating quantum algorithms that encapsulates the principles of quantum mechanics such as superposition and entanglement in a natural way for physicists *and* programmers.
2. Find efficient technology-independent optimization algorithms that work well on realistic classes of quantum circuits, and develop strategies for adapting generic circuits to specific implementation technologies.
3. Develop simulation techniques for quantum circuits and high-level programs that will allow designers to evaluate meaningful design blocks.
4. Identify fault-tolerant architectural strategies that can be used with emerging quantum device technologies such as ion traps.
5. Find efficient optimization algorithms for fault-tolerant circuits that minimize the number of fault paths, size of code and the number of gates.

As the scale of quantum computing increases, design verification and device testing may also require software support, e.g., for circuit-equivalence checking and test-vector generation. However, such topics have not been widely covered in quantum computing literature, and there still seems to be little empirical context for these aspects of automation.

**Acknowledgements** The authors are grateful to Stephen Edwards for many helpful comments on computer-aided design flows.

## References

[1] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Unpublished*, 2003.

[2] D. Aharonov and M. Ben-Or. Fault tolerant computation with constant error. *Proc. ACM Symposium on the Theory of Computing (STOC)*, pages 176–188, 1997.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] A. Barenco et al. Elementary gates for quantum computation. *PRA*, 52:3457–3467, 1995. arXive e-print quant-ph/9503016.

[5] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *Eur. Phys. J.*, 25:181–200, 2003.

[6] S. S. Bullock and I. L. Markov. Asymptotically optimal circuits for arbitrary n-qubit diagonal computations. *Quantum Inf. and Computation*, 4(1):27–47, January 2004.

[7] A. R. Calderbank and P. W. Shor. *Good quantum error-correcting codes exist*. *Phys. Rev. A.*, 54:1098–1105, 1996.

[8] A. J. G. Hey ed. *Feynman and Computation: Exploring the Limits of Computers*. 1999.

[9] D. Gottesman. Stabilizer codes and quantum error correction. *PhD thesis, Cal. Inst. Tech*, 1997.

[10] D. Gottesman. Theory of fault-tolerant quantum computation. *Phys. Rev. A*, 57:1, January 1998.

[11] D. Gottesman. The Heisenberg representation of quantum computers. *Intl. Conf. on Group Theoretic Methods in Physics*, 1998. quant-ph/9807006.

[12] D. Gottesman. Fault-tolerant quantum computation with local gates. *Unpublished*, 1999. arXive e-print quant-ph/9903099.

[13] G. Hachtel and F. Somenzi. *Synthesis and Verification of Logic Circuits*. Kluwer, 2000.

[14] D. Kielpinski, C. Monroe, and D.J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417:709–711, 2002.

[15] E. Knill. Conventions for quantum pseudocode. *Technical Report LAUR-96-2186, Los Alamos National Laboratory*, 1996.

[16] D. Leibfried et al. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, 422:412–415, 2003.

[17] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, England, 2000.

[18] B. Oemer. A procedural formalism for quantum computing. *Ph.D. Thesis, Univ. of Vienna*, 1998.

[19] B. Oemer. Quantum programming in QCL. *Masters Thesis, Univ. of Vienna*, 2000.

[20] M. Oskin and A. Petersen. A new algebraic foundation for quantum programming languages. *Second Workshop on Non-Silicon Computing*, 2003.

[21] K. N. Patel, I. L. Markov and J. P. Hayes. Efficient synthesis of linear reversible circuits. *Intl. Workshop on Logic and Synthesis*, June 2004. arXive e-print quant-ph/0302002.

[22] M. A. Rowe et al. Transport of quantum states and separation of ions in a dual RF ion trap. *Quantum Information and Computation*, 2:257–271, 2002. arXive e-print quant-ph/0205094.

[23] J. Sanders and P. Zuliani. Quantum programming. *Technical report, Oxford Univ.*, 1999.

[24] F. Schmidt-Kaler et al. Realization of the Cirac-Zoller controlled-NOT quantum gate. *Nature*, 422:408–411, 2003.

[25] P. Selinger. Towards a quantum programming language. *Math. Struct. in Comp. Sci.*, 2004.

[26] V. V. Shende, I. L. Markov and S. S. Bullock. Finding small two-qubit circuits. *Proc. SPIE*, volume 5436, April 2004.

[27] V. V. Shende, A.K. Prasad, I. L. Markov, and J. P. Hayes. Synthesis of reversible logic circuits. *IEEE Trans. on CAD* 22, pp. 710–722, June 2003.

[28] A. Sorensen and K. Molmer. Entanglement and quantum computation with ions in thermal motion. 2000. arXive e-print quant-ph/0002024.

[29] A. M. Steane. Error correcting codes in quantum theory. *Phys. Rev. Lett.*, 77:793, 1996.

[30] J. J. Vartiainen, M. Mottonen, and M. Salomaa. Efficient decomposition of quantum gates. *Phys. Rev. Lett.*, 92:177902, 2004.

[31] G. F. Viamontes, I. L. Markov, and J. P. Hayes. More efficient gate-level simulation of quantum circuits. *Quantum Info. Processing*, 2(5):347–380, 2003. arXive e-print quant-ph/0309060.

[32] G. F. Viamontes, I. L. Markov, and J. P. Hayes. Graph-based simulation of quantum computation in the state-vector and density-matrix representation.*Proc. SPIE*, 5436, April 2004.

[33] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Phys. Rev. Lett.*, (91):147902, 2003.

[34] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies,* Princeton Univ. Press, 329–378, 1956.

[35] D. J. Wineland et al. Experimental issues in coherent quantum-state manipulation of trapped atomic ions. *Journal of Research of the National Institute of Standards and Technology*, 103:259–328, 1998.

[36] S. Winograd and J. D. Cowan. Reliable computation in the presence of noise. MIT Press, Cambridge, MA. 1967.

[37] X. Zhou, D. Leung, and I. L. Chuang. Methodology for quantum logic gate construction. *Phys. Rev. A* 62:52316, 2000.

[38] P. Zuliani. Quantum programming. *PhD thesis, St. Cross, Oxford Univ.*, 2001.