Simulating and Compiling Code for the Sequential Quantum Random Access Machine

Rajagopal Nagarajan
Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom
biju@dcs.warwick.ac.uk

Nikolaos Papanikolaou
Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom
nikos@dcs.warwick.ac.uk

David Williams
School of Informatics
City University
London EC1V 0HB
United Kingdom
david@david-williams.info

Abstract

We present the SQRAM architecture for quantum computing, which is based on Knill's QRAM model. We detail a suitable instruction set, which implements a universal set of quantum gates, and demonstrate the operation of the SQRAM with Deutsch's quantum algorithm.

The compilation of high-level quantum programs for the SQRAM machine is considered; we present templates for quantum assembly code and a method for decomposing matrices for complex quantum operations. The SQRAM simulator and compiler are discussed, along with directions for future work.

1. Introduction

The rapidly growing field of quantum computation and quantum information is still in its infancy, largely due to the lack of a substantial, practical quantum computing device. However, the theoretical potential of such devices is widely acknowledged. Presently, the only realistic avenue of investigation for an interested computer scientist is the use of quantum computer simulators.

Owing to the large state spaces of quantum–mechanical systems, a complete simulator of subatomic phenomena cannot be implemented efficiently on a classical computer. Nobel laureate Richard Feynman observed in 1985 that [6]:

> "…if a description of an isolated part of Nature with N particles requires a general function of N variables and if a computer simulates this by actually computing or storing this function then doubling the size of Nature ($N \to 2N$) would require an exponentially explosive growth in the size of the simulating computer."

Focusing on quantum mechanics in particular, Feynman points out that:

> "…the full description of quantum mechanics for a large system with R particles is given by a function $\psi(x_1, x_2, \ldots, x_R, t)$ which we call the amplitude to find the particles at $x_1, x_2, \ldots, x_R$ and therefore, because it has too many variables, it *cannot be simulated* with a normal computer with a number of elements proportional to R […]."

Our goals in this paper are substantially more modest; we are interested in local quantum computation on a finite number of quantum bits (*qubits*). In particular, we will discuss the design of a hybrid classical–quantum computer architecture, which we will call the Sequential Quantum Random Access Memory machine, or SQRAM for short. The SQRAM design is based on Knill's QRAM model [8]. In addition, we will define an instruction set for a hypothetical implementation of the SQRAM, and illustrate the operation of such a device when running Deutsch's algorithm for determining the balance of a boolean function [9]. We have implemented a simulator of the SQRAM machine using the OpenQubit library [12].

In light of recent proposals for quantum programming languages, including QPL [13], QCL [10], CQP [7] and qSpec [11], we feel it is suitable to consider compilation of high–level quantum programs; we discuss techniques for this and present a compiler we have developed for a subset of QPL.

We begin with a summary of basic quantum computing concepts. We will then proceed to describe the proposed SQRAM architecture and instruction set; this is followed by a walkthrough of Deutsch's algorithm, as implemented on the SQRAM. Finally, we will turn to compilation of high–level quantum programs in QPL, a functional quantum programming language due to Selinger.

2. Related Work

Currently several quantum simulators are available, including tools for analysing quantum circuits and interpreters for quantum programming languages [3, 10].

With the notable exception of a joint Columbia and MIT project [14], there has been little work to date on the development of a quantum computer architecture which is realisable using current technology. In [14], a multi-layer framework is defined, which models different levels of abstraction for a quantum computer simulator; however, the authors account for specific aspects of physical implementation; on the contrary, we simply rely on the hypothesis that the proposed system architecture may be implemented with present–day hardware, and do not concern ourselves with details of the physics.

3. Quantum Computing Fundamentals

A few preliminaries are in order; we are assuming no prior knowledge of quantum computing.

A *quantum bit* or *qubit* is a physical system which has two basis states, conventionally written $|0\rangle$ and $|1\rangle$, corresponding to one-bit classical values. These could be, for example, spin states of a particle or polarization states of a photon, but we do not consider physical details. According to quantum theory, a general state of a quantum system is a *superposition* or linear combination of basis states. A qubit has state $\alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$; states which differ only by a (complex) scalar factor with modulus 1 are indistinguishable. States can be represented by column vectors: $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right] = \alpha|0\rangle + \beta|1\rangle$. Formally, a quantum state is a unit vector in a Hilbert space, i.e.

a complex vector space equipped with an inner product satisfying certain axioms.

The basis $\{|0\rangle, |1\rangle\}$ is known as the *standard* basis. Other bases are sometimes of interest, especially the *diagonal* (or *dual*, or *Hadamard*) basis consisting of the vectors

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Evolution of a closed quantum system can be described by a *unitary transformation*. If the state of a qubit is represented by a column vector then a unitary transformation U can be represented by a complex-valued matrix (u_{ij}) such that $U^{-1} = U^*$, where U^* is the conjugate-transpose of U (i.e. element ij of U^* is $\bar{u}_{ji}$). U acts by matrix multiplication:

$$\begin{bmatrix}\alpha' \\ \beta'\end{bmatrix} = \begin{bmatrix}u_{00} & u_{01} \\ u_{10} & u_{11}\end{bmatrix}\begin{bmatrix}\alpha \\ \beta\end{bmatrix}$$

A unitary transformation can also be defined by its effect on basis states, which is extended linearly to the whole space. For example, the *Hadamard* operator is defined by

$$\begin{array}{lcl}|0\rangle & \mapsto & |+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \\ |1\rangle & \mapsto & |-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\end{array}$$

which corresponds to the matrix $H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1 \\ 1 & -1\end{bmatrix}$. The *Pauli* operators, denoted by $\sigma_0, \sigma_1, \sigma_2, \sigma_3$, are defined by

$$\sigma_0 = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} \qquad \sigma_1 = \begin{bmatrix}0 & 1 \\ 1 & 0\end{bmatrix}$$
$$\sigma_2 = \begin{bmatrix}0 & -i \\ i & 0\end{bmatrix} \qquad \sigma_3 = \begin{bmatrix}1 & 0 \\ 0 & -1\end{bmatrix}$$

Measurement plays a key role in quantum physics. If a qubit is in state $\alpha|0\rangle + \beta|1\rangle$ then measuring its value gives the result 0 with probability $|\alpha|^2$ (leaving it in state $|0\rangle$) and the result 1 with probability $|\beta|^2$ (leaving it in state $|1\rangle$).

For example, if a qubit is in state $|+\rangle$ then a measurement (with respect to the standard basis) gives result 0 (and state $|0\rangle$) with probability $\frac{1}{2}$, and result 1 (and state $|1\rangle$) with probability $\frac{1}{2}$. If a qubit is in state $|0\rangle$ then a measurement gives result 0 (and state $|0\rangle$) with probability 1.

To go beyond single-qubit systems, we consider tensor products of spaces (in contrast to the cartesian products used in classical systems). If spaces U and V have bases $\{u_i\}$ and $\{v_j\}$ then $U \otimes V$ has basis $\{u_i \otimes v_j\}$. In particular, a system consisting of n qubits has a 2^n-dimensional space whose standard basis is $|00\ldots0\rangle \ldots |11\ldots1\rangle$. We can now consider measurements of single qubits or collective measurements of multiple qubits. For example, a 2-qubit system has basis $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ and a general state is $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ with $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$. Measuring the first qubit gives result 0 with probability $|\alpha|^2 + |\beta|^2$ (leaving the system in state $\frac{1}{\sqrt{|\alpha|^2+|\beta|^2}}(\alpha|00\rangle + \beta|01\rangle)$) and result 1 with probability $|\gamma|^2 + |\delta|^2$ (leaving the system in state $\frac{1}{\sqrt{|\gamma|^2+|\delta|^2}}(\gamma|10\rangle + \delta|11\rangle)$); in each case we renormalize the state by multiplying by a suitable scalar factor. Measuring both qubits simultaneously gives result 0 with probability $|\alpha|^2$ (leaving the system in state $|00\rangle$), result 1 with probability $|\beta|^2$ (leaving the system in state $|01\rangle$) and so on; the association of basis states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ with results $0, 1, 2, 3$ is just a conventional choice. The power of quantum computing, in an algorithmic sense, results from calculating with superpositions of states; all of the states in the superposition are transformed simultaneously (*quantum parallelism*) and the effect increases exponentially with the dimension of the state space. The challenge in quantum algorithm design is to make measurements which enable this parallelism to be exploited; in general this is very difficult.

The *controlled not* (CNOT) operator on pairs of qubits performs the mapping $|00\rangle \mapsto |00\rangle$, $|01\rangle \mapsto |01\rangle$, $|10\rangle \mapsto |11\rangle$, $|11\rangle \mapsto |10\rangle$, which can be understood as inverting the second qubit (the *target*) if and only if the first qubit (the *control*) is set. The action on general states is obtained by linearity.

Systems of two or more qubits may be in *entangled* states, meaning that the states of the qubits are correlated. For example, consider a measurement of the first qubit of the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. The result is 0 (and the resulting state is $|00\rangle$) with probability $\frac{1}{2}$, or 1 (and the resulting state is $|11\rangle$) with probability $\frac{1}{2}$. In either case, a subsequent measurement of the second qubit gives a definite, non–probabilistic result which is identical to the result of the first measurement. This is true even if the entangled qubits are physically separated. Entanglement illustrates the key difference between the use of the tensor product (in quantum systems) and the cartesian product (in classical systems): an entangled state of two qubits is one which cannot be expressed as a tensor product of single-qubit states. The Hadamard and CNOT operators can be combined to create entangled states: $\mathrm{CNOT}((H \otimes I)|00\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

4. The SQRAM Architecture

In this section we propose a system architecture for a hybrid classical–quantum computer, with a conventional classical subsystem (consisting of a CPU, a classical data store and program store) and a separate quantum–mechanical unit. Such a device, termed a QRAM machine, was proposed by Knill in [8]. The quantum–mechanical unit consists of a quantum memory register and a means of manipulating its contents. We will describe the details of this architecture, including its operating cycle and instruction set.

4.1 The Classical Component

The CPU contains an Arithmetic–Logic Unit for evaluating classical expressions, a control unit, and a program counter, which keeps track of the current point of execution within the program store (the program store is separate from the data store for simplicity). The CPU does not contain any registers as all operations are performed on the data store. The data store operates a stack–based model both for the evaluation of expressions and for the allocation of variables; this fits well with the functional programming paradigm and simplifies code generation for the compiler.

Program execution begins with the program counter, local base, and stack top all cleared to 0. An instruction is retrieved from the location given by the program counter and executed, the process is then repeated. Most instructions cause the program counter to be incremented but some (such as jumping and halting instructions) have different effects (see Table 1 for a listing of the available classical instructions). Program execution is finished once the program counter goes past the end of the program store.

4.2 The Quantum Component

The quantum subsystem comprises a quantum register and a Quantum Hardware Interface (QHI), which receives instructions from the CPU and manipulates qubits accordingly. Figure 2 illustrates the stages of a typical quantum algorithm.

In the first stage, the hardware resets the qubits to the $|0\rangle$ state and then applies some transformation to place them in the desired initial state. Applying this transformation could be considered to be part of the computation, but conceptually it helps to consider it as part of the initialisation. The second stage is where the manipulation of the quantum state actually takes place. After the computation is complete, the result is measured. Each measured qubit

Table 1. SQRAM Machine Classical Instruction Set

Instruction	Effect
ADD	$st \leftarrow st - 1$ $DS[st-1] \leftarrow DS[st-1] + DS[st]$ $pc \leftarrow pc + 1$
HALT	$pc \leftarrow size(PS)$
JUMPZ *address*	$st \leftarrow st - 1$ $if(DS[st-1] = 0):$ $\quad pc \leftarrow address$ $else:$ $\quad pc \leftarrow pc + 1$
LOAD *offset*	$DS[st] \leftarrow DS[lb + offset]$ $st \leftarrow st + 1$ $pc \leftarrow pc + 1$
LOADL*value*	$DS[st] \leftarrow value$ $st \leftarrow st + 1$ $pc \leftarrow pc + 1$
SAVE *offset*	$st \leftarrow st - 1$ $DS[lb + offset] \leftarrow DS[st]$ $pc \leftarrow pc + 1$
SUBTRACT	$st \leftarrow st - 1$ $DS[st-1] \leftarrow DS[st-1] - DS[st]$ $pc \leftarrow pc + 1$

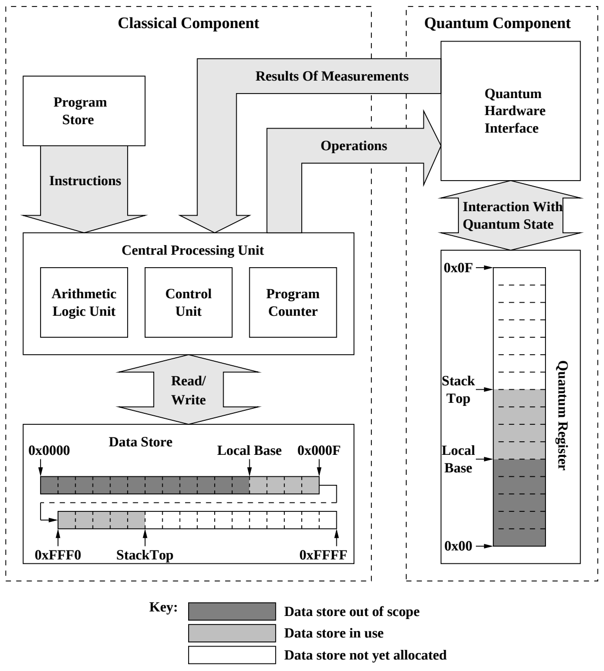


Figure 1. Design of the SQRAM machine

yields a binary value $\{0,1\}$ which is returned to the CPU and can be used for conditional control purposes. The final stage checks the validity of the result and performs the computation again if necessary. Incorrect results may occur either due to problems with the hardware, allowing quantum states to become damaged, or due to the algorithm being probabilistic by nature and hence not always producing the desired result.

As was explained in Section 3, the state is transformed by applying a sequence of operations to it; these may operate on an arbitrary number of qubits and the only restriction is that they must be unitary. While this is accurate from a theoretical point of

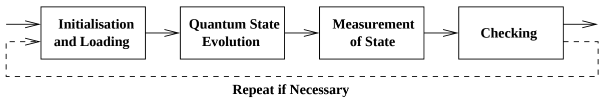


Figure 2. SQRAM Operating Cycle

Table 2. SQRAM Machine Quantum Instruction Set

Instruction	Effect
AQBIT	$qst \leftarrow qst + 1$ $pc \leftarrow pc + 1$
CNOT tar cont inv	$QR[tar] \leftarrow tar \times cnot(cont, inv, \ldots)$ $pc \leftarrow pc + 1$
GATE tar a b c d	$QR[tar] \leftarrow tar \times gate(a,b,c,d)$ $pc \leftarrow pc + 1$
HDMD tar	$QR[tar] \leftarrow tar \times gate(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$ $pc \leftarrow pc + 1$
MSRE tar	$DS[st] \leftarrow measure(tar)$ $st \leftarrow st + 1$ $pc \leftarrow pc + 1$
PHASE tar	$QR[tar] \leftarrow tar \times gate(1,0,0,i)$ $pc \leftarrow pc + 1$
PI tar	$QR[tar] \leftarrow tar \times gate(1,0,0,e^{i\pi/4})$ $pc \leftarrow pc + 1$

view it is at the present time very difficult to implement arbitrary operations on arbitrary numbers of qubits. Fortunately it is known that there is a small set of operations (actually an infinite number of such sets) which is universal in that it is able to *approximate* an arbitrary operation to any given accuracy [9].

We now introduce the operations which make up one of these universal sets (known as the *standard set*). The first operation is the Controlled–NOT (CNOT), which we described previously. This operation can be combined with arbitrary single qubit operations to *exactly* implement any quantum operation on an arbitrary number of qubits.

The universal set also includes *approximate* arbitrary single qubit operations. Within the standard set these are the Hadamard Operator (denoted by H), the Phase Operator (S) and the $\pi/8$ operator. We use the standard universal set as the basis for the instruction set of the SQRAM, along with instructions for measurement and initialisation, and a classical set of instructions for control purposes. We also include an instruction for performing arbitrary operations on single qubits. The 'quantum part' of the instruction set is summarised in Table 2.

4.3 Deutsch's Algorithm on the SQRAM

We now present an example of a program written for the SQRAM machine. We will be using a revised and improved version of Deutsch's algorithm as given in [5]; this paper should be consulted for further details of the algorithm, as the description given here will be necessarily brief.

We are presented with a black–box which performs some function $f(x)$ on a single bit x. There are four possible functions which $f(x)$ could perform, these being $f(x) = 0$, $f(x) = 1$, $f(x) = NOT(x)$, and $f(x) = x$. Of these the first two are called *constant* because they always give the same result, while the second two are called *balanced* because half the inputs result in 0 and half result in 1. The problem is to determine, using as few function evaluations as possible, whether $f(x)$ is constant or balanced.

If done classically, this requires two function evaluations, one with an input of 0 and the other with an input of 1, and a comparison of the results. However, using Deutsch's algorithm on a quantum computer it is possible to use just one function evaluation. Note that if the function is constant then $f(0) \oplus f(1) = 0$, while if it is balanced then $f(0) \oplus f(1) = 1$. Using the circuit in Figure 3 it is possible to evaluate $f(0) \oplus f(1)$ with out ever finding out the values of $f(0)$ and $f(1)$.

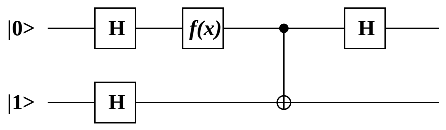


Figure 3. A Circuit Implementing Deutsch's Algorithm.

To illustrate this as SQRAM assembly code we choose a function to test; we will work with the NOT gate as it is straightforward. The NOT gate is balanced, so the result of evaluating $f(0) \oplus f(1)$ should be 1. The code to be executed on the SQRAM machine is given in Algorithm 4.3.

Compared to the circuit representation the code requires additional initialisation as all qubits are automatically initialised to $|0\rangle$ where as the second qubit needs to begin in state $|1\rangle$; a NOT operation resolves this. Note that the NOT operation has been realised using the GATE instruction, it could equally well be done using a CNOT with no controls.

The example mostly illustrates quantum instructions as the classical ones should be familiar to most readers. The only classical instruction used is SAVE, which stores the result of the measurement at the top of the classical stack. Further code could conditionally jump based on this value to give feedback to the user.

4.4 Simulating the SQRAM

In order to evaluate and analyse the design of the SQRAM, we have developed a software simulator. Simulation of quantum mechanical systems is known to be a highly complex problem (as observed by Richard Feynman in [6]), and so our simulator is restricted to a relatively small number of qubits. However, there are several known quantum algorithms which only make use of a few qubits; we have succeeded in modelling these.

The simulator makes use of the 'OpenQubit' library [12]. This is an Open Source library designed to be used in projects involving the modelling of quantum systems. The library (written in C++) provides a class to represent the state of a system (by storing a potentially large, complex vector) and a set of classes representing valid transformations which can be applied to that state. Our simulator implements the classical component and fetch–execute cycle of the SQRAM machine directly and also makes use of the OpenQubit library to implement the quantum part. As far as possible the architecture of the simulated SQRAM machine matches that presented earlier in Figure 1, and the instruction set also matches the one specified.

One key difference between the simulator and the design presented in Figure 1 is an additional layer of abstraction placed between the quantum register and the processor. The simulator provides a 'universe' of qubits and the quantum register is actually a collection of references to qubits within the universe, as shown in Figure 5. It is designed in this way so as to simplify the process of transmitting qubits from one machine to another; it is just a case of passing references. Although the issue of such communication is not discussed here it is a topic for future work.

We provide in Appendix A an example of the SQRAM machine simulator performing a simulation of Deutsch's algorithm, using the bytecode presented in Algorithm 4.3. The algorithm is testing the NOT function, so we expect it to determine that the function is *balanced*. We have broken the output into sections which are numbered to allow them to be referenced from the text.

```
AQBIT                            ;allocate initial qubits
AQBIT
GATE  0x01   0.0  0.0            ;initialise second qubit to 1
      1.0 0.0 1.0 0.0
      0.0 0.0
HDMD  0x00                       ;apply Hadamards
HDMD  0x01
GATE  0x00   0.0  0.0            ;NOT gate is our test function
      1.0 0.0 1.0 0.0
      0.0 0.0
CNOT  0x01 1 0 0                 ;apply the CNOT gate
HDMD  0x00                       ;the last Hadamard
MSRE  0x00                       ;measure the result
SAVE  0x00                       ;save to address 0x00 for later
```

Figure 4. SQRAM Program Implementing Deutsch's Algorithm

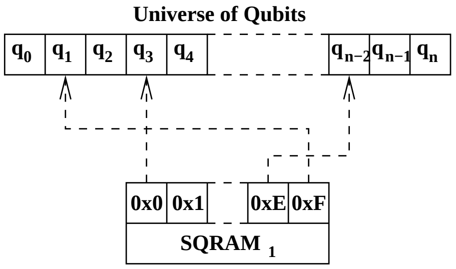


Figure 5. SQRAM machine accessing qubits via references

The simulator begins by creating an instance of the QRAM class (with three qubits) and initialising it to its starting state (step 1). As always the sum of the probability amplitudes must be 1.0. The program is then read from disk and loaded into the SQRAM machine (step 2). The machine starts off with all qubits sets to $|0\rangle$, it can optionally output the state of classical memory as well but we won't be using that here. Note that the display (step 3) of the machine state always shows all three qubits even though none of them have been allocated yet, and even though we will only be using two of them. Execution of the program begins with the allocation of two qubits (step 4), a result of two successive AQBIT instructions. Qubits are initialised to the $|0\rangle$ state.

It can be seen from Figure 3 that the lower qubit needs to start in state $|1\rangle$ instead. A GATE operation implementing the NOT function is applied (step 5) to change this. The two Hadamards are applied next (step 6), eventually leaving the system is an equal superposition of the states $|000\rangle$,$|001\rangle$,$|010\rangle$,$|011\rangle$. Again remember that the first qubit is still $|0\rangle$ because it has not been allocated.

In (step 7) we apply the test function, which in our case is the GATE instruction implementing the NOT function. We then perform a CNOT operation (step 8), conditional on the result of our test function. Some of the output is generated by the OpenQubit library rather than by our simulator, hence the target and control qubits are referred to as 1 and 2 respectively, whereas we have previously called them 0 and 1.

The last Hadamard is applied (step 9) which leaves qubit 0 in state $|0\rangle$ (as expected). Of course, although we can see the result by 'peeking' into the simulator, in reality we can't tell until we perform the measurement. During the measurement the OpenQubit library generates some more output (step 10), the eventual result being that the qubit is correctly measured to be 1. This indicates our test function was balanced.

5. Compiling High-Level Languages for the SQRAM

It is, of course, impractical to write large programs by hand using the instructions set presented in the previous section as this is a task which is both time consuming and error prone. The solution in the field of classical computing has been to develop programming languages which can be automatically compiled into corresponding machine code; this is also one of the solutions for quantum computers.

It is by no means the only solution as many theoretical breakthroughs are made by studying the underlying mathematics of quantum mechanics and another notation known as 'quantum circuits' has also proved popular. However there are several reasons why there has been a growing interest in the use of quantum programming languages.

Perhaps the most important point is that use of a language elegantly allows a mixture of classical control structures with quantum operations; this is something which is very difficult to do with a quantum circuit model. It fits well with the model of computation used by our SQRAM machine (and by most quantum algorithms) in which the majority of the operations are classical. A second advantage (as in classical programming) is that the generation of repetitive code structures is handled automatically by the compiler and it is able to perform optimisations which would be difficult by hand. Finally, the programming model is far more intuitive to most computer scientists and will greatly ease the development of algorithms.

5.1 Quantum Language Requirements

It is possible to identify a set of properties which a quantum programming language should possess in order to maximise the effectiveness of the language. Some of the ideas given have been previously suggested by other research groups [2, 8, 13] while some are based on experience.

Classical Characteristics: Many years of research on classical languages have identified properties such as a clean syntax and intuitive set of keywords as being important for languages.

Completeness: The language should be universal such that it can represent all quantum algorithms. This gives it the same expressive power as the mathematical model or quantum circuit model.

Expressivity: The language should present the programmer with a sufficient set of primitives (such as qubits) and constructs (such as measurements) to allow programs to be constructed easily.

Separability: It should be easy to achieve separation of those parts of the program which are quantum in nature from those parts which are classical as this can simplify compilation and execution of the code.

Hardware Independence: Although in this paper we present the SQRAM machine as a model of computation it is important to realise that other models may be developed and a language should ideally work on these. A language should be kept as general as possible perhaps by using the quantum Turing machine as the target.

Extension of Classical Languages: Much research has been undertaken analysing the advantages of different programming paradigms and by extending an existing language which uses one of these we get a language with which programmers are familiar and which has an existing set of tools. Hence much unnecessary work can be avoided duplicating features.

$$
\begin{aligned}
\mathit{QPLTerms}\ P, Q ::= {} & \mathbf{new\ bit}\ b\ :=\ 0 \\
& \mid \mathbf{new\ qbit}\ q\ :=\ 0 \\
& \mid \mathbf{discard}\ x \mid b\ :=\ 0 \mid b\ :=\ 1 \\
& \mid q_1, \ldots, q_n {*}{=}\ S \mid \mathbf{skip} \mid P; Q \\
& \mid \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q \\
& \mid \mathbf{measure}\ q\ \mathbf{then}\ P\ \mathbf{else}\ Q \\
& \mid \mathbf{while}\ b\ \mathbf{do}\ P \\
& \mid \mathbf{proc}\ X : \Gamma \to \Gamma' \{P\}\ \mathbf{in}\ Q \\
& \mid y_1, \ldots, y_m = X(x_1, \ldots, x_n)
\end{aligned}
$$

Figure 6. The syntax of Peter Selinger's QPL.

5.2 Particularities of Quantum Systems Affecting Language Designs

The inability to clone an unknown quantum state has a direct effect on the behaviour of statements which involve assignment; these include direct assignment and passing values to functions. Because it is not possible to actually copy the value many languages make use of references and hence have many variables pointing to the same qubit. Other languages may forbid the direct assignment of quantum variables.

Although as noted previously it is not possible to assign one qubit to another it is possible to assign a qubit to a classical bit; this involves an implicit measurement. Unlike in classical programming this will modify the variable on the right hand side of the assignment and it is not possible to retrieve the previous value.

It is possible for two qubits identified by separate variable names to become entangled such that the manipulation of one variable has an effect on the other. There is no analogy to this in classical programming although it is not so much an issue for the language but rather for the programmer who designs the algorithms.

5.3 A Functional Language: QPL

Given the requirements which have been set out there are several languages which meet them to varying degrees [2, 8, 10]. Our work focuses on one in particular, Peter Selinger's QPL [13], due to its clean and elegant design, and its suitability for implementation on the SQRAM machine.

Selinger identifies the static type system as being one of the key features of his language as this allows the syntax to enforce certain properties of quantum mechanics such as the no–cloning theorem, by forbidding direct assignment between qubits. As well as the usual control constructs such as loops and conditional statements QPL allows the definition of recursive functions. This is useful when operating over lists and trees which QPL allows to contain quantum data as well as classical. The syntax of QPL is reproduced from [13] in Figure 5.3.

5.4 Code Templates for Quantum Operations

Code generation, for our purposes, is the process of producing bytecode instructions for each of the nodes in the abstract syntax tree corresponding to a high–level program. We have designed code templates for the quantum constructs within the QPL language and have also considered the decomposition of large operations, so they may be implemented directly in bytecode.

Code templates are used within compilers to provide a set of byte–code instructions which correspond to a particular construct in the source program, such as an expression, a loop, or a variable declaration. We specify code templates for those constructs which are quantum in nature but do not give details of classical constructs as these are very similar to existing languages. Specifically, we cover the declaration of quantum types, the manipulation of those types, and their eventual measurement.

Within QPL a new qubit is declared using a statement such as:

$$(\textbf{new qbit } \mathsf{q} := 0)$$

This allocates a new qubit, referred to by the variable name q, and initialises it to the state $|0\rangle$. To implement this we simply use the AQBIT instruction:

$$\textit{translate}[\textbf{new qbit } \mathsf{q}{:=}0] = \textbf{AQBIT}$$

where *translate* is a function that specifies a translation of terms in QPL to SQRAM bytecode.

A transformation is applied to a quantum data type using the $* =$ operator. For example, the built-in unitary transformation U could be applied to q as follows:

$$(\mathsf{q} \mathrel{*}= \mathsf{U})$$

There are two situations to consider here. Firstly U might be a single qubit operation which we wish to implement directly using the GATE instruction. This becomes:

$$\textit{translation}[\mathsf{q} \mathrel{*}= \mathsf{U}] = \textbf{GATE } \mathsf{q}\ \mathsf{U}$$

Alternatively U might be a multi–qubit operation (in which case q would need to be a multi–qubit data type), or it might be a single qubit operation which we wish to decompose into gates from the universal set of operations. Either way, we move into the decomposition process which is discussed in Section 5.5.

Measurement is the most complex of the code templates (assuming we don't get involved with decomposition when manipulating quantum types). QPL performs measurement by the following statement:

$$\textbf{measure } \mathsf{q} \textbf{ then } \mathsf{P} \textbf{ else } \mathsf{Q}$$

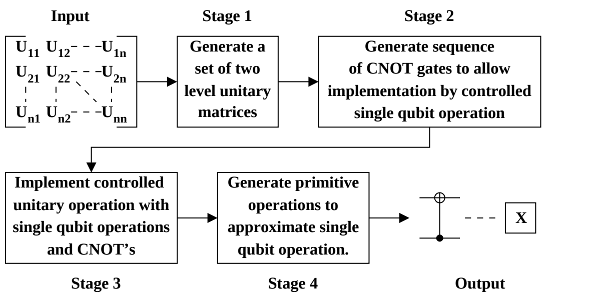
A measurement is performed on the qubit q. If the result of the measurement is $|1\rangle$ then the command corresponding to P is executed, otherwise the command corresponding to Q is executed. The code template for this looks as follows:

```
translation[measure q then P else Q] =
MSRE     q            ;Perform the measurement
JUMPZ    ELSE         ;If it's |0⟩ then jump to else part
execute  P            ;Else execute command P
LOADL    0            ;Unconditionally jump to end
JUMPZ    END          ;by loading 0 and jumping if 0
ELSE:    execute Q    ;Execute command Q
END:
```

If the measurement of q gives a value of 1 then the JUMPZ instruction is ignored and the program proceeds to execute P before unconditionally jumping over the code to execute Q. On the other hand, if q is measured as 0 the first JUMPZ jumps over the execution of P straight to the point where Q is executed.

5.5 Decomposition of Operator Matrices

In Section 4.2 we discussed the principle of universality, stating that any quantum operation can be broken down and implemented in terms of a small set of universal gates. Hence our SQRAM machine only provides operations corresponding to these universal gates and it is the job of the compiler to perform the decomposition. This decomposition is a complex process and work has been done on it by a variety of different people and research groups. We bring this work together to form a complete compilation process and provide an analysis of its efficiency.



Input
Stage 1
Stage 2
U11 U12 U1n
U21 U22 U2n
Un1 Un2 Unn
Generate a set of two level unitary matrices
Generate sequence of CNOT gates to allow implementation by controlled single qubit operation
Implement controlled unitary operation with single qubit operations and CNOT's
Generate primitive operations to approximate single qubit operation.
X
Stage 3
Stage 4
Output

Figure 7. Decomposition of an Arbitrary Unitary Matrix

5.5.1 Overview

Decomposing a matrix into primitive operations is a multistage process (outlined in Figure 7); each of the stages shown is described in the following sections. The mathematical proofs for the validity of each stage of the process are well established and work has previously been done looking at the optimal number of gates which can be used to approximate a given unitary matrix. Therefore this work focuses on designing algorithms to implement the process and performing classical efficiency analysis on these algorithms. It is to our knowledge the first system to implement the complete process from arbitrary operations to quantum byte–code within a compiler.

A working compiler has been implemented to test the concepts presented in the following sections but we will not discuss its implementation here. For details of this including design approaches, examples, and sample output please refer to [15].

5.5.2 Generating Two-Level Unitary Matrices

Two-level unitary matrices are those which act non-trivially on only 2 vector components of the system state; that is ,when the vector is multiplied by the matrix only two elements are changed as most elements in the matrix are identity. Such a matrix has a structure as follows:

$$R_k = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \\ \cdots & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots \\ \cdots & 0 & \alpha & 0 & \cdots & 0 & \gamma & 0 & \cdots \\ \cdots & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots \\ & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \\ \cdots & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & \cdots \\ \cdots & 0 & \beta & 0 & \cdots & 0 & \delta & 0 & \cdots \\ \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & \cdots \\ & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (1)$$

The initial step is to decompose the original matrix U of side length s into a sequence of two-level unitary matrices (also of side length s). The product of the matrices in this sequence must be equal to the input matrix U, so that applying them to the system in the correct order has the same effect as applying U.

Performing this decomposition is not only necessary for the next stage, it is also a result in its own right. A description of a technique for implementing such transformations with beam–splitter devices is presented in [16], with the result that simply performing this stage could bring arbitrary operations closer to being realisable.

We will not go deeply into the mathematics involved as it can become reasonably complicated; for a coverage of this see [16, 15, 9]. It has been observed [9] that such a process will decompose the original matrix into at most $\frac{s(s-1)}{2}$ two level matrices. However, no analysis of the efficiency of such an algorithm was provided and it is a useful result to determine this. In [15] we make some assumptions about the efficiency of variations and show the complexity to be approximately $\Theta(n^5)$. This is clearly not a fast algorithm, and it should be noted that the previous analysis was with respect to the size of the matrix (which is exponential in the size of the system). Hence the algorithm requires exponential time overall, but it should also be remembered that it will typically be operating on small values of n, corresponding to a small number of qubits. Also, the difficulty in performing this decomposition makes it clear that it is necessary to have a quantum byte–code which can be stored as it too difficult to generate in real time.

5.5.3 Generating Controlled Unitary and CNOT Gates

We obtain from the previous stage a set of two level unitary matrices, for example a matrix of the form:

$$U = \begin{bmatrix} \alpha & 0 & 0 & \gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta & 0 & 0 & \delta \end{bmatrix} \tag{2}$$

A matrix such as this acts on two components of the system (in this particular case it acts on $|00\rangle$and $|11\rangle$), and leaves the other components unaffected as follows:

$$\begin{bmatrix} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{bmatrix} \overset{U}{\rightarrow} \begin{bmatrix} \alpha\,|00\rangle + \gamma\,|11\rangle \\ |01\rangle \\ |10\rangle \\ \beta\,|00\rangle + \delta\,|11\rangle \end{bmatrix} \tag{3}$$

We wish to implement this matrix in terms of a controlled single qubit operation, or Controlled–U gate, but note that a single qubit operation cannot act on both $|00\rangle$ and $|11\rangle$ as they differ by more than one bit. Therefore we use a series of CNOT gates (in this simple case the series contains just one gate) to swap states around such that the target states are adjacent to each other. The Controlled-U is then applied to the one bit which still differs, and the reverse series of CNOT gates is used to arrange the states back to their original position.

To clarify this procedure, the operation given by Equation 2 is implemented by the circuit in Figure 8, where T is the sub–matrix of U given by:

$$T = \begin{bmatrix} \alpha & \gamma \\ \beta & \delta \end{bmatrix} \tag{4}$$

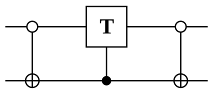


Figure 8. Circuit implementing Equation 2.

Note that the CNOT gates are active when the control qubit is $|0\rangle$, rather than the more conventional $|1\rangle$. The problem then is how to generate the series of CNOT gates which rearrange the computational states in the appropriate way. A solution involving the use of *Gray codes* is covered by [9] and a description within the context of our compiler is provided by [15].

5.5.4 Implementing Controlled Unitary Gates

The output from the procedure described in Section 5.5.3 consists of two types of gates; Controlled–NOT gates and Controlled–U gates. Our SQRAM machine is able to directly implement Controlled–NOT gates through the CNOT instruction, but Controlled–U gates require further decomposition. This section shows briefly how this is done, building on work presented by Barenco *et al.* in [1].

Barenco *et al.* make the observation that for any unitary matrix U of side length 2 (i.e. operating on a single qubit) it is possible to find 3 more unitary matrices A, B, and C such that:

$$A \times B \times C = I$$

and:

$$S \times A \times NOT \times B \times NOT \times C = U$$

where S is defined as:

$$S = \begin{bmatrix} e^{i\delta} & 0 \\ 0 & e^{i\delta} \end{bmatrix}$$

A controlled–S gate can be simulated by a unitary operator E acting on the control bit, hence it is possible to produce an implementation of an arbitrary operator U using a circuit such as the one shown in Figure 9. For unitary gates controlled by multiple qubits the procedure is similar; we find a set of unitaries which can either implement the original unitary matrix or can implement the identity matrix, depending on the use of CNOT gates in between. However the actual process of generating both the gates and the sequence of CNOTs is considerably more complex (see [1, 15]).

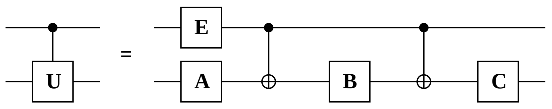


Figure 9. Implementation of an Arbitrary Unitary

The problem then becomes determining suitable values for the operators A, B, C, and E, expressions for doing so are established in [1] though we will not re–iterate them here.

6. Conclusions and Future Work

In this work we have presented an architecture for quantum computation, studied quantum programming languages and their relationship to our architecture, and looked at the issues involved in compiling quantum languages to byte–code.

We began by presenting a quantum computer based on the QRAM architecture originally suggested by Knill. We designed the instruction set and method of operation for the classical component such that it could be used as a stand–alone processor or as a control mechanism for a quantum component. We then designed the quantum component with an instruction set which makes it universal for quantum computation we allow data to be passed between the two components.

After studying the low level byte–code programs which can be written directly on the SQRAM machine we studied higher level languages and the advantages of using them to write quantum algorithms. The general ideas surrounding quantum languages were discussed and we then provided a brief comparison of the features available in several of the languages available today.

We then moved on to the issue of generating instructions for our SQRAM machine based on a program written in QPL. Part of this involved creating ‘code templates’ for the various constructs in the QPL language, and part of it involved decomposing complex operations into those suitable for our SQRAM model. This decomposition process was based on work by various people but due to space limitations we avoided going too deeply into the mathematics involved.

We consider the work carried out to date to be a great success but there is the potential for much more work to be done. Some of the ideas presented here have been started and are already showing promise.

6.1 SQRAM Model and Simulator

We stated that the instruction set provided was universal for quantum computing; that is not to say it cannot be improved. There are different universal sets available and there are also advantages to having a certain amount of redundancy (as with the GATE instruction). An analysis of the advantages and disadvantages of different instruction sets could yield a more efficient SQRAM architecture. There is also scope for expanding the classical instruction set as the current one is just a proof–of–concept allowing us to focus on the quantum work. More sophisticated conditional control statements (as opposed to simply using the JUMPZ instruction) would ease the development of complex control structures and a greater range of instructions for manipulating classical data would also be desirable.

A discussion of the actual physics involved in building a quantum computer has been avoided in this paper and, as far as possible, in the SQRAM model. In practice there are many real physical aspects which would affect a SQRAMs behaviour. For example, when using the ion trap technique [4] it is easier to perform operations on multiple qubits if they are adjacent to each other. It would be interesting to integrate such constraints into the model.

A related idea is to model the effects of ‘quantum decoherence’ on the SQRAM machine. Quantum decoherence is the process of errors arising due to undesirable interaction with an external system (something which is impossible to avoid in practice). The QPL language was designed for ‘perfect’ hardware in which such interactions do not occur but, given the impossibility of building such hardware, it would be useful to introduce errors into the results of the simulation so that techniques for combating them can be developed. Existing methods can also be tested and their effectiveness determined within the context of the QPL/SQRAM system.

Lastly, the tools developed so far have been for our own internal research purposes. They are slightly obscure, command–line affairs which require a degree of inside knowledge to use effectively. We feel that it would be beneficial to the larger research community if these tools could be more widely used and to this end we intend to refine the tools and provide them with a more friendly and intuitive user interface. There are plans to develop the tools into a complete environment for the development and testing of quantum algorithms.

6.2 The QPL Compiler

One of the distinguishing features of the QPL language is a static type checking system which allows certain errors to be detected at compile time rather than run time. For example, the static type system is able to enforce the no–cloning principle of quantum mechanics within QPL programs. We have not yet implemented such static type checking within our compiler but aim to do so in the near future. This should look at type checking issues when working with more complex quantum structures (lists, trees, etc) and could also consider type checking within the higher–order version of QPL currently being worked on by Peter Selinger.

More generally, the compiler needs work to make it more ‘complete’. It implements only a subset of QPL, the focus being on those parts which were necessary to test ideas presented in this paper. More work on the classical control structures would enable a wider range of programs to be implemented and better data structures (currently only limited support for lists is available) would allow more interesting algorithms. We also plan to extend the compiler with features for concurrency and communication (discussed next).

6.3 Communication and Concurrency

Although it has not been described in this paper we have also undertaken some work to integrate constructs for communication and concurrency into the QPL language and SQRAM simulator. These changes to QPL have yielded a new language, CQP [7], which is able to describe algorithms such as quantum key exchange and quantum teleportation. We aim to integrate this into our compiler; the simulator has already had such support added.

References

[1] A. Barenco, C. Bennett, R. Cleve, D. Divincenzo, N. Margolus, P. Shor, T. Sleator, J. Smoli, and H. Weinfurter. Elementary gates for quantum computation. *Submitted to Physical Review A*, 1995.

[2] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal*, 25(2):181–200, 2003.

[3] P. E. Black and A. W. Lane. Modeling quantum information systems, 2004. Unpublished.

[4] J. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74:4091, 1995.

[5] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Submitted to Phil. Trans. R. Soc. Lond. A*, 1997.

[6] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6&7):467–488, 1982.

[7] S. Gay and R. Nagarajan. Communicating quantum processes. In *POPL ’05: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages, Long Beach, California*, January 2005.

[8] E. Knill. Conventions for quantum pseudocode, 1996.

[9] M. Nielson and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[10] B. Omer. A procedural formalism for quantum computing, 1998.

[11] N. Papanikolaou. QSPEC: A programming language for quantum communication systems design. In *Proceedings of PREP2004 Postgraduate Research Conference in Electronics, Photonics, Communications & Networks, and Computing Science*. EPSRC, 2004.

[12] Y. Pritzker. Simulation of quantum computation on intel-based architectures, 1999.

[13] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[14] K. Svore, A. Cross, A. Aho, I. Chuang, and I. Markov. Toward a software architecture for quantum computing design tools. *Proceedings of the 2nd International Conference on Quantum Programming Languages*, pages 145–162, 2004.

[15] D. Williams. Quantum computer architecture, assembly language and compilation, 2004.

[16] A. Zeilinger, M. Reck, H. Bernstein, and P. Bertani. Experimental realization of any discrete unitary operator. *Physical Review Letters*, 73(1):58–61, 1994.

A. Sample Simulator Output

We provide here the output of our simulator when performing a simulation of Deutsch’s Algorithm as described in Section 4.4.

```
Initialising QRAM machine...        } (1)
Sum of probabilities: 1.0000        }

   Loading program...                    }
   Loading AQbitInstruction              }
   Loading GateInstruction               } (2)
   Loading CNotInstruction               }
   Loading MsreInstruction               }
   Load succeeded                        }

   *** QRAM Quantum State ***  } (3)
        1.000000 |000>         }
```

```
Allocating new QBit
*** QRAM Quantum State ***
     1.000000 |000>

Allocating new QBit
*** QRAM Quantum State ***
     1.000000 |000>
```
(4)

```
Applying Gate operation to bit 1
*** QRAM Quantum State ***
     1.000000 |010>
```
(5)

```
Applying Gate operation to bit 0
*** QRAM Quantum State ***
     0.707100 |010> +...
     ...0.707100 |011>

Applying Gate operation to bit 1
*** QRAM Quantum State ***
     0.499990 |000> +...
     ...0.499990 |001> +...
     ...-0.499990 |010> +...
     ...-0.499990 |011>
```
(6)

```
Applying Gate operation to bit 0
*** QRAM Quantum State ***
     0.499990 |000> +...
     ...0.499990 |001> +...
     ...-0.499990 |010> +...
     ...-0.499990 |011>
```
(7)

```
Performing CNot Instruction
Controlling: 2 Controlled: 1
In common: 0
(-0.49999,0)(-0.49999,0)...
...(0.49999,0)(0.49999,0)...
...(0,0)(0,0)(0,0)(0,0)
(-0.49999,0)(-0.49999,0)...
...(0.49999,0)(0.49999,0)...
...(0,0)(0,0)(0,0)(0,0)
*** QRAM Quantum State ***
     -0.499990 |000> +...
     ...-0.499990 |001> +...
     ...0.499990 |010> +...
     ...0.499990 |011>
```
(8)

```
Applying Gate operation to bit 0
*** QRAM Quantum State ***
     -0.707086 |000> +...
     ...0.707086 |010>
```
(9)

```
Measuring QBit 0
Got probabilities...p0=1.000,...
     ...p1=0.000
Sum of probabilities: 0.99994
Normalized amplitudes: 0.99994
norm(*this)+RND_ERR =...
...0.9999424611046097144395617
norm(*this)-RND_ERR =...
...0.9999424611026097586830019
p0+p1 should be between those...
...and it is 0.9999424611
Set bit state to 0
*** QRAM Quantum State ***
     -0.707107 |000> +...
     ...0.707107 |010>
```
(10)