QPL 2005 Preliminary Version

Simulating and compiling code for the Sequential Quantum Random Access Machine

Rajagopal Nagarajan [1,3]

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

Nikolaos Papanikolaou [2,4]

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

David Williams [5]

School of Informatics
City University
London EC1V 0HB
United Kingdom

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.

Key words: Quantum computation, quantum programming, quantum simulators, QRAM, compilers.

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:

> "...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." (excerpt from [7])

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 [9]. 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 balancedness of a boolean function [11]. We have implemented a simulator of the SQRAM machine using the OpenQubit library [14].

In light of recent proposals for quantum languages, including for example QPL [15], QCL [12], CQP [8] and qSpec [13], we feel it is in order 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.

[1] R. Nagarajan is supported by EPSRC grant GR/S34090 and the EU Sixth Framework Programme (Project SecoQC: *Development of a Global Network for Secure Communication based on Quantum Cryptography*).

[2] N. Papanikolaou is supported by a Postgraduate Fellowship by the Department of Computer Science, University of Warwick.

[3] Email: `biju@dcs.warwick.ac.uk`

[4] Email: `nikos@dcs.warwick.ac.uk`

[5] Email: `d.p.williams@city.ac.uk`

We begin with a summary of basic quantum computing concepts. We will then proceed to describe the proposed SQRAM architecture and instruction set; to illustrate the instruction set, we show how Deutsch's algorithm would be implemented in assembly language for the SQRAM. Finally, we will turn to compilation of high–level quantum programs in QPL, a functional quantum programming language.

2 Related Work

Currently several quantum simulators are available, including tools for analyzing quantum circuits and interpreters for quantum programming languages [3,12]. The work presented here is also closely related to that described in [2,12,15], which deals with the issues involved with the design of quantum programming languages. Our work develops a more complete suite of tools consisting of separate (but interacting) parts for compilation and simulation; it also allows compiled code to be stored.

In [16], 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 an electron 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

$$\begin{aligned} |0\rangle &\mapsto |+\rangle = \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle \\ |1\rangle &\mapsto |-\rangle = \tfrac{1}{\sqrt{2}}|0\rangle - \tfrac{1}{\sqrt{2}}|1\rangle \end{aligned}$$

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:

$$\begin{aligned}
|00\rangle &\mapsto |00\rangle \\
|01\rangle &\mapsto |01\rangle \\
|10\rangle &\mapsto |11\rangle \\
|11\rangle &\mapsto |10\rangle
\end{aligned}$$

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 Machine

In this section we propose a system architecture for a hybrid classical–quantum computer, consisting of a classical computing machine with control over a purely quantum–mechanical component. Such a device, termed a QRAM machine, was proposed by Knill in [9].

The classical component comprises a Central Processing Unit (CPU), a classical data store (DS) and a program store (PS). The quantum–mechanical component is divided into a Quantum Hardware Interface (QHI), a quantum memory register, and a means of manipulating its contents; note that our discussion remains independent of particular hardware implementations and associated physical details.

Figure 1 illustrates the overall design of the SQRAM machine, including both the classical component and the quantum component. We will now describe the details of this design, including its operating cycle and instruction set.

4.1 The Classical Component

The classical component has two distinct memory locations, one for programs and one for data. We have chosen to deviate slightly from the conventional Von Neumann model, where programs are treated in the same way, and in the same location, as the data they manipulate. The CPU keeps track of the current instruction in a program through the program counter, which indexes a location in the program store.

The CPU also contains an Arithmetic–Logic Unit, for evaluating classical expressions, and a control unit, for other operations. The CPU does not contain any registers, since all operations are performed within the data store.

The data store operates a *stack–based model* for evaluation of expressions and the allocation of variables; this fits well with the functional programming paradigm and simplifies code generation for the compiler. Global data (if applicable) is stored from the address $0x0000$ while the local base points to the the beginning of the data local to the current function (this is modified as functions are entered and left). Stack top points just after the end of the valid data area.

Table 1 details the principal instructions used to control the classical component of the SQRAM and describes the effect of each instruction in an algorithmic notation. The notation includes the symbols st (the Stack Top), lb (the Local Base), pc (the value of the Program Counter), $DS[i]$ (the contents of location i in the Data Store), and $PS[i]$ (the contents of location i in the Program Store).

The operating cycle of the SQRAM is as follows. Program execution begins with the program counter, local base, and stack top all initialized 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 for a listing of the available classical instructions). Program execution is finished once the program counter goes past the end of

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	$st \leftarrow st - 1$ $DS[st-1] \leftarrow DS[st-1] - DS[st]$ $pc \leftarrow pc + 1$

Table 1
SQRAM Machine Classical Instruction Set

the program store.

4.2 The Quantum Component

The quantum component of the SQRAM consists of 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 of Figure 2, the hardware resets the qubits to the $|0\rangle$ state and then applies some transformation to place them in the desired initial state. 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 yields a binary value $\{0, 1\}$ which is passed to the CPU and can then be used for conditional control purposes. The final stage checks the validity of the result and repeats the computation if necessary. Incorrect results may occur, either due to hardware problems such as decoherence, which damages quantum states, or due to the probabilistic nature of quantum algorithms.

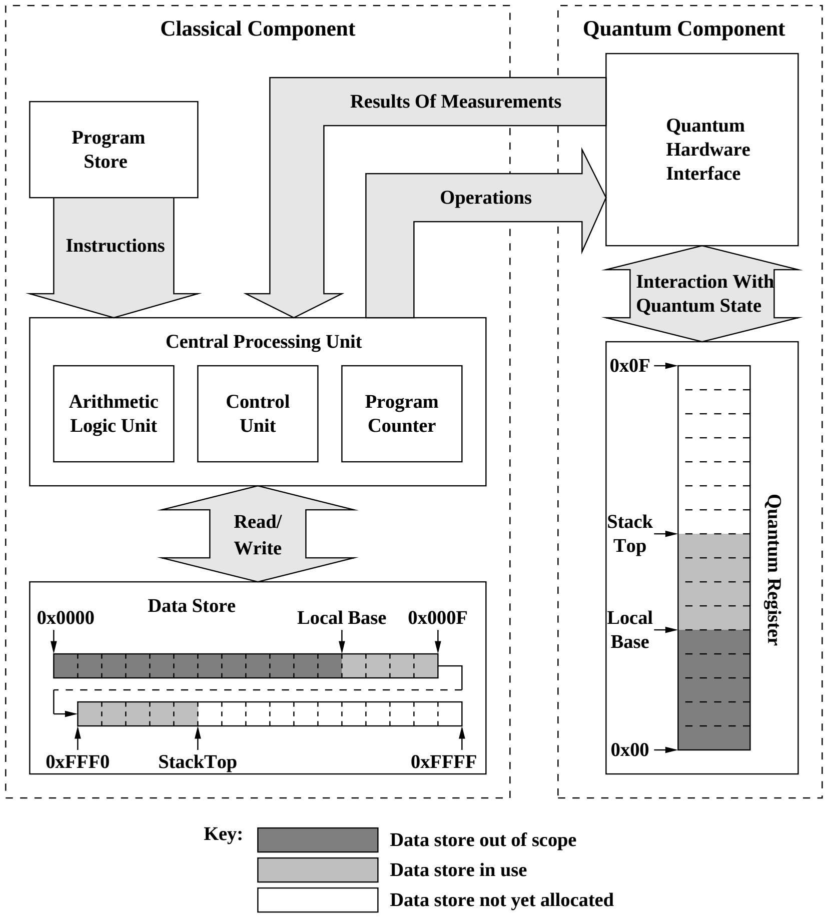

Fig. 1. Design of the SQRAM machine

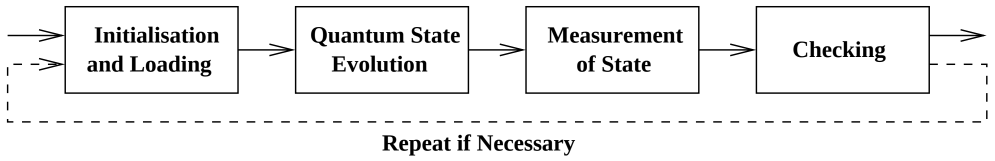

Fig. 2. SQRAM Operating Cycle

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

Table 2
SQRAM Machine Quantum Instruction Set

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 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 [11].

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.

To complete the standard set it is necessary to *approximate* arbitrary single qubit operations. Within the standard set this is done with the Hadamard

Operator (denoted by H), the Phase Operator (S) and the $\pi/8$ operator. We use the standard 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 instructions' for the SQRAM are summarized in Table 2. Special notation used in this table includes the symbols *target* (for *target* qubit), *control* (for *control* qubit), and *invert* (indicates the CNOT is active when the control is in the state $|0\rangle$ rather than $|1\rangle$); otherwise, the notation is mostly self–explanatory.

4.3 Deutsch's Algorithm on the SQRAM Machine

We now present an example of a program written for the SQRAM machine. We will be using Deutsch's algorithm as given in [5]; that reference 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 $f(1)$.

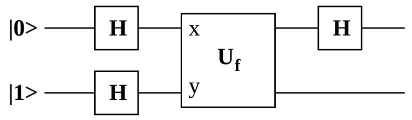

Fig. 3. A Circuit Implementing Deutsch's Algorithm.

An implementation, for the SQRAM machine, of Deutsch's algorithm for testing the balancedness of the NOT gate is given in Figure 4. The NOT gate is balanced, so the result of evaluating $f(0) \oplus f(1)$ should be 1.

Compared to the circuit representation, this code involves additional initializations as all qubits are automatically placed in state $|0\rangle$. The second qubit actually needs to be placed in state $|1\rangle$; this is achieved using the NOT operation. Note that the NOT operation has been realized using the **GATE** instruction; it could have been implemented just as well using a **CNOT** with no controls. The operation U_f is a reversible form of an *f(x)–controlled–NOT*;

```
AQBIT                                            ;allocate initial qubits
AQBIT
GATE    0x01 0.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0     ;initialise second qubit to 1
HDMD    0x00                                     ;apply Hadamards
HDMD    0x01
CNOT    0x01 1 0 1                               ;apply the CNOT gate
HDMD    0x00                                     ;the last Hadamard
MSRE    0x00                                     ;measure the result
SAVE    0x00                                     ;save to address 0x00 for later
```

Fig. 4. SQRAM Assembly Program Implementing Deutsch's Algorithm for the NOT gate

it performs the mapping $|x\rangle|y\rangle \rightarrow |x, y \oplus f(x)\rangle$. The *f(x)–controlled–NOT* operation is, in our example, a *NOT–controlled–NOT* which in turn is implemented as a CNOT with its control inverted to trigger when in the state $|0\rangle$.

The example mostly illustrates quantum instructions; the functions performed by classical instructions should be familiar to most readers. The only classical instruction used in this example 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 analyze 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 discussed in the Introduction), and so our simulator handles only a relatively small number of qubits. There are, of course, several known quantum algorithms which only make use of a few qubits; these we have already succeeded in modelling.

The simulator makes use of the *OpenQubit* library [14]. This is an Open Source C++ library designed to be used in projects involving the modelling of quantum systems. It provides classes for representing the states of quantum systems (which store large, complex–valued vectors), and a set of classes representing transformations which can be applied to such states. Our simulator implements the classical component and the fetch–execute cycle of the SQRAM machine directly; it makes use of the OpenQubit library to simulate the quantum component. As much as possible, the architecture of the simulated SQRAM machine matches the one presented earlier.

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.

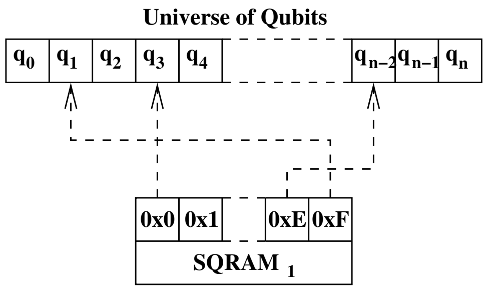

Fig. 5. SQRAM machine accessing qubits via references

5 Compiling High-Level Languages for the SQRAM

It is impractical to write large programs by hand using the assembly language instructions presented in the previous section; this is time consuming and error prone. The emergence of quantum programming languages, which can automatically be compiled into corresponding machine code, facilitates the task of 'quantum programming' significantly.

A quantum programming language provides an elegant mixture of classical control structures and quantum operations; this is something which is very difficult, if not impossible, to implement in the standard quantum circuit model. The use of quantum programming languages fits in well with the model of computation used by our SQRAM machine (and by most quantum algorithms). This computational model is more familiar to computer scientists than the circuit model, and we expect it to greatly ease the development of new quantum algorithms.

In this section, we consider high–level quantum programming languages and how programs written in such languages may be translated into assembly code for the SQRAM machine. We begin by identifying certain desirable characteristics and particularities of such languages, and then we proceed to discuss Selinger's QPL [15], a functional quantum programming language. We have implemented a translation from a subset of QPL to SQRAM assembly code, which is also described briefly.

5.1 Quantum Language Requirements

According to [2,9,15], the desired features of high–level quantum programming languages are:

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 high–level languages.

Completeness: A quantum programming language should be universal, so that it can represent all quantum algorithms. This gives it the same expressive power as the quantum circuit model.

Expressivity: The language should present the programmer with a sufficient set of primitives and constructs to allow quantum programs to be constructed easily.

Separability: It should be simple to separate those parts of the program which are quantum–mechanical in nature from those parts which are classical, as this can simplify compilation and execution of a given program.

Hardware Independence: A quantum programming language should be portable, i.e., independent of a particular hardware platform. A language should be kept as general as possible, perhaps even using the *Quantum Turing machine* as its target platform.

Extension of Classical Languages: Quantum programming languages which extend known classical languages are likely to find a wider user base than completely new languages.

5.2 Particularities of Quantum Systems Affecting Language Designs

The inability to clone an unknown quantum state has a direct effect on the behavior 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 classical programming, this will modify the variable on the right hand side of the assignment, and it is not possible to restore the previous value.

It is possible for two qubits identified by separate variable names to become entangled, so that the manipulation of one variable has an effect on the other. There is no analogy to this in classical programming, but this is more of an issue for the programmer than it is for the language designer.

5.3 A Functional Language: QPL

There are several languages which meet the requirements set out previously to varying degrees [2,9,12]. Our work focuses on one in particular, Peter Selinger's QPL [15], due to its 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

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

Fig. 6. The syntax of Peter Selinger's QPL, reproduced from [15].

QPL; this allows the syntax to enforce certain requirements of quantum theory, such as the no–cloning theorem. 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 on lists and trees, which in QPL, may contain quantum as well as classical data. The syntax of QPL is reproduced from [15] in Figure 6.

5.4 Code Templates for Quantum Operations

Code generation, for our purposes, is the process of producing machine 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 in the QPL language, and we have also considered the decomposition of large operations, so they may be implemented directly using the limited number of instructions available.

Code templates are used in compilers to provide a set of assembly instructions which correspond to a particular construct in the source program, such as an expression, a loop, or a variable declaration. We will now define code templates for those constructs which are quantum in nature; we will not give details of classical constructs here. Specifically, we will cover the declaration of quantum data, its manipulation and eventual measurement. The mapping of a high–level QPL construct to the corresponding SQRAM assembly code is expressed as a function

$$\textit{Translation} : \text{QPLTerms} \mapsto \text{SQRAMTerms}$$

In QPL, a new qubit is declared using the statement:

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

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

$$\textit{Translation}[(\textbf{new qbit } \text{q:=0})] = \textsf{\textbf{AQBIT}}$$

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: $(q* = U)$. 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:

$$\textit{Translation}[(q* = U)] = \textsf{\textbf{GATE}}\ q\ U$$

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:

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

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 result is 0, jump to ELSE
        P              ; Execute command P if result is not 0.
        LOADL   0      ; Unconditionally jump to END,
        JUMPZ   END    ; by loading 0 and jumping to 0.
ELSE:   Q              ; Execute command Q.
END:                   ; 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.

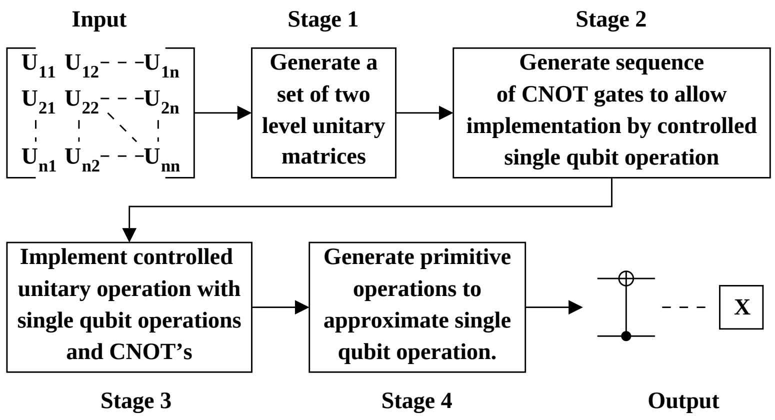
Input
U_{11} U_{12} - - U_{1n}
U_{21} U_{22} - - U_{2n}
U_{n1} U_{n2} - - U_{nn}
Stage 1
Generate a set of two level unitary matrices
Stage 2
Generate sequence of CNOT gates to allow implementation by controlled single qubit operation
Implement controlled unitary operation with single qubit operations and CNOT's
Stage 3
Generate primitive operations to approximate single qubit operation.
Stage 4
X
Output

Fig. 7. Decomposition of an Arbitrary Unitary Matrix

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.

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 [11]. 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 [18].

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:

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

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 [20], with the result that simply performing this stage could bring arbitrary operations closer to being realizable.

We will not go deeply into the mathematics involved as it can become reasonably complicated; for a coverage of this see [20,18,11]. It has been observed [11] 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 [18] 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:

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

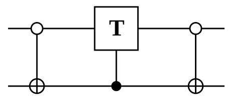

Fig. 8. Circuit implementing Equation 2.

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 [11] and a description within the context of our compiler is provided by [18].

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]. Note that we will only consider Controlled–U gates with a single control; for details of how the techniques apply to more controls please consult [18].

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 [1,18]).

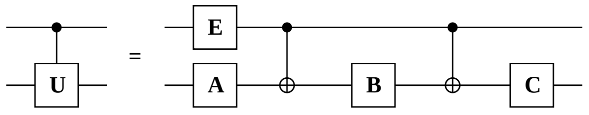

Fig. 9. Implementation of an Arbitrary Unitary

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

We have presented a simple machine architecture for practical quantum computation, and shown, in broad terms, how high–level quantum programming languages may be compiled to assembly language targeted at this architecture.

We began by presenting a machine design based on the QRAM model due to Knill. We discussed the instruction set and method of operation for the classical component so that it could be used as a standalone processor or as a control mechanism for a quantum component. We then discussed the quantum component and its instruction set, which is universal for quantum computation.

We entered into the issue of generating instructions for the SQRAM machine from a QPL program. Part of this involves creating 'code templates' for the various constructs in the QPL language, and part of it involves decomposing complex operations into those suitable for our SQRAM model.

We consider the work presented here a great success; there is potential for much improvement and refinement.

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 physical matters which would affect the behavior of a real SQRAM device. 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 our design.

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.

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 developed by Peter Selinger.

The QPL compiler 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.

6.3 Communication and Concurrency

Williams' thesis [18] describes a preliminary effort to integrate constructs for communication and concurrency into the QPL language and SQRAM simulator. Such constructs are available in the language CQP due to Gay and Nagarajan [8], which allows the description of quantum protocols, such as quantum key distribution and quantum teleportation. We aim to incorporate support for CQP in the QPL compiler; the simulator has already had such support added.

Acknowledgements

We gratefully acknowledge the feedback provided by the anonymous reviewers.

References

[1] Barenco, D., C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin and H. Weinfurter, *Elementary gates for quantum computation*, Phys. Rev. A **52** (1995), p. 3457.

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

[3] Black, P. E. and A. W. Lane, *Modeling quantum information systems* (2004), unpublished.

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

[5] Cleve, R., A. Ekert, C. Macchiavello and M. Mosca, *Quantum algorithms revisited*, Proc. Royal Soc. London, Series A **454:1969** (1998), pp. 339–354.

[6] Dirac, P., "Principles of Quantum Mechanics," Oxford Science Publications, 1958, fourth edition.

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

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

[9] Knill, E., *Conventions for quantum pseudocode* (1996), Technical Report LAUR-96-2724, Los Alamos National Laboratory, http://citeseer.ist.psu.edu/knill96conventions.html.

[10] Moore, G., *Cramming more components onto integrated circuits*, Electronics **38** (1965).

[11] Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information," Cambridge University Press, 2000.

[12] Omer, B., "A Procedural Formalism for Quantum Computing," Master's thesis, Department of Theoretical Physics, Technical University of Vienna (1998).

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

[14] Pritzker, Y., *Simulation of quantum computation on Intel-based architectures*, http://citeseer.ist.psu.edu/217822.html.

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

[16] Svore, K., 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 (2004), pp. 145–162.

[17] Turing, A., *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Soc **2** (1936), pp. 230–265.

[18] Williams, D., "Quantum Computer Architecture, Assembly Language and Compilation," Master's thesis, Department of Computer Science, University of Warwick (2004).

[19] Wootters, W. and W. Zurek, *A single quantum cannot be cloned*, Nature **299** (1982).

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