Contents

List of Figures v

List of Algorithms vii

I Introduction to Quantum Computing 3

1 Introduction 5
- 1.1 Background and History . . . 5
- 1.2 Structure of This Work . . . 6
- 1.3 Identification of New Work . . . 7

2 Background 9
- 2.1 Theory of Quantum Computation . . . 9
  - 2.1.1 Qubits . . . 9
    - 2.1.1.1 Multiple Qubits . . . 10
  - 2.1.2 Gates and Operators . . . 11
    - 2.1.2.1 Multi–Qubit Gates . . . 11
    - 2.1.2.2 Controlled Gates . . . 12
  - 2.1.3 Circuits . . . 13
  - 2.1.4 Key Principles . . . 14
    - 2.1.4.1 Reversibility . . . 14
    - 2.1.4.2 Quantum Parallelism . . . 14
    - 2.1.4.3 Measurement . . . 15
    - 2.1.4.4 Entanglement . . . 15
    - 2.1.4.5 No Cloning . . . 16
  - 2.1.5 Notation . . . 16

3 Literature Review 17
- 3.1 Linear Algebra and Mathematics . . . 17
- 3.2 Theory of Quantum Computation . . . 17
- 3.3 Quantum Hardware . . . 18
- 3.4 Quantum Programming Languages . . . 19
- 3.5 Quantum Circuit Generation . . . 20

3.6 Concurrency and Communication 20
3.7 General work on compiler design 21

II A Quantum Computation Architecture and its Programming Methods 23

4 A Model of Quantum Computation 25
4.1 Using Models 25
4.1.1 Quantum Turing Machines 25
4.1.2 The QRAM Machine 26
4.1.3 Physical Implementations 26
4.1.3.1 Nuclear Magnetic Resonance 26
4.1.3.2 Ion Traps 27
4.2 Design of a QRAM Machine 27
4.2.1 Overview 27
4.2.2 The Classical Part 27
4.2.2.1 A Classical Instruction Set 29
4.2.3 The Quantum Part 30
4.3 Universal Operations 31
4.3.1 A Universal Set of Gates 31
4.3.2 A Quantum Instruction Set Extension 33
4.3.3 An Example - Deutsch's Algorithm 34
4.4 Simulation of a QRAM Machine 35

5 Quantum Programming Languages 37
5.1 Limitations of the Circuit Model 37
5.2 Programming Languages 38
5.2.1 Requirements for a Quantum Programming Language 38
5.2.2 Issues in Language Design 39
5.2.2.1 Problems Caused by the No–Cloning Theorem 39
5.2.2.2 Problems Caused by Destructive Measurement 40
5.2.2.3 Problems Caused by the Principle of Entanglement 40
5.3 Existing Languages 41
5.3.1 Features 41
5.3.2 Languages 41
5.4 QPL 42
5.4.1 Static Type System 42
5.4.2 Data and Control Structures 42
5.4.3 Syntax 43
5.4.4 Limitations and Future Directions 43
5.5 Changes to Selingers version 44
5.5.1 Allow Definition of New Operators 44

5.5.2 Different Data Types 44

6 The QPL Compiler 45

6.1 Compilation Issues Not Concerned with Decomposition 46
6.2 Code Templates for Quantum Operations 46
6.2.1 A classical example 46
6.2.2 Declaration of Quantum Types 47
6.2.3 Manipulation of Quantum Types 47
6.2.4 Measurement of Quantum Types 47
6.2.5 A Quantum Example 48
6.3 Decomposition of Complex Operations 48
6.3.1 Overview 49
6.3.2 Generating Two-Level Unitary Matrices 50
6.3.3 Generating Controlled Unitary and CNOT Gates 54
6.3.4 Implementing Controlled Unitary Gates 58
6.3.4.1 Implementing Single Control Unitary Gates 58
6.3.4.2 Implementing Multiple Control Unitary Gates 59
6.3.5 Implementing Single Qubit Gates 64
6.3.6 Optimisations 65
6.3.6.1 Merging Single Qubit Gates 65
6.3.6.2 CNOT Optimisations 66
6.3.6.3 'Almost Equivelent' Circuits 66

III Integration of Communication and Concurrency 69

7 Communication 71

7.1 Classical Communication 71
7.1.1 A Classical Channel 71
7.1.2 QRAM Extensions 72
7.1.3 The Complete Classical System 73
7.1.4 An Example - Quantum Teleportation 73
7.2 Quantum Communication 75
7.2.1 Properties of Communicating Quantum Systems 75
7.2.2 A Quantum Channel 76
7.2.3 Quantum QRAM Extensions 77
7.2.4 The Complete Quantum System 78
7.2.5 An Example - Super–Dense Coding 78
7.3 Simulation of Communication and Concurrency 80
7.4 CQP and its relation to the Parallel QRAM model 82

8 Conclusions and Future Work (More to be done...) 85
8.1 Conclusion 85
8.2 Future Work 86
8.2.1 The QRAM Model and Simulator 86
8.2.2 The QPL Compiler 87

Bibliography 88

A Design of the QRAM Simulator 93

B Simulation of Deutsch's Algorithm 95

C Output of Compilation Process 99

D Output of Decomposition Process 103

List of Figures

2.1 A Quantum Circuit for Creating an Entangled Pair 14

4.1 A QRAM Machine 28
4.2 The Process of Quantum Computation 30
4.3 A Circuit Implementing Deutsch's Algorithm. 34

6.1 The Compilation Process 45
6.2 Decomposition of ArbitaryArbitrary Unitary Matrix 49
6.3 Circuit implementing Equation 6.11. 55
6.4 Implementation of Equation 6.1 as CNOTs and Controlled-Us. 56
6.5 Implementation of an Arbitrary Special Unitary 58
6.6 Implementation of an Arbitrary Unitary 59
6.7 Implementation of 3 control–line unitary operation due to Barenco *et al*. 60
6.8 Implementing a Controlled–U gate in terms of Xor–Controlled gates. 62
6.9 Our Implementation of a 3 control–line unitary operation. 63
6.10 Removing Redundant Single Qubit Unitaries 65
6.11 Replacements to make our implementation equal to Barenco *et al's*. 67

7.1 Classical Communication between two QRAM machines. 73
7.2 A Circuit for Quantum Teleportation 74
7.3 Concurrency diagram for Quantum Teleportation. 75
7.4 Quantum Communication between two QRAM machines. 78
7.5 Circuit for Super–Dense Coding 79
7.6 Concurrency Diagram for Super–Dense Coding 80
7.7 Two QRAM machines sharing the universe of qubits. 81

List of Algorithms

1 A QRAM Program Implementing Deutsch's Algorithm. 35
2 QPL Example Program 48
3 Compiled Version of QPL Example Program 49
4 Generation of two-level unitary matrices from arbitary operation. 53
5 Generation of CNOTs and Controlled-Us from two–level unitaries 57
6 Implementation of Controlled–Us with Single Control. 60
7 Implementation of Controlled–Us with Multiple Controls. 63
8 Code for Alice's Machine Implementing Quantum Teleportation 75
9 Code for Bob's Machine Implementing Quantum Teleportation 75
10 QRAM Byte Code For Alice Implementing Super–Dense Coding 79
11 QRAM Byte Code For Bob Implementing Super–Dense Coding 80
12 Quantum Teleportation in CQP 82

Abstract

We present the design for a quantum computer based on the QRAM architecture, detailing it's method of operation when executing both classical and quantum algorithms. We also define an assembly–like instructions set for this machine which allows it to be universal in the set of quantum algorithms it can represent.

Next we look at the issue of compiling a high–level quantum programming language into 'byte–code' for the machine defined previously. We study transformation of programming language constructs, decomposition of complex quantum operations, and some possible optimisation techniques.

Lastly we introduce the notion of communication and concurrency to our quantum computer. We add the necessary instructions to it and demonstrate quantum teleportation and the dense–coding protocol. We then look at high–level languages allowing communication and discuss how they may be compiled to our new architecture.

Part I

Introduction to Quantum Computing

Chapter 1

Introduction

1.1 Background and History

It was as long ago as 1965 that Gordon Moore first made his famous observation that the number of components which were being fitted onto integrated circuits had been growing at an exponential rate [Moo65]. This trend (which has continued right up till the present day) is a result of research conducted by academic and industrial institutions with a desire to solve increasingly complex problems on their machines. Despite these advances there remain many problems which can still not be solved today, meaning there is as much enthusiasm as ever to keep the field moving forward at the speed it has been.

It has been suggested in the past that this rate of technological advancement cannot continue indefinitely. Had an engineer in the field 20 years ago been asked to speculate on the situation today, it is unlikely that they would have imagined that the miniaturisation process would have continued in the way that it has. However, there is ultimately a limit which must be reached, and which is enforced by the laws of physics rather than our own ability to build devices. Computers store their internal state by means of transistors holding an electric charge, and it is the tendency for smaller and smaller transistors to be developed which has allowed Moores law to remain true. At the current rate it is expected that transistors will shrink to the size of an atom by the year 2020.

The real problem is not simply one of manufacturing devices of such small size, it is the far more significant problem that even if we could manufacture them they would not behave in the same way as we are used to. As these components get progressively smaller, the laws of classical physics which describe their behavior become insufficient. Eventually it is conceivable that the storage device for system state will be reduced to the size of a single atomic particle. At this point it is the laws of quantum mechanics which start to govern their behavior, and these can be radically different from those we have grown accustomed too.

Currently integrated circuit designers and engineers spend much and effort time trying to overcome these quantum effects or find new ways around them. However, over the last 20 years it has been realised that these effects can actually be put to use to improve the computational power of the machines. The interest started when Richard Feynman observed [Fey82] that it was not possible to simulate efficiently certain quantum processes using a classical computer, and he suggested

that simulation could be made more efficient if it actually made use of certain aspects of quantum behavior (although he didn't clarify details on how this could be done).

Although Feynman inspired a significant amount of interest in the field it was only with the development of algorithms (designed exclusively for quantum hardware) which had provable advantages over classical algorithms that the rest of the computing community started to take notice of what was happening. The most significant of these was perhaps Peter Shors algorithm for the factorisation of large numbers (due to the implications it had for breaking down the security of public key cryptography systems[1]), but others (such as Grovers database search) also aroused a good deal of interest.

Unfortunately quantum computing is a field in which the theory is very much ahead of the practice. Although there is a fairly good understanding of quantum computation and quantum information theory (and it is now an active research area), it will be a long time before we see an actual quantum computer capable of performing useful computational tasks. This is largely due to the physical difficulties involved in building and controlling a device working at such small scales, and although real devices have been built (using techniques such as ion traps or polarisation of photons) they are very much in the realm of the research lab.

Despite problems building actual quantum machines, researchers have a fairly good knowledge of the underlying theories and rules. Classical computing defines methods of storage (bits) and logical descriptions of the operations which can be performed on them (boolean algebra), and quantum machines have similar concepts. In some ways quantum architectures can be thought of as extensions to the classical architectures, though it is often not appropriate to do so. Algorithms and ideas can be developed based on these models in the hope that they can be realised sometime in the future.

1.2 Structure of This Work

Quantum computation is still a new and rather specialised discipline within computer science, and so in order to keep this work as self contained as possible the necessary background material will be reviewed. This is provided by Chapter 2, and then Chapter 3 provides coverage of the important literature in the field to date.

Chapters 4,5 and 6 then cover the core of the work, presenting an architecture for a quantum computer and discussing the design and implementation of a language suitable for describing quantum algorithms. This includes a detailed look at the process of generating low level machine code instructions from higher level descriptions of operations.

The model of a quantum computer is then extended in Chapter 7 with the capability to operate concurrently with another quantum computer, and also to communicate both classical and quantum data between them. It also builds on this with a study of the changes which need to be made to a quantum programming language in order to support such concurrency and communication.

[1] As well as potentially providing a way to weaken classical cryptographic systems, quantum computing has also provided a new way of carrying out secure communication. Systems using this new method of 'Quantum Key Distribution' are available today, and are not even vulnerable to quantum computer attacks.

1.3 Identification of New Work

As this thesis forms the basis for a Masters degree course it is desirable (though not strictly required) that some form of original material is presented. Some parts of it are simply a review of existing works, some parts apply existing work in new ways, and some genuinely new. This section aims to clarify what can be found in each chapter.

Chapter 2 and 3 contain (as mentioned) coverage of the necessary background theory and a review of existing work to date. Needless to say this is not original work but it is necessary to make the thesis as self contained as possible.

Chapter 4 contains the design of a QRAM machine which is, to the best of our knowledge, original. We do not claim to have come up with all the ideas behind it (the literature review covers some existing work) but it's design and instruction set are ours.

The information on quantum programming languages in Chapter 5 is largely not original, it is a summery of existing works. It does, however, contain some contrasting and comparison between the different available languages.

Chapter 6 describes several mathematical procedures which are used to decompose high–level operations into low–level operations as part of the compilation process. These procedures are not new, though they did come from a variety of different sources and have been expressed algorithmically instead of just as a mathematical procedure. Also they have been integrated in a new way and some efficiency analysis has been performed which has not been previously studied.

The material in Chapter 7 studying the concurrent operation of multiple QRAM machines is, again, new to the best of our knowledge. We present our own models of execution and the extensions to the design of the QRAM machine are also ours. The extensions to QPL (which are also covered in this chapter) are the work of Simon Gay and Rajagopal Nagarajan, this is new work in the field of quantum programming languages. The work in this thesis covering the relationship between these extensions and the concurrent QRAM model is therefore new by it's very nature.

Chapter 2

Background

2.1 Theory of Quantum Computation

Quantum computation is a large and complex field, and it is the purpose of this chapter to provide the background information required to understand the rest of the work. A more detailed explanation of most of the concepts presented here can be found in [NC00], though for some topics it will be necessary to consult other works referenced throughout the text.

2.1.1 Qubits

The fundamental element in quantum computation is the quantum bit, or qubit. It is a concept analogous to the bit in classical computing in that it is the smallest unit of storage, but is much more powerful (even a single qubit can have real uses). Like a classical bit it can exist in the 2 basic states of $\{0,1\}$, but unlike the classical bit it can also exist in a superposition (or linear combination) of the 2 states, essentially giving an infinite number of possible values in a continuous space.

Despite the fact that this qubit can exist anywhere between the states 0 and 1, it can only be measured to be either a 0 or a 1. Furthermore, performing this measurement actually changes the state of the qubit so that is no longer in a superposition, hence any further measurements of its state will always yield the same value. The state of a qubit can be represented using the following Dirac notation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

In Equation 2.1 the (complex) values α and β represent the probability amplitudes for the qubit, where the probability amplitude is the square root of the probability that the qubit will collapse to the corresponding state when measured. Hence :

$$\alpha^2 + \beta^2 = 1 \tag{2.2}$$

Alternatively a vector notation can be used, this is often more useful when applying operators

to qubits as the operators can be represented by matrices. Equation 2.1 would be expressed as:

$$\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.3}$$

When using the vector notation we imagine listing the base states of the system in ascending binary order in a column format. The probability amplitudes are then listed vertically (forming the vector) to correspond with the position of the base states. For our single qubit system there are only two base states, $|0\rangle$ and $|1\rangle$, so when listed vertically next to the definition of ψ above it can be seen that α is the probability amplitude for state $|0\rangle$ while β is the probability amplitude for the state $|1\rangle$.

2.1.1.1 Multiple Qubits

A system which is composed of multiple qubits $\{x_1, x_2 \ldots x_n\}$ is represented by any of the following Dirac notations, all equivalent:

$$\begin{aligned} \psi &= |x_1 x_2 \ldots x_n\rangle && (2.4) \\ &= |x_1\rangle |x_2\rangle \ldots |x_n\rangle && (2.5) \\ &= |x_1\rangle \otimes |x_2\rangle \otimes \ldots \otimes |x_n\rangle && (2.6) \end{aligned}$$

The operator $\otimes$ denotes the tensor product, and is used to combine smaller systems of qubits into larger ones. As an example, the two single qubit systems:

$$|\psi_1\rangle = \alpha|0\rangle + \beta|1\rangle, |\psi_2\rangle = \gamma|0\rangle + \delta|1\rangle \tag{2.7}$$

Can be combined to form the two qubit system:

$$\begin{aligned} |\psi_1\psi_2\rangle &= (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) && (2.8) \\ &= \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle && (2.9) \end{aligned}$$

It can be seen that this yields the 4 base states $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, and more generally the number of base states is given by 2^n where n is the number of qubits in the system. As before, the system can exist in a superposition of all base states, collapsing to one of them when measured. It is this behavior which makes quantum parallelism (discussed later) such a powerful tool for computation.

Again the state of the system can be represented using the (sometimes more useful) vector notation, as before we imagine the base states being listed in ascending binary order:

$$\psi = \begin{bmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{bmatrix} \tag{2.10}$$

The length of the vector representing a system of qubits grows exponentially with the number of qubits in the system.

2.1.2 Gates and Operators

Given a set of qubits, it is possible to apply operations to them to change their state, much like in classical computing. However the quantum operators which are available are much more powerful and flexible, and are represented by matrices which operate on the vectors in a complex vector space. Any matrix can be used provided it satisfies the condition expressed in Equation 2.11.

$$A^{\dagger}A = I \tag{2.11}$$

$$\Rightarrow A^{\dagger} = A^{-1} \tag{2.12}$$

Where $A^{\dagger}$ denotes the adjoint (also known as the conjugate transpose).

Such a matrix is called *unitary*, and this requirement for the matrix to be unitary is the only restriction on the operators which can be used. The restriction firstly ensures that the operation is reversible (as it will always have an inverse equal to its adjoint), and this is a requirement of operations in quantum computing as will be discussed later. Secondly the unitary property prevents the operation from modifying the length of the vector, essential because the squares of the components must always sum to 1.

An example of an operation satisfying the above criteria is given below:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.13}$$

This is known as the Hadamard gate (note that we will be using the terms *operator* and *gate* interchangeably) and it is widely used within quantum computing. It will be described properly when we make use of it in Chapter 4; for now simply observe that it satisfies the properties of a valid gate.

2.1.2.1 Multi–Qubit Gates

When specifying an operation on a single qubit system the unitary matrix used is always $2x2$. Naturally this can be extended to systems with a higher number of qubits, and it will come as no surprise that this results in exponentially larger matrices.

The composition of single qubit operations into operations on multiple qubits is done by application of the matrix variant of the tensor product. For example if we have a 2 qubit system and we wish to apply the Hadamard gate to both qubits, then the resulting system transformation is

given by:

FILL THIS IN...

2.1.2.2 Controlled Gates

Many gates fall into the category of controlled gates; that is they have a behaviour which is dependant on on some qubit other than the one on which they are operating. A common example of this would be to have a unitary operation applied to a qubit (called the target) only if another qubit (called the control) is set to $|1\rangle$. If the state of the control qubit is $|0\rangle$ the target qubit remains unchanged.

It is useful to see in more detail how this works on as such controlled operations are used quite heavily later on. Let us assume we have a two–qubit system (called ψ) where our qubits are called t and c for target and control respectively, and c is the most significant bit. We start by assuming that c is in the state $|0\rangle$and so ψ is given by:

$$\psi = \begin{bmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{bmatrix} \tag{2.14}$$

We also have a single–qubit operator U which we wish to apply to t in a controlled manner:

$$U = \begin{bmatrix} w & x \\ y & z \end{bmatrix} \tag{2.15}$$

Because U is a $2x2$ matrix it is necessary to somehow embed it into a $4x4$matrix so that it can be applied to the two qubit system. While doing this we also wish to establish the property of$U$ being applied to t conditional on the state of c. The two–qubit unitary operator which does this is given by:

$$Cont-U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & w & x \\ 0 & 0 & y & z \end{bmatrix} \tag{2.16}$$

If we now apply $Cont-U$ to the state ψ then ψ is transformed as follows:

$$\psi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & w & x \\ 0 & 0 & y & z \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{bmatrix} \tag{2.17}$$

$$\Rightarrow \psi = \begin{bmatrix} \alpha \\ \beta \\ 0 \\ 0 \end{bmatrix} \tag{2.18}$$

That is, t has remained unaffected (as it should because c is $|0\rangle$). If we now consider the situation where c is $|1\rangle$ then ψ is given by:

$$\psi = \begin{bmatrix} 0 \\ 0 \\ \gamma \\ \delta \end{bmatrix} \tag{2.19}$$

And hence and application of $Cont-U$ to ψ yields:

$$\psi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & w & x \\ 0 & 0 & y & z \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \gamma \\ \delta \end{bmatrix} \tag{2.20}$$

$$\Rightarrow \psi = \begin{bmatrix} 0 \\ 0 \\ w\gamma + x\delta \\ y\gamma + z\delta \end{bmatrix} \tag{2.21}$$

Which *does* represent a manipulation of the system (as expected because c was $|1\rangle$). Note that it is possible to have gates with inverted controls (and we will do so later) such that the state $|0\rangle$ causes the operation to be applied. Also the situation can become more complex such that controlled gates may have multiple target bits as well as multiple control bits, allowing the state of a *set* of qubits to determine the application of an operator to another *set* of qubits.

2.1.3 Circuits

The use of Dirac notation or vector/matrix form is useful for a mathematical analysis of a quantum algorithm, but sometimes it helps to be able to visually 'see' what is happening. Quantum circuits provide a way of describing algorithms which is more intuitive. The use of quantum circuits to express algorithms comes a step closer to showing something which can actually be created, rather than the abstract notation of the mathematics. It must be realised however that this is strictly

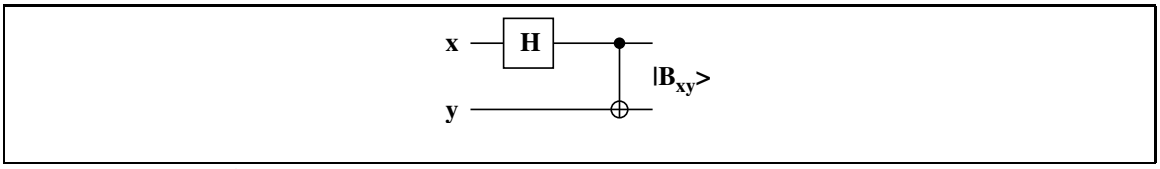


Figure 2.1: A Quantum Circuit for Creating an Entangled Pair

a notational issue, as they are fundamentally capable of describing the same algorithms.

In a similar way to classical circuits, quantum circuits consist of a set of gates which operate on their inputs. The operations performed by these gates is again specified by a matrix, which must meet the same criteria as outlined previously. Qubits are represented by parallel lines which form the inputs and outputs of the gates, though it is important to realise that these lines do not necessarily correspond directly to physical connections. The circuit notation is independent of the actual physical system used to implement it, and the lines could instead represent another property such as the passage of time. As an example a simple circuit for creating an entangled pair is presented in Figure 2.1.

Despite the fact that such circuits bear a superficial resemblance to classical circuits, there are different rules which govern what may and may not be done. Due to the requirement for reversibility there can be no fan-in (as this implies a loss of information) and circuits may not contain loops, that is they must be acyclic. Fan out is also not permissible due to the no-cloning property making it impossible to produce copies of the state.

2.1.4 Key Principles

There are several principles of quantum physics which cause computers based on them to behave in a very different way to conventional computers. These principles have an important effect on the theory of computation, and on the way that algorithms are designed.

2.1.4.1 Reversibility

FILL THIS IN...

2.1.4.2 Quantum Parallelism

We have described how, given a quantum system in some state, it is possible to apply operations to it to manipulate that state provided the operations meet some basic requirements outlined earlier. We have also described how it is possible for a quantum system to exist in a superposition of multiple states at once. It is the combination of these two abilities which allows for quantum parallelism.

If an input register is prepared in a superposition of several values and some quantum operator is applied, then the result is a superposition of all the output values. That is, a single application of the operator can operate upon multiple inputs at the same time.

2.1.4.3 Measurement

An arrangement involving quantum parellism implies huge computational advantages, as a function can be applied to a number of inputs (exponential in the number of qubits) simultaneously. The problem is that upon measuring the qubits they collapse into the value of one of the outputs, and the other values are lost. Worse still, it is not even possible to choose which of the results you get. It would appear that this loses any computation advantage gained by the use of quantum parallelism.

Although it is not possible to directly read all the outputs of the system, there are techniques which can be applied in order to gain useful information. For example it may be possible to apply a set of transformations in order to increase the chance of measuring the desired output (this is used by Grovers search algorithm), or alternatively it may be possible to deduce some property of multiple outputs (such as the period, as used by Shors algorithm).

2.1.4.4 Entanglement

It was observed in Section 2.1.1.1 that the tensor product can be used to express several single qubit systems as one larger system. It is generally possible to perform the reverse of this operation, that is to factorise a larger system into its single qubit components:

$$|\psi\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle \tag{2.22}$$

$$\Rightarrow |\psi\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) \tag{2.23}$$

However, if we consider the state:

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.24}$$

And relate the coefficients to those in Equation 2.22 we see that $\alpha\gamma = \frac{1}{\sqrt{2}}$ and $\beta\delta = \frac{1}{\sqrt{2}}$, therefore none of $\alpha, \beta, \gamma, \delta$ can be 0. Yet we also see that $\alpha\delta = 0$ and $\beta\gamma = 0$, implying that α or δ must be 0 and β or γ must be 0. This contradiction has arisen due to the incorrect assumption that a state of the form in Equation 2.22 can always be expressed in the form of Equation 2.23. A state which cannot be factorised in this way is said to be *entangled*, and operations on one qubit in the system can affect another.

The results of entanglement are most clearly seen during a measurement operation. Consider the non-entangled state:

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \tag{2.25}$$

If a measurement is performed on the first qubit, and determines it to be a $|1\rangle$, then the system collapses into the state:

$$|\psi\rangle = \frac{c|10\rangle + d|11\rangle}{\sqrt{|c|^2 + |d|^2}} \tag{2.26}$$

From this state the second qubit could be measured as either a 0 or 1. The measurement of the first qubit has not had any effect on the measurement of the second qubit; they are not entangled. Next consider the entangled state in equation 2.24.

This is a more surprising result as the 2 qubits in this state are always going to collapse to the same value, and so the measurement of the system will either result in $|00\rangle$ or $|11\rangle$. An important point to realise is that not only does the measurement of the first qubit determine the value found when the second qubit is measured, but it actually causes the second qubit to collapse *at that same moment in time*, even though the two qubits may be separated by a significant physical distance.

2.1.4.5 No Cloning

The no cloning property of quantum mechanics states that it is not possible to take a qubit existing in an unknown state and duplicate it. This was shown by Wootters and Zurek in [WZ82], who proved there is no unitary matrix which can perform this operation. It is of course not possible to perform the duplication by measuring the first qubit as this only a probabilistic measurement giving one of the base states, and it destroys the value of the original qubit anyway. Realise, however, that it is perfectly possible to clone a qubit which exists in a *known* state.

2.1.5 Notation

e.g $A^\dagger$ vs A^*

Chapter 3

Literature Review

It is the purpose of this chapter to review existing works which are relevant to the work undertaken here. Some of these are of a more general nature covering the linear algebra and other mathematics necessary for an understanding of quantum mechanics, while others cover more specific issues such as the process of quantum compilation.

3.1 Linear Algebra and Mathematics

An understanding of certain areas of mathematics (specifically linear algebra) is crucial to fully appreciate the operation of quantum mechanical systems. The book by Barnett and Kearns [BK90] covers more general mathematics while the work by Kolman [Kol93] has a greater focus on linear algebra.

This thesis also involves a small amount of discrete maths (mainly based around sets and sequences) for which the book by Ross and Wright [RW99] provides good coverage. Only the earlier parts of this book were relevant where the notation and techniques of sets, sequences, and functions are discussed.

Lastly some mathematical web resources were useful for various parts of the work. Specifically `http://sosmath.com/` provided useful information on several areas including eigen systems, while the Wolfram (creators of Mathematica) site at `http://mathworld.wolfram.com/` was an excellent reference.

3.2 Theory of Quantum Computation

It is crucial to have an understanding of the underlying principles of quantum computation before more specialist areas can be studied, and several papers have aimed to address this need by providing a general overview of the field at an introductory level. A brief though useful summery of the history of quantum computation and the key principles are provided by Knill and Nielson [KN00], while a more detailed and comprehensive guide is given by Aharonov in [Aha98] and by Rieffel and Polak in [RP00]. These papers cover the mathematical and physical ideas behind quantum mechanics, though they focus on the concepts relevant to quantum computation such as

qubits, gates, algorithms, and error correction.

For a more thorough coverage of the field Nielson and Chuang provide an excellent reference [NC00], though some of the material contained in the book is beyond the scope of this thesis. Chapters 1-3 provide useful background material, while chapter 4 and appendix 3 were among the most useful resources for the work on quantum compilation (discussed later). Gruska has produced another classic work in the field [Gru99], which takes an approach more aimed at computation rather than general quantum mechanics, covering algorithms and complexity in more detail. Chapter 7 was particularly useful for some of the ideas behind creating quantum processors, describing both abstract theoretical models and real physical hardware.

3.3 Quantum Hardware

The first proper model of a Quantum Turing Machine was proposed by Deutsch in [Deu85], although this work built off ideas presented by Benioff and Feynman. The 'tape' of the Turing machine now holds an infinite sequence of qubits, and the state is given by that of a finite set of qubits within the sequence. Deutsch claims his model is a "universal quantum computer" as it can simulate any other quantum computer with arbitrarily high accuracy. As well as [Deu85], a good overview of the Quantum Turing Machine (and the models that led up to it) is given by Gruska [Gru99].

A more practical model of quantum computation was provided by Knill [Kni96] with his QRAM machine, essentially a classical machine augmented with a set of qubits. Control is provided by the classical machine, and it is able to set the initial state and apply quantum operations to the qubits. Most of the quantum languages designed to date are built around this model of computation. A discussion of some of the practical problems which may be faced when implementing a QRAM machine is given in [BS03], as well as a suggestion of using quantum 'byte code' for the interface between the classical and quantum components.

The work of Xue *et al.* [XH03] does not actually build upon the QRAM machine, but is appropriate for integration with it. Xue *et al.* give an architecture for a quantum CPU, which operates by performing CNOT operations and rotations along 2 axis. It is claimed that this combination allows the CPU "to approximate any n-qubit computation in a deterministic fashion".

Gruska also covers some of the physical processes which can be used to implement quantum computers, such as the Magnetic Ion Trap, Cavity QED, and Nuclear Magnetic Resonance. It is not strictly necessary to understand these for this paper, but some understanding of what can and cannot be done is useful when simulating. For example, a method of implementing unitary matrices operating on only 2 dimensions is given in [RB94], this provides motivation for part of the compilation process covered later.

Of course, the focus of this work is more on the simulation of a QRAM machine than the physical implementation. Both Omer [Ome96] and Pritzker [Pri99] are concerned with this simulation aspect, and both provide software libraries covering certain aspects of the process. The library produced by Omer is used as the basis for his work on QCL (discussed shortly), while Pritzker worked on the open source 'OpenQubit' library. Either can be used as the 'core' of a simulation

engine.

3.4 Quantum Programming Languages

The first attempt to define a programming language for use with quantum algorithms was put forward by Knill, at the same time that he suggested the QRAM machine as an appropriate architecture for running them [Kni96]. Knills language was more of a pseudocode than a complete language with proper semantics, but formed the basis for much of the work that followed.

Omer provided the first proper quantum programming language when he defined QCL [Ome98]. This was a procedural language similar in concept to C or Pascal, but with the ability to operate on quantum data. The language provided classical constructs such as loops and conditional statements for control, while allowing operations to be performed on quantum registers and for the results of these operations to be measured. Omer also provided an interactive interpreted environment for experimenting with the language, building upon his earlier work [Ome96]. He has also more recently [Ome02] given a more detailed account (with examples) of the quantum concepts behind his language.

A slightly more abstract language (qGCL) was described by Sanders and Zuliani in [SZ00]. This was a more mathematical representation, aimed primarily at being a specification language rather than a true language for computation. It was based on Dijkstra's guarded command language, and was significant because it was the only language to possess formal operation semantics.

Bettelli *et al.* present an alternative approach [BS03]. Rather than define a new language they simply extend an existing language (C++) with a set of quantum primitives. Operators are treated as C++ objects, and these objects hold the stream of quantum 'byte–code' which represents the operation. It is argued that by embedding in the object the definition of the corresponding circuit "it is possible to automatically manipulate this definition in a number of different ways". This provides opportunities for building derived circuits and performing simplification routines. Bettelli *et al.* also provide arguments for extending a language rather than defining a new one, suggesting that a new language "would invariably fall behind whenever standard programming techniques improve".

The most recent proposal for a quantum language is that given by Peter Selinger, and is known as QPL [Sel03]. Selingers work on QPL forms the basis for much of the work being undertaken here. Unlike previous languages QPL is functional (although it has a superficially imperative syntax), in that each function operates by transforming a set of inputs to outputs, rather than modifying global variables. The language is also statically typed, and admits a denotational semantics. Two key areas which Selinger fails to address are error correction (he assumes idealised hardware) and communication, the latter forms part of the work here.

Stephen Blaha takes a different approach in [Bla02] in that the focus is on producing a set of low level 'assembly' instructions rather than a high level programming language. This is similar in some ways to the byte–code presented by Bettelli *et al.* and the ideas are used for the interface between the compiler and QRAM machine in this work.

3.5 Quantum Circuit Generation

The process of breaking complex quantum operations into simpler primitive operations is a multistage process. It is widely accepted that this is possible, and many proofs of the universality of different sets of gates are available [Aha03, BW95, DiV94, Har02].

Given an arbitrary unitary matrix, Zeilinger *et al.* describe a method of decomposing it into a product of matrices acting in only 2 dimensions [RB94]. As motivation for this work they also describe the beam splitter hardware which can implement such transformations. Nielson and Chuang [NC00] provide a summery of this process with examples.

Nielson and Chuang also present a method for further breaking down such a matrix into a set of CNOT gates and a single qubit controlled unitary operation. They use 'Grey codes' to generate the sequence of CNOT gates required to swap the inputs to the original matrix, bringing the non–identity terms adjacent to each other. This allows the controlled single qubit operation to replace the multi–qubit operation.

Due to the difficulty of implementing such an operation, Barenco *et al.* present a method [BW95] for transforming it into a set of single qubit operations with no controls and CNOT gates. Due to the large number of CNOT gates produced by the above process, Iwama derives a set of transformation rules [Iwa02] which can be used to simplify CNOT circuits.

Work has been done on the compilation of single qubit operations by Harrow, who provides an implementation and anaysis of the algorithms [Har01]. This is similar to our work on decomposition except we extend it to the multi–qubit case. Harrow also verifies the upper bound on the number of gates required to implement a single qubit transformation, originally given by Solovay (unpublished) and Kitaev [Kit97].

3.6 Concurrency and Communication

One of the most useful texts for understanding the formal description of systems involving communication and concurrency was by Robin Milner [Mil89], which introduces and explains the ideas behind Milners 'Calculus of Communicating Systems' (CCS). This calculus allows a system to be described in terms of the behaviour of the agents within the system and the actions which they may perform at any given time. Agents can act concurrently and may communicate in order to synchronise their actions.

Milner later built upon his own work to eventually produce the π–calculus, the final version being described in [Mil99] but earlier papers also covered it [Mil90]. The π–calculus adds the concept of *mobile* processes to CCS, and also aims to simplify it by treating things in a more uniform way. It aims to be universal for concurrent communication in the same way that the λ–calculus is universal for functional computation.

A natural extension to work discussed earlier on quantum programming languages is to try to integrate work on communication and concurrency. This type of work has been undertaken independently by Gay and Nagarajan in [GN04] and by Jorrand and Lalire in [Lal04], the result in both cases is an algebra capable of describing communicating quantum processes. The key

difference currently between the two works is that Gay and Nagarajan build upon Peter Selingers QPL language and hence have the benefit of a static type checking system.

3.7 General work on compiler design

As well as the quantum aspects of languages discussed earlier it was necessary to do some research into the design of classical compilers. Techniques such as parsing, building abstract syntax trees, and code generation apply equally well to quantum languages (although such techniques are not supposed to be the focus of this work). Appel and Ginsberg provide good coverage of all these key areas in their book [App97], as they introduce a new language and take the reader through the theory and practice necessary to implement a compiler for it.

This material is also covered by [GBJ+00] (although it is from a more theoretical and less practical standpoint), but in addition Bal *et al.* cover less common topics such as the design of compilers for object–oriented and functional languages. It also discusses the use of tools for the automatic generation of scanners and parsers, although these were ultimately not used in this work.

However the most useful source of information was the book by Brown and Watt [Bro00]. This provides a design for a compiler for a new language, and implements it in Java. Although the work here was written in C++ there were enough similarities for design techniques (such as the use of the *visitor* pattern when working with ASTs) to be reused. This book also provides useful information on the design of virtual machines which was useful for writing the QRAM simulator.

Part II

A Quantum Computation Architecture and its Programming Methods

Chapter 4

A Model of Quantum Computation

This chapter aims to present our model of quantum computation which we then use in later chapters as a target for our compiler. We begin by discussing the existing Quantum Turing Machine model and present some of the current ideas physicists are using considering for real quantum computers. We then take the QRAM machine (originally presented simply as a concept by Knill) and flesh out the details of it's operation along with our instruction set for both classical and quantum computing. Lastly we present an example of our QRAM machine performing the well known Deutsch's algorithm, and also talk briefly about the software we wrote as a simulator for our QRAM machine.

4.1 Using Models

The effects and processes described in Chapter 2 can be put to use in a *quantum computer* to allow computation to be performed in a manner more efficient than is possible with a classical computer. At the present time we do not actually a have quantum computer, only models which dictate how such a machine might work. The first of these is the Quantum Turing Machine which provides a hypothetical model for all quantum computation, and the second is the QRAM machine which is a more realistic and feasible idea of how a quantum computer might operate.

4.1.1 Quantum Turing Machines

The classical Turing machine was proposed by Alan Turing in [Tur36]. It does not provide an actual working computer, but rather a model which is able to simulate any other computer. Hence any problem which is computable on a classical computer is computable on a Turing machine, and so the Turing machine provides a model of computability. This model has remained consistent and applicable since it's conception, despite huge advances in the speed and size of computers over the past 60 years.

The Turing machine is a simple device essentially consisting of an infinitely long tape holding a stream of 0s and 1s, and a read/write head which moves over tape. The head maintains an internal state, and also a set of instructions indicating the action which should be taken for a particular combination of internal state and tape state. Such an action might involve modifying the

internal state, the tape, or moving the head to a new position. It is fairly clear how this relates to a classical computer.

The Quantum Turing Machine (the result of work in [Ben80, Deu85]) adapts the classical (deterministic) Turing machine in two important ways. Firstly a particular state no longer has exactly one successor state, it is possible for the machine to include an element of randomness in choosing the next state. This results in a non-deterministic Turing machine, and with such a device it is possible to trade the probability of a correct result against the time needed to acquire that result. Secondly, each state is no longer required to just exist in the states 0 or 1, it can also exist in a superposition of states. This reflects the use of qubits rather than bits for the storage of data.

Although a useful model for quantum computation, the Quantum Turing Machine is not a real usable device. An idea for a more realistic quantum computer is given in [Kni96], where the QRAM machine is presented.

4.1.2 The QRAM Machine

Although it is not known for sure how a quantum computer will be realised, the majority algorithms are described in a way which is compatible with the QRAM machine. This provides a classical computer which has access to (and controls) a quantum device to be used for computation. The result of this is a device which can be controlled in a conventional way using the same kind of programming constructs as are found in conventional computing (loops, branches, etc.), but which has access to the the computational power of the quantum device when it needs it.

Where possible processing is done on the classical machine, and states are only transfered to the quantum device when necessary (in order to reduce the chances of decoherence occurring, and the problems associated with this). The quantum state is then allowed to evolve before one of the measurement techniques is applied and the results returned to the classical part of the system.

4.1.3 Physical Implementations

We aim to present a design for a QRAM machine without concerning ourselves too much with the physical implementation of such a model, but it would be wise to at least have an idea of the strategies which are being used and the limitations which are being encountered. It is also worth realising that physical implementations of quantum machines tend to work on a low number of qubits (whereas the machine to be presented works on arbitrarily high numbers of qubits), and are also relatively slow.

4.1.3.1 Nuclear Magnetic Resonance

A quantum computer based on *nuclear magnetic resonance* (NMR) uses the spin on atoms within a molecule as a basis for representing different states. A spin can be aligned with or against a strong external magnetic field, this is used as a representation of the two base states. Application of electromagnetic waves can cause the direction of alignment to change, these can be directed at

individual qubits by using a different type of molecule for each. The main drawback with NMR techniques is that it is difficult to prepare the system into the initial state.

Actual working quantum computers have been developed using this technique, including one with 7 qubits. It is therefore a useful technique (which has been used to prove the realisability of quantum computers) but there are doubts as to whether it can be expanded to many more qubits.

4.1.3.2 Ion Traps

Ions are atoms which have either gained or lost electrons, and as a result have an electric charge. It is therefore possible to control their movement by the use of an electric field, an *ion trap* [Cir95] makes use of this to confine a group of ions. The equipment typically consists of a set of bar electrodes arranged parallel to each other to form a tube (the ions form a line down the centre of this tube) and electrodes at each end to prevent the ions escaping.

The ion is used as a qubit by allowing the outer electron to exist in one of two electron shell levels, and these represent the $|0\rangle$ and $|1\rangle$ states. At the start of the computation the ions are brought to their base state by cooling them, a laser can then be used to shift the electron between states, as also to perform measurement at the end of the process. The main drawback of the ion trap technique is the states can be unstable and tend to be short–lived.

4.2 Design of a QRAM Machine

4.2.1 Overview

A QRAM machine consists of two parts, these being the classical part and the quantum part (see Figure 4.1). The classical part is very similar in design to any other stand alone computer, it has a Central Processing Unit for overall control and for evaluating arithmetic expressions, and access to a data store such as Random Access Memory. It is also capable of interacting with a user, perhaps via a keyboard and VDU.

The quantum part contains a quantum register (a collection of qubits) and the necessary hardware for manipulating their state. It is this quantum register which, through the controlled evolution of it's state, performs the desired quantum computation. This quantum part is under the control of the classical part, which issues commands to modify the state of the system or to perform a measurement on a particular qubit.

4.2.2 The Classical Part

The classical part of the QRAM machine should be familiar to any computer scientist, and so in many ways it is the least interesting. It is none the less a crucial aspect of the system as a whole, and so it's operation is worth covering at least briefly. The aim in designing it has been to create the simplest possible system able to do what is required of it, in order to allow the focus of the work to be on the quantum part of the system. It would in theory be possible to take a different design (even an existing processor) and modify it to work with the quantum hardware.

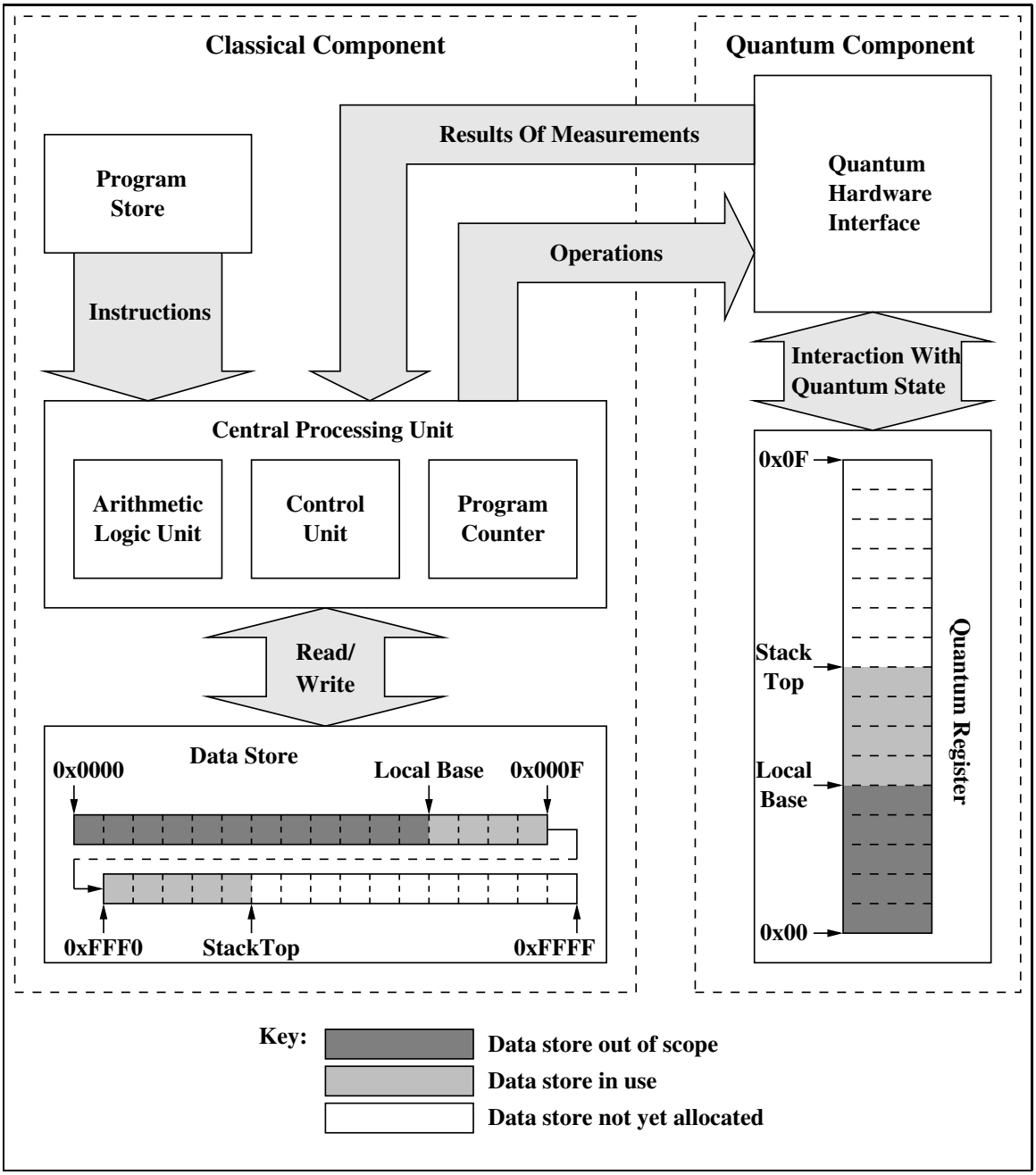


Figure 4.1: A QRAM Machine

The core is the Central Processing Unit (CPU), which in turn consists of an Arithmetic Logic Unit (ALU) and a control unit. The ALU is responsible for the evaluation of simple mathematical expressions, which are actually performed on the data store. It should be noted that the CPU does not have any general purpose registers as most real processors do, this is because the stack model provides a simpler instruction set and it is also easier for a compiler to generate code for. The CPU receives its instructions from the Program Store, again this is kept separate for simplicity as in a real processor the program and data are typically held in the same memory.

The Data Store is used for holding variables and data structures during the execution of a program. It is (as mentioned) implemented as a stack (rather than the more conventional Random Access Memory) as firstly it is used for the evaluation of expressions, and secondly it is a more intuitive model for a programming language based on the *functional* paradigm. Global variables (if supported by the language) can be stored from the address 0x0000, while the Local Base indicates the offset for variables declared in the current function. This moves as functions are entered and left. The Stack Top represents both the point where new variables are declared and the point where evaluation of expressions takes place.

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. Program execution is finished once the Program Counter goes past the end of the Program Store.

4.2.2.1 A Classical Instruction Set

Like any processor, the classical component presented above has an instruction set, that is a set of instructions used to control the flow of execution and perform calculations. There is nothing unusual about the instruction set for this part of the machine, but it is given for completeness as its complementary *quantum instruction set* is given later.

ADD: Removes the top two bytes from the top of the stack, and then appends the sum of the two items just removed. The Stack Top pointer is therefore decreased by one as two bytes have been replaced by a single byte, and the Program Counter is incremented.

HALT: Causes execution of the program to be terminated by setting Program Counter to the end of the Program Store.

JUMPZ address: If the value on the top of the stack is zero, the Program Counter is set to the value of *address*, transferring execution to this point. Otherwise the Program Counter is incremented and execution continues as normal.

LOAD offset: Retrieves the value at the address given by the specified *offset* (relative to the Local Base), and appends it to the top of the stack. Both the Stack Top and Program Counter are incremented by this operation.

LOADL value: Appends the literal *value* to the top of the stack. Both the Stack Top and Program Counter are incremented by this operation.

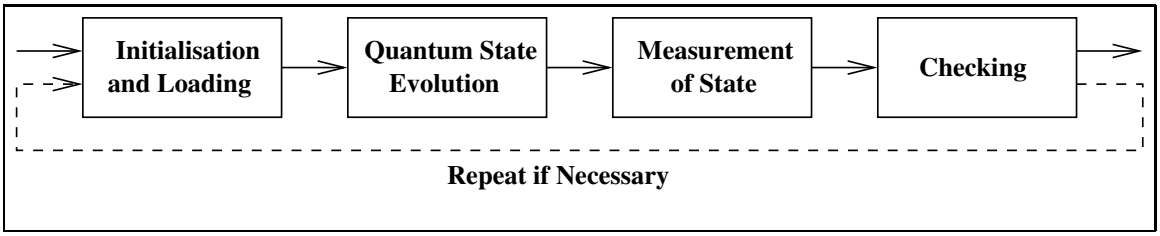


Figure 4.2: The Process of Quantum Computation

SAVE offset: Removes the value at the top of the stack and stores it at the address given by the specified *offset* (relative to the Local Base). The Stack Top pointer is decremented by this operation, while the Program Counter is incremented.

SUBTRACT: Subtracts the value at the top of the stack from the value at the address just below it. Both values are removed and the result is the appended to the stack. As with the ADD instruction the Stack Top pointer is decreased by one as two bytes have been replaced by a single byte, and the Program Counter is incremented.

4.2.3 The Quantum Part

The quantum part of the QRAM machine is where the interest really lies, and is the focus of this work. It consists of a quantum register, which is a set of qubits which can be manipulated to perform quantum computation, and a Quantum Hardware Interface (QHI) which is the hardware capable of performing the manipulation. The QHI receives from the CPU those instructions which are of a quantum nature, and via some method (laser pulses, etc) manipulates the qubits which are stored in some form. We keep away from the physics as far as possible, as the precise method of implementation is not important.

A quantum computation typically follows a pattern as outlined by Figure 4.2. For the first stage we assume the quantum hardware has a method of reseting a qubit to the $|0\rangle$ state, this is not too much to ask as it could, for example, simply require setting a photon to a particular polarisation. From this $|0\rangle$ state a series of transformations can be applied to load in the initial value. This is a process similar to the second stage, but conceptually it helps to think of it as separate so that the qubit is initialised and then manipulated.

The second stage is where the actual computation takes place. This is done by evolving the quantum system in line with the rules of quantum mechanics presented in Chapter 2, it is an important stage and so the details of it are deferred for a thorough discussion in a later section.

At the end of the computation, it is usual to perform some kind of measurement on the system to determine the results. The result of performing this measurement is that the measured qubits collapses into one of the base states, $|0\rangle$ or $|1\rangle$. The classical bit (0 or 1) corresponding to this base state is then returned to the CPU and can be used directly as a result or as a means of controlling some conditional statement.

The final stage is the checking of the result yielded by the quantum computation, to check that it gave something sensible. The result may be incorrect due to the limited capabilities of the hardware (decoherence may cause damage to the quantum state), or it may be the case that the

algorithm does not reliably give correct results. Either way, checking can usually be performed on the classical component of the QRAM machine, for example it is easy to verify that the results from Shor's algorithm are indeed factors of the input. If necessary the system can return to the first stage to perform the computation again.

As with the classical data store the quantum register is implemented as as stack. As in the classical case this gives us the ability to have qubits which are local to a particular function, but unlike the classical case we don't use it for the purpose of evaluating expressions (as conceptually this doesn't make sense). As a result the quantum instructions to be presented shortly do not generally operate on the top of the stack, but instead take a parameter indicating the address of the qubit which is to be operated on.

4.3 Universal Operations

In Section 4.2.3 we skipped very quickly over the part of the computation process in which the quantum state undergoes an evolution. Here, we come back to this issue to look at the operations which might be performed during this process.

It was stated in Chapter 2 that if the state of a quantum system is represented by a vector of probability amplitudes, then any unitary matrix of the correct size represents a valid transformation of this state. Hence we might expect that if we wished to perform an operation on three qubits we would be able to provide the Quantum Hardware Interface with and 8×8 unitary matrix and have it modify the state accordingly. The reality is that todays quantum hardware can struggle to perform simple manipulations reliably, it would be asking too much for it to perform arbitrary operations on multiple qubits.

In classical circuit theory it is well known that the NAND gate is universal; that is it is possible to implement any classical logical function using just NAND gates. There is no *single* universal quantum gate, but there are several *sets* of gates which are universal, in fact it is believed that there are infinitely many such sets of gates. These universal gates operate on one or two qubits, and are far simpler to implement than arbitrary multi-qubit transformations.

4.3.1 A Universal Set of Gates

We now present a fairly common set of gates which are known to be universal, and which form the core of the instructions in our quantum instruction set:

Hadamard: This commonly used gate creates a superposition state from a base state. That is, it performs the transformation:

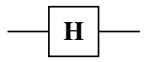
$$|0\rangle \rightarrow \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{4.1}$$

$$|1\rangle \rightarrow \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{4.2}$$

Hence each base state is taken to a state 'halfway' between a $|0\rangle$ and a $|1\rangle$, in hardware it can be implemented (for example) by passing photons through a semi-silvered mirror. The operation is more precisely defined for a single qubit as:

$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{4.3}$$

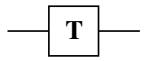
It can be seen that the Hadamard is it's own inverse as $H^2 = I$, and so applying two in succession has no effect. The gate can also therefore be used to reduce a superposition back to one of it's base states. In quantum circuit diagrams it is represented as shown:



$\pi/8$–Gate: This gate has a slightly confusing name (for historical reasons) as it does not involve a value of $\pi/8$, rather a value of $\pi/4$ as shown in the definition below:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$$

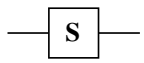
It is a single qubit operation usually represented in circuit form as follows:



Phase Gate: Another single qubit operation, the phase gate is defined by:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

Note that it can actually be implemented by two $\pi/8$ gates as $T^2 = S$, but this is obviously less efficient and as it is simple to implement it forms part of the universality set. It circuit notation it is shown as:



Controlled–NOT: The NOT operation can be applied to a single qubit to switch the probability amplitudes of the base states. Hence on a base state it acts like a classical NOT, but it can also be applied to superpositions and performs the transformation:

$$\alpha|0\rangle + \beta|1\rangle \rightarrow \beta|0\rangle + \alpha|1\rangle \tag{4.4}$$

The Controlled-NOT (or CNOT) gate is a controlled extension (as described in Chapter 2)

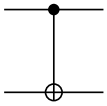
to the NOT gate. It usually has only one control bit but can have more if necessary. A CNOT gate with one control bit performs the transformation:

$$\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \rightarrow \alpha|00\rangle + \beta|01\rangle + \delta|10\rangle + \gamma|11\rangle \tag{4.5}$$

And can be represented by a matrix as:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.6}$$

Within a quantum circuit it is shown as follows, where the top line is the control and the lower line is the target:



4.3.2 A Quantum Instruction Set Extension

We now reach a point where we can extend the classical instruction set for the QRAM machine with quantum instructions for use by the Quantum Hardware Interface. This includes instructions for allocating new qubits, performing operations on their state, and measuring the result.

AQBIT: Allocates a new qubit from the top of the stack. The qubit is initialised to the state $|0\rangle$ and the quantum stack pointer is incremented.

CNOT target no_controls <control inv_flag>: Performs a NOT operation on *target* conditional on the values of the control bits. The number of control bits is given by *no_controls,* and the following sequence of pairs specifies the controls themselves. For each pair, *control* gives the bit to be used as a control while *inv_flag* is 1 if the control is to be treated as an inverted control and 0 otherwise.

GATE target ar ai br bi cr ci dr di: Applies an arbitrary single qubit operation to *target*. The values following the *target* describe the matrix representing the operation as follows:

$$GATE = \begin{bmatrix} \mathbf{ar} + \mathbf{ai}i & \mathbf{cr} + \mathbf{ci}i \\ \mathbf{br} + \mathbf{bi}i & \mathbf{dr} + \mathbf{di}i \end{bmatrix}$$

HDMD target: Applies a Hadamard gate to *target*.

MSRE target: Performs a measurement on *target*, causing *target* to collapse to one of the base states. The result of this measurement (0 or 1) is also pushed onto the classical data stack, causing the stack pointer to be incremented.

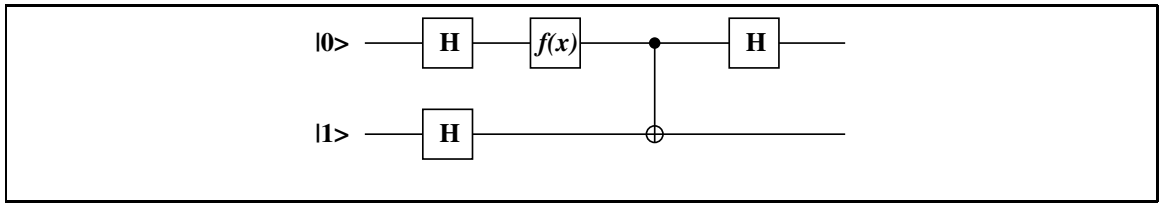


Figure 4.3: A Circuit Implementing Deutsch's Algorithm.

PHASE target: Applies a Phase gate to *target*.

PI target: Applies a $\pi/8$ gate to *target*.

It can be seen that we have chosen to include a GATE instruction, despite the fact (as was noted earlier) that any single qubit operation can be broken down into simpler operations in accordance with the principle of universality. So while it is not strictly necessary to include the GATE operation, we have done so for convenience. Performing a decomposition by hand is a long and tedious process, so including this instruction makes it much easier to write quantum assembly programs by hand. Later on we shall look at how a compiler might be used to perform this decomposition for us.

A second reason for including the gate instruction is that (as we shall see later) an implementation in terms of the universal set is only approximate (though we can approximate to an arbitrary degree of accuracy). For a more precision we use the GATE operation directly.

4.3.3 An Example - Deutsch's Algorithm

To finish this chapter on the QRAM machine, and to help consolidate the ideas presented so far, we now present an example of a program written for the QRAM machine. We will be using a revised and improved version of Deutsch's algorithm as given in [CM97], this paper should be consulted for further details of how the algorithm works as the description presented here will be fairly 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 be, 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 4.3 it is possible to evaluate $f(0) \oplus f(1)$ with out ever finding out the values of $f(0)$ and $f(1)$.

To illustrate this as QRAM assembly code we choose a function to test, we'll work with the NOT gate as it is straight forward. The NOT gate is balanced, so the result of evaluating

Algorithm 1 A QRAM Program Implementing Deutsch's Algorithm.

```
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
GATE   0x00 0.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0    ;NOT gate is our test function
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
```

$f(0) \oplus f(1)$ should be 1. The code to be executed on the QRAM machine is given in Algorithm 1.

There is a clear resemblance between the code representation and 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 takes care of 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, 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. The output of Deutsch's algorithm running on the QRAM machine simulator is given in Appendix B.

4.4 Simulation of a QRAM Machine

Part of the work associated with this chapter has been the development of a QRAM simulator to test the ideas presented. Simulation of a quantum system is difficult task, due to the exponential complexity of such systems as introduced in Chapter 2. In [Pri99] some ideas are presented for the design and implementation of a library for simulating quantum systems, the result of this work was the OpenQubit project.

OpenQubit was run as a collaborative effort to develop an open–source library which could be used in projects (such as this one) which were based on 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.

The simulator makes use of the OpenQubit library to implement the quantum part of the QRAM machine, and implements itself the classical part and the fetch–execute cycle. As far as possible the architecture of the simulated QRAM machine matches that presented earlier in Figure 4.1, and the instruction set also matches the one specified.

For those who are interested, an overview of the architecture of the simulator can be found in Appendix A, though it should be noted that this includes additions which are made in Chapter 7 to allow communication to take place between several machines.

Chapter 5

Quantum Programming Languages

Quantum systems are inherently more complex than classical systems, and this extra complexity carries through to the method used to describe the algorithms that run on them. The linear algebra which governs quantum systems can be a powerful tool for the analysis of their behavior, but it is not always the most appropriate mechanism for describing them in a way which can actually be implemented. Other methods are being developed which have a greater resemblance to those used in classical computing, and which can be more intuitive for the developers of algorithms.

Although quantum circuits are useful as a tool for expressing the design of quantum algorithms, and are universal in the set of such algorithms which can be represented, they are not an appropriate method for specifying algorithms to a computer. What is needed is a programming language which can describe (through the use of elementary constructs such as loops, conditional execution, etc) an algorithm in a step-by-step manner, in much the same way as is done when programming classical computers.

This chapter covers three key areas. Firstly we look at the limitations of existing methods of representing algorithms and ask how quantum languages could be an improvement. Secondly we determine what the key criteria of a quantum language are, and have a brief overview of some of the languages which have been presented to date. In the final section we take a more detailed look at Peter Selinger's QPL, the language upon which some later work is based.

5.1 Limitations of the Circuit Model

Although the circuit model is a useful and intuitive method for representing circuit, it is not ideal in all circumstances. At the 2nd International Workshop on Quantum Programming Languages Richard Josza made the statement:

> 'Quantum algorithms are actually classical algorithms punctuated by quantum effects'

This was made with particular reference to Shors factoring algorithm (although it applies to many others as well), the key point being that this algorithm features relatively few quantum operations compared to classical ones. Surely it doesn't make sense therefore to try and describe it using the purely quantum notation of circuits?

The quantum circuit model is also not particularly intuitive as a method for *developing* quantum algorithms. Programmers are more used to working at a 'higher level', rather than worrying about low level details of implementation. Indeed, most classical programming languages aim to abstract away such details so they need no longer be of concern.

5.2 Programming Languages

In order to combat the problems and limitations with circuit notation, quantum programming languages have become increasingly popular as a method of expressing algorithms. These languages typically have some resemblance to classical programming languages, featuring some sort of classical logic and control structures and combining this with the ability to operate upon quantum data types.

Such a language can be compiled to low-level operations appropriate for the target hardware, this achieves the desired abstraction mentioned above. The compiler can also be responsible for performing various forms of optimisation to try and increase the efficiency of the resulting circuit.

One of the key ideas behind most languages is replacing the 'quantum control' of circuits with a more 'classical control', such that they include control structures like loops and conditional statements. Indeed, Selinger states that the approach of QPL can be summarised by the slogan:

> 'Quantum data, classical control'

It has been asked in the past whether this classical control in any way limits the capabilities of the languages with respect to the range of algorithms which it can describe, it might not be surprising if this were the case. However, it has been shown in [Nie97] that this is not an issue provided the program is executed deterministically (which it is within the context of our QRAM machine).

Because of this classical control idea the QRAM machine (with it's classical processor controlling it's quantum register) makes an ideal target platform for such languages. However they are not restricted to a QRAM implementation, they are general enough to be applied to other architectures.

5.2.1 Requirements for a Quantum Programming Language

Several researchers and groups [BS03, Kni96, Sel03] have investigated the design of quantum programming languages, and have built their ideas based on certain key requirements which it is expected a such languages will meet. These are outlined below, though it should be noted that not all quantum languages to date exhibit all the characteristics:

Classical Characteristics: This simply refers to the principles which language designers have established over the years for classical languages, such as a clean syntax to make parsing easy and a set of keywords with an intuitive meaning to the programmer.

Completeness: The language must have the same expressive power as the mathematical model or quantum circuit model, by which it is meant that each algorithm which can be repre-

sented mathematically has a corresponding implementation as a program. The language is therefore universal in the set of algorithms it can represent.

Expressivity: The language must present the programmer with a sufficiently powerful set of primitives and logical constructs so that the programmer is easily able to express the design of the algorithm through the program. These should be high level primitives such as qubits, loops, and assignment, the low level implementation of which is handled separately.

Separability: There are likely to be a mixture of classical and quantum elements within the algorithm, and these should ideally be kept as separate as possible. This is important for a QRAM machine because tasks which do not explicitly gain from execution on the quantum device should be run on the classical machine to avoid decoherence problems.

Hardware Independence: The QRAM machine seems like a likely candidate for a real quantum computer, but other ideas may be presented in the future. It is also possible that some hardware will be created which is more restricted to particular tasks, such as quantum communication. Designers should therefore aim to keep their languages as general as possible, perhaps using a quantum Turing machine as their target platform.

Extension of Classical Languages: This is a less important aspect, but there are several good reasons why it could be advantageous for a quantum language to be an extension to an existing classical language. Investigation into classical languages has been going on for many years, and the result is a set of paradigms which are capable of expressing algorithms in ways which are intuitive to the programmer. Procedural, object orientated, and functional languages have all been popular, and by extending an existing language in one of these paradigms we get a language which programmers are already familiar with and which come with a set of tools ready to use. Also many classical languages are continually refined with new features, and it is desirable to avoid effort duplicating these features into new languages.

5.2.2 Issues in Language Design

Despite having a nice set of requirements, the classical and quantum aspects of a programming language do not always mix together in an ideal fashion. In Chapter 2 we introduced some of the unusual properties of quantum mechanical systems, here we look at how these properties affect the behavior of quantum languages.

5.2.2.1 Problems Caused by the No–Cloning Theorem

One of the major differences is in the behavior of assignment, and other places where variable copying is used (such as when calling functions), due to the no–cloning theorem. Within the context of classical languages we are used to statements such as:

integer x, y;
x = 10;
y = x;

The expected behavior would be for x and y to end up both containing the value 10, but with quantum data the behaviour is not so simple. Firstly there is the issue regarding initialisation, as was mentioned in the Chapter 4 the qubit cannot be easily initialised to an arbitrary state. If a language allowed a new qubit to be specified with such an arbitrary state it would be the job of the compiler to generate a series of transformations to take it there from the state $|0\rangle$.

The second problem is the direct assignment of the value in x to the variable y. With quantum data types such a simple operation is forbidden by the no–cloning theorem, meaning we need to find a creative way to achieve the same effect. One possibility is too remember that the no–cloning theorem only applies to qubits in an *unknown* state, and so it is theoretically possible for the compiler to keep track of the operations applied to a qubit and know it's state. Upon encountering an assignment the compiler could reset the target qubit to $|0\rangle$and then apply the same transformations. In practice this is likely to be difficult partly due to the overhead involved in tracking the transformations, but more importantly due to the complex entangled states which can arise meaning the operations can affect other qubits than those at which they are targeted.

A more realistic solution is likely to involve the use of *references*, as can be used in classical programming to avoid the overhead of copying large data structures. A variable is considered to point to a qubit, rather that to be it's direct representation. An assignment of x to y will then result in both variables pointing at the same qubit, and so any future operations on x will also affect y in the same way. The qubit to which x previously referred may or may not be left unreferenced.

5.2.2.2 Problems Caused by Destructive Measurement

It would not be expected within classical programming that the act of reading back a variable could actually change it's value, yet this is the very behaviour which can arise when dealing with variables of a quantum data type. There is nothing which can be done about this, it is simply a result of the underlying quantum mechanics. it was noted previously that direct assignment of one qubit to another is not possible, but what is possible is for a qubit to be assigned to a bit, this operation performs an implicit measurement. We will look at the syntax of a real language later, but as an example:

bit b;
qubit q;
... //operations on q
b = q; //measurement of q

This corresponds neatly to the MSRE instruction of the QRAM machine presented in Chapter 4, as this similarly measures a qubit and stores the result in a bit.

5.2.2.3 Problems Caused by the Principle of Entanglement

To use another classical analogy, consider the following example;

integer x;
integer y;
... //operations on x and y
print x; //prints e.g 12
... //operations on y
print x;

It would be extremely confusing if the final print statement did not yield the value 12 as the previous one did because only operations on y have been carried out. However if x and y were of a quantum data type, and the first set of operation caused them to become entangled, then this kind of behaviour is quite possible. This is not really an issue that must be directly addressed by the programming language, it is more something the programmer must be aware of.

5.3 Existing Languages

Before looking in detail at the QPL language we provide an overview of existing languages. Although there are many differences between them, they share some common features which have become well established.

5.3.1 Features

The semantics of many of the control structures are borrowed from classical languages, helping to keep quantum languages in line with the 'classical control' principle which many of them seem to adhere to. They typically have conditional statements (if... else... end) and loops which can be executed a fixed number of times or while a condition is true (loop... until... end). Some work has also been done on incorporating *probabilistic* structures which follow a particular path of execution with some probability.

The languages all give the ability to operate over quantum data types in addition to the usual classical data types. They define at least a 'qubit' data type and have a set of operations which are allowed to be performed. In addition to this some languages allow data structures (such as lists or trees) to be built up from individual qubits.

Most languages so far also adhere to one of the established programming paradigms such as functional, object–oriented, etc. It will be interesting to see if future languages continue to do so, or if some new paradigm more appropriate for quantum computation is established.

5.3.2 Languages

Given the principles established so far, there have been several attempts to provide languages with the desired features. Work done on these is discussed briefly below, though detailed discussion of the concepts involved in these languages is left until later chapters when they can be compared and contrasted.

Some of the earliest was done by Knill in [Kni96] which consolidated knowledge gained up to that point and laid down the basic ideas for other languages to build on. Knill proposed a

quantum pseudocode, aiming to provide a standard method for describing algorithms to computers as is done with conventional pseudocode. This however was a combination of program code and mathematical notation, not suitable for immediate entry into a machine.

Soon after the work done by Knill on quantum pseudocode, the language QCL was proposed by Bernhard Omer in [Ome98]. This provided a procedural language with syntax similar to 'C' or Pascal, and was the first proper language which could be used for development. Omer also provided a simulator and tools for the development of programs in his language.

The authors of [BS03] took a similar approach to QCL in terms of the architecture of their language and it's treatment of the machine on which it ran, but choose to extend the existing language 'C++'. This naturally gave their language a more object orientated nature, and they claimed that treating operators as objects rather than functions made it easier to create them and aided in their simplification and optimisation.

5.4 QPL

The most recent language to appear on the scene is Peter Selingers QPL [Sel03], and this is the language on which our work is to be based. We provide an overview of it's structure and features here but for more details the original paper should be consulted. QPL is the first language to take a functional approach[1] compared to the imperative approach of it's predecessors. It also admits a denotational semantics by the assignment of superoperators to each program fragment, however the denotational semantics will not be studied any further in this work (for more details see [Sel03]).

The language aims to be hardware independent (though Selinger specifically mentions the QRAM machine as an appropriate model), though it does assume idealised hardware. This means either hardware which does not suffer from errors and decoherence, or hardware which provides a means of correcting these problems when they occur. Perfect hardware is hard to build in practice, but it is not a problem for our simulator.

5.4.1 Static Type System

QPL features a static type system which Selinger claims prevents errors at run time due to the violation of some of the rules of quantum mechanics. For example, the syntax of QPL enforces the no–cloning principle by forbidding the assignment of qubit variable to another. It was mentioned earlier that a language could perform assignment by the use of references rather direct values, QPL does not do this it simply forbids direct assignment of qubits.

5.4.2 Data and Control Structures

As well as the expected loops and conditional control structures, QPL allow recursive procedures to be defined. This is useful because recursion can provide a natural approach for operating on

[1] Selinger is keen to emphasize in his paper that QPL is a functional language, despite the fact that the syntax looks imperative. Functions map inputs to outputs rather than operating on a set of global variables.

data structures such as lists and trees, which QPL also allows to be defined. Selinger observes that lists and trees are best represented with a classical 'structure' containing quantum data in order to fit with the 'classical control, quantum data' approach used elsewhere in the language.

5.4.3 Syntax

Selinger is more concerned with defining appropriate semantics for the QPL language rather than focusing on the syntax, for this reasons the majority of the paper is presented using flow-charts as a method of representation (after introducing how *quantum* flowcharts work). However this is not particularly appropriate for defining algorithms such that they can be compiled, and so a textual syntax for the language is also given:

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

We take the time to reproduce this here because part of the work in the next section covers the syntactic changes which have been made to the language in order to adapt it to our needs.

5.4.4 Limitations and Future Directions

There are two key weaknesses of QPL which are identified by Selinger in his paper. Firstly QPL does not allow for the definition of higher–order functions; that is the ability to treat functions as data and pass them as arguments to other functions[2]. He does raise the question of whether this is possible, and it is an ongoing area of research by Selinger himself. Progress on this was presented at the 2nd International Workshop on Quantum Programming Languages and in the associated proceedings [Sel04], the conclusion so far being that there are many semantic challenges still to be overcome despite restrictions on the number of base types and a requirement for all functions to be linear.

There is also no concept of concurrency or communication in the QPL language, Selinger argues that "the denotational semantics of the resulting language is not currently very well understood". Further work on this has again been presented at the 2nd International Workshop on Quantum Programming Languages, in two separate papers [GN04, Lal04]. The work of Gay and Nagarajan is particularly relevant as it is used later in this thesis when studying extensions to the QRAM model to allow communication.

[2]There are known quantum algorithms where this would be useful. For example Deutsch's algorithm, presented previously, requires a function as input to determine whether it is constant or balanced.

5.5 Changes to Selingers version

It has been necessary during the course of this work to make some adaptations to Selingers original definition of QPL in order to better fit in with the aims of this thesis.

5.5.1 Allow Definition of New Operators

A significant part of this work (see Chapter 6) has been concerned with the decomposition of high–level operations into the low–level operations provided by our QRAM model. Selinger does assume his language to have a built-in set of unitary transformations (he doesn't say what these are), but he doesn't provide a method of extending this set. That is, we need a syntax for defining new unitary transformations.

Selinger already allows code such as:

$$\begin{aligned} &\textbf{new}\,\textbf{qbit}\,q := 0; \\ &q* = S; \end{aligned}$$

Where S is some arbitrary unitary transform operating on one qubit (in this case). We simply extend this to allow operations to be written explicitly, so that to apply a Hadamard for example we could say:

$$\begin{aligned} &\textbf{new}\,\textbf{qbit}\,q := 0; \\ &q* = [0.707 + 0.0i,\, 0.707 + 0.0i,\, 0.707 + 0.0i,\, -0.707 + 0.0i]; \end{aligned}$$

Clearly we could extend this further if we wished and allow this new definition to be assigned to a variable to be used again later (this would simplify life for the compiler which would only have to perform the decomposition once) but for our testing purposes the above is sufficient.

5.5.2 Different Data Types

In addition to the **qbit** data type for quantum operations Selinger provides the type **bit** to represent classical bits. We replace this with the type **int** (although it could simply be added instead of replaced) as this is more appropriate for our target platform (which operates on integers). It is of course possible to represent, when necessary, a bit using the integer data type by considering all non–zero values to be 'true'. This is done within conditional jumps, for example. We also allow typical arithmetic operations (such as addition and subtration) to be performed on the interger data type.

Chapter 6

The QPL Compiler

In Chapter 4 the architecture of the QRAM machine was presented, along with a description of the available byte code commands which it can execute. Chapter 5 then specified a variation of Peter Selingers QPL language. It is the job of the compiler, discussed in this section, to transform a program defined in QPL into it's corresponding set of byte code instructions.

The task of designing and implementing compilers is an established field of computer science, so the focus of this chapter is to be on those aspects of compiling which are specifically related to quantum computation. There are several different approaches to designing a compiler but it is typically a multi–stage process similar to that shown in Figure 6.1.

The first stage, *Syntactic Analysis*, is the process of reading the input program, checking it conforms to the syntax of the language (for QPL this syntax was presented in Chapter 5), and then transforming it into a suitable representation of it's structure. This is often an *Abstract Syntax Tree*, or AST, where the nodes of the AST represent key constructs in the source file such as commands, expressions, and identifiers. However, the QPL language has a straight forward syntax and the syntactic analysis is not significantly more complex than for a classical language (at least the parts we implemented within our compiler). Hence we are not discussing syntactic analysis further.

Contextual Analysis involves checking that the languages contextual constraints are met. Any applied occurance of an identifier can be checked to make sure it has been declared, and that it is of the correct type. The type of expressions can also be infered and any type errors detected. information gathered during the contextual analysis stage is added to the AST resulting in a *decorated AST*. QPL is a statically–typed language, and it is during this stage that the type checking would take place. However, in our implementation we chose to focus on the last stage, *Code Generation*, and so will not discuss contextual analysis very much.

Code generation is the process of producing byte code instructions for each of the nodes in

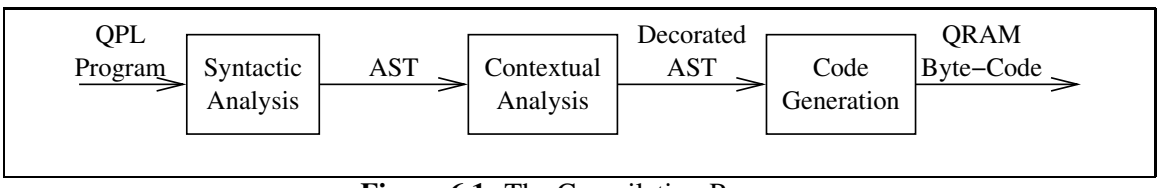


Figure 6.1: The Compilation Process

the AST, and this is where the majority of the work has been done. We have designed code templates for the quantum constructs within the QPL language, and have also done work on the decomposition of large operations into smaller ones which we can implement in our byte–code. This chapter will mainly be concerned with discussing issues in code generation.

6.1 Compilation Issues Not Concerned with Decomposition

how fan-in, fan-out, and reversibility cause problems.

6.2 Code Templates for Quantum Operations

Code templates are used within compilers to provide a set of byte–code instructions which correspond to a particular construct in our source program, such as an expression, a loop, or a variable declaration. We introduce the idea by presenting classical examples from the QPL language and QRAM machine, though we won't cover all of these as it is an issue for classical compiler design. We will instead move on to specify code templates for those constructs which are quantum in nature.

6.2.1 A classical example

In describing our code templates we will be using the syntax adopted in [Bro00], for further details on it this work should be consulted. The 'while' loop in QPL is a classical control structure, the code template for which can be specified as follows:

$execute[\textbf{while}\ b\ \textbf{do}\ P] =$

START:	**LOAD**	b	*;Start by testing if condition b is true*
	JUMPZ	END	*;If not we've finished and so jump to the end.*
	execute	*P*	*;Execute the code denoted by P*
	LOAD	0	*;Unconditionaly jump to start by*
	JUMPZ	START	*;loading zero and jumping if not zero*
END:			*;Finished*

The template name 'execute' is one of several templates called execute, overloaded on thier parameters. There are 'execute' templates for all the commands including single commands, blocks of commands, if 'commands', and while 'commands'. There are other types of template such as 'evaluate' for expressions and 'elaborate' for declarations.

The execute template for the while loop begins by testing if the condition is true, if not the contents of the loop is skipped completly. If it *is* true then the execute template for *P* (whatever type of command *P* is) is inserted. The template then jumps back to the start to retest the condition.

As mentioned there are templates for all the different constructs available in the language, but many of thenm are clasical in nature and so not of interest to us (they form part of most classical compiler design courses). We are instead interested in templates for quantum constructs,

specifically we cover the declaration of quantum types, the manipulation of those types and thier eventual measurement.

6.2.2 Declaration of Quantum Types

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

$$\textbf{new qbit}\, q := 0;$$

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

$$\begin{array}{l} elaborate[\textbf{new qbit}\, q := 0] = \\ \quad \textbf{AQBIT} \qquad \textit{;Allocates a qubit and sets it to } |0\rangle \end{array}$$

Of course, there is some work going on behind the scenes by the compiler to keep track of where the variable q is actually pointing, but this works in the same way as the classical case.

6.2.3 Manipulation of Quantum Types

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 a single qubit operation which we wish to implement directly using the GATE instruction. This is as simple as:

$$\begin{array}{l} execute[q *= U] = \\ \quad \textbf{GATE} \quad \text{q U} \quad \textit{;Applies the gate U to the target qubit q} \end{array}$$

Alternativly 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 the universal set of operation. Either way, we move into the decomposition process which is discussed in Section 6.3.

6.2.4 Measurement of Quantum Types

Mesurement is the most complex of the code templates (assuming we don't get involved with decomposition when manipulating quantum types). QPL performs mesurement 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 comand corresponding to $Q$is executed. The code template for this look as follows:

Algorithm 2 QPL Example Program

```
new qbit q := 0;
q *= H;
measure q then
  q *= S;
else
  q *= X;
```

$execute[\textbf{measure}\ q\ \textbf{then}\ P\ \textbf{else}\ Q] =$

```
        MSRE    q       ;Perform the measurement
        JUMPZ   ELSE    ;If it is |0⟩ then jump to else part
        execute P       ;Otherwise execute command P
        LOADL   0       ;Unconditionaly jump to end by
        JUMPZ   END     ;loading zero and jumping if not zero
ELSE:   execute Q       ;Execute command Q
END:                    ;Finished
```

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.

6.2.5 A Quantum Example

We finish by using the above templates to generate code for a complete (though short) quantum example. The example declares a qubit (initialised to $|0\rangle$) and applies a Hadamard operation to it to put it in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. A measurement is then performed and on the basis of this either a NOT gate or Phase gate is applied. Note that this results in one of two possible states *not* in a superposition of states. The QPL code for this is given in Algorithm 2, which assumes the existance of built–in unitary operations H, X, and S for the Hadamard, Not, and Phase gates respectivly.

By applying the code templates discussed previously we arrive at the byte–code implementation as presented in Algorithm 3. Note that we have also appended a HALT instruction, this results from an execute 'program' template which we have not yet defined but would look as given below:

$execute[P] =$

```
  execute[P]    ;Executes the commands within P
  HALT          ;Finished
```

6.3 Decomposition of Complex Operations

In Chapter 4 we discussed the principle of universiality, stating that any quantum operation can be broken down and implemented in terms of a small set of universal gates. Hence our QRAM

Algorithm 3 Compiled Version of QPL Example Program

```
        AQBIT                                   ;Allocate the qubit, will have address 0
        HDMD  0x00                              ;Apply the Hadamard
        MSRE  0x00                              ;Measure the qubit
        JUMPZ ELSE                              ;If it is |0⟩ then jump to else part
        PHASE 0x00                              ;Otherwise apply the Phase gate
        LOADL 0x00                              ;Unconditionaly jump to end by
        JUMPZ END                               ;loading zero and jumping if not zero
ELSE:   GATE  0x00 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0  ;Apply the Not gate
END:    HALT                                    ;Finished
```

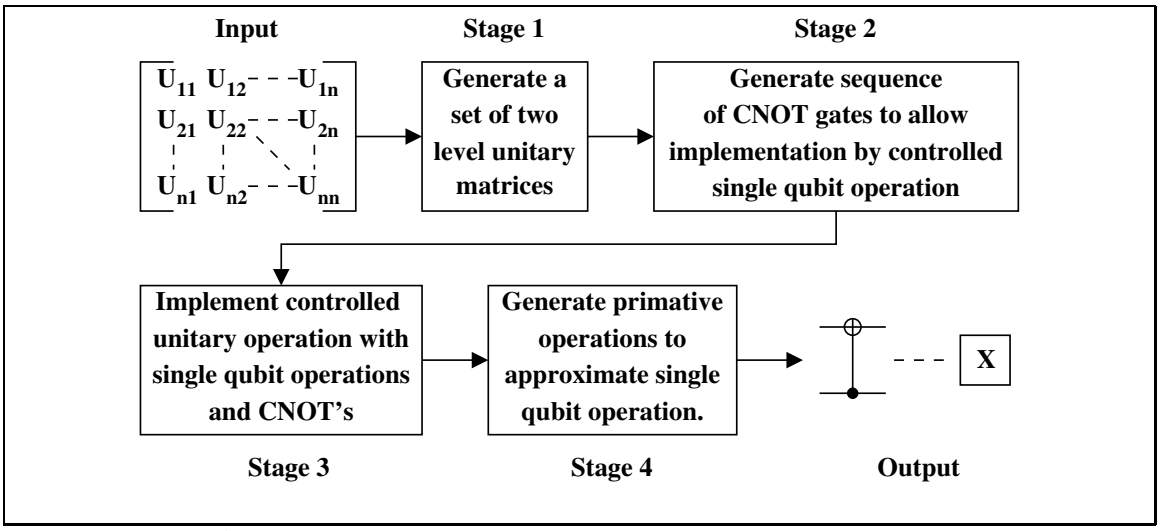


Figure 6.2: Decomposition of ArbitaryArbitrary Unitary Matrix

machine only provides operations corresponding to these universal gates, and it is the job of the compiler to peform the decomposition. This decomposition is a complex process, and work has been done on in by a variety of different people and research groups. We bring this work together to form a complete compilation process, and provide an anaysis of it's efficency.

6.3.1 Overview

Decomposing a matrix into primitive operations is a multistage process, outlined by Figure 6.2. Each of the stages shown is described in detail in the following sections. For completeness, and in order to provide a thorough explanation of the process, an example unitary matrix will be decomposed into primitive operations. The following unitary matrix is a special case of the quantum Fourier transform, and will be used as an example.

$$U = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \tag{6.1}$$

The mathematical proofs for the validity of each stage of the process are well established, and

Function	Description	Example
abs	Returns the absolute value, or magnitude, of it's argument. Defined for complex and real numbers	$abs(x) = \|x\|$
adj	Returns the adjoint, or conjugate transpose, of it's argument. Defined for matrices.	$adj\begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^{\dagger}$
conj	Returns the complex conjugate of it's argument. Defined for complex numbers.	$conj(x) = x^*$
pad	Increases the size of it's argument by 1, by appending identity elements around the top and left edges. Defined for matrices.	$pad\begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & a & c \\ 0 & b & d \end{bmatrix}$
reverse	Returns a list containing the same elements as the input, but in the reverse order.	$reverse(3,4,5) = (5,4,3)$
size	Returns the number of elements in a list, of the side length of a matrix.	$size(5,6,7) = 3$
sqrt	Returns the square root of it's argument. Defined for real and complex numbers.	$sqrt(x) = \sqrt{x}$
sub	Returns a matrix consisting of some of the elements of it's argument. The exact nature will be made clear from the context within which it is used.	$sub\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & j \end{pmatrix} = \begin{bmatrix} e & h \\ f & j \end{bmatrix}$

Table 6.1: Assumed functions in pseudocode.

work has 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 my knowledge the first system to implement the complete process from arbitrary operations to quantum byte code within a compiler.

The compiler is written in C++, but for generality and readability the algorithms in the following sections are written using a form of pseudocode. The syntax of the code is fairly straight forward, but because it is operating on quantum data it assumes the existence of appropriate operators for matrix and vector types. It also assumes the language has a sophisticated set of types (including lists, vectors, and matrices), is equipped with a basic set of functions capable of operating on these types. These functions are outlined in the Table 6.1.

6.3.2 Generating Two-Level Unitary Matrices

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). Two-level unitary matrices are those which act non-trivially on only 2 vector components of the system state. The product of these matrices 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 it's own right. Zeilinger et al (1994) provide a description of their technique for implementing

such transformations with beam-splitter devices, meaning that simply performing this stage could bring arbitrary operations closer to being realisable. The method described is based on the work of Zeilinger et al (1994), and also the method presented by Nielson and Chuang (2000).

The idea is to produce an ordered set R of $l = \frac{s(s-1)}{2}$unitary matrices such that:

$$\prod_{k=l}^{1} R_k \cdot U = I \tag{6.2}$$

And such that each matrix R_k is two-level. If the input matrix U is given by:

$$U = \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1n} \\ U_{21} & U_{22} & \cdots & U_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n1} & U_{n2} & \cdots & U_{nn} \end{bmatrix} \tag{6.3}$$

Then the first $s-1$ entries of R are given by:

$$R_n = \begin{bmatrix} \frac{U_{11}^*}{\sqrt{|U_{11}|^2+|U_{n1}|^2}} & 0 & \cdots & 0 & \frac{U_{n1}^*}{\sqrt{|U_{11}|^2+|U_{n1}|^2}} & 0 & \cdots \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots \\ \frac{U_{n1}}{\sqrt{|U_{11}|^2+|U_{n1}|^2}} & 0 & \cdots & 0 & \frac{-U_{11}}{\sqrt{|U_{11}|^2+|U_{n1}|^2}} & 0 & \cdots \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{6.4}$$

Where all the elements of R_n are identity except the elements in the set $\{R_{11}, R_{(n+1)1} R_{1(n+1)} R_{(n+1)(n+1)}\}$. If we define the $s \times s$ matrix T as the adjoint of the product of the first $s-1$ elements of R:

$$T = \left(\prod_{k=s-1}^{1} R_k \cdot U \right)^{\dagger} \tag{6.5}$$

Then T will always be of the form:

$$T = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & T_{22} & \cdots & T_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & T_{n2} & \cdots & T_{nn} \end{bmatrix} \tag{6.6}$$

Hence we have decomposed the original matrix U into a set of $s-1$ two-level unitary matrices, and the matrix T. All that remains is to recursively apply the procedure to the $(s-1) \times (s-1)$ sub–matrix in the lower right-hand corner of T, and then to pad the result with identity elements so it is of size $s \times s$. We now have an ordered set R fulfilling the condition in equation 6.2. By re-arrangement of this equation, it follows that:

$$\prod_{k=1}^{l} R_k^{\dagger} = U \tag{6.7}$$

And hence we have accomplished our goal of decomposing U into a set of two-level unitary matrices. The above procedure will now be clarified by means of a concrete example, the decomposition of the matrix presented in equation 6.1. By application of equation 6.4 we obtain:

$$R_1 = \begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & 0 & 0 \\ \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_2 = \begin{bmatrix} \frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} & 0 & \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}} & 0 & -\frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_3 = \begin{bmatrix} \sqrt{\frac{3}{4}} & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & -\sqrt{\frac{3}{4}} \end{bmatrix}$$

A value for T is now yielded by application of equation 6.5 giving:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{4} - \frac{\sqrt{2}}{4}i & \sqrt{\frac{1}{2}} & \frac{\sqrt{2}}{4} + \frac{\sqrt{2}}{4}i \\ 0 & \frac{\sqrt{\frac{3}{4}}}{2\sqrt{\frac{1}{2}}} + \frac{\sqrt{\frac{1}{2}}}{4\sqrt{\frac{3}{4}}}i & -\frac{\sqrt{\frac{1}{2}}}{2\sqrt{\frac{3}{4}}} & \frac{\sqrt{\frac{3}{4}}}{2\sqrt{\frac{1}{2}}} - \frac{\sqrt{\frac{1}{2}}}{4\sqrt{\frac{3}{4}}}i \\ 0 & \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}}i & \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}} & -\frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}}i \end{bmatrix}$$

Which is of the form required by equation6.6. This clearly shows the recursive nature of the algorithm, as it is now applied to the lower–right sub–matrix of T to eventually give a complete set R. As indicated by equation 6.7 it is actually the adjoint of these entries which are required, and so the final results are:

$$R_1^{\dagger} = \begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & 0 & 0 \\ \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_2^{\dagger} = \begin{bmatrix} \frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} & 0 & \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}} & 0 & -\frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.8}$$

$$R_3^{\dagger} = \begin{bmatrix} \sqrt{\frac{3}{4}} & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & -\sqrt{\frac{3}{4}} \end{bmatrix} \quad R_4^{\dagger} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{\frac{3}{4}}}{2} - \frac{\sqrt{\frac{3}{4}}}{2}i & \frac{3}{4} - \frac{1}{4}i & 0 \\ 0 & \frac{3}{4} + \frac{1}{4}i & -\frac{\sqrt{\frac{3}{4}}}{2} - \frac{\sqrt{\frac{3}{4}}}{2}i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.9}$$

$$R_5^{\dagger} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} & 0 & -\frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}}i \\ 0 & 0 & 1 & 0 \\ 0 & \frac{\frac{1}{2}}{\sqrt{\frac{3}{4}}}i & 0 & -\frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{3}{4}}} \end{bmatrix} \quad R_6^{\dagger} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}}i \\ 0 & 0 & -\sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}}i \end{bmatrix} \tag{6.10}$$

Algorithm 4 Generation of two-level unitary matrices from arbitary operation.

input : a square, two dimensional, unitary matrix of arbitrary size, the elements of which may be complex numbers.

output : a list of unitary matrices each of the same size as the input, the product of which is equal to the input matrix.

```
function generate_two_level_unitaries(matrix u) : list
  list result ← {}
  integer size ← size(u)
  if(size = 2) then
    u ← adj(u)
    result ← result ⊣ u
  else
    matrix t ← u
    for(integer j ← 1 to size)
      complex a ← u_11
      complex b ← u_j1
      complex d ← sqrt(abs(a) × abs(a) + abs(b) × abs(b))
      matrix t ← I
      t_11 ← conj(a) ÷ d
      t_1j ← conj(b) ÷ d
      t_j1 ← b ÷ d
      t_jj ← -a ÷ d
      result ← result ⊣ t
      product ← t × product
    matrix s ← sub(product)
    list s_result ← generate_two_level_unitaries(s)
    for(matrix p in s_result)
      p ← pad(p)
      result ← result ⊣ p
  return result
```

An algorithm for performing this decomposition is given in Algorithm 4, though note that it actually gives the set of matrices $\{R_1 \dots R_n\}$ rather than $\{R_1^\dagger \dots R_n^\dagger\}$. This adjoint operation is performed afterwards and is not shown.

It has been observed (Nielson and Chuang, 2000) that a process such as the one above 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 is provided and it is a useful result to determine this. It is of course partly dependant of the efficiency of the functions which the algorithm calls, and an estimated complexity for each of these was given earlier. For other operations such as multiplication and assignment we assume $\Theta(1)$for numbers, $\Theta(n)$ for lists, and $\Theta(n^2)$ for matrices. We then proceed to analyse the complexity by starting with the inner code blocks and working up to the function as a whole.

If statement:

$$T(n) \quad = \quad \Theta(c)$$

First loop

$$\begin{aligned} T(n) &= n^3 + n^2 \\ \Rightarrow T(n) &\approx \Theta(n^3) \end{aligned}$$

Second loop:

$$\begin{aligned} T(n) &= \sum_{i=1}^{(n-1)(n-2)/2} (n^2) \\ \Rightarrow T(n) &= \frac{n^4 - 3n^3 + 2n^2}{2} \\ \Rightarrow T(n) &\approx \Theta(\frac{n^4}{2}) \end{aligned}$$

Else statement:

$$\begin{aligned} T(n) &= T(n-1) + \frac{n^4}{2} + n^3 + 2n^2 \\ \Rightarrow T(n) &\approx \Theta(T(n-1) + \frac{n^4}{2}) \end{aligned}$$

Overall:

$$\begin{aligned} T(n) &= \begin{cases} c & if\, n = 2 \\ T(n-1) + \frac{n^4}{2} & otherwise \end{cases} \\ \Rightarrow T(n) &= T(n-1) + \frac{n^4}{2} + c \\ \Rightarrow T(n) &= \sum_{i=2}^{n} \frac{i^4}{2} + c \\ \Rightarrow T(n) &= \int_2^n \frac{n^4}{2} \cdot dn \\ \Rightarrow T(n) &\approx \Theta(n^5) \end{aligned}$$

With an approximate complexity of $\Theta(n^5)$, this is clearly not a fast algorithm. However, it is polynomial rather than exponential and so in theory is computable, and 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 to compile to and store, as it too difficult to generate in real time. DISCUSSION OF EFFICIENCY STUFF BEST SAVED UNTIL END?

6.3.3 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:

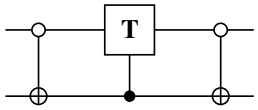


Figure 6.3: Circuit implementing Equation 6.11.

$$U = \begin{bmatrix} \alpha & 0 & 0 & \gamma \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \beta & 0 & 0 & \delta \end{bmatrix} \tag{6.11}$$

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:

$$\begin{array}{l} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \xrightarrow{U} \begin{cases} \alpha|00\rangle + \gamma|11\rangle \\ |01\rangle \\ |10\rangle \\ \beta|00\rangle + \delta|11\rangle \end{cases} \tag{6.12}$$

We wish to implement this in terms of a controlled single qubit operation, or Controlled-U gate, but note that a single qubit operation can not 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 it requires just one) 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, the operation given by Equation 6.11 is implemented by the circuit in Figure 6.3, where T is the sub–matrix of U given by:

$$T = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \tag{6.13}$$

It can be seen that this does indeed implement U because:

$$\overbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{CNOT} \times \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 0 & c \\ 0 & 0 & 1 & 0 \\ 0 & c & 0 & d \end{bmatrix}}^{Controlled-T} \times \overbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{CNOT} = \overbrace{\begin{bmatrix} a & 0 & 0 & c \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b & 0 & 0 & d \end{bmatrix}}^{U} \tag{6.14}$$

And hence the transformation of states is as shown below, giving the result specified in Equation 6.12:

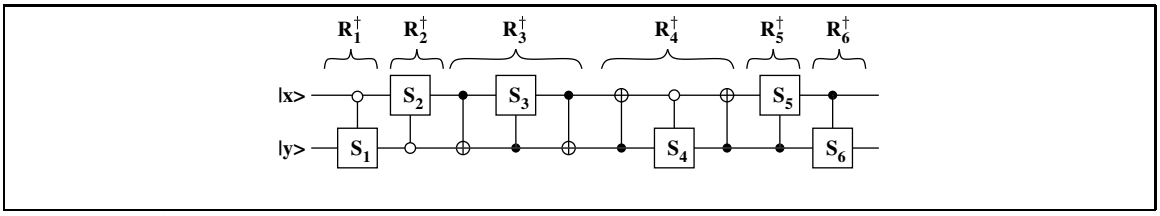


Figure 6.4: Implementation of Equation 6.1 as CNOTs and Controlled-Us.

$$\begin{matrix}|00\rangle\\|01\rangle\\|10\rangle\\|11\rangle\end{matrix} \xrightarrow{CNOT} \begin{cases}|01\rangle\\|00\rangle\\|10\rangle\\|11\rangle\end{cases} \xrightarrow{Cont-T} \begin{cases}|01\rangle\\ \alpha|00\rangle+\gamma|11\rangle\\|10\rangle\\ \beta|00\rangle+\delta|11\rangle\end{cases} \xrightarrow{CNOT} \begin{cases}\alpha|00\rangle+\gamma|11\rangle\\|01\rangle\\|10\rangle\\ \beta|00\rangle+\delta|11\rangle\end{cases} \tag{6.15}$$

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 (Nielson and Chuang, 2000) involves the use of *Grey codes*. A Grey code is a sequence of binary numbers such each number differs from it's predecessor by exactly one bit, and we use the computational basis states as the start and end values. Hence for our (rather trivial) example above we have the Grey code:

$$g = \left\downarrow \begin{matrix}00\\01\\11\end{matrix} \right\downarrow \tag{6.16}$$

The length l of g is at most $q+1$, where q is the number of qubits in the system. For each adjacent pair of entries in the Grey code $\{(g_1,g_2),\ldots,(g_{l-1},g_l)\}$the bit which differs is identified. In the case of the pair (g_{l-1},g_l) this is the target of the Controlled-T operation, for all other pairs it is the target of the CNOT operation. The remaining bits (which are guaranteed to match within the pair) indicate whether the control placed on that qubit is activated by $|0\rangle$ or $|1\rangle$. A common 0 means it is activated by the state $|0\rangle$ and a common 1 means it is activated by the state $|1\rangle$.

To further illustrate the procedure, we apply it to our test case matrix presented earlier in Equation 6.1. As a result of Section 6.3.2 we have already decomposed it into an ordered set R of two-level unitary matrices. If we define a set of matrices S to be the sub–matrices of $R^\dagger$ as given by Equation 6.13, then the final circuit is as shown in Figure 6.4.

MENTION DIFFERENT OUTPUTS FOR R3 AND R4? MENTION OPTIMISATION OPPORTUNITIES?

As before we now move on to look at an algorithm which is able to perform the process described. As well as the functions introduced in Table 6.1, the code also makes use of some new functions such as make_grey_code(), add_control(), etc. These functions have not been formally defined as their purpose should be fairly self explanatory from their names, and they are assumed to be $\Theta(n)$. They are used just to make the code simpler and shorter. With this in mind, the pseudocode is presented in Algorithm 5.

Algorithm 5 Generation of CNOTs and Controlled-Us from two–level unitaries

input: A two-level unitary matrix.
output: A set of CNOT gates and controlled unitary gates which implement the input matrix.

```
function generate_cnot_and_unitaries(matrix u) : list
  integer a_index ← 0
  integer d_index ← 0
  for(integer i ← 1 to size(u))
    if(u_ii ≠ 1.0)
      if(a_index = 0)
        a_index ← i
      else
        d_index ← i
  matrix subu ← sub(u, a_index, d_index)
  list grey_code ← make_grey_code(a_index - 1, d_index - 1)
  list cnots ← { }
  matrix controlled_t ← I
  matrix temp← I
  for(integer i ← 1 to size(grey_code) - 1)
  binary first ← grey_code_i
  binary second ← grey_code_i+1
  for(integer b ← 1 to size(first))
    if(first_i · second_i)
      add_control(temp)
    else if(¬first_i · ¬second_i)
      add_inverted_control(temp)
    else if(i = size(grey_code) - 1)
      set_matrix_operation(subu, temp)
      controlled_t ← temp
    else
      set_matrix_operation(not, temp)
      cnots ← cnots ⊣ temp
  list result ← cnots ⊣ controlled_t ⊣ reverse(cnots)
  return result
```

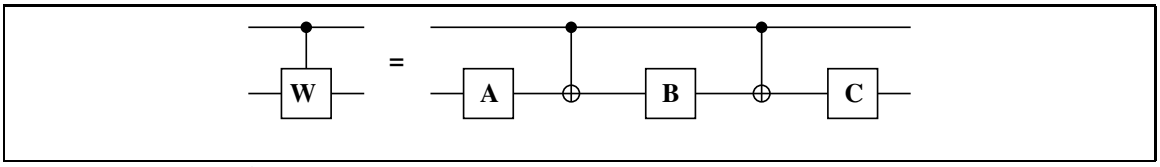


Figure 6.5: Implementation of an Arbitrary Special Unitary

Fortunately it can be seen that this algorithm is significantly more efficient than that presented in Algorithm 4...TO BE DONE

6.3.4 Implementing Controlled Unitary Gates

The output of the previous stage consists of two types of gates; Controlled–NOT gates and Controlled–U gates. Our QRAM machine is able to directly implement Controlled–NOT gates through the CNOT instruction, but Controlled–U gates require further decomposition. This section shows how this is done, building on work presented by Barenco et al.

We break this stage into two sub–stages. Firstly we look at the decomposition of a Controlled–U gate with a single control bit, and then extend this to Controlled–U gates with multiple control bits. We will only be looking at non–inverted control bits (those which are active in the state $|1\rangle$) because inverted controls can be inverted simply by applying a NOT gate.

6.3.4.1 Implementing Single Control Unitary Gates

Barenco et al make the observation that for any special unitary[1] matrix W 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:

$$A \times NOT \times B \times NOT \times C = W$$

where NOT is as has been defined previously. This allows for an implementation of of W in terms of CNOT gates and single qubits gates as shown in Figure 6.5.

Barenco et al then note that any single qubit unitary operator U can be represented by:

$$U = S \times W$$

where S is given by:

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

for some δ. A controlled–S gate can be simulated by a unitary operator E acting in the control bit, hence it is possible to produce an implementation of an arbitrary operator U using a circuit

[1] A special unitary matrix is a unitary matrix of unity determinant.

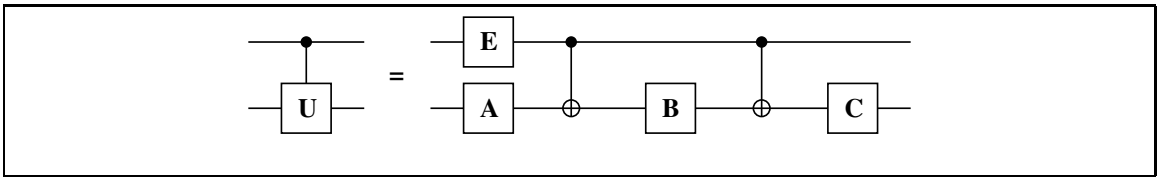


Figure 6.6: Implementation of an Arbitrary Unitary

such as the one shown in Figure 6.6.

The problem then is how to determine suitable values for the operators A, B, C, and E; and we must also make our usual analysis of the efficiency of this process.

It is well known (since the rows and columns are orthonormal, see [NC00]) that any single qubit unitary operator can be represented in the form:

$$U = \begin{bmatrix} e^{i(\delta+\alpha/2+\beta/2)}cos\theta/2 & e^{i(\delta+\alpha/2-\beta/2)}sin\theta/2 \\ -e^{i(\delta-\alpha/2+\beta/2)}sin\theta/2 & e^{i(\delta-\alpha/2-\beta/2)}cos\theta/2 \end{bmatrix}$$

where α, β, γ, and θ are all real valued. Barenco et al provide expressions for the matrices A, B, C, and E in terms of α, β, γ, and θ. The details of these expressions are not important here, but see [BW95] for further details if required (they could also be deduced from the pseudocode given later).

We now move on to provide as an example the decomposition of the controlled gate S_1 from the previous stage, this happens to be a Controlled–H gate. The compilation process yields the following values for the matrices:

$$A = \begin{bmatrix} e^{\frac{i\pi}{2}}Cos\left(\frac{\pi}{8}\right) & e^{\frac{i\pi}{2}}Sin\left(\frac{\pi}{8}\right) \\ -e^{-\frac{i\pi}{2}}Sin\left(\frac{\pi}{8}\right) & e^{-\frac{i\pi}{2}}Cos\left(\frac{\pi}{8}\right) \end{bmatrix}, B = \begin{bmatrix} e^{-\frac{i\pi}{4}}Cos\left(-\frac{\pi}{8}\right) & e^{-\frac{i\pi}{4}}Sin\left(-\frac{\pi}{8}\right) \\ -e^{-\frac{i\pi}{4}}Sin\left(-\frac{\pi}{8}\right) & e^{\frac{i\pi}{4}}Cos\left(-\frac{\pi}{8}\right) \end{bmatrix}$$

$$C = \begin{bmatrix} e^{-\frac{i\pi}{4}} & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}, E = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

which fit into Figure 6.6.

6.3.4.2 Implementing Multiple Control Unitary Gates

We have now shown how a single qubit operation with a single control line might be implemented, but more generally we need to be able to implement a single qubit operation with an *arbitrary* number of control lines. The solution (suggested by Barenco et al) is to implement it in terms of single control line operations and CNOT gates, so that we can them implement the single control line CNOT gates as discussed previously.

Such an implementation is shown in Figure 6.7 for a controlled operation with three control lines, the matrix V must be chosen such $V^{2^{n-1}} = U$ where n is the number of control lines. So in Figure 6.7 $V^4 = U$ and so $V = \sqrt[4]{U}$. Barenco et al provide a proof in their paper that such an implementation is correct, and they also provide limits on the number of gates which are required. We do not wish to reiterate their work here by trying to explain the process in too much detail so

Algorithm 6 Implementation of Controlled–Us with Single Control.

input: A matrix representing the gate to be conditionally performed. The target for the gate. A control bit for the gate.
output: A set of single qubit operation and CNOT gates equivalent to the input.

function implement_single_controlled_unitary(**matrix** u, **integer** target, **integer** control) : **list**
  real theta ← **arctan**(**abs**(u_{12}) ÷ **abs**(u_{11})) × 2
  real g1 ← **log**(u_{11} ÷ **cos**(theta ÷ 2) ÷ i
  real g2 ← **log**(u_{12} ÷ **sin**(theta ÷ 2) ÷ i
  real g3 ← **log**(-u_{21} ÷ **sin**(theta ÷ 2) ÷ i
  real d_val ← (g2 + g3) ÷ 2
  real a_val ← g1 + g2 - (2 × d)
  real b_val ← (2 ×g1) - (2 ×d) - a
  matrix a ← **rz**(a_val) × **ry**(theta ÷ 2)
  matrix b ← **ry**(-theta ÷ 2) × **rz**(-(a+b) ÷2)
  matrix c ← **rz**((b-a) ÷ 2)
  matrix e ← **rz**(-d) ×**ph**(-d ÷ 2)
  matrix s ← **ph**(d)
  matrix cnot ← [0 1; 1 0]
  set_target(cnot, target)
  add_control(cnot, control)
  list result ← e ⊣ a ⊣ cnot ⊣ b ⊣ cnot ⊣ c
  return result

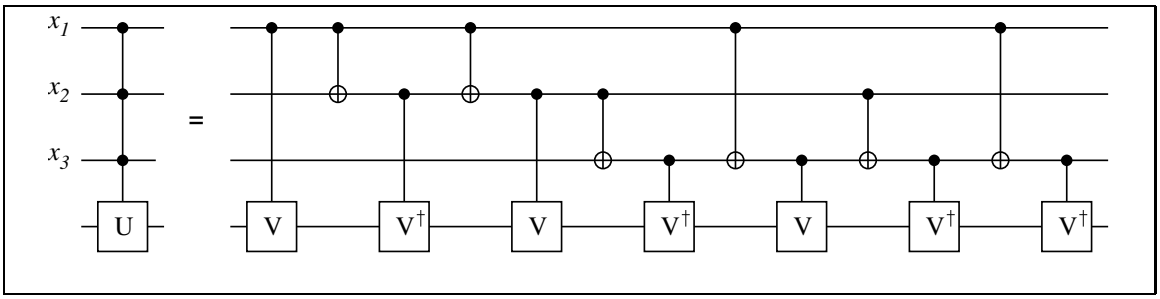


Figure 6.7: Implementation of 3 control–line unitary operation due to Barenco *et al*.

it is recommended that the reader be familiar with it. We will instead focus on two things; the calculation of the matrix V from U and the generation of the set of CNOT gates.

To calculate the arbitrary root of a matrix U it is first necessary to perform a *matrix diagonalisation* of U. This is the process of finding two matrices, Z and D, of the same dimensions as U such that:

$$U = Z \times D \times Z^{\dagger}$$

Z and D can be found by performing an *eigen decomposition* of U, Z then consists of the eigenvectors arranged as columns and D consists of the corresponding eigenvalues arranged along the diagonal (other elements are 0). We then take the appropriate root r of each element along the diagonal of D (e.g. to find the fourth root of the matrix U we take the fourth root of each element of D) and call this $\sqrt[r]{D}$. The value of V is now given by:

$$\begin{aligned} V &= \sqrt[r]{U} \\ &= Z \times \sqrt[r]{D} \times Z^{\dagger} \end{aligned}$$

As an example let us now compute a value for V if we were implementing a Controlled–H gate with three control lines. The root we require is $r = 2^{n-1} = 4$ therefore we find $V = \sqrt[4]{H}$. The details of computing eigenvalues and eigenvectors were covered in Chapter 2, applying this process and constructing the results into matrices yields values for Z and D of:

$$Z = \begin{bmatrix} 0.38268 & 0.92388 \\ 0.92388 & 0.38268 \end{bmatrix}, \; D = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

We can then compute $\sqrt[4]{D}$ as:

$$\sqrt[4]{D} = \begin{bmatrix} \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i & 0 \\ 0 & 1 \end{bmatrix}$$

and hence V is given by:

$$\begin{aligned} V &= Z \times \sqrt[4]{D} \times Z^{\dagger} \\ &= \begin{bmatrix} 0.9571 + 0.10355i & 0.10355 - 0.25i \\ 0.10355 - 0.25i & 0.75 + 0.60355i \end{bmatrix} \end{aligned}$$

We can then easily verify that $V^4 = U$. Once we are able to calculate V in this way[2], the only

[2]It should be noted that the actual implementation of the compiler which was produced for testing purposes has a limitation with regard to computing these roots of matrices. As mentioned, the process relies on matrix diagonalisation and, in turn, eigen decomposition. However the GNU Scientific library (`http://www.gnu.org/software/gsl/`) upon which the compiler is based is only able to perform an eigen decomposition on hermitian matrices.

A hermitian matrix is one which is self–adjoint such that $U = U^{\dagger}$. Many unitary matrices are also hermitian (the Hadamard operator, for example), but some are not. Hence the compiler can fail in certain cases, but performance is sufficient to show that the technique works in principle.

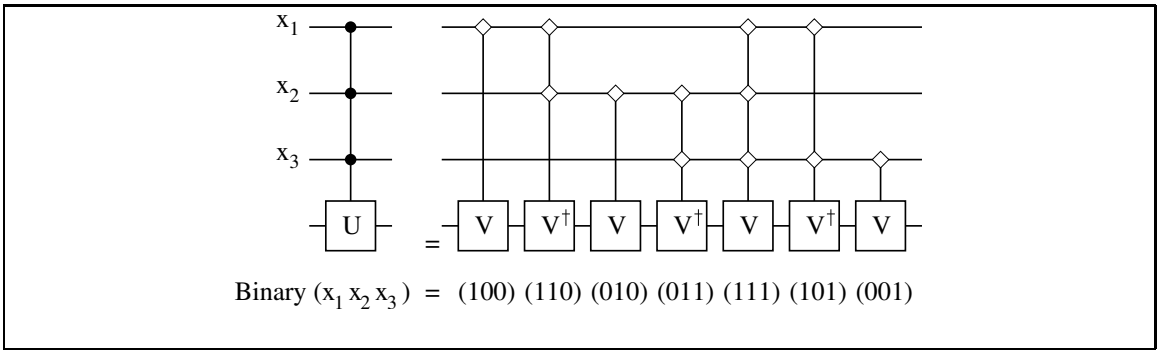


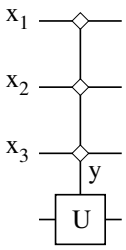
Figure 6.8: Implementing a Controlled–U gate in terms of Xor–Controlled gates.

remaining task is to find a way of generating the required sequence of CNOT gates. Barenco et al do mention ideas for doing this, we summarise these here and derive an algorithm yielding a similar set of gates (the set of gates aren't exactly the same, we come to that later).

We begin by implementing the Controlled–U in terms of what we shall call XOR–controlled gates (DO THESE HAVE A PROPER NAME?). This is a gate with an arbitrary number of control lines $(x_1,\ldots,x_n)$ which is active when the XOR of the control lines is 1 $(x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \oplus x_n = 1)$. More formally, the operation OP performed on the target is given by:

$$OP = \begin{cases} U\ if\ (x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \oplus x_n = 1) \\ I\ otherwise \end{cases}$$

We have defined our own circuit notation for it, representing a control by $\diamond$, so a three–input XOR–controlled gate would look as follows:



It is possible to use these XOR–controlled gates to implement the controlled–U gate as shown in Figure 6.8. Every possible combination of control lines are used (except for having *no* controls), and the gate being controlled is V if there are an odd number of control lines and $V^\dagger$ if there are an even number of control lines. Barenco et al state that the circuit is more efficient if the pattern of the control lines forms a grey code (as shown at the bottom of Figure 6.8). There are many possible grey code sequences, ours is generated by the *binary reflection* method. Note that we find a *complete* grey code (containing all combinations) rather than the *shortest possible* grey code which we were finding in Section 6.3.3.

It is relatively straight forward to implement a XOR–controlled gate. Each pair of controls is replaced by a CNOT gate which uses the first control as *its* control and the second control as its target. This gives us a sequence of CNOT gates which we apply, followed by a single–controlled U gate, and then the sequence of CNOT gates in reverse. So a three–input XOR–controlled gate would be implemented as follows:

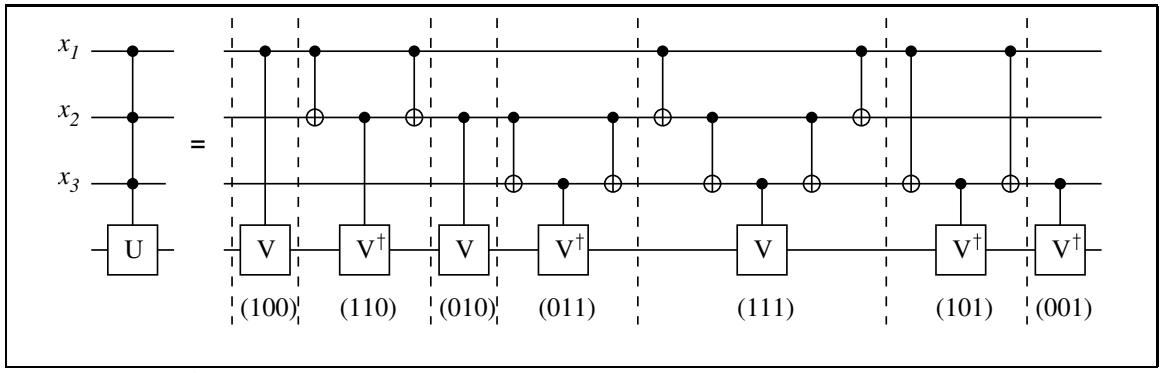
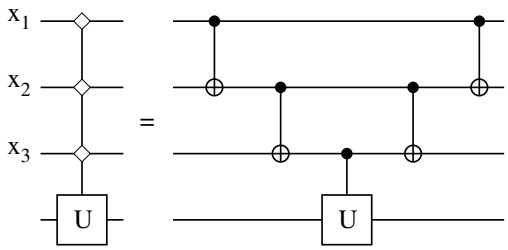


Figure 6.9: Our Implementation of a 3 control–line unitary operation.

Algorithm 7 Implementation of Controlled–Us with Multiple Controls.

input: A matrix representing the gate to be conditionally performed. The target for the gate. A set of control bits for the gate.
output: A set of single qubit operation and CNOT gates equivalent to the input.

```
function implement_multiple_controlled_unitary(matrix u, integer target, list controls) : list
  list result ← {}
  matrix root ← root(u, 2^(size(controls)-1))
  list grey_code ← make_long_grey_code(size(controls))
  for(binary grey_code_entry in grey_code)
    list cnots ← generate_gates_from_grey_code(grey_code_entry)
    list gate_imp ← implement_single_controlled_unitary
                    (u, target, controls[last_set_bit(grey_code_entry)])
    result ← result ⊣ cnots ⊣ gate_imp ⊣ reverse(cnots)
    root ← adjoint(root)
  return result
```



Hence using this we can expand Figure 6.8 to yield an implementation of controlled–U with an arbitrary number of gates, in terms of CNOT gates and Controlled–Us with only one control (and we learned how to implement these in Section 6.3.4.1). Our final implementation is given in Figure 6.9.

A distinct similarity can be seen between our implementation shown in Figure 6.9 and that due to Barenco et al (shown in Figure 6.7). However, our implementation can be observed to be less efficient as it uses a greater number of CNOT gates. The version we have given is that which is actually generated by this stage in our compiler, we shall reduce the number of CNOT gates later during the optimisation stage. This makes the algorithm at this stage simpler and allows us to perform all optimisations in one place.

6.3.5 Implementing Single Qubit Gates

We have now reached the point where our original multi–qubit operation has been reduced to a set of CNOT gates and single qubit operations. As such, we are now able to implement it on our QRAM machine by the use of CNOT and GATE instructions. However it is possible to take this decomposition further and implement the single qubit operations in terms of the universal operations which were presented in Chapter 4. There are at least two reasons for doing this:

Ease of implementation: It is (perhaps not suprisingly) much easier to implement in hardware the ability to perform a small set of fixed operations than it is to implement arbitary operations. Hence in the development of real quantum computers we are likely to reach this point first.

Fault tolerance: We have avoided the issue of fault tolerance so far, assuming our QRAM machine is able to execute any of it's instructions without error. In practice error correction and fault toleance is much more straight forward when working with a *discrete* set of gates rather than gates which form a *continuum* (such as general single qubit gates).

However there are also drawbacks to proceeding further with the decomposition, and so it is not always appropriate to do so. Such drawbacks include:

Approximation: The implementation of a single qubit gate by gates from the universal set is only an *approximation*, which has some error ε(dependant on the number of gates used). This is unfortunate because up to this point our decompositions have been exact.

More operations: If a single qubit operation can be perfomed using a single GATE instruction then this is likely to be faster than having to execute a whole set of HDMDs, PIs, etc. The exact number of universal operations required to implement a single qubit operation depends on the accuracy to which it is to be implemented, but the Solovay–Kitaev REFERENCE theorem shows that for an error $\varepsilon > 0$ it is possible to implement any single qubit gate U with $\Theta(log^c(1/\varepsilon))$ gates where c is a constant roughly equal to 2.

For the purposes of our simulator the issues of fault tolerance and ease of implementation are not important, as it is able to perform any single qubit operation with equal ease and without any error. This is one of the reasons we have not implemented further decomposition in our compiler, but the real reason is that significant work has already ben done on the subject by Aram Harrow. In REFERENCE Harrow presents an algorithm for decomposing single qubit gates, and also provides an analysis of the efficency showing that the running time is $O(log^\alpha(1/\varepsilon))$ where $\alpha = \frac{log(3)}{log(3/2)} \approx 2.7$, and hence that the algorithm is efficent.

Harrow's aims are in some ways similar to those of this chapter, in that he tries to produce algorithms for decomposition and provide an analysis of thier efficency. However, it should be noted that Harrow's work is strictly limited to the single qubit case whereas we work woth multiple qubits.

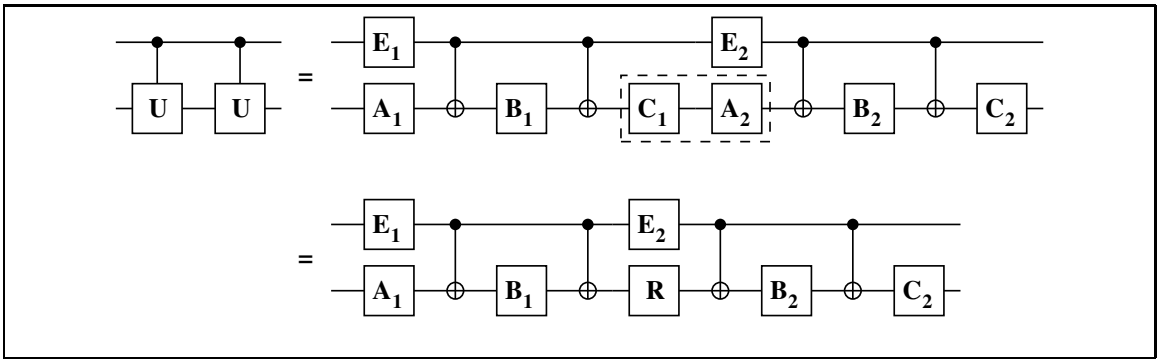


Figure 6.10: Removing Redundant Single Qubit Unitaries

6.3.6 Optimisations

Having generated a set of operations which are equivelent to the multi–qubit operation which was provided as input, we have either a sequence of CNOT gates and Single Qubit Unitaries or we have a sequence of gates from the universal set. Either way, we certainly pass through the point where we have just CNOTs and single qubit unitaries (at the end of stage 3) and this is an ideal time to perform some optimisations.

We choose to perform all optimisations at the end of the compilation process rather than during each stage for two reasons. Firstly it simplifies the algorithms for compilation and as a result makes them more efficent, and secondly it allows the optimiser to have a 'global' view of the program it is compiling rather than having only small parts of it at a time. That said, the optimisations which are performed tend to be 'local' optimisations which simply involve swapping a set of nearby gates for another set which performs the same function more efficently.

6.3.6.1 Merging Single Qubit Gates

If, in a quantum circuit, we have two or more adjacent single qubit operations they can usually be merged into one operation which is equal to the product of it's components. More formally, if we have a sequence of single qubits operations $\{U_1 \dots U_n\}$ we can replace them with a single gate R such that:

$$R = \prod_{i=1}^{n} U_i$$

If we are implementing these single qubit operations via a GATE instruction (rather than by further decomposition), and we assume that a GATE instruction always takes a constant amount of time to execute, then the resulting circuit is more efficent by a factor of n.

Such situations so occur during the compilation process. For example, imagine we have 2 single controlled unitaries adjacent to eachother. By applying the method desribed in Section 6.3.4.1 we achieve the decomposition shown in Figure 6.10. We can see here that the two adjactent gates C_1 and A_2 can be replaced by the gate R where $R = C_1 \times A_2$.

6.3.6.2 CNOT Optimisations

Apart from the single unitary transformations, the other key component of our compiled programs is the CNOT gate, which are sometimes generated in long chains (perhaps to rearrange qubits). It is often the case that such chains of qubits can be replaced by another chain with equivilent functionality, but which is optimised in some sense. This may be to reduce the number of gates in the circuit, or to increase the gates but decrease the number of controls which they have (CNOTs with more controls are harder to implement in pratice).

Research into the optimisation of such CNOT chains has already been undertaken by [Iwa02], they provide set of transformation rules which can be applied to a CNOT circuit without changing its functionality. It is posible to use these rules to generate optimised circuits, or to prove that any two given circuits are equivelent. We do not present these rules here, but instead refer the reader to the original paper.

The rules are applicable to CNOT chains which are specified directly by the programmer, but are not of so much use with chains generated by the compiler. These tend to be shorter and difficult to optimise, but there are occasions when they can be improved. After the second stage of decomposition it is possible to have two adjacent CNOTs with identical target and control bits, the transformation rule mentioned in Iwama *et al.* states that these may be removed as they cancel each other out.

6.3.6.3 'Almost Equivelent' Circuits

It is sometimes possible to optimise a circuit further if the resulting circuit does not perform quite the same function as the original, but the differences do matter further down the line. By means of an example, we return to the implementation of multi–controlled opeations discussed in Section 6.3.4.2. There we provided an implementation but observed that it was not as optimal as the version provided by Barenco *et al* (see Figures 6.7 and 6.9).

In order to reduce our circuit to thiers we have to make two replacements of CNOT sequences. These are given in Figure 6.11, however it can be noted that the replacements do not perform exactly the same function as the sequences they are replacing. For Set 1, outputs a' and c' are the same while output $b'$is inverted when a is $|1\rangle$. For set 2, the outputs are only correct if input b is inverted when a is $|1\rangle$. Hence when combined the two sets perform the same function, and the controlled unitary gate gate which sits between them in Figures 6.7 and 6.9 is only dependant on the bit c which is always correct.

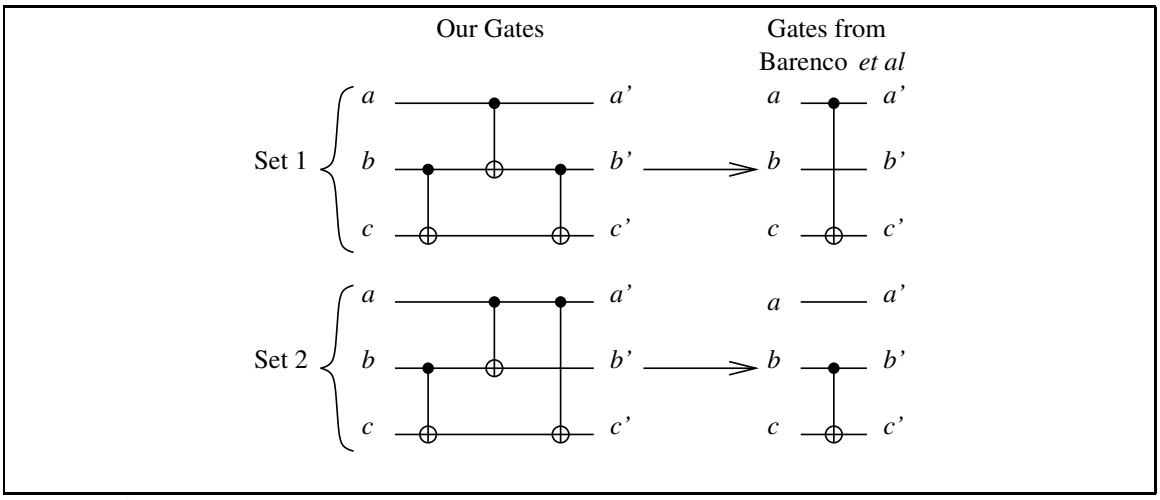


Figure 6.11: Replacements to make our implementation equal to Barenco *et al's*.

Part III

Integration of Communication and Concurrency

Chapter 7

Communication

Communication in the world of classical computing is immensely important, and it's importance looks set to continue with the arrival and continued development of the Internet. Although the QRAM model of quantum computation presented in Chapter 4 is widely accepted, there has to our knowledge been little work on the subject of communication between two (or more) QRAM machines. This chapter aims to address this gap.

As has been discussed already, there are certain tasks and operations which can be performed much more efficiently on a quantum computer than a classical one; the same applies to communication. The method of 'dense coding' provides a way to efficiently transmit data, while 'quantum cryptography' provides a method of securely transmitting data such that even a quantum computer cannot crack it. Although these techniques are known, their implementation on networked QRAM machines has not been previously studied.

This chapter begins with a look at classical communication, and adding support for it to the QRAM machine. This is a fairly trivial task but is important because it provides a basis for adding quantum communication, and also highlights the differences in the behavior between classical and quantum channels. An example is provided implementing quantum teleportation. Next we discuss the issues involved in quantum communication, and provide extensions to the QRAM machine which allow it to exchange qubits with another machine. Again an example is provided showing how this can be used to implement dense–coding. Lastly we look at some upcoming work which is focused on extending the QPL language with primitives from the π–calculus in order to specify algorithms which contain communication and concurrency. Although these extensions are not actually implemented within the QPL compiler, their implications are discussed.

7.1 Classical Communication

7.1.1 A Classical Channel

Apart from the QRAM machines (the details of which have been discussed already) the key ingredient of a classical communication system is the *channel*. This channel is responsible for the transmission of data between its source and its destination, and its type is given by the type of data that is transmitted. For the classical channel this type is *integer*, as this is the basic type on which

the QRAM machine operates. The channel operates as a queue such that the order in which items are removed from the channel is the same as the order in which they are placed on it. It will also have some maximum capacity, though the value of this is not important for our discussion. We shall also not concern ourselves with the physical implementation of this channel, or with issues such as making sure data actually reaches it destination, as for classical systems plenty of work has been done on this. The channel is also considered to be one–way, though this is not a limitation as a second channel could be supplied for communication in the other direction.

The channel can specified more formally by the use of the π–calculus from [Mil99], a calculus for describing communication and concurrency which extends and improves upon CCS. Using this, and assuming a channel capacity of c, we obtain the following formal specification:

$$\begin{aligned}
Classical_c\,\langle\rangle &\stackrel{\text{def}}{=} ctransmit(x).Classical_c\,\langle x\rangle \\
Classical_c\,\langle v_1,\ldots,v_c\rangle &\stackrel{\text{def}}{=} \overline{creceive}(v_c).Classical_c\,\langle v_1,\ldots,v_{c-1}\rangle \\
Classical_c\,\langle v_1,\ldots,v_k\rangle &\stackrel{\text{def}}{=} ctransmit(x).Classical_c\,\langle x,v_1,\ldots,v_k\rangle \\
&\quad +\overline{creceive}(v_k).Classical_c\,\langle v_1,\ldots,v_{k-1}\rangle \quad (0<k<c)
\end{aligned}$$

7.1.2 QRAM Extensions

To make use of the provided classical channel, the capabilities of the QRAM machines need to be extended; specifically they need the ability to send and receive data. To provide this capability the instruction set given in Chapter 4 is extended with two new instructions:

TRANS: Places the value on the top of the data stack onto the channel. The stack pointer is then decremented, effectively removing the value from the data stack. If the channel is full the value cannot be placed on it, in this case execution is blocked until the channel has space.

RCVE: Removes a value from the channel and places it in the top of the data stack, causing the stack pointer to be incremented. If the channel is empty then execution is blocked at this point until some data becomes available.

The only restrictions on the use of these instructions is that a TRANS instruction cannot be executed if there is no data on the stack, and the RCVE instruction cannot be executed if the data stack is full. Bearing this in mind, and specifying the maximum stack size by s, it is possible to more formally define the behaviour as follows:

$$\begin{aligned}
QRAM_s\,\langle\rangle &\stackrel{\text{def}}{=} creceive(x).QRAM_s\,\langle x\rangle \\
QRAM_s\,\langle v_1,\ldots,v_s\rangle &\stackrel{\text{def}}{=} \overline{ctransmit}(v_s).QRAM_s\,\langle v_1,\ldots,v_s\rangle \\
QRAM_s\,\langle v_1,\ldots,v_k\rangle &\stackrel{\text{def}}{=} crecieve(x).QRAM_s\,\langle v_1,\ldots,v_k,x\rangle \\
&\quad +\overline{ctransmit}(v_k).QRAM_s\,\langle v_1,\ldots,v_k\rangle \quad (0<k<s)
\end{aligned}$$

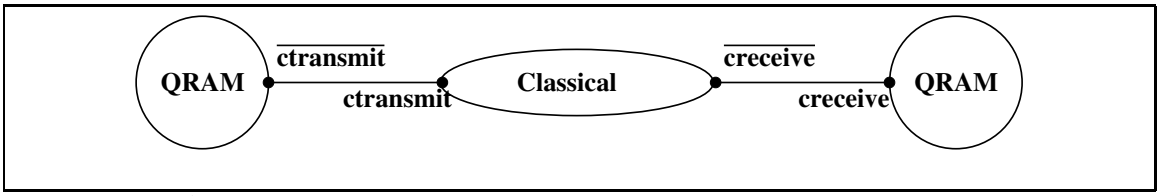


Figure 7.1: Classical Communication between two QRAM machines.

The definition bears a strong resembalence to that of the classical channel, but note that the actions are the complements of those for the channel which allow the connections to be made. Observe also the other subtle difference, the definition of QRAM receives and transmits data from the *same* end of the stack whilst the channel receives and transmits at *opposite* ends (other wise the data wouldn't get anywhere!).

7.1.3 The Complete Classical System

Now we have defined the two key components we can combine them into a more complete system. Figure 7.1 shows a system consisting of two QRAM machines and a single channel connecting them. Because there is only one channel it is only possible to transmit data in one direction, but this is sufficient for an example and it is not hard to see how it can be expanded with a second channel to allow transmission in both directions.

More formally this can be viewed as three concurrently active processes with the restriction operator applied to prevent additional connections (there are times when we might allow additional connections, for the moment we won't). In terms of π–calculus:

$$ClassicalSystem \stackrel{\text{def}}{=} new\, ctransmit, creceive\,(QRAM_s \mid Classical_c \mid QRAM_s)$$

7.1.4 An Example - Quantum Teleportation

We now finish this section with an example of two QRAM machines implementing a well-known quantum protocol; quantum teleportation. The aim is to transmit the state of of an unknown qubit from one machine (which we shall call Alice's machine) to another (which we shall call Bob's machine), simply using a classical channel. Intuitively this would seem like a difficult (if not impossible task), firstly because we are not able to get all the information about a qubits state from a single measurement, and secondly because even if we could get all this information it exists in a continuous space and hence would take forever to transmit.

The protocol therefore relies on an entangled pair being shared between the two QRAM machines. For the moment we will consider this pair to have come from some external source, or *entanglement server*; however in the next section we introduce a quantum channel which will allow the machines to create and share and entangled pair themselves.

The protocol is shown in circuit form in Figure 7.2. Alice's machine begins by interacting the unknown qubit ψ which it wishes to transmit with the first qubit of the entangled pair (which starts in the state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$). Alice's machine then measures both the qubit it is trying to send and the first qubit of the entangled pair, the results of these measurements are sent to Bob's

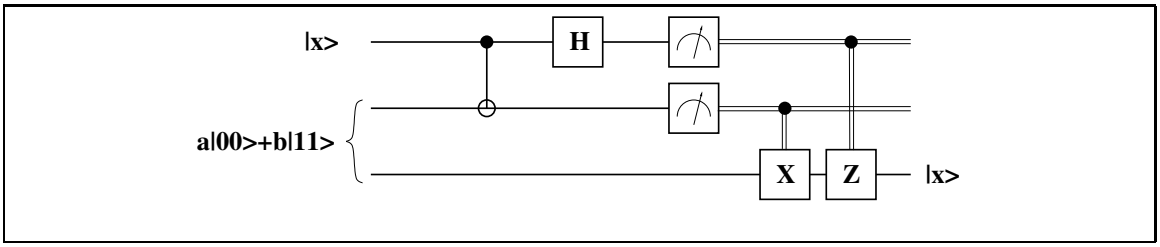


Figure 7.2: A Circuit for Quantum Teleportation

Received Bits	State	Gates to be Applied
00	$\alpha\|0\rangle + \beta\|1\rangle$	None
01	$\alpha\|1\rangle + \beta\|0\rangle$	Z
10	$\alpha\|0\rangle - \beta\|1\rangle$	X
11	$\alpha\|1\rangle - \beta\|0\rangle$	X,Z

Table 7.1: Corrections to be applied during Quantum Teleportation.

machine. Based on the values received, Bob's QRAM machine can determine the state of it's half of the entangled pair relative to ψ. It can then apply set transformations to it's half to correct the state so that it matches ψ. These are given in Table 7.1, for further details on this algorithm see NIELSON OR ORIGINAL SOURCE.

Before presenting the algorithm in it's byte–code form, there is one more issue worthy of note. The circuit shown in Figure 7.2 makes use of a CNOT gate and a Controlled-Z gate in order to perform corrections to the final state. This is a potential problem as our QRAM machine does not have a byte-code instruction for performing a Controlled-Z. There are, however, at least two possible solutions:

1. In Section 6.3.4 a method was presented for implementing Controlled-U gates using only single qubits operations and CNOT gates. This was necessary for some parts of the compilation process, and the same technique could be applied here to decompose the Controlled-Z gate.

2. Because the control of the Controlled-Z comes after the qubit has been measure, it will actually be in one of the classical base states. Hence it is possible to use the classical part of the QRAM machine to determine what transformations should be applied and to then apply them.

We will use the second option because it is easier to code, faster (at run time), and allows us to make use of the classical control structures which have not been demonstrated until now. The QRAM byte-code which actually implements the quantum teleportation protocol is shown in Algorithms 8 and 9.

Due to the blocking effect of the communication instructions they not only act as a form of communication but also as a form of *synchronisation* (see Figure 7.3). Although both machines may be started at the same time, the second QRAM does not initially execute any code because the RCVE instruction blocks execution while the channel is empty. Although it may seem trivial for

Algorithm 8 Code for Alice's Machine Implementing Quantum Teleportation

```
        CNOT    0x01 1 0 0      ;Interact the unknown state with the
        HDMD    0x00            ;first qubit of the entangled pair
        MSRE    0x01            ;Measure first qubit of entangled pair
        TRANS                   ;And transmit the result
        MSRE    0x00            ;Measure the original unknown qubit.
        TRANS                   ;And transmit the result
        HALT                    ;Finish
```

Algorithm 9 Code for Bob's Machine Implementing Quantum Teleportation

```
        RCVE                                    ;Receive measured value
        JUMPZ   PAULI                           ;If not zero, carry on...
        GATE    0x00 0.0 0.0 1.0 0.0 1.0 0.0 0.0
                0.0                             ;Perform the NOT operation
PAULI:  RCVE                                    ;Receive other measured value
        JUMPZ   END                             ;If not zero, carry on...
        GATE    0x00 1.0 0.0 0.0 0.0 0.0 0.0 -1.0
                0.0                             ;Perform Pauli-Z operation
END:    HALT                                    ;Finish
```

this example, synchronisation is always important between concurrent processes and the example of 'dense coding' in the next section will make further use of it.

7.2 Quantum Communication

Although important, the classical communication between two QRAM machines described previously is not a particularly new or exciting result. Of far more interest are the possibilities which present themselves once two QRAM machines are connected via a *quantum* channel, which permits *qubits* to be passed between machines rather than the classical bits which were allowed previously. There are known algorithms (examples will be presented later) which can make use of this quantum channel to achieve things not possible through classical communication.

7.2.1 Properties of Communicating Quantum Systems

The behaviour of a communicating quantum system is different to that of a classical system due to some of the properties of quantum mechanics which were introduced in Chapter 2. Specifically,

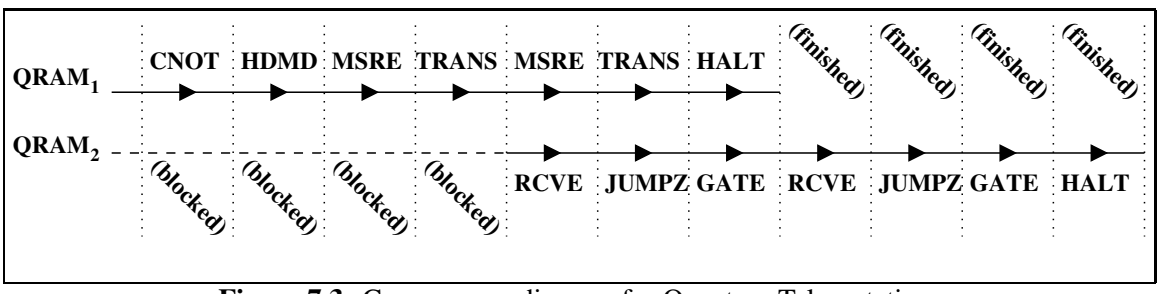


Figure 7.3: Concurrency diagram for Quantum Teleportation.

the no–cloning theorem and entanglement make important differences:

No–Cloning: In a classical system, a value held by a machine can be placed onto a channel and transmitted to another machine; the first machine can then continue to perform operations on the original data. We think nothing of doing this, but it is something the no–cloning theorem of quantum mechanics forbids when dealing with quantum data. Because it is not possible to copy the state of an unknown qubit, the qubit itself must be transmitted rather than a copy of it. The physical method of doing this is not important (for example, it could be a optic cable carrying a polarised photon), but the key point is that the first machine no longer has the qubit and so cannot perform any further operations on it.
Essentially we are saying that when a classical bit is transmitted we do not actually send the bit itself but rather some representation of it. The electrons which could transmitted from the original machine are not even necessarily the same ones which reach the target. But when transmitting a qubit it is the qubit itself which moves.

Entanglement: It has been stated that two or more qubits may become entangled with each other in a way such that their state can only be considedconsidered as a set of qubits rather than being considered individually. If this happens an operation on one qubit within the group can affect the state of others. If a QRAM machine applies a series of operations to a set of qubits in a way so as to create such a state, then after transmitting a qubit from the set to another machine it may continue to alter qubits still in its possession and so alter the state of the one it has already transmitted even though it no longer has access to it. Furthermore, if the qubit which was transmitted then gets entangled with qubits on the second QRAM machine then the first machine is able to affect the state of qubits it has never even seen.
This property of quantum mechanics is crucial to most of the algorithms involving quantum communication, including quantum teleportation and dense coding.

7.2.2 A Quantum Channel

Despite the differences outline above, the definition of a quantum channel is the same as that of the classical channel with the exception that it operates on a different type. The property of entanglement is not represented explicitly, as it is a property of quantum mechanics and quantum systems rather than being a property of quantum communication per se. Hence the fact that entanglement can occur is implicit in the fact that it is operating on quantum data.

The property of no–cloning *is* represented explicitly, but not in the channel. The quantum channel behaves in the same way as the classical one in that it transmits values, the loss of the original value from the QRAM machine is represented in the model of the QRAM machine rather than the channel; hence this is where the difference in the models can be seen. The quantum channel can now be formally defined as:

$$
\begin{aligned}
Quantum_c\,\langle\rangle &\stackrel{\text{def}}{=} qtransmit(q).Quantum\,\langle q\rangle \\
Quantum_c\,\langle v_1,\ldots,v_c\rangle &\stackrel{\text{def}}{=} \overline{qreceive}(v_c).Quantum_c\,\langle v_1,\ldots,v_{c-1}\rangle \\
Quantum_c\,\langle v_1,\ldots,v_k\rangle &\stackrel{\text{def}}{=} qtransmit(q).Quantum_c\,\langle q,v_1,\ldots,v_k\rangle \\
&\quad +\overline{qreceive}(v_k).Quantum_c\,\langle v_1,\ldots,v_{k-1}\rangle \quad (0<k<c)
\end{aligned}
$$

Because (as previously mentioned) the behaviour of the quantum channel is very similar to that of the classical channel, it is possible to simplify the model by providing a more abstract channel model and then using the π–calculus relabeling operator to provide typed instances. The abstract version is described as follows:

$$
\begin{aligned}
Channel_c\,\langle\rangle &\stackrel{\text{def}}{=} transmit(x).Channel_c\,\langle x\rangle \\
Channel_c\,\langle v_1,\ldots,v_c\rangle &\stackrel{\text{def}}{=} \overline{receive}(v_c).Channel_c\,\langle v_1,\ldots,v_{c-1}\rangle \\
Channel_c\,\langle v_1,\ldots,v_k\rangle &\stackrel{\text{def}}{=} transmit(x).Channel_c\,\langle x,v_1,\ldots,v_k\rangle \\
&\quad +\overline{receive}(v_k).Channel_c\,\langle v_1,\ldots,v_{k-1}\rangle \quad (0<k<c)
\end{aligned}
$$

The classical and quantum channels can now be described by:

$$
\begin{aligned}
Classical_c &\stackrel{\text{def}}{=} Channel_c\left\{ctransmit/transmit,\overline{creceive}/\overline{receive}\right\} \\
Quantum_c &\stackrel{\text{def}}{=} Channel_c\left\{qtransmit/transmit,\overline{qreceive}/\overline{receive}\right\}
\end{aligned}
$$

7.2.3 Quantum QRAM Extensions

It is again necessary to augment the existing QRAM instruction set with instructions for the transmission and reception of quantum data. The operation of these instructions is similar to those presented for classical communication, except rather than operate on values from the top of the stack they take an address parameter indicating which qubit should be transmitted or where the result should be stored.

QTRANS address: Causes the qubit at *address* to be transmitted, reducing the size of the quantum system by one qubit. If the channel is full this instruction blocks execution until space is available. The program counter is also incremented.

QRCVE address: Causes a qubit to be removed from the channel and stored at *address*, replacing if necessary a qubit already there. If the channel is empty this instruction blocks execution until a qubit is available. The program counter is also incremented.

In the π–calculus model presented below we omit the details of the classical communication presented earlier as these make the model unnecessarily complicated.

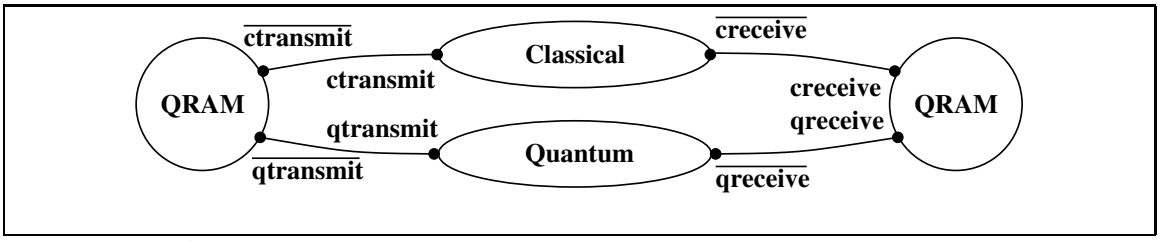


Figure 7.4: Quantum Communication between two QRAM machines.

$$
\begin{array}{rcl}
QRAM_s\langle\rangle & \stackrel{\text{def}}{=} & qreceive(x).QRAM_s\langle x\rangle \\
QRAM_s\langle v_1,\ldots,v_s\rangle & \stackrel{\text{def}}{=} & \overline{qtransmit}(v_k).QRAM_s\langle v_1,\ldots,v_{k-1},v_{k+1},\ldots,v_s\rangle \quad (0<k<s) \\
QRAM_s\langle v_1,\ldots,v_k\rangle & \stackrel{\text{def}}{=} & qrecieve(x).QRAM_s\langle v_1,\ldots,v_{n-1},x,v_n,\ldots,v_k\rangle \\
 & & +\overline{qtransmit}(v_n).QRAM_s\langle v_1,\ldots,v_{n-1},v_{n+1},\ldots,v_k\rangle \quad (0<n<k<s)
\end{array}
$$

There are two main differences between this and the classical model. Firstly, when transmitting the size of the quantum register in the transmitting machine is decreased; this is due to the qubit actually leaving the register as discussed previously. Secondly, during transmission and reception the qubit does not necessarily get added or removed from the top of the quantum stack, it can come from anywhere. That is, in a system with k qubits the qubit at position n can be removed where $(0<n<k)$.

7.2.4 The Complete Quantum System

Finally we can define our complete communicating system as the classical system described in Section 7.1 augmented with the quantum features described in this section. This is shown in Figure 7.4, and is presented more formally below.

$$
\begin{array}{rcl}
QuantumSystem & \stackrel{\text{def}}{=} & new\,ctransmit, creceive, qtransmit, qreceive \\
 & & (QRAM_s \mid Classical_c \mid Quantum_c \mid QRAM_s)
\end{array}
$$

This ability to communicate in a quantum manner now allows QRAM machines operating in parallel to execute a range of new algorithms. We illustrate this by means of an example in the following section.

7.2.5 An Example - Super–Dense Coding

The technique of 'super–dense coding' provides a more interesting example than that of quantum teleportation given earlier, because it relies on a quantum channel rather than the purely classical one which was used previously. The effect of the protocol is to transmit the values of two classical bits by the actual transmission of just one quantum bit. It's implementation is shown in Figure 7.5.

As before the protocol requires both Alice and Bob to have one qubit from an entangled pair.

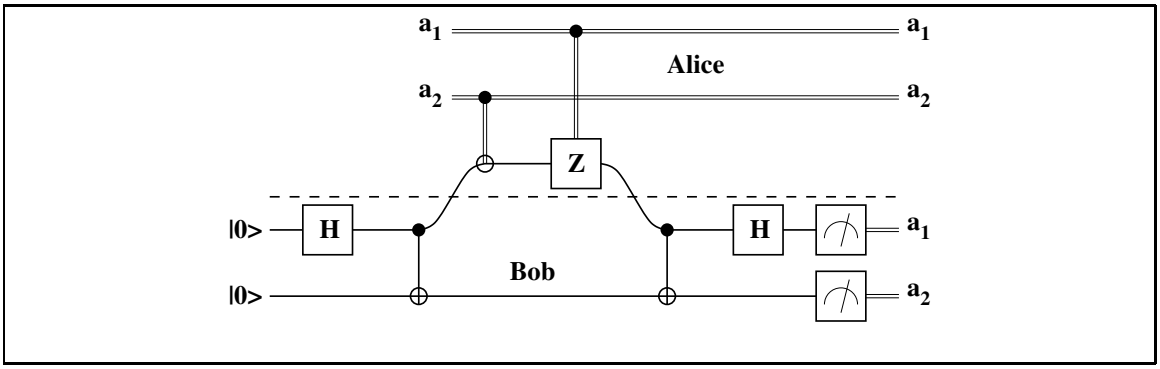


Figure 7.5: Circuit for Super–Dense Coding

Algorithm 10 QRAM Byte Code For Alice Implementing Super–Dense Coding

	LOAD	0x01	*;Load a_2 on the top of the stack.*
	QRCVE	0x00	*;Receive half the entangled pair.*
	JUMPZ	PAULI	*;If a_2 is 0 skip the Not gate.*
	GATE	0x00 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0	*;Else perform the Not gate.*
PAULI:	**LOAD**	0x00	*;Load a_1 on the top op the stack.*
	JUMPZ	END	*;If a_1 is 0 skip the Pauli gate.*
	GATE	0x00 1.0 0.0 0.0 0.0 0.0 0.0 -1.0 0.0	*;Else perform the Pauli gate.*
END	**QTRANS**	0x00	
	HALT		*;Finish Execution*

However before we introduced the notion of a classical channel it was necessary to use an entanglement server to generate this pair and get it to each party by some means, whereas in this example the entangled pair is created by Bob and on half of it is sent to Alice before the protocol begins.

The pair begins in the state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, and Alice applies a Not gate (controlled by her second classical bit) and a Pauli gate (controlled by her first classical bit). This result in the pair being in one of the four Bell states (possibly the one it started in). Alice returns the qubit to Bob who can then reverse the sequence of operation which he applied to generate the entangled pair, yielding two qubits which are actually in base state corresponding to the states of Alice's classical bits. Measurement of these qubits gives Bob the original value. For more details on the operation of this protocol please see ORIGINAL SOURCE.

Within the context of the QRAM machine, we consider the values a_1 and $a_2$to be stored on Alice at addresses 0x00 and 0x01 respectivlyrespectively. The aim is therefore to get these two addresses on Bob to contain the same values. The protocol presented above can be implemented as QRAM byte code as shown in Algorithms 10 and 11.

Although both machines start executing at the same time Alice only executes one instruction before being forced to stop and wait for Bob to provide her with a qubit. Once she receives this she can use classical conditional control to decide whether to apply the Not and Pauli Gates, rather than being forced to use Controlled–U operations (the benefits of this were discussed previously). Once half the entangled pair has been modified according to the values to be transmitted, Alice

Algorithm 11 QRAM Byte Code For Bob Implementing Super–Dense Coding

```
AQBIT                    ;Allocate the qubits which will
AQBIT                    ;form the entangled pair
HDMD    0x00             ;Create the superposition using
CNOT    0x01 1 0 0       ;CNOT and Hadamard
QTRANS  0x00             ;Send half to Alice
QRCVE   0x00             ;Receive it back after changes
CNOT    0x01 1 0 0       ;Reverse the entanglement process
HDMD    0x00
MSRE    0x00             ;Measure the results
SAVE    0x00             ;And save them to memory.
MSRE    0x01             ;Measure the results
SAVE    0x01             ;And save them to memory.
HALT                     ;Finished Execution
```

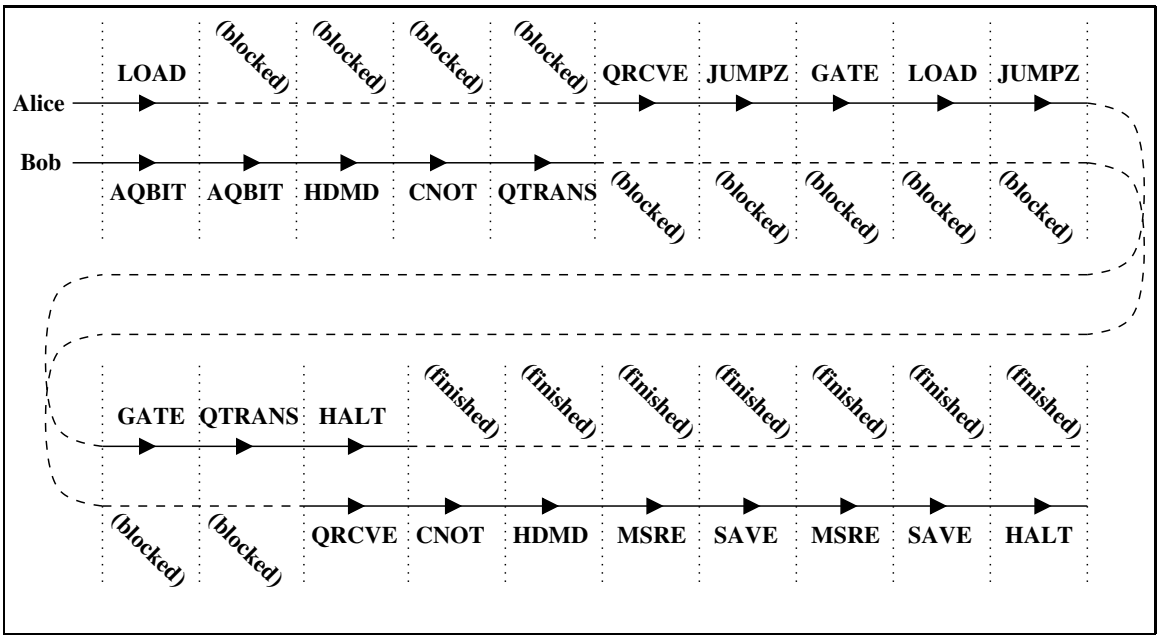


Figure 7.6: Concurrency Diagram for Super–Dense Coding

job is complete.

Bob begins by generating the entangled pair and sends half to Alice (this is part of what couldn't be done in the teleportation example, we had to rely on an external entanglement server). He sends half to Alice and must wait for her to return the modified qubit before continuing to retrieve and store the result. This example clearly involves more complex synchronisation than the last one, and it is shown in Figure 7.6. As before this synchronisation is implemented via the blocking nature of the communication instructions.

7.3 Simulation of Communication and Concurrency

The simulator mentioned in Chapter 4 has been extended to allow concurrency and communication to take place; several key extensions have been made.

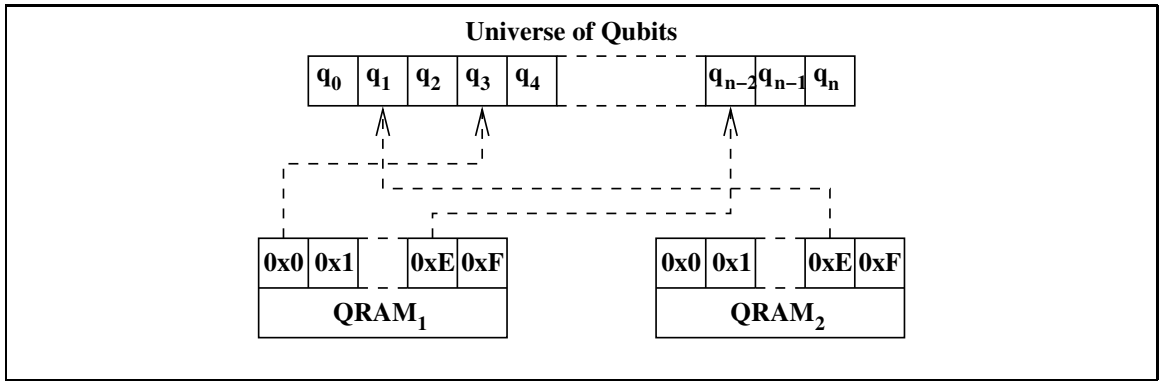


Figure 7.7: Two QRAM machines sharing the universe of qubits.

Extended instruction set: We have presented in this chapter the four new instructions TRANS. RCVE, QTRANS, QRCVE. Support for these instructions has been added to the simulators instruction set.

Classes for channels: New classes have been created to represent the classical and quantum channels (inheriting from a base channel class) and these are used by the new instructions.

Concurrent execution of QRAM machines: Two instances of the QRAM machine class can now be instantiated, and share channels for communication. During execution the simulator simply alternates between each machine to execute an instruction provided the machine is not currently blocked.

It has been necessary to give some thought to how the simulation of qubits being passed between QRAM machines is to be implemented. It is, of course, not possible to pass a qubit simply by passing it's probability amplitudes because the behaviour of a qubit depends on more than just it's own state; it may be entangled with other qubits and this is lost if just the probability amplitudes are passed.

The solution which has been used within the simulator is to pass references to qubits instead of qubits themselves. We create a 'universe' of all qubits, and the quantum register of each QRAM machine simply contains references to these. This is shown in Figure 7.7.

Although the simulator design works in principle there have been implementation difficulties due to the use of the OpenQubit library. When the simulator for the QRAM machine was written for Chapter 4 it was not anticipated that further work would be carried out to look at communication and concurrency issues. Hence this was not taken into consideration when choosing the OpenQubit library as a basis.

Due to the way the OpenQubit library is designed it is not possible to perform some operations (namely controlled operations) indirectly, that is via the references. So although the communication aspect works it is not possible to run some moderately complex byte–code programs (such the dense–coding protocol).

Algorithm 12 Quantum Teleportation in CQP

$$Alice(x : Qbit, c : \hat{}[0..3], z : Qbit) = \{z, x* = CNot\}.\{z* = H\}.c![measure\, z, x].0$$

$$Bob(y : Qbit, c : \hat{}[0..3]) = c?[r : 0..3].\{y* = \sigma_r\}.Use(y)$$

$$System(x : Qbit, y : Qbit, z : Qbit) = (new\, c\hat{}[0..3])(Alice(x, c, z)|Bob(y, c))$$

7.4 CQP and its relation to the Parallel QRAM model

After studying the QRAM machine without its extensions for communication and conncurency we were able to look at the relationship between our QRAM byte–code and a higher level programming language (QPL), and discuss how QPL programs might be compiled into QRAM byte–code. Now that we have added communication and conncurency primatives to the QRAM machine we move on to look at higher level languages which are extended in a similar way. Again we hope to identify a link between high level programs and the extended QRAM byte–code, though we do not study it in the same degree of detail as we did previously

The advantages of doing so are much the same as before (ad as for the classicla case). Firstly it is usually faster to write programs in a high level language, and they a less error prone. But more importantly it is often posible to apply formal verification techniques (depending on the language) to prove the program does what is expected.

Quantum languages which include constructs for communication are a new idea and reletivly little work has been done on them. The two main contenders are the Quantum Process Algebra (QPAlg) by [Lal04] and Communicating Quantum Processes (CQP) due to [GN04]. Both these works allow protocols to be described which involve computation and both classical and quantum communication, CQP also has the advantage that it provides a static type checking system to enforce, for example, the no–cloning theorem. We will be focusing on CQP partly because it is more familiar to us , and partly because it is in some ways similar to Peter Selinger's language QPL.

The syntax of CQP is a combination of QPL for the computation aspects and the π–calculas for describing communication. Gay and Nagarajan present Quantum Teleportation and Dense Coding as informal examples, the description of Quantum Teleportation is duplicated in Algorithm ALGORITHM. They also define a syntax for the language which is given below:

$$\begin{aligned}
T &::= Int|Uint|Qbit\hat{}[\tilde{T}]|Op(1)|Op(2)|\ldots \\
v &::= x|0|1|\ldots|unit|H|\ldots \\
e &::= v|measure\,\tilde{e}|\tilde{e}* = e|e+e \\
P &::= 0|(P|P)|e?[\tilde{x} : \tilde{T}].P|e![\tilde{e}].P|\{e\}.P|(new\, x; T)P|(qbit\, x)P
\end{aligned}$$

Our aim within this section is to try and relate some aspects of CQP to the QRAM model, in a similar way as we did for QPL. There we presented some byte–code templates which corresponded to constructs within QPL, and we can take a similar approach for CQP. It should be noted that our compiler does not actually implement the CQP language (it is limited to QPL) so the ideas

presented here are more speculative than those in Chapter 6, but should none the less provide an insight into how CQP might be executed.

CPQ provides the ! and ? operators for transmission and reception of data respectivly. Each of these operate upon a *typed* channel $\tilde{x}$, and the type $\tilde{T}$ dicates the code template which is to be used. If $\tilde{T}$ indicates w are working with classical data we have:

$$
\begin{array}{l}
execute[e![\tilde{e}]] = \\
\quad \textbf{LOAD} \quad \text{e} \quad \textit{;Loads value from address e onto top of stack} \\
\quad \textbf{TRANS} \qquad \textit{;Transmits value on top of the stack}
\end{array}
$$

and

$$
\begin{array}{l}
execute[e?[\tilde{x} : \tilde{T}]] = \\
\quad \textbf{RCVE} \qquad \textit{;Receives value onto top of the stack.} \\
\quad \textbf{SAVE} \quad \text{e} \quad \textit{;And saves it to address of e}
\end{array}
$$

The QRAM model as we have defined it only has one channel, and so is not general enough for arbitary communication according to CQP. We ignore x and transmit or recieve on the only channel we have available, though this still allows many interestingprotocols to be implemented.

For quantum channels the code templates are slighty simpler, as the byte–code instructions for the transmissions and reception of data take a parameter indicating the address rather than requiring the item to be transmitted to be at the top of the stack. Hence in the quantum case we have:

$$
\begin{array}{l}
execute[e![\tilde{e}]] = \\
\quad \textbf{QTRANS} \quad \text{e} \quad \textit{;Transmits value at address e.}
\end{array}
$$

and

$$
\begin{array}{l}
execute[e?[\tilde{x} : \tilde{T}]] = \\
\quad \textbf{QRCVE} \quad \text{e} \quad \textit{;Receives value and stores at address e}
\end{array}
$$

While the above allows basic communication, another key concept in CQP is concurrency. A statement such as $(P \,|\, Q)$ represents two proceses P and Q running conncurently, for example the proceses *Alice* and *Bob* in Algorithm 12. Within the QRAM model these processes correspond to seperate QRAM machine which are running concurrently. The process *System* would refer to some external influence responsible for preparing the machines and starting thier execution.

Bearing the above in mind it is posible to se the correspondance between the CQP description of quantum teleportation presented in Algorithm 12 and the byte–code version given in Algorithms 8 and 9. Gay and Nagarajan also present an implementation of the dense–coding protocol in thier paper, which can be compared to the byte–code version in Algorithms 10 and 11. The byte–code does not correspond directly to the code templates as some optimisation have been performed to remove redundant instructions such as a SAVE imidiatly followed by a LOAD to the same address.

Chapter 8

Conclusions and Future Work (More to be done...)

8.1 Conclusion

In this work we have presented an architecure for quantum computation, have studied quantum programming languages and their relationship to our architecture, and have looked at extensions to both the architecture and the languages to allow communcation and concurrency.

We began by presenting a quantum computer based on the QRAM architecture originaly presented 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 allowed data to be passed between the two components.

After studying the low level byte–code programs which can be written directly on the QRAM machine, we studied higher level languages and the advantages of writing quantum algorithms in these. The ideas surrounding quantum languages are disscussed in a general way, and we then provide a brief comparison of the features available in several of the languages to date. One language (QPL) is then discussed in more detail as we provide an implemtation of it.

We then moved on to the issue of generating instructions for our QRAM machine based on a program written in QPL. That is, the issue of compiling QPL to QRAM byte–code. Part of this involves creating 'code templates' for the various constructs in the QPL language, and partly it involves decomposing complex operations into those suitable for our QRAM model. This decomposition process is based on work by various people, we have brought the existing work together in one compiler and performed an efficency analyis of the varios methods it uses. We also discuss various optimisations which can be performed on the resulting byte–code once compilation is finished.

We had by this point defined a complete architecture for quantum computation along with programming methodologies. Next we addressed the issues of communication and concurrency (important for many quantum protocols to be implemented). This required us to extend our QRAM machine with new instructions and introduce the concept of a channel to connect two machines,

the π–calculus was used to describe how the system behaves. It was shown that these extensions allow the implemntation of both quantum teleportation and dense coding, details of the execution of these was provided. Lastly we looked at the relationship between our enhanced QRAM model and a language designed for communicating quantum systems, CQP.

8.2 Future Work

The work performed to date has most certainly ben sucessful, but in many ways has only scratched the surface of what could be done. The work has covered several areas and there is plenty of further work which could be done on any of these, here we take the appraoch of presenting each area with posible improvemnts.

8.2.1 The QRAM Model and Simulator

Due to the limited time available during an MSc the QRAM machine was kept as simple as possible. Its instruction set was fairly minimal and, although universal for quantum computing, could usefully gain some additional classical instructions. More flexible jumping would be useful, rather than always relying on the conditional JUMPZ instruction, and more powerful instructions for the manipulation of different classical data types could make programming easier. Simpler control over the quantum register could be achieved by adding instructions to initialise the register to arbitary starting points, or adding instructions to perform common operations such as creating entangled systems. Whether such extensions would prove to be useful remains to be seen.

We did briefly cover some of the physical methods which might be used to implement quantum computers, but did not pay much attention to how they would actually affect the operation or our QRAM machine. These physical implemetnation issues might restrict the type of operations we are able to perform (limiting our choice of instruction set), and may also affect the design of algorithms. For example, within the ion trap model it is easier to perform operations on multiple qubits if they are adjacent to eachother. This requirement could affect the way algorithms are written and optimisations are performed.

When discusing his QPL language, Peter Selinger states that it is designed for 'idealised hardware' which never produces flawed results. This is exactly what our simulator provides, but it is very difficult to implement in practice, as real quantum computers suffer from deterioration of the quantum states and dificulty in applying transformations to arbitarily high precision. There has been considerable research into method of quantum error detection and error correction, it would be interesting to introduce such errors into the simulator and judge the effectivness of these techniques when used with the QPL language.

Lastly, from the point of view of useability the simulator could use much work. It is currently a command–line based system which, although sufficent for our purpose so far, often requires adjustments and recompilation for different algorithms and a certain amount of inside knowledge to intepret the results. It is likely that this could be considerably improved with the addition of a graphical user interface, perhaps allowing viewing and adjustment of the quantum system as it evolves. This might be especssally useful combined with the ideas for error handling.

8.2.2 The QPL Compiler

One of the distinguishing features of the QPL language is the static type checking system, but unfortuntly this has not not en implemented in our compiler. We were originally going to work on this, but decided instead to pursue the work on communication. Static type checking would none the less be an important addition to our simulator and something we would like to look at in the future.

Bibliography

[Aha98] D Aharonov. Quantum computation. *To appear in Annual Reviews of Computational Physics*, 1998.

[Aha03] D Aharonov. A simple proof that toffoli and hadamard are quantum universal, 2003.

[App97] M Appel, A. Ginsberg. *Modern Compiler Implementation in C : Basic Techniques*. Cambridge University Press, 1997.

[Ben80] P Benioff. The computer as a physical system: A microscopic quantum mechanical model of computers as represented by turing machines. *Journal of Statistical Physics*, 22:563–591, 1980.

[BK90] Raymond Barnett and Thomas Kearns. *Elementary Algebra: Structure and Use*. McGraw–Hill, fifth edition, 1990.

[Bla02] S Blaha. Quantum computers and quantum computer languages: Quantum assembly language and quantum c, 2002.

[Bro00] D Brown, D. Watt. *Programming Language processors in Java : Compilers and Interpreters*. Prentice Hall, 2000.

[BS03] T. Bettelli, S. Calarco and L Serafini. Toward an architecture for quantum programming. *The European Physical Journal*, 25(2):181–200, 2003.

[BW95] C. Cleve R. Divincenzo D. Margolus N. Shor P. Sleator T. Smoli J Barenco, A. Bennett and H. Weinfurter. Elementary gates for quantum computation. *Submitted to Physical Review A*, 1995.

[Cir95] P Cirac, J. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74:4091, 1995.

[Cle99] R Cleve. An introduction to quantum complexity theory. *To appear in Collected Papers on Quantum Computation and Quantum Information Theory*, 1999.

[CM97] A. Macchiavello C. Cleve, R. Ekert and M Mosca. Quantum algorithms revisited. *Submitted to Phil. Trans. R. Soc. Lond. A*, 1997.

[Deu85] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985.

[DiV94] D.P. DiVincenzo. Two-bit gates are universal for quantum computation. *Accepted by Physical Review A*, 1994.

[Fey82] R Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6&7):467–488, 1982.

[GBJ+00] D Grune, H. E. Bal, C. Jacobs, K. G. Langendoen, Koen Langendoen, and Henri Bal. *Modern Compiler Design*. John Wiley & Sons, Inc., 2000.

[GN04] S. Gay and R. Nagarajan. Communicating quantum processes. *Proceedings of the 2nd International Conference on Quantum Programming Languages*, pages 91–107, 2004.

[Gru99] J Gruska. *Quantum Computing*. McGraw-Hill, 1999.

[Har01] A.W. Harrow. Quantum compiling, 2001.

[Har02] A.W. et al Harrow. Efficient discrete approximations of quantum gates, 2002.

[Iwa02] K. et al Iwama. Transformation rules for designing cnot-based quantum circuits, 2002.

[Kit97] A Kitaev. Quantum computation: Algorithms and error correction. *Russ. Math. Surv*, 52:1991–1249, 1997.

[KN00] E.H Krill and M.A Nielson. Quantum computation, theory of. *Encyclopaedia of Mathematics*, Supplement III, 2000.

[Kni96] E. Knill. Conventions for quantum pseudocode, 1996.

[Kol93] Bernard Kolman. *Introductory Linear Algebra with Applications*. Macmillan, fifth edition, 1993.

[Lal04] P Lalire, M. Jorrand. A process algebraic approach to concurrent and distributed quantum computation: operational semantics. *Proceedings of the 2nd International Conference on Quantum Programming Languages*, pages 109–126, 2004.

[Mil89] R Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] R Milner. A calculus of mobile processes, part 1, 1990.

[Mil99] R Milner. *communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[Moo65] G.E Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[NC00] M.A Nielson and I.L Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[Nie97] I.L Nielsen, M.A. Chuang. Programmable quantum gate arrays. *Physical Review Letters*, 79, 1997.

[Ome96] B Omer. Simulation of quantum computers, 1996.

[Ome98] B Omer. A procedural formalism for quantum computing, 1998.

[Ome02] B Omer. Classical concepts in quantum programming, 2002.

[Pri99] Y Pritzker. Simulation of quantum computation on intel-based architectures, 1999.

[RB94] A. Bernstein H. Reck, M. Zeilinger and P Bertani. Experimental realization of any discrete unitary operator. *Physical Review Letters*, 73(1):58–61, 1994.

[RP00] E. Rieffel and W. Polak. An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, 32(3):300–335, 2000.

[RW99] Kenneth Ross and Charles Wright. *Discrete Mathematics*. Prentice Hall, fourth edition, 1999.

[Sel03] P Selinger. Towards a quantum programming language. *To appear in Mathematical Structures in Computer Science*, 2003.

[Sel04] P Selinger. Towards a semantics for higher–order quantum computation. *Proceedings of the 2nd International Conference on Quantum Programming Languages*, pages 127–143, 2004.

[SZ00] J.W. Sanders and P. Zuliani. Quantum programming. *Mathematics of Program Construction*, 2000.

[Tur36] A.M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc*, 2(42):230–265, 1936.

[WZ82] W.K Wootters and W.H Zurek. A single quantum cannot be cloned. *Nature*, 299, 1982.

[XH03] Z. Shi M. Zhou X. Du J. Xue, F. Chen and R. Han. An architecture of deterministic quantum central processing unit. *Submited to Physics Letters A*, 2003.

Appendix A

Design of the QRAM Simulator

We present here an overview of the design of the QRAM simulator, useful for those who want to know how such a program is constructed but not crucial for an understanding of the main text.

Appendix B

Simulation of Deutsch's Algorithm

We provide here an example of the QRAM machine simulator performing a simulation of Deutsch's algorithm, using the byte–code presented in Algorithm 1. The algorithm is testing the NOT function, so we expect it to determine that the function is *balanced*.

The simulator begins by creating an instance of the QRAM class (with three qubits) and initialising it to its starting state. As always the sum of the probability amplitudes must be 1.0.

```
Initialising QRAM machine...
Sum of probabilities:  1.000000000000000000000000
```

The program is then read from disk and loaded into the QRAM machine.

```
Loading program...
Loading AQbitInstruction
Loading AQbitInstruction
Loading GateInstruction
Loading GateInstruction
Loading GateInstruction
Loading GateInstruction
Loading CNotInstruction
Loading GateInstruction
Loading MsreInstruction
Load sucedded
```

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 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.

```
--------------------Initial States--------------------
Primary Machine
*** QRAM Quantum State ***
1.000000 |000>
*************************
-----------------------------------------------------
```

Execution of the program begins with the allocation of two qubits, a result of two successive AQBIT instructions. Qubits are initialised to the $|0\rangle$ state.

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

It can be seen from Figure 4.3 that the lower qubit needs to start in state $|1\rangle$ instead. A GATE operation implementing the NOT function is applied to change this.

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

The two Hadamards are applied next, 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.

```
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>
*************************
```

Here we apply the test function, which in our case is the GATE instruction implementing the NOT function.

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

We perform a CNOT operation, conditional on the result of our test function. Some of the output below 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.

```
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>
**************************
```

The last Hadamard is applied 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 was can't tell until we perform the measurement.

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

During the measurement the OpenQubit library generates some more output, the eventual result being that the qubit is correctly measured to be 0. This indicates our test function was balanced.

```
Measuring QBit 0
Got probabilities...p0=1.000, p1=0.000
Sum of probabilities:  0.99994; Normalized amplitudes:  0.99994
norm(*this) + ROUND_ERR = 0.99994246110460971443956l7
norm(*this) - ROUND_ERR = 0.99994246110260975868300l9
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>
**************************
```

Appendix C

Output of Compilation Process

We present here an example of our compiler working to compile a simple QPL program, as presented in Chapter 6. The input to the compiler is:

new qbit q := 0;
q *= H;
measure q **then**
  q *= S;
else
  q *= X;

We annotate an edited version of the output below to show the stages the compiler goes through. Note that this compilation process does not require and decomposition to be performed, as the operation are all on one qubit and can be implemented with the GATE instruction.

The compiler begins by parsing the program and building an AST internally. Each time it enters a function which parses a particular construct in the language it prints out the name of the function, allowing the progress of the parser to be checked for debugging purposes.

```
Parser::parseProgram()
Parser::parseCommand()
Parser::parseSingleCommand()
Parser::parseCommand()
Parser::parseSingleCommand()
Parser::parseTypeDenoter()
Parser::parseIdentifier()
Parser::parseIdentifier()
Parser::parseSingleCommand()
Parser::parseIdentifier()
Parser::parseQuantumExpression()
Parser::parseMatrixLiteral()
Parser::parseSingleCommand()
```

```
Parser::parseExpression()
Parser::parsePrimaryExpression()
Parser::parseSingleCommand()
Parser::parseIdentifier()
Parser::parseQuantumExpression()
Parser::parseMatrixLiteral()
Parser::parseSingleCommand()
Parser::parseIdentifier()
Parser::parseQuantumExpression()
Parser::parseMatrixLiteral()
```

Once the AST has been built the checker visists each node to perform contextual analyisis. Had static type checking been implemented, this stage is where it would have been performed.

```
Looking at Abstract Syntax Tree
Checker::visitProgram
Checker::visitSequentialCommand
Checker::visitSequentialCommand
Checker::visitDeclarationCommand q
Checker::visitSimpleTypeDenoter
Checker::visitTimesEqualsCommand
Checker::visitIdentifier q
Checker::visitMatrixExpression
Checker::visitMatrixLiteral
Checker::visitIfCommand
Checker::visitQuantumExpression
Checker::visitTimesEqualsCommand
Checker::visitIdentifier q
Checker::visitMatrixExpression
Checker::visitMatrixLiteral
Checker::visitTimesEqualsCommand
Checker::visitIdentifier q
Checker::visitMatrixExpression
Checker::visitMatrixLiteral
```

Having finished the contextual analysis we are now ready to generate the byte code. The encoder visits each node and uses a template (as discussed in Chapter 6) to determine which instructions to emit.

```
Performing code Generation
Encoder::visitProgram
```

```
Encoder::visitSequentialCommand
Encoder::visitSequentialCommand
Encoder::visitDeclarationCommand q0x807a250
Encoder::visitTimesEqualsCommand
Encoder::visitMatrixExpression
```

Here it encounters the first gate, which is the Hadamard. It parses the matrix and prints some debug statements to check it is getting things right.

```
Encoder::visitMatrixLiteral [0.707+0.0i,0.707+0.0i,0.707+0.0i,-0.707+0.0i]
matrixString is [0.707+0.0i,0.707+0.0i,0.707+0.0i,-0.707+0.0i]
removed whitespace - 0.707+0.0i,0.707+0.0i,0.707+0.0i,-0.707+0.0i,
got component (0.707,0)
got component (0.707,0)
got component (0.707,0)
got component (-0.707,0)
Side lengh is 2 - This should be an integer.
q = 0x807a254
```

It then generates the instrunctions which implement the gate. For this example this is a single GATE instruction.

```
Encoder::generateGatesFor()
[ (0.707,0) (0.707,0) ]
[ (0.707,0) (-0.707,0) ]
```

Next we come to the 'if' command. Chapter 6 discussed the code template used to implement this.

```
Encoder::visitIfCommand
Encoder::visitQuantumExpression
Encoder::visitTimesEqualsCommand
Encoder::visitMatrixExpression
```

We have a two more matrices to parse and emit instructions for, these are the Phase gate and the Not gate. This is the same process as for the Hadamard earlier.

```
Encoder::visitMatrixLiteral [1.0+0.0i,0.0+0.0i,0.0+0.0i,0.0+1.0i]
matrixString is [1.0+0.0i,0.0+0.0i,0.0+0.0i,0.0+1.0i]
removed whitespace - 1.0+0.0i,0.0+0.0i,0.0+0.0i,0.0+1.0i,
got component (1,0)
```

```
got component (0,0)
got component (0,0)
got component (0,1)
Side lengh is 2 - This should be an integer.
q = 0x807a254
Encoder::generateGatesFor()
[ (1,0) (0,0) ]
[ (0,0) (0,1) ]
Encoder::visitTimesEqualsCommand
Encoder::visitMatrixExpression
Encoder::visitMatrixLiteral [1.0+0.0i,0.0+0.0i,0.0+0.0i,1.0+0.0i]
matrixString is [1.0+0.0i,0.0+0.0i,0.0+0.0i,1.0+0.0i]
removed whitespace - 1.0+0.0i,0.0+0.0i,0.0+0.0i,1.0+0.0i,
got component (1,0)
got component (0,0)
got component (0,0)
got component (1,0)
Side lengh is 2 - This should be an integer.
q = 0x807a254
Encoder::generateGatesFor()
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
```

Having completed the compilation process we now only have to save it to disk

```
InstructionSet::save()
```

Appendix D

Output of Decomposition Process

We give here the output of our compiler performing the decomposition discussed in Chapter 6. The input is the two qubit unitary operation:

$$U = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

which is to be implemented in terms of CNOT gates and single qubit unitaries. We omit the initial parsing stage which was shown in the previous appendix and jump to the point where the compiler has identified the matrix to be decomposed.

```
Encoder::generateGatesFor()
[ (0.5,0) (0.5,0) (0.5,0) (0.5,0) ]
[ (0.5,0) (0,0.5) (-0.5,0) (0,-0.5) ]
[ (0.5,0) (-0.5,0) (0.5,0) (-0.5,0) ]
[ (0.5,0) (0,-0.5) (-0.5,0) (0,0.5) ]
Got multiple qubit gate
```

The compiler determine values for the two level unitary operations whose product gives the original input. These correspond to the matrices given in Equations 6.8 to 6.10.

```
Two level Unitaries are:
[ (0.707107,0) (0.707107,0) (0,0) (0,0) ]
[ (0.707107,0) (-0.707107,0) (0,0) (0,0) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0,0) (0,0) (0,0) (1,0) ]

[ (0.816497,0) (0,0) (0.57735,0) (0,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
```

```
[ (0.57735,0) (0,-0) (-0.816497,-0) (0,-0) ]
[ (0,0) (0,0) (0,0) (1,0) ]

[ (0.866025,0) (0,0) (0,0) (0.5,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0.5,0) (0,-0) (0,-0) (-0.866025,-0) ]

[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (0.433013,-0.433013) (0.75,-0.25) (0,0) ]
[ (0,0) (0.75,0.25) (-0.433013,-0.433013) (0,0) ]
[ (0,0) (0,0) (0,0) (1,0) ]

[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (0.816497,0) (0,0) (0,-0.57735) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0,-0) (-0,0.57735) (0,-0) (-0.816497,-0) ]

[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
[ (0,0) (0,0) (0.707107,0) (0,0.707107) ]
[ (0,-0) (0,-0) (-0.707107,-0) (-0,0.707107) ]
Successfully generated two level unitaries
```

Each of these two level unitaries is now implemented in terms of controlled unitaries and CNOT gates. The first two–level unitary is shown below:

```
generateCUnitaryAndCNots()
[ (0.707107,0) (0.707107,0) (0,0) (0,0) ]
[ (0.707107,0) (-0.707107,0) (0,0) (0,0) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0,0) (0,0) (0,0) (1,0) ]
```

The decomposition process required the generation of a Grey code for each two–level unitary, this was used to identify the sequence of CNOT gates to be used. The Grey code and its result are given next:

```
00
01
Adding following inverted control for Controlled Unitary - 0
```

```
Setting U target to 1
```

Next we implement the controlled unitary from the previous stage according to the procedure described in Section 6.3.4.1. This requires determining the matrices A, B, C, E, and S.

```
implementSingleCUnitary()
Target 1
Matrix
[ (0.707107,0) (0.707107,0) ]
[ (0.707107,0) (-0.707107,0) ]
Control 0
theta = 1.5708
a = 3.14159
b = 0
d = -1.5708
[ (6.12303e-17,1) (0,0) ]
[ (0,0) (6.12303e-17,-1) ]
[ (0.92388,0) (0.382683,0) ]
[ (-0.382683,0) (0.92388,0) ]
A =
[ (0,0.92388) (0,0.382683) ]
[ (0,0.382683) (0,-0.92388) ]
B =
[ (0.653281,-0.653281) (-0.270598,-0.270598) ]
[ (0.270598,-0.270598) (0.653281,0.653281) ]
C =
[ (0.707107,-0.707107) (0,0) ]
[ (0,0) (0.707107,0.707107) ]
E =
[ (1,0) (0,0) ]
[ (0,0) (0,-1) ]
S =
[ (0,-1) (0,0) ]
[ (0,0) (0,-1) ]
```

We check the matrices to ensure that $A \times B \times C = I$ and that $S \times A \times X \times B \times X \times C$ gives the unitary operation which is being controlled.

```
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
```

```
S*A*X*B*X*C =
[ (0.707107,0) (0.707107,0) ]
[ (0.707107,0) (-0.707107,0) ]
```

We then repeat the procedure for each of the other 5 two-level unitaries which were found earlier.

```
generateCUnitaryAndCNots()
[ (0.816497,0) (0,0) (0.57735,0) (0,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
[ (0.57735,0) (0,-0) (-0.816497,-0) (0,-0) ]
[ (0,0) (0,0) (0,0) (1,0) ]
00
10
Setting U target to 0
Adding following inverted control for Controlled Unitary - 1
implementSingleCUnitary()
Target 0
Matrix
[ (0.816497,0) (0.57735,0) ]
[ (0.57735,0) (-0.816497,-0) ]
Control 1
theta = 1.23096
a = 3.14159
b = 0
d = -1.5708
[ (6.12303e-17,1) (0,0) ]
[ (0,0) (6.12303e-17,-1) ]
[ (0.953021,0) (0.302905,0) ]
[ (-0.302905,0) (0.953021,0) ]
A =
[ (0,0.953021) (0,0.302905) ]
[ (0,0.302905) (0,-0.953021) ]
B =
[ (0.673887,-0.673887) (-0.214186,-0.214186) ]
[ (0.214186,-0.214186) (0.673887,0.673887) ]
C =
[ (0.707107,-0.707107) (0,0) ]
[ (0,0) (0.707107,0.707107) ]
E =
[ (1,0) (0,0) ]
```

```
[ (0,0) (0,-1) ]
S =
[ (0,-1) (0,0) ]
[ (0,0) (0,-1) ]
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
S*A*X*B*X*C =
[ (0.816497,0) (0.57735,0) ]
[ (0.57735,0) (-0.816497,0) ]

generateCUnitaryAndCNots()
[ (0.866025,0) (0,0) (0,0) (0.5,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0.5,0) (0,-0) (0,-0) (-0.866025,-0) ]
00
01
11
Setting U target to 0
Adding following control for Controlled Unitary - 1
implementSingleCUnitary()
Target 0
Matrix
[ (0.866025,0) (0.5,0) ]
[ (0.5,0) (-0.866025,-0) ]
Control 1
theta = 1.0472
a = 3.14159
b = 0
d = -1.5708
[ (6.12303e-17,1) (0,0) ]
[ (0,0) (6.12303e-17,-1) ]
[ (0.965926,0) (0.258819,0) ]
[ (-0.258819,0) (0.965926,0) ]
A =
[ (0,0.965926) (0,0.258819) ]
[ (0,0.258819) (0,-0.965926) ]
B =
[ (0.683013,-0.683013) (-0.183013,-0.183013) ]
[ (0.183013,-0.183013) (0.683013,0.683013) ]
```

```
C =
[ (0.707107,-0.707107) (0,0) ]
[ (0,0) (0.707107,0.707107) ]
E =
[ (1,0) (0,0) ]
[ (0,0) (0,-1) ]
S =
[ (0,-1) (0,0) ]
[ (0,0) (0,-1) ]
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
S*A*X*B*X*C =
[ (0.866025,0) (0.5,0) ]
[ (0.5,0) (-0.866025,0) ]

generateCUnitaryAndCNots()
[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (0.433013,-0.433013) (0.75,-0.25) (0,0) ]
[ (0,0) (0.75,0.25) (-0.433013,-0.433013) (0,0) ]
[ (0,0) (0,0) (0,0) (1,0) ]
01
00
10
Setting U target to 0
Adding following inverted control for Controlled Unitary - 1
implementSingleCUnitary()
Target 0
Matrix
[ (0.433013,-0.433013) (0.75,-0.25) ]
[ (0.75,0.25) (-0.433013,-0.433013) ]
Control 1
theta = 1.82348
a = 2.03444
b = -0.463648
d = -1.5708
[ (0.525731,0.850651) (0,0) ]
[ (0,0) (0.525731,-0.850651) ]
[ (0.897879,0) (0.440243,0) ]
[ (-0.440243,0) (0.897879,0) ]
A =
```

```
[ (0.472043,0.763781) (0.231449,0.374493) ]
[ (-0.231449,0.374493) (0.472043,-0.763781) ]
B =
[ (0.829532,-0.343603) (-0.406731,-0.168474) ]
[ (0.406731,-0.168474) (0.829532,0.343603) ]
C =
[ (0.811242,-0.58471) (0,0) ]
[ (0,0) (0.811242,0.58471) ]
E =
[ (1,0) (0,0) ]
[ (0,0) (0,-1) ]
S =
[ (0,-1) (0,0) ]
[ (0,0) (0,-1) ]
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
S*A*X*B*X*C =
[ (0.433013,-0.433013) (0.75,-0.25) ]
[ (0.75,0.25) (-0.433013,-0.433013) ]

generateCUnitaryAndCNots()
[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (0.816497,0) (0,0) (0,-0.57735) ]
[ (0,0) (0,0) (1,0) (0,0) ]
[ (0,-0) (-0,0.57735) (0,-0) (-0.816497,-0) ]
01
11
Setting U target to 0
Adding following control for Controlled Unitary - 1
implementSingleCUnitary()
Target 0
Matrix
[ (0.816497,0) (0,-0.57735) ]
[ (-0,0.57735) (-0.816497,-0) ]
Control 1
theta = 1.23096
a = 1.5708
b = 1.5708
d = -1.5708
[ (0.707107,0.707107) (0,0) ]
```

```
[ (0,0) (0.707107,-0.707107) ]
[ (0.953021,0) (0.302905,0) ]
[ (-0.302905,0) (0.953021,0) ]
A =
[ (0.673887,0.673887) (0.214186,0.214186) ]
[ (-0.214186,0.214186) (0.673887,-0.673887) ]
B =
[ (0.673887,-0.673887) (-0.214186,-0.214186) ]
[ (0.214186,-0.214186) (0.673887,0.673887) ]
C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
E =
[ (1,0) (0,0) ]
[ (0,0) (0,-1) ]
S =
[ (0,-1) (0,0) ]
[ (0,0) (0,-1) ]
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
S*A*X*B*X*C =
[ (0.816497,0) (0,-0.57735) ]
[ (0,0.57735) (-0.816497,0) ]

generateCUnitaryAndCNots()
[ (1,0) (0,0) (0,0) (0,0) ]
[ (0,0) (1,0) (0,0) (0,0) ]
[ (0,0) (0,0) (0.707107,0) (0,0.707107) ]
[ (0,-0) (0,-0) (-0.707107,-0) (-0,0.707107) ]
10
11
Adding following control for Controlled Unitary - 0
Setting U target to 1
implementSingleCUnitary()
Target 1
Matrix
[ (0.707107,0) (0,0.707107) ]
[ (-0.707107,-0) (-0,0.707107) ]
Control 0
theta = 1.5708
```

```
a = 0
b = -1.5708
d = 0.785398
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
[ (0.92388,0) (0.382683,0) ]
[ (-0.382683,0) (0.92388,0) ]
A =
[ (0.92388,0) (0.382683,0) ]
[ (-0.382683,0) (0.92388,0) ]
B =
[ (0.853553,0.353553) (-0.353553,0.146447) ]
[ (0.353553,0.146447) (0.853553,-0.353553) ]
C =
[ (0.92388,-0.382683) (0,0) ]
[ (0,0) (0.92388,0.382683) ]
E =
[ (1,0) (0,0) ]
[ (0,0) (0.707107,0.707107) ]
S =
[ (0.707107,0.707107) (0,0) ]
[ (0,0) (0.707107,0.707107) ]
A*B*C =
[ (1,0) (0,0) ]
[ (0,0) (1,0) ]
S*A*X*B*X*C =
[ (0.707107,0) (0,0.707107) ]
[ (-0.707107,0) (0,0.707107) ]
```

Having finished the decomposition we now save the results to disk.

```
InstructionSet::save()
```