Techniques for Quantum Computing

State Generation, Discrete Logarithms in Elliptic Curve Groups,

Reliable Global Control Schemes and Algorithmic Cooling

by

Phillip R. Kaye

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Phillip R. Kaye, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Phillip R. Kaye

Abstract

This thesis is about techniques for quantum computing. A common theme throughout this work is the examination of how quantum algorithms and protocols might be implemented in practice. I explore this question at the level of algorithmic details and computer architecture, and not at the level of specific physical systems for performing quantum computation.

The first problem I consider is the generation of quantum states. Many results in quantum information theory require the generation of specific quantum states, such as Bell states. Some states can be efficiently created using standard quantum computational primitives such as preparing a qubit in the state $|0\rangle$ and applying a sequence of quantum gates (from a finite set). For example, a Bell state can be prepared from the state $|0\rangle|0\rangle$ using a Hadamard gate and a controlled-NOT gate. However, many states cannot be efficiently created. Chapter 1 of this thesis focusses on the generation of quantum states.

In Chapter 2, I explore implementations of Shor's quantum algorithm for computing discrete logarithms. This algorithm is particularly significant because it threatens to undermine the security of widely used elliptic curve cryptosystems. I give a strategy for implementing Shor's algorithm for finding discrete logarithms in groups of points on elliptic curves over fields of characteristic 2.

Chapter 3 is about *globally controlled arrays*, which is a paradigm for implementing quantum computers that may prove to be more feasible in practice than the quantum circuit model. I explore strategies for implementing error correction in such global control models, so that they might be implemented more robustly. I also cast the various global control schemes that have appeared in the literature into a unified framework so that their properties can be studied somewhat independently of the differences in low-level details. Using this framework, I consider the main challenges and obstacles to implementing quantum computing fault tolerantly using globally controlled arrays.

Finally, in Chapter 4, I consider algorithmic cooling—a technique that is potentially important for making quantum computation using nuclear magnetic resonance (NMR) feasible. Given the constraints imposed by the NMR approach to quantum computing, the most likely cooling algorithms to be practicable are those based on simple reversible polarization

(RPC) operations acting locally on small numbers of bits. Algorithms using 2- and 3-bit RPC operations have appeared in the literature, and these are the algorithms I consider in Chapter 4. Specifically, I show that the RPC operation used in all these algorithms is essentially a majority-vote of 3 bits, and prove the optimality of the best such algorithm (in a restricted setting). I go on to derive some theoretical bounds on the performance of these algorithms under some specific assumptions about errors. These bounds are independent of implementation details and low-level algorithmic details.

Acknowledgements

I am indebted to my supervisor, Professor Michele Mosca, for his guidance, support and interest in my work. I would also like to thank Prof. Raymond Laflamme, Christof Zalka, Donny Cheung, Carlos Perez, Alastair Kay, Mark Saaltink and Lawrence Ioannou for the many useful conversations that have been important to the development of the work presented in this thesis. I also wish to thank my wife Janine for her love and support during my years as a graduate student.

The work presented in this thesis has been supported by MITACS (Mathematics of Information Technology and Complex Systems), NSERC (National Science and Engineering Research Council), CSE (Communications Security Establishment), CFI (Canadian Foundation for Innovation), ORDCF (Ontario Research and Development Challenge Fund), and PREA (Premier's Research Excellence Awards).

Contents

Preface **1**

1 Quantum circuits for generating quantum states **4**

1.1 Background 4

1.2 Generating the phase factors 5

1.3 Generating the state with real nonnegative amplitudes 7

1.3.1 The algorithm 7

1.3.2 Implementing the U_j^Ψ 9

1.4 An example: symmetric states 11

1.5 Precision 12

1.5.1 Precision in the generation of $|\hat{\Psi}\rangle$ 12

1.5.2 Precision in the generation of phases 16

1.6 Conclusions 17

2 Discrete logarithms for elliptic curve groups **18**

2.1 Background 18

2.1.1 Shor's algorithm 18

2.1.2 Circuits for modular arithmetic 19

2.1.3 The elliptic curve group operation 20

2.2 Elliptic curves over GF(2^m) 21
2.3 Representations of the group elements 23
2.4 The discrete-logarithm problem 24
2.5 Decomposing the group operation 25
2.6 The extended Euclidean algorithm for polynomials 27
2.7 Naive implementation of the extended EEA 31
2.7.1 Implementing some tools 32
2.7.2 Long division 37
2.8 The problem of synchronization 38
2.9 An optimized implementation 40
2.9.1 The implementation 40
2.9.2 Space complexity 46
2.10 Conclusions and future work 47

3 Globally controlled quantum arrays **48**

3.1 Background 48
3.1.1 Quantum cellular automata and globally controlled arrays 48
3.1.2 Between quantum circuits and simple spin chains 50
3.2 The basic GCA model 51
3.2.1 The language `SPA` 52
3.2.2 Implementations of `SPA` for some example architectures 55
3.2.3 Implementation on lattices with a distinguished site 66
3.2.4 `SPA` programs to simulate quantum circuits 68
3.2.5 GCA, QCA, and error correction 72
3.3 Two approaches to error correction for GCA 74
3.4 Implementation-level error correction 76

3.4.1 Dissipative pulses—removing unwanted entropy 76
3.4.2 A bit-flip code for a GCA memory 78
3.4.3 A 9-qubit code for a GCA memory 81
3.4.4 A 1-dimensional implementation of the 9-qubit code 86
3.4.5 Scaling the 9-qubit code 86
3.4.6 From a robust GCA memory to a robust implementation of SPA . . . 90
3.4.7 A general construction for implementation-level codes on lattices . . 94
3.5 GCA models with parallelism 95
3.5.1 The language MPA(k) 95
3.5.2 An approach to implementing MPA(k) 99
3.6 Data-level error correction 103
3.6.1 The general approach 103
3.6.2 Example: the Steane code 104
3.6.3 Code-concatenation requires more sophisticated parallelism 106
3.7 Unresolved problems 106
3.8 Conclusions and future work 108

4 Cooling algorithms based on the 3-bit majority 110

4.1 Background 110
4.2 Architecture 112
4.3 The reversible polarization compression step 114
4.3.1 The 2-bit RPC step 115
4.3.2 The 3-bit RPC step 116
4.3.3 Equivalence between the 2BC and 3BC operations 118
4.4 Efficiency 119
4.4.1 The simple recursive algorithm 119

4.4.2 Algorithms using a heat bath 120
4.4.3 Accounting for the heat bath as a computational resource 124
4.5 Accounting for errors in an analysis of cooling 125
4.6 The symmetric bit-flip channel 126
4.6.1 3BC followed by a symmetric bit-flip error 127
4.6.2 Symmetric bit-flip errors during application of 3BC 128
4.7 Debiasing errors 130
4.7.1 3BC followed by a debiasing error 132
4.7.2 Debiasing errors during application of 3BC 134
4.8 More general algorithms based on 3BC 135
4.9 Conclusions and other considerations 136

A Proofs of correctness for sequences in Section 3.2.2.2 138

B Implementing switching stations for the architecture of Section 3.2.2.2 143

List of Figures

1.1 A circuit to generate $|\hat{\Psi}\rangle$ 8

1.2 A circuit to generate $|\Psi\rangle$ 9

1.3 A circuit implementing U_j^Ψ 10

1.4 A circuit for computing the Hamming weight 12

2.1 A circuit to compute the degree of $A \in GF(2^m)$ 33

2.2 A circuit to compute $|k\rangle \leftrightarrow |k+1\rangle$ 34

2.3 The quantum SWAP gate 36

2.4 A cyclic left shift gate 36

2.5 A circuit for $|\theta\rangle|s\rangle \leftrightarrow |\theta \ll s\rangle|s\rangle$ 37

2.6 Desynchronization example 39

2.7 The positions of A, B, a, b for register sharing 43

2.8 Example of long division by hand 44

2.9 Example of optimized implementation of long division 45

3.1 A fragment of a circuit to be simulated by `SPA` 69

3.2 A circuit equivalent to Figure 3.1 69

3.3 The circuit in Figure 3.2 rewritten with no gates acting in parallel 70

3.4 A nearest-neighbour version of the circuit shown in Figure 3.3 70

3.5 A construction for implementing a distance-4 CNOT gate 70

3.6 Some optimizations applied to the circuit constructed in Figure 3.4 72
3.7 A nearest-neighbour encoding circuit for the three-qubit code 79
3.8 A quantum circuit performing encoding for the Shor code 83
3.9 A circuit for performing encoding for the Steane code 104
3.10 A circuit for performing error correction for the Steane code 105

4.1 A circuit for the 3BC step using CNOT gates and generalized Toffoli gates . 117
4.2 An alternative circuit for the cooling step 118
4.3 The majority circuit with relevant error positions shown 129

Preface

Quantum computing as a discipline is still largely confined to theoretical activities in computer science aimed at finding new algorithms and protocols, and to experimental work seeking to identify and learn about the physical systems that might be plausible candidates for implementation of large-scale quantum computers. Not as much attention has been paid to the low-level design and optimization of algorithm implementations and quantum computer architecture. It is in these areas that I have directed my efforts.

There are a few general threads of research directed at low-level algorithmic implementation of quantum computing. One is the design and optimization of detailed sequences of quantum gates for implementing algorithms and protocols in the quantum circuit model. Quantum algorithms and protocols are often expressed in the circuit model at a high level, using black boxes to represent circuits for performing subroutines (e.g. arithmetic or group operations), using particular initial states that may have to be prepared ahead of time by an algorithmic technique, or using particular measurements that may require implementing some basis change. Physical implementation of an algorithm or protocol in a quantum computer will require a more complete specification of a sequence of quantum gates drawn from a (universal) finite set. Another thread of research is the design and study of the computational models we use to discover and articulate algorithms and protocols. An important goal is to develop models that are more likely to map naturally to a winning technology for implementing quantum computing machines. A third area of research is the design of detailed strategies for implementing techniques of quantum error correction and fault tolerance, so that the computing models can be made more robust against errors. This thesis contains elements of all three of the above activities.

I (with my co-author) gave the first detailed scheme for implementing quantum circuits for generating arbitrary quantum states, supposing we are provided a reasonably compact (classical) description of the desired states [KM02]. We show in detail how an important class of states (the symmetric states) can be created by our method. This work is the subject of Chapter 1.

Shor's algorithm is particularly interesting because of its ability to compromise real-world cryptographic systems. The computational bottleneck in an implementation of Shor's algorithm is the exponentiation step (either modular exponentiation, or exponentiation of some other group operation). An understanding of the exact complexity of implementing these operations is required for cryptographers to prepare for the post-quantum era, so that existing cryptosystems can be strengthened or replaced. Detailed circuits for modular exponentiation (needed for compromising the RSA and Diffie-Hellman protocols) are known, as is an implementation of the group operation for elliptic curves over prime fields (needed for compromising elliptic curve cryptosystems based on these fields). However, many real-world cryptosystems are based on elliptic curves over binary fields. I gave the first detailed reversible implementation of the group operation for these curves [Kaye05]. Such an implementation would be required for a quantum computer running Shor's algorithm to compromise the real-world cryptosystems based on elliptic curves over binary fields. This work is presented in Chapter 2.

The difficulties associated with realizing the quantum circuit model in a physical system have motivated the search for alternative quantum computing paradigms. One such approach has removed the requirement for the system to provide full local control over the qubits on which we are computing. Proposals for "globally controlled arrays" of qubits hold the potential to be more easily implemented (for example by nuclear magnetic resonance) than the circuit model. The proposals for globally controlled arrays differ in their details, but above a certain level of abstraction are organized in essentially the same way. In Chapter 3, I present a unified framework for studying these schemes. Globally controlled arrays are known to support universal quantum computation, but the question of whether they can support fault-tolerant quantum computation has not been adequately resolved. I discuss the main obstacles to fault tolerance for globally controlled arrays, suggest some approaches, and point out some of the shortcomings of existing proposals.

One of the challenges of quantum computing with NMR is the difficulty in obtaining a pure initial state. One approach to resolving this problem has been the development of techniques for algorithmic cooling. Because algorithmic cooling is likely to be useful for constrained systems like globally controlled arrays, it is important to find cooling algorithms that can be realistically implemented on these architectures. The cooling algorithms that have been proposed as practical candidates for quantum computation have been based on a reversible polarization compression (RPC) step involving either two or three qubits. In Chapter 4, I show that the RPC operation used in all these algorithms is essentially a majority-vote of 3 bits, and prove the optimality of the best such algorithm (where "best" is defined in terms of a specific characterization of the performance of cooling algorithms) in a restricted setting. I go on to derive some theoretical bounds on the performance of these algorithms under some specific assumptions about errors. These bounds are independent of implementation details and low-level algorithmic details. At the time of writing this thesis, this work on algorithmic cooling has been selected for publication [Kaye07].

Chapter 1

Quantum circuits for generating quantum states

1.1 Background

Many results in quantum information theory require the generation of specific quantum states, such as Bell states, or the implementation of specific quantum measurements, such as a von Neumann measurement in a Fourier-transformed basis. Some states and measurements can be efficiently implemented using standard quantum computational primitives such as preparing a qubit in the state $|0\rangle$ and applying a sequence of quantum gates (from a finite set). A Bell state can be prepared from the state $|0\rangle|0\rangle$ using a Hadamard gate and a controlled-NOT gate. A von Neumann measurement in the Fourier basis can be efficiently realized by applying an inverse quantum Fourier transform and performing a von Neumann measurement in the standard computational basis. However, many states and basis changes cannot be efficiently realized, and for those that can, it is not always clear how. For example, in [HMP+98] an improved frequency standard experiment is presented that requires the preparation of specific symmetric states. In this chapter I describe a general algorithm for preparing quantum states for which we have a reasonably compact (classical) description. One example for which this algorithm is efficient is the preparation of the symmetric states required by [HMP+98].

Suppose we want to generate $|\Psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x e^{2\pi i\gamma_x}|x\rangle$, where α_x are nonnegative reals and γ_x are real numbers in $[0,1)$. If we can first generate $|\hat{\Psi}\rangle = \sum_x \alpha_x|x\rangle$, then using methods described in Section 1.2 we can introduce phase factors to estimate $|\Psi\rangle$ arbitrarily well. I begin in Section 1.2 by showing how to generate the phase factors, as we will also need this technique in Section 1.3 where I show how to generate a state that is a good approximation of $|\hat{\Psi}\rangle = \sum_x \alpha_x|x\rangle$. In Section 1.4 I will give an example implementation of the state-generation algorithm, and in Section 1.5 I will examine the issues associated with finite precision.

1.2 Generating the phase factors

Let us begin by reviewing the procedure of [CEMM98] for generating arbitrary interference patterns. If we have first created the state $|\hat{\Psi}\rangle = \sum_{x\in\{0,1\}^n} \alpha_x|x\rangle$, this procedure will enable us to generate the state $|\Psi\rangle = \sum_x \alpha_{x\in\{0,1\}^n} e^{2\pi i\gamma_x}|x\rangle$.

The goal is to implement a circuit that performs

$$|x\rangle \mapsto e^{2\pi i\gamma_x}|x\rangle \tag{1.1}$$

for each basis state $|x\rangle$, where γ_x are real numbers in $[0,1)$. Here we will assume that each $\gamma_x = \frac{\hat{\gamma}_x}{2^m}$ for some m-bit integer $\hat{\gamma}_x$.

We suppose the phases are encoded in an operator $\text{ADD}_{\hat{\gamma}}$ that has the following effect

$$\text{ADD}_{\hat{\gamma}} : |x\rangle|y\rangle \mapsto |x\rangle|y + \hat{\gamma}_x \bmod 2^m\rangle. \tag{1.2}$$

We assume that we are provided with a compact description of the phases γ_x, and that this allows us to construct an efficient circuit for $\text{ADD}_{\hat{\gamma}}$ that will add the appropriate values of $\hat{\gamma}_x$ to the second register, controlled on the state $|x\rangle$ in the first register. The addition can be performed using standard reversible arithmetic circuits (e.g. [VBE95]).

We will make use of an m-bit auxiliary register, initially in the state $|1\rangle$. We then apply the inverse quantum Fourier transform, $\text{QFT}_{2^m}^{-1}$, to obtain the state

$$\frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{\frac{-2\pi i}{2^m}y}|y\rangle. \tag{1.3}$$

Claim 1.2.1 *For each basis state $|x\rangle$, $0 \leq x \leq 2^n - 1$, the state $|x\rangle \mathrm{QFT}_{2^m}^{-1}|1\rangle$ is an eigenstate of $\mathrm{ADD}_{\hat{\gamma}}$ with eigenvalue $e^{2\pi i \gamma_x}$.*

Proof:

$$\begin{aligned}
\mathrm{ADD}_{\hat{\gamma}}\left(|x\rangle \mathrm{QFT}_{2^m}^{-1}|1\rangle\right) &= \mathrm{ADD}_{\hat{\gamma}}|x\rangle \left(\frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{\frac{-2\pi i}{2^m} y}|y\rangle\right) && (1.4)\\
&= \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} \mathrm{ADD}_{\hat{\gamma}}|x\rangle e^{\frac{-2\pi i}{2^m} y}|y\rangle && (1.5)\\
&= \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} |x\rangle e^{\frac{-2\pi i}{2^m} y}|y + \hat{\gamma}_x \bmod 2^m\rangle && (1.6)\\
&= \left(e^{\frac{-2\pi i}{2^m}\hat{\gamma}_x}\right)\left(\frac{1}{\sqrt{2^m}}\right) \sum_{y=0}^{2^m-1} |x\rangle e^{\frac{-2\pi i}{2^m} y}|y \bmod 2^m\rangle && (1.7)\\
&= e^{-2\pi i \gamma_x}|x\rangle \mathrm{QFT}_{2^m}^{-1}|1\rangle. \quad \square && (1.8)
\end{aligned}$$

So applying $\mathrm{ADD}_{\hat{\gamma}}$ to $|x\rangle\left(\mathrm{QFT}_{2^m}^{-1}|1\rangle\right)$ and associating the eigenvalue (which is a global phase factor in this case) with the first register results in the state

$$e^{2\pi i \gamma_x}|x\rangle\left(\mathrm{QFT}_{2^m}^{-1}|1\rangle\right). \tag{1.9}$$

By applying $\mathrm{ADD}_{\hat{\gamma}}$ to $|\hat{\Psi}\rangle = \sum_x \alpha_x |x\rangle\left(\mathrm{QFT}_{2^m}^{-1}|1\rangle\right)$ we therefore get

$$\begin{aligned}
&\sum_x \alpha_x e^{2\pi i \gamma_x}|x\rangle\left(\mathrm{QFT}_{2^m}^{-1}|1\rangle\right) && (1.10)\\
=&|\Psi\rangle\left(\mathrm{QFT}_{2^m}^{-1}|1\rangle\right) && (1.11)
\end{aligned}$$

and, tracing out the first register, we have generated $|\Psi\rangle$.

In Section 1.3 we will need to use an operator

$$S_\omega : |\omega\rangle|x\rangle \mapsto |\omega\rangle e^{2\pi i (-1)^x \omega}|x\rangle \tag{1.12}$$

where the second register is a single qubit. This operator S_ω can be implemented using the procedure described above, by taking $\gamma_x = (-1)^x \omega$ and implementing $\mathrm{ADD}_{\hat{\gamma}}$ so that it is conditioned on the state $|\omega\rangle$ as well as on the state $|x\rangle$.

1.3 Generating the state with real nonnegative amplitudes

Ignoring for now the issues associated with precision, I present the basic technique for generating the state $|\hat{\Psi}\rangle = \sum_{x\in\{0,1\}^n} \alpha_x|x\rangle$. The approach will be to implement a series of n controlled-rotations, with the state of the kth rotation controlled by the state of the previous $k-1$ qubits for $k > 0$.

1.3.1 The algorithm

We begin by extending the definition of α_x to $x \in \{0,1\}^j$ for $1 \le j < n$. Suppose we had a copy of $|\Psi\rangle$ and we measured the leftmost j qubits in the computational basis. Let $\alpha_{x_1x_2\ldots x_j}$ be the nonnegative real number so that $\alpha^2_{x_1x_2\ldots x_j}$ equals the probability that the measurement result is $x_1x_2\ldots x_j$. Then $\left(\alpha_{x_1x_2\ldots x_{j-1}x_j}/\alpha_{x_1x_2\ldots x_{j-1}}\right)^2$ gives the conditional probability that a measurement of the first j qubits would yield x_j in the jth qubit, given that it yielded $x_1x_2\ldots x_{j-1}$ in the first $j-1$ qubits. For $2 \le j \le n$, define a controlled rotation U_j by

$$|x_1\rangle|x_2\rangle\ldots|x_{j-1}\rangle|0\rangle \xmapsto{U_j^\Psi} |x_1\rangle|x_2\rangle\ldots|x_{j-1}\rangle\left(\frac{\alpha_{x_1x_2\ldots x_{j-1}0}}{\alpha_{x_1x_2\ldots x_{j-1}}}|0\rangle + \frac{\alpha_{x_1x_2\ldots x_{j-1}1}}{\alpha_{x_1x_2\ldots x_{j-1}}}|1\rangle\right). \quad (1.13)$$

Define U_1^Ψ to be the single-qubit (uncontrolled) rotation that performs

$$|0\rangle \xmapsto{U_1^\Psi} (\alpha_0|0\rangle + \alpha_1|1\rangle)\,. \quad (1.14)$$

The algorithm for generating the n-qubit state $|\hat{\Psi}\rangle$ is a sequence of n such rotations, as shown in Figure 1.1.

The following claim shows that the circuit of Figure 1.1 generates $|\psi\rangle$.

Claim 1.3.1 *After the operation U_j^ψ in Figure 1.1, the state of the first j qubits is*

$$\sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \alpha_{x_1x_2\ldots x_j}|x_1x_2\ldots x_j\rangle. \quad (1.15)$$

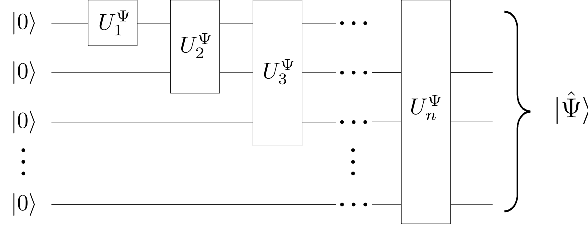


Figure 1.1: A circuit to generate $|\hat{\Psi}\rangle$.

Proof: The proof is by induction on k. The claim is clearly true for the first rotation by definition of U_1^Ψ. Suppose we have generated the state

$$\sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \alpha_{x_1x_2\ldots x_j}|x_1x_2\ldots x_j\rangle$$

and we apply U_{j+1}^Ψ as in Figure 1.1. Then we have

$$U_{j+1}^\Psi\left(\sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \alpha_{x_1x_2\ldots x_j}|x_1x_2\ldots x_j\rangle|0\rangle\right) \tag{1.16}$$

$$= \sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \alpha_{x_1x_2\ldots x_j}\left(U_{j+1}^\Psi|x_1x_2\ldots x_j\rangle|0\rangle\right) \tag{1.17}$$

$$= \sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \alpha_{x_1x_2\ldots x_j}|x_1x_2\ldots x_j\rangle\left(\frac{\alpha_{x_1x_2\ldots x_j0}}{\alpha_{x_1x_2\ldots x_j}}|0\rangle + \frac{\alpha_{x_1x_2\ldots x_j1}}{\alpha_{x_1x_2\ldots x_j}}|1\rangle\right) \tag{1.18}$$

$$= \sum_{x_1x_2\ldots x_j\in\{0,1\}^j} \left(\alpha_{x_1x_2\ldots x_j0}|x_1x_2\ldots x_j\rangle|0\rangle + \alpha_{x_1x_2\ldots x_j1}|x_1x_2\ldots x_j\rangle|1\rangle\right) \tag{1.19}$$

$$= \sum_{x_1x_2\ldots x_j\in\{0,1\}^{j+1}} \alpha_{x_1x_2\ldots x_jx_{j+1}}|x_1x_2\ldots x_jx_{j+1}\rangle. \quad \square \tag{1.20}$$

Putting the circuit to generate $|\hat{\Psi}\rangle$ together with the phase generation procedure described in Section 1.2, the circuit in Figure 1.2 generates the desired state $|\Psi\rangle$.

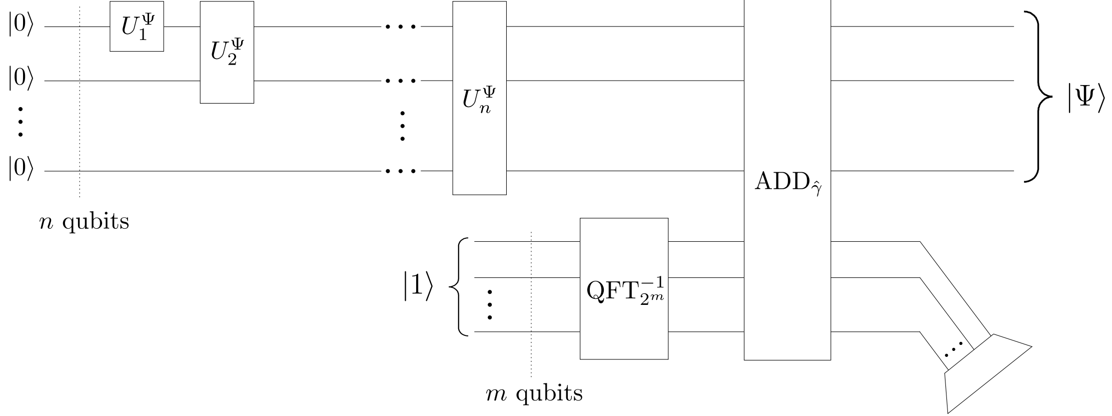


Figure 1.2: A circuit to generate $|\Psi\rangle$.

1.3.2 Implementing the U_j^Ψ

In this section I show how to implement the U_j^ψ rotations. Assume that we have access to a quantum register $|\bar{\Psi}\rangle$ that encodes some "classical" description of the state $|\Psi\rangle$. The state $|\bar{\Psi}\rangle$ must contain enough information to allow the probabilities α_x^2 (or a related quantity, such as the ω_j defined below) to be efficiently computed. We also use an ancillary register initialized to the state $|0\rangle$. For each $1 \leq j \leq n$, assume we can efficiently implement the operators V_j^Ψ defined as follows:

$$|\bar{\Psi}\rangle|0\rangle|x_1\rangle \dots |x_{j-1}\rangle \xmapsto{V_j^\Psi} |\bar{\Psi}\rangle|\omega_j\rangle|x_1\rangle \dots |x_{j-1}\rangle, \tag{1.21}$$

where ω_j satisfies

$$\cos^2(2\pi\omega_j) = \left(\frac{\alpha_{x_1 \dots x_{j-1}0}}{\alpha_{x_1 \dots x_{j-1}}}\right)^2. \tag{1.22}$$

Using the method described in Section 1.2 we can implement the operator

$$S_\omega : |\omega\rangle|x\rangle \mapsto |\omega\rangle e^{2\pi i(-1)^x\omega}|x\rangle. \tag{1.23}$$

Assume we also have access to the 1-qubit gate

$$W = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}. \tag{1.24}$$

With these components, a circuit implementing U_j^Ψ is shown in Figure 1.3

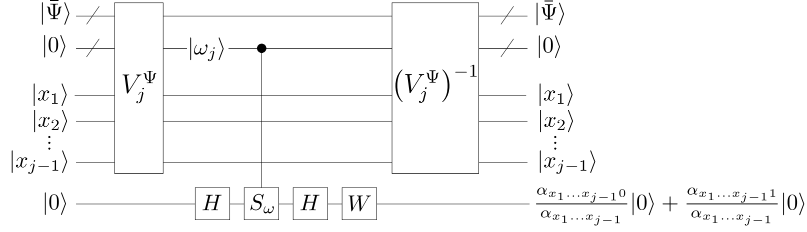


Figure 1.3: A circuit implementing U_j^Ψ.

We can verify that the above circuit has the desired effect by following the state through each step of the circuit:

$$|\bar{\Psi}\rangle|0\rangle\left(|x_1\dots x_{j-1}\rangle|0\rangle\right) \tag{1.25}$$

$$\xmapsto{V_j^\Psi}|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle|0\rangle\right) \tag{1.26}$$

$$\xmapsto{H}|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle\frac{1}{\sqrt{2}}\left(|0\rangle+|1\rangle\right)\right) \tag{1.27}$$

$$\xmapsto{\mathrm{c}-S_\omega}|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle\frac{1}{\sqrt{2}}\left(e^{2\pi i\omega_j}|0\rangle+e^{-2\pi i\omega_j}|1\rangle\right)\right) \tag{1.28}$$

$$\xmapsto{H}|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle\frac{1}{2}\left(\left(e^{2\pi i\omega_j}+e^{-2\pi i\omega_j}\right)|0\rangle+\left(e^{2\pi i\omega_j}-e^{-2\pi i\omega_j}\right)|1\rangle\right)\right) \tag{1.29}$$

$$\xmapsto{W}|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle\frac{1}{2}\left(\left(e^{2\pi i\omega_j}+e^{-2\pi i\omega_j}\right)|0\rangle-i\left(e^{2\pi i\omega_j}-e^{-2\pi i\omega_j}\right)|1\rangle\right)\right) \tag{1.30}$$

$$=|\bar{\Psi}\rangle|\omega_j\rangle\left(|x_1\dots x_{j-1}\rangle\left(\cos(2\pi\omega_j)|0\rangle+\sin(2\pi\omega_j)|1\rangle\right)\right) \tag{1.31}$$

$$\xmapsto{\left(V_j^\Psi\right)^{-1}}|\bar{\Psi}\rangle|0\rangle\left(|x_1\dots x_{j-1}\rangle\left(\cos(2\pi\omega_j)|0\rangle+\sin(2\pi\omega_j)|1\rangle\right)\right) \tag{1.32}$$

$$=|\bar{\Psi}\rangle|0\rangle U_j^\Psi\left(|x_1\dots x_{j-1}\rangle|0\rangle\right). \tag{1.33}$$

The above algorithm works for a general family of states with classical descriptions $|\bar{\Psi}\rangle$. If we are only interested in producing a specific state $|\Psi\rangle$, the circuit can be simplified by

removing the register containing $|\bar{\Psi}\rangle$ and simplifying each V_j^Ψ to work only for that specific $|\Psi\rangle$.

Note that the overall efficiency of our algorithm depends on how efficiently we can implement the V_j^Ψ; in other words, how efficiently we can compute the conditional probabilities $\left(\alpha_{x_1 \ldots x_{j-1}0}/\alpha_{x_1 \ldots x_{j-1}}\right)^2$. In the next section I give an example for which this is easy: the symmetric states.

1.4 An example: symmetric states

In this section I describe an example of a family of states for which we can efficiently implement the state generation algorithm described above. These are the *symmetric states*. The symmetric state $|S_r\rangle$ is defined to be an equally-weighted superposition of the computational basis states $|x\rangle$ that have Hamming weight $\mathrm{H}(x) = r$ ($\mathrm{H}(x)$ is the number of bits of x that equal 1). That is,

$$|S_r\rangle = \frac{1}{\sqrt{\binom{n}{r}}} \sum_{\mathrm{H}(x)=r} |x\rangle. \tag{1.34}$$

The conditional probability $\left(\frac{\alpha_{x_1 \ldots x_{j-1}1}}{\alpha_{x_1 \ldots x_{j-1}}}\right)^2$ is

$$\frac{r - \mathrm{H}(x_1 x_2 \ldots x_{j-1})}{n - j + 1} \tag{1.35}$$

for $1 \leq j \leq n$. These values can be computed using standard reversible circuits for arithmetic operations (e.g. [VBE95]) and using a circuit for computing the Hamming weight $\mathrm{H}(x_1 x_2 \ldots x_{j-1})$. A circuit for the Hamming weight was given in [KM01] and is reproduced in Figure 1.4.

We will see a detailed implementation of the controlled-increment (+1) gates in Section 2.7.1.

We have

$$\left(\frac{\alpha_{x_1 \ldots x_{j-1}0}}{\alpha_{x_1 \ldots x_{j-1}}}\right)^2 = 1 - \left(\frac{\alpha_{x_1 \ldots x_{j-1}1}}{\alpha_{x_1 \ldots x_{j-1}}}\right)^2 \tag{1.36}$$

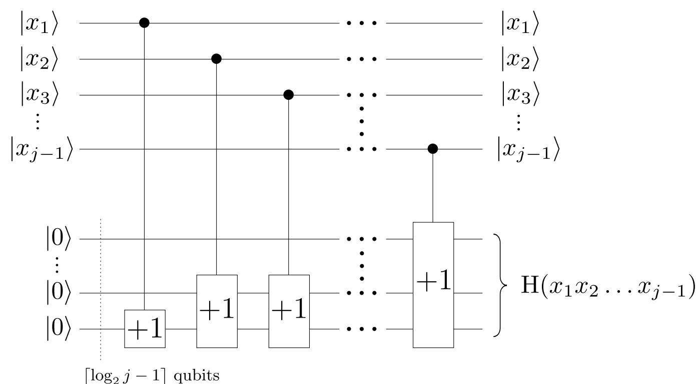


Figure 1.4: A circuit for computing the Hamming weight $\mathrm{H}(x_1x_2\ldots x_{j-1})$.

and using standard methods for arithmetic operations we can efficiently compute the ω_j satisfying

$$\cos^2(2\pi\omega_j) = \left(\frac{\alpha_{x_1\ldots x_{j-1}0}}{\alpha_{x_1\ldots x_{j-1}}}\right)^2. \tag{1.37}$$

Another example for which we can efficiently implement the V_j^Ψ is for more general symmetric pure states

$$\sum_{j=0}^{n} \beta_j |S_j\rangle \tag{1.38}$$

where we are given the β_j values (as required in [HMP+98]).

1.5 Precision

1.5.1 Precision in the generation of $|\hat{\Psi}\rangle$

We want to implement the algorithm described in Section 1.3 to generate a state $|\tilde{\Psi}\rangle$ that is a good approximation of the state $|\hat{\Psi}\rangle$. One measure of the quality of such an approximation is the *fidelity* between the states.

The classical fidelity between two probability distributions $\{p_x\}$ and $\{q_x\}$ is defined by

$$F(p_x, q_x) \equiv \sum_x \sqrt{p_x q_x}. \tag{1.39}$$

The classical fidelity is the inner product between vectors on the unit sphere with components $\sqrt{p_x}$ and $\sqrt{q_x}$. The classical fidelity between identical probability distributions is 1.

The quantum fidelity between two general quantum states ρ and σ is defined to be

$$F(\rho, \sigma) = \mathrm{Tr}\sqrt{\rho^{\frac{1}{2}}\sigma\rho^{\frac{1}{2}}}. \tag{1.40}$$

The fidelity is a nonnegative number that equals 1 when $\rho = \sigma$. So we want to estimate $|\hat{\Psi}\rangle$ with fidelity close to 1. For $|\hat{\Psi}\rangle = \sum_x \alpha_x |x\rangle$ and $|\tilde{\Psi}\rangle = \sum_x \tilde{\alpha}_x |x\rangle$, which are pure states having nonnegative real amplitudes, the fidelity is

$$\begin{aligned}
F(|\hat{\Psi}\rangle, |\tilde{\Psi}\rangle) &= \mathrm{Tr}\sqrt{\left(\sum_x \alpha_x^2 |x\rangle\langle x|\right)^{\frac{1}{2}} \left(\sum_x \tilde{\alpha}_x^2 |x\rangle\langle x|\right) \left(\sum_x \alpha_x^2 |x\rangle\langle x|\right)^{\frac{1}{2}}} && (1.41)\\
&= \mathrm{Tr}\sqrt{\sum_x \alpha_x^2 \tilde{\alpha}_x^2 |x\rangle\langle x|} && (1.42)\\
&= \sum_x \alpha_x \tilde{\alpha}_x && (1.43)\\
&= \langle \hat{\Psi} | \tilde{\Psi} \rangle && (1.44)\\
& && (1.45)
\end{aligned}$$

which equals the classical fidelity between the probability distributions that would result from measurements of the states in the $\{|x\rangle\}$-basis.

Suppose we can estimate the state $|\hat{\Psi}\rangle$ by a state $|\tilde{\Psi}\rangle$ satisfying

$$|\tilde{\alpha}_x - \alpha_x| \leq \frac{\varepsilon}{\sqrt{2^n}} \tag{1.46}$$

for each x and for some $\varepsilon > 0$. Then we have

$$\sum_x |\tilde{\alpha}_x - \alpha_x| \leq \sum_x \frac{\varepsilon}{\sqrt{2^n}} \tag{1.47}$$

$$\sum_x (\tilde{\alpha}_x - \alpha_x) \leq \sum_x \frac{\varepsilon}{\sqrt{2^n}} \tag{1.48}$$

$$\sum_x \tilde{\alpha}_x^2 - \sum_x \alpha_x \tilde{\alpha}_x \leq \sum_x \tilde{\alpha}_x \frac{\varepsilon}{\sqrt{2^n}} \tag{1.49}$$

$$\langle \hat{\Psi} | \tilde{\Psi} \rangle > 1 - \sum_x \tilde{\alpha}_x \frac{\varepsilon}{\sqrt{2^n}} \tag{1.50}$$

$$\geq 1 - \varepsilon \tag{1.51}$$

where the last inequality follows from the Cauchy-Schwartz inequality which says

$$\left(\sum_x \frac{\alpha_x}{\sqrt{2^n}} \right)^2 \leq \left(\sum_x \alpha_x^2 \right) \left(\sum_x \frac{1}{2^n} \right) = 1. \tag{1.52}$$

So (1.46) is sufficient to estimate $|\hat{\Psi}\rangle$ with fidelity at least $1 - \varepsilon$. Suppose we compute the $\tilde{\alpha}_x$ with k bits of precision so that $|\tilde{\alpha}_x - \alpha_x| \leq 2^{-k}$ for each x. Then (1.46) is satisfied if we choose k so that

$$2^{-k} \leq \frac{\varepsilon}{\sqrt{2^n}} \tag{1.53}$$

$$\Rightarrow \quad -k \leq \log_2 \left(\frac{\varepsilon}{\sqrt{2^n}} \right) \tag{1.54}$$

$$\Rightarrow \quad k \geq \log_2 \left(\frac{\sqrt{2^n}}{\varepsilon} \right). \tag{1.55}$$

We can reformulate the precision requirement in terms of the conditional probability amplitudes $\left(\alpha_{x_1 x_2 \ldots x_{j-1} x_j} / \alpha_{x_1 x_2 \ldots x_{j-1}}\right)$ that are computed by the V_j^{Ψ}. This will tell us how many qubits we must use for the register into which $|\omega_j\rangle$ is computed (as described in Section 1.3.2).

For $2 \leq j \leq n$ let $P_j = \left(\alpha_{x_1 x_2 \ldots x_j} / \alpha_{x_1 x_2 \ldots x_{j-1}}\right)$, and let $P_1 = \alpha_{x_1}$. Then each coefficient α_x for $x \in \{0,1\}^n$ is

$$\alpha_x = \Pi_{j=1 \ldots n} P_j. \tag{1.56}$$

If the V_j^Ψ produces estimates $\tilde{P}_j$ of the conditional probability amplitudes P_j, then the state generation algorithm produces a state with amplitudes

$$\tilde{\alpha}_x = \Pi_{j=1\dots n}\tilde{P}_j \tag{1.57}$$

(this follows from the proof of Claim 1.3.1).

We can rewrite (1.46) in terms of the P_j and $\tilde{P}_j$, as follows:

$$\Pi_j\tilde{P}_j \geq \Pi_j P_j - \frac{\varepsilon}{\sqrt{2^n}}. \tag{1.58}$$

We want to know how precisely we should compute each $\tilde{P}_j$ in order that (1.58) is satisfied. Suppose $j = 2$ and our estimates of P_j satisfy

$$P_1 - \tilde{P}_1 < \delta \tag{1.59}$$
$$P_2 - \tilde{P}_2 < \delta. \tag{1.60}$$

Then

$$P_1P_2 \leq \tilde{P}_1\tilde{P}_2 + \delta\left(\tilde{P}_1 + \tilde{P}_2\right) + \delta^2 \tag{1.61}$$
$$\leq \tilde{P}_1\tilde{P}_2 + 2\delta + \delta^2 \tag{1.62}$$
$$\leq \tilde{P}_1\tilde{P}_2 + 3\delta \tag{1.63}$$

since $a < 1$. So if we form the product by recursively taking the P_j (and $\tilde{P}_j$) in pairs, then we get

$$3^{\log_2 n}\delta \leq \frac{\varepsilon}{\sqrt{2^n}} \tag{1.64}$$
$$\Longrightarrow \delta \leq \frac{\varepsilon}{\sqrt{2^n}n^{\log_2 3}}. \tag{1.65}$$

Suppose we use m bits to compute the P_j (i.e. suppose we use m qubits for $|\omega_j\rangle$) so that $\delta \leq 2^{-m}$. Then taking

$$m \geq \log_2\left(\frac{\sqrt{2^n}n^{\log_2 3}}{\varepsilon}\right) \tag{1.66}$$

will be sufficient.

1.5.2 Precision in the generation of phases

Recall in Section 1.2 we used an m-qubit ancillary register to introduce phases of the form $\frac{\hat{\gamma}}{2^m}$. In this section we investigate how large m should be if we want to generate the phases with some desired fidelity $1-\varepsilon$. Suppose we have generated the state $|\hat{\Psi}\rangle = \sum_x \alpha_x |x\rangle$ with perfect fidelity. The goal is to generate the phases to create the state $|\Psi\rangle = \sum_x \alpha_x e^{2\pi i \gamma_x}|x\rangle$. We use the procedure in Section 1.2 to generate the state $|\tilde{\Psi}\rangle = \sum_x \alpha_x e^{2\pi i \tilde{\gamma}_x}|x\rangle$ where $\tilde{\gamma}_x = \frac{\hat{\gamma}_x}{2^m}$. We desire the fidelity between the states to be greater than $1-\varepsilon$. That is, we want

$$F(|\Psi\rangle, |\tilde{\Psi}\rangle) = \left|\langle\Psi|\tilde{\Psi}\rangle\right| > 1 - \varepsilon. \tag{1.67}$$

We have $|\tilde{\gamma}_x - \gamma_x| < \frac{1}{2^m}$ for all x, and $\sum_x \alpha_x^2 = 1$. Then

$$\begin{aligned}
\left|\langle\Psi|\tilde{\Psi}\rangle\right| &= \left|\sum_x \alpha_x^2 e^{-2\pi i \gamma_x} e^{2\pi i \tilde{\gamma}_x}\right| && (1.68)\\
&\geq \mathrm{Re}\left(\sum_x \alpha_x^2 e^{2\pi i (\tilde{\gamma}_x - \gamma_x)}\right) && (1.69)\\
&= \sum_x \alpha_x^2 \mathrm{Re}\left(e^{2\pi i (\tilde{\gamma}_x - \gamma_x)}\right) && (1.70)\\
&= \sum_x \alpha_x^2 \cos\left(2\pi\left(\tilde{\gamma}_x - \gamma_x\right)\right) && (1.71)\\
&= \sum_x \alpha_x^2 \cos\left(2\pi\left|\tilde{\gamma}_x - \gamma_x\right|\right) && (1.72)\\
&> \sum_x \alpha_x^2 \cos\left(\frac{2\pi}{2^m}\right) && (1.73)\\
&= \cos\left(\frac{2\pi}{2^m}\right). && (1.74)
\end{aligned}$$

So to estimate $|\Psi\rangle$ with a fidelity at least $1-\varepsilon$, when generating the phases we should take

$$m \geq \left\lceil \log_2\left(\frac{2\pi}{\cos^{-1}(1-\varepsilon)}\right)\right\rceil. \tag{1.75}$$

1.6 Conclusions

In this chapter I have described a general algorithm that will efficiently generate any desired quantum state $|\Psi\rangle$ for which we have a compact description; that is, for which we can efficiently implement the V_j^Ψ. I have analyzed the precision requirements for the algorithm to generate the state with a desired fidelity. For the specific example of the symmetric states I have given efficient circuits for implementing the V_j^Ψ.

Chapter 2

Discrete logarithms for elliptic curve groups

2.1 Background

2.1.1 Shor's algorithm

A very significant potential application of quantum computers lies in their ability to efficiently solve the problems of finding orders of elements in finite Abelian groups and of finding discrete logarithms over these groups. It is this ability that makes quantum computers capable, in principle, of undermining the security of public-key cryptographic systems that are widely used by industry and government to protect sensitive information. There are no known classical algorithms for solving the order-finding or discrete-logarithm problems in polynomial time. In 1994, Peter Shor [Sho94] described a quantum algorithm that solves both problems in polynomial time.

A key ingredient in the quantum algorithms for finding orders and discrete logarithms is a circuit for exponentiation. For example, the order-finding algorithm works by applying an inverse quantum Fourier transform to the state $\sum_x |x\rangle|a^x\rangle$ where a is a fixed group element. This state is typically created by first using a quantum Fourier transform to create a superposition $\sum_x |x\rangle$ in the first register. Then the desired state is created by

applying a controlled exponentiation circuit c-U_a^x that computes a^x into the second register conditioned on the value x in the first register.

For factoring and discrete logarithms of integers, this exponentiation is done for the integers modulo a prime. For discrete logarithms over groups of points on elliptic curves, this exponentiation is done relative to the elliptic curve group operation. There has been significant interest in designing efficient quantum circuits to perform these operations. There has also been interest in finding optimized versions of these circuits, since the construction of medium- or large-scale quantum computers is an enormous technological challenge.

In the next section, I will review prior work on quantum circuits for modular arithmetic. In the following section, I will review a method for implementing the group operation for elliptic curve groups over fields of prime characteristic. The remainder of the chapter will describe my own work extending this to curves over fields of characteristic 2.

2.1.2 Circuits for modular arithmetic

To implement Shor's factoring algorithm we must first generate the state $\sum_x |x\rangle|a^x\rangle$ where a is a fixed (known in advance) element of $\mathbb{Z}_N^*$ (that is, the multiplicative group of integers modulo N) and then apply the inverse quantum Fourier transform to this state. For the integer-discrete-logarithm algorithm we want to generate the state $\sum_{x,y} |x\rangle|y\rangle|b^x a^y\rangle$. These states can be created if we have quantum circuits for doing modular arithmetic. Reversible circuits for one-parameter modular integer multiplication (i.e. multiplication of a variable parameter by a fixed constant) appeared in [VBE95]. The first circuit discussed in that paper is a "plain adder" (also called a "ripple-carry adder") that implements $|a\rangle|b\rangle \mapsto |a\rangle|a+b\rangle$ (we might refer to this as *two-parameter in-place addition*). An adder mod N is implemented by first adding a and b using the plain adder, and then checking whether $a+b$ is bigger than N. If it is, N is subtracted from the result, achieving the modular reduction. This approach to modular addition requires about n ancillary bits (where n is the number of bits of the modulus) to keep track of the carries, and a flag to indicate whether modular reduction is required. The modular adder is used as a building block for a controlled modular multiplication circuit in [VBE95], that computes $|x\rangle|0\rangle \mapsto |x\rangle|ax\rangle$ conditioned on the state of some control qubit being $|1\rangle$ (these expressions are understood

to be mod N). Note that this implements modular integer multiplication by a *fixed factor* a, which is effectively hard-wired into the multiplication circuit. I will refer to this type of multiplication as *one-parameter out-of-place multiplication.* Note that given the out-of-place multiplication circuit M'_a we can construct an in-place version M_a, so long as we can classically precompute the inverse of the fixed value a. The in-place version makes use of a reusable ancillary qubit in the state $|0\rangle$, as follows:

$$|x\rangle|0\rangle \xmapsto{M'_a} |x\rangle|ax\rangle \xmapsto{\text{SWAP}} |ax\rangle|x\rangle \xmapsto{(M'_{a^{-1}})^{-1}} |ax\rangle|0\rangle. \tag{2.1}$$

The controlled modular multiplication circuit is used as a building block for a one-parameter modular exponentiation circuit that performs $|x\rangle|0\rangle \mapsto |x\rangle|a^x \bmod N\rangle$. A number of optimizations and improvements can be made to the circuits presented in [VBE95], some of which are mentioned in that paper and some of which appeared later (e.g. [ME99], [PP05], [Dra00]). The optimizations are summarized in [Bea03].

2.1.3 The elliptic curve group operation

The security of elliptic curve cryptography is based on the difficulty of solving the discrete-logarithm problem for groups of points on elliptic curves. Shor's algorithm can be employed to solve this problem efficiently, but it requires an efficient implementation of the elliptic curve group operation (so that the group exponentiation can be implemented).

Prior to [PZ03], none of the work on reversible implementations of arithmetic for quantum computers explicitly addressed the problem of reversibly performing *two-parameter in-place multiplication*

$$|x\rangle|y\rangle \mapsto |x\rangle|xy\rangle \tag{2.2}$$

(without generating additional junk). Indeed, for Shor's algorithms for finding orders and discrete logarithms for integers, there is no need for such an implementation (the one-parameter versions described above are sufficient). To implement Shor's discrete-logarithm algorithm for elliptic curve groups, we need to be able to compute the group operation, and again a one-parameter implementation of this suffices. The problem is that to compute the elliptic curve group operation itself requires performing multiplications of the underlying

field elements (integers mod p for curves over GF(p), or binary polynomials for curves over GF(2^m)), and *we have to be able to do this for two variable parameters* without generating additional junk. So to implement the elliptic curve group operation requires an implementation of the two-parameter in-place multiplication. This can be done if we have a reversible method for computing inverses, since we can then use a two-parameter out-of-place multiplication circuit[1] to multiply by the inverse of one of the operands to uncompute it. Proos and Zalka described how to reversibly compute the inverses of integers mod p using the *extended Euclidean algorithm* [PZ03]. This allows the implementation of the discrete-logarithm algorithm for elliptic curves over the fields GF(p). In this Chapter, I extend the work of [PZ03] to implement the group operation for elliptic curves over GF(2^m).

2.2 Elliptic curves over GF(2^m)

Many real-world elliptic curve cryptosystems are based on elliptic curves over the binary fields GF(2^m) [FIPS]. It is therefore important to examine implementations of the discrete-logarithm algorithm for elliptic curve groups over these binary fields. In this direction, I will first show how to decompose the group operation into a series of smaller, individually reversible, steps. Some of these steps will involve divisions of elements in the binary field GF(2^m). To solve this problem, I will give an efficient implementation of the extended Euclidean algorithm for polynomials.

An *elliptic curve* over a field F is the set of points $(x, y) \in F^2$ satisfying

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_5, \tag{2.3}$$

subject to some additional conditions on the constants $a_1, \ldots, a_5 \in F$, together with a 'point at infinity', denoted $\mathcal{O}$. For the particular case of curves over the finite fields GF(2^m), the defining equation and additional conditions simplify as follows.

Case 1: $a_1 \neq 0$ *(non-supersingular curves)*

Using the change of variables $(x, y) \to \left(a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4+a_3^2}{a_1^3}\right)$, and the fact that

[1] The circuit in [VBE95] for $|x\rangle|0\rangle \mapsto |x\rangle|ax\rangle$ can be adapted to give a circuit for $|x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|xy\rangle$.

the field has characteristic 2, the defining formula simplifies to

$$y^2 + xy = x^3 + ax^2 + b \quad , \quad b \neq 0. \tag{2.4}$$

Case 2: $a_1 = 0$ *(supersingular curves)*

Using the change of variables $(x, y) \to (x + a_2, y)$, and the fact that the field has characteristic 2, the defining formula simplifies to

$$y^2 + cy = x^3 + ax + b \quad , \quad c \neq 0. \tag{2.5}$$

An elliptic curve over $\mathrm{GF}(2^m)$ is the set of points $(x, y) \in \mathrm{GF}(2^m) \times \mathrm{GF}(2^m)$ that satisfy one of the above two formulae, together with the point at infinity $\mathcal{O}$. A particular curve of one of the above types is specified by giving values to the constants a, b (and c in the case of a supersingular curve). The set of points on a given elliptic curve forms a group with identity element $\mathcal{O}$, under the following operation of addition. Let $P = (x, y)$ and $R = (\alpha, \beta)$, where $P \neq R$, be two distinct points on a curve over $\mathrm{GF}(2^m)$. The point $P + R = (x', y')$ is defined as follows (my choice of labels for the curve points here is made to be consistent with the context in which I use them later).

Case 1: *non-supersingular curves*

$$P + R = \begin{cases} \mathcal{O} & \text{if } (\alpha, \beta) = (x, x + y) \\ (x', y') & \text{otherwise,} \end{cases} \tag{2.6}$$

$$\text{where} \quad x' = \lambda^2 + \lambda + x + \alpha + a \quad , \quad y' = \lambda(x + x') + x' + y \tag{2.7}$$

$$\lambda = \frac{y + \beta}{x + \alpha}. \tag{2.8}$$

Case 2: *supersingular curves*

$$P + R = \begin{cases} \mathcal{O} & \text{if } (\alpha, \beta) = (x, y + c) \\ (x', y') & \text{otherwise,} \end{cases} \tag{2.9}$$

$$\text{where} \quad x' = \lambda^2 + x + \alpha \quad , \quad y' = \lambda(x + x') + y + c \tag{2.10}$$

$$\lambda = \frac{y + \beta}{x + \alpha}. \tag{2.11}$$

Note that the parameter λ is guaranteed to exist, since GF(2^m) has characteristic 2 and $\alpha = -x = x$ is handled separately in the first case (for both supersingular and non-supersingular curves).

A more detailed treatment of elliptic curves and all of the above formulae can be found in Chapter 3 of [HMV03].

2.3 Representations of the group elements

A representation of the points on an elliptic curve must begin with a representation of the underlying field elements. The elements in a finite field of order 2^m can be represented by polynomials with binary coefficients (that is, polynomials in $\mathbb{Z}_2[x]$). We need a notion of congruence between polynomials. Suppose $g(x)$ and $f(x)$ are two polynomials, and the degree of f is m. Then dividing $g(x)$ by $f(x)$ (by the usual long division of polynomials) yields a unique quotient $q(x)$ and remainder $r(x)$ satisfying

$$g(x) = q(x)f(x) + r(x) \tag{2.12}$$

where the degree of $r(x)$ is strictly less than m. The remainder polynomial $r(x)$ is referred to as "$g(x)$ reduced modulo $f(x)$", sometimes written $g \bmod f$.

We also need a notion analogous to primality for integers. This is given by the following definition.

Definition 2.3.1 *A polynomial $f(x) \in \mathbb{Z}_2[x]$ is said to be* irreducible *if there do not exist polynomials $f_1(x), f_2(x) \in \mathbb{Z}_2[x]$ such that*

$$f(x) = f_1(x)f_2(x) \tag{2.13}$$

where $\deg(f_1) > 0$ *and* $\deg(f_2) > 0$.

The field GF(2^m) can be represented by the set of binary polynomials of degree at most $m-1$, with addition and multiplication defined as the usual operations on polynomials, followed by reduction modulo an irreducible binary polynomial of degree m.

In a computer (or quantum computer) register, the polynomials of degree at most $m-1$ can be represented by binary strings of length m, with each element in the string representing the value of one of the coefficients of the polynomial.

The points on an elliptic curve over GF(2^m) can be represented by the corresponding ordered pairs (x, y) of elements over GF(2^m). We also need a representation of the point at infinity $\mathcal{O}$. One possibility is to use an ordered pair (x, y) that is not on the curve. An implementation of the group operation would have to be tailored accordingly. For implementing the discrete-logarithm algorithm, however, we can simplify the group operation by ignoring the cases $P = R$, $P = \mathcal{O}$ and $R = \mathcal{O}$. The target register for the controlled-exponentiation operations in Shor's algorithms is usually specified as starting in the state $|1\rangle$ (the group identity). For discrete logarithms over elliptic curve groups, this translates to the point at infinity $|\mathcal{O}\rangle$. However, the algorithm will also work if we initialize this register to a random group element [2]. The control registers will contain superpositions of all 2^m group elements. So each time the controlled group operation is performed in the discrete-logarithm algorithm, about $2/2^m$ elements in superposition will be of the unsupported type (i.e. addition of inverses of points, and point doubling will both be implemented incorrectly). Since we only perform m controlled group exponentiations, only an exponentially small number of elements in the superposition will be corrupted, and so the fidelity loss will be exponentially small.

2.4 The discrete-logarithm problem

Let G be a cyclic group, and let a be a generator for G. The discrete-logarithm problem with respect to the base a is the following. Given a group element $b \in G$, find the unique integer $d \in [0, |G| - 1]$ such that $b = a^d$. The first step in Shor's quantum algorithm for solving the discrete-logarithm problem is to create the state

$$|x\rangle|y\rangle|a^x b^y\rangle. \tag{2.14}$$

[2] This is most easily seen by analyzing the second register in terms of the eigenbasis of the operator performing the group operation (see [CEMM98]).

One way to create this state is to implement a circuit that performs

$$\sum_{x,y} |x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|a^x b^y\rangle \tag{2.15}$$

where x and y are integers in the range $[0, \ldots, |G|-1]$, and a, b are fixed elements in the group G. A circuit for performing this operation can be built from circuits for performing the group operation[3] as

$$|s\rangle \mapsto |sa\rangle \tag{2.16}$$

and

$$|s\rangle \mapsto |sb\rangle. \tag{2.17}$$

Consider an elliptic curve E and let P be a point on E. Consider the cyclic subgroup of the elliptic curve group generated by P. We are interested in solving the discrete-logarithm problem for this subgroup. The group operation is written additively, so the discrete-logarithm problem is the following. Given a point Q in the subgroup generated by P, find the unique integer $d \in [0, \ldots, \text{order}(P)-1]$ such that $Q = dP$. Then, for the discrete-logarithm algorithm it suffices to be able to implement

$$|S\rangle \rightarrow |S+A\rangle \quad S, A \in E \text{ and } A \text{ is fixed and classically known.} \tag{2.18}$$

Writing $S = (x, y)$ and $A = (\alpha, \beta)$, we want to implement the elliptic curve group operation

$$|(x,y)\rangle \mapsto |(x,y) + (\alpha, \beta)\rangle. \tag{2.19}$$

2.5 Decomposing the group operation

I will now show how to decompose the group operation for curves over $\text{GF}(2^m)$ into a sequence of individually reversible steps. I use the notation $x \mapsto y$ to refer to a (not necessarily invertible) map transforming the value x to the value y. This map represents a (not necessarily reversible) computation. I will write $x \leftrightarrow y$ to refer to an invertible map

[3]Circuits for the group operation can be extended to perform exponentiation in the same way that modular multiplication circuits are extended to give modular exponentiation circuits.

transforming x to y (i.e. such that there exists an inverse mapping from y to x). This invertible map represents a computation that is reversible.

For a fixed elliptic curve point (α, β), define $(x', y') \equiv (x, y) + (\alpha, \beta)$. We want to decompose the operation

$$|(x, y)\rangle \mapsto |(x', y')\rangle. \tag{2.20}$$

For simplicity, in the following I will write the expressions without the Dirac *ket* symbols.

We will need to use the following identities, both of which are easily verified using the fact that the parameters are all in a field of characteristic 2, so that $+1 = -1$ (note the following identities do not hold in general, but do hold for the binary fields $\mathrm{GF}(2^m)$).

Identity 2.5.1

$$\lambda = \frac{y + \beta}{x + \alpha} = \frac{x' + y'}{x' + \alpha}. \tag{2.21}$$

Identity 2.5.2

$$\lambda = \frac{y + \beta}{x + \alpha} = \frac{y' + c + \beta}{x' + \alpha}. \tag{2.22}$$

Case 1: *non-supersingular curves*

The group operation is decomposed as

$$\begin{aligned} x, y \;\leftrightarrow\; x + \alpha, y + \beta \;\leftrightarrow\; x + \alpha, \lambda = \tfrac{y+\beta}{x+\alpha} \;\leftrightarrow\; x' + \alpha, \lambda = \tfrac{x'+y'}{x'+\alpha} \\ \leftrightarrow\; x' + \alpha, x' + y' \;\leftrightarrow\; x', x' + y' \;\leftrightarrow\; x', y'. \end{aligned} \tag{2.23}$$

The second step in the above decomposition is a division of the form $A, B \leftrightarrow A, B/A$, and the fourth step is a multiplication of the form $A, B \leftrightarrow A, BA$. Both of these are field operations performed on two parameters, in-place (that is, one of the operands is effectively uncomputed in the process). In the third step we use the group operation formula $x' = \lambda^2 + \lambda + x + \alpha + a$, and simplify using the fact that the field has characteristic 2. This third step requires the squaring of λ. In the third step I also rewrite the expression for λ using Identity 2.5.1. The only other operations we require are additions.

Case 2: *supersingular curves*

The group operation is decomposed as

$$x, y \leftrightarrow x+\alpha, y+\beta \leftrightarrow x+\alpha, \lambda = \tfrac{y+\beta}{x+\alpha} \leftrightarrow x'+\alpha, \lambda = \tfrac{y'+c+\beta}{x'+\alpha} \\ \leftrightarrow x'+\alpha, y'+c+\beta \leftrightarrow x', y'. \tag{2.24}$$

As in the non-supersingular case, the second step in the above decomposition is a division, and the fourth step is a multiplication, where both are of the two-parameter, in-place type. The other steps involve only additions, and one squaring. In the third step I have used Identity 2.5.2.

In both the supersingular and non-supersingular case, computing the group operation requires a method for reversibly performing in-place multiplication (division) of two parameters (that is, one of the operands is uncomputed so that no additional junk is generated). Consider the division operation. It can be decomposed into the following four reversible steps:

$$x, y \stackrel{E}{\longleftrightarrow} 1/x, y \stackrel{m}{\longleftrightarrow} 1/x, y, y/x \stackrel{E}{\longleftrightarrow} x, y, y/x \stackrel{m^{-1}}{\longleftrightarrow} x, 0, y/x. \tag{2.25}$$

The letters over the arrows are m for standard out-of-place polynomial multiplication, and E for "Euclid's algorithm" to compute inverses of field elements (polynomials in $\mathrm{GF}(2^m)$). We know how to implement the out-of-place multiplication in $\mathrm{GF}(2^m)$ (using $2m$ qubits) by [BBF03]. It remains to show how to implement the extended Euclidean algorithm for polynomials to compute inverses in $\mathrm{GF}(2^m)$.

2.6 The extended Euclidean algorithm for polynomials

Suppose $A(z)$ and $B(z)$ are two binary polynomials in the variable z, of degrees less than m (i.e. $A, B \in \mathrm{GF}(2^m)$). Suppose A and B are not both 0, and are such that $\deg(A) \leq \deg(B)$. The *greatest common divisor* of A and B, denoted $\gcd(A, B)$, is the binary polynomial of highest degree that divides both A and B. The classical Euclidean algorithm for finding $\gcd(A, B)$ is based on the fact that $\gcd(A, B) = \gcd(B - CA, A)$, for all

binary polynomials C. If we divide B by A (by standard long division of polynomials), we obtain a quotient polynomial $q(z)$ and a remainder polynomial $r(z)$ satisfying $B = qA + r$, and $\deg(r) < \deg(A)$. By the fact observed above, we have $\gcd(A, B) = \gcd(r, A)$. The classical Euclidean algorithm for polynomials makes this replacement repeatedly until one of the arguments is 0. If we set $r_0 = A$ and $r_1 = B$, the Euclidean algorithm performs the following sequence of divisions:

$$\begin{aligned}
r_0 &= q_1 r_1 + r_2, && 0 < \deg(r_2) < \deg(r_1)\\
r_1 &= q_2 r_2 + r_3, && 0 < \deg(r_3) < \deg(r_2)\\
&\vdots && \vdots\\
r_{m-2} &= q_{m-1} r_{m-1} + r_m, && 0 < \deg(r_m) < \deg(r_{m-1})\\
r_{m-1} &= q_m r_m + 0. &&
\end{aligned} \tag{2.26}$$

We then have the sequence of equalities:

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \ldots = \gcd(r_{m-1}, r_m) = \gcd(r_m, 0). \tag{2.27}$$

At this point we have the result, since $\gcd(r_m, 0) = r_m$. The algorithm is guaranteed to terminate, since the degree of one of the arguments strictly decreases in each step. Moreover, the algorithm is efficient because the number of iterations is bounded by the degree of A (which is at most m).

Recall that the gcd of two integers a, b can always be written as a linear combination of a and b having integral coefficients. The same is true for the gcd of two polynomials A, B. That is, there exist polynomials k, k' in $\mathrm{GF}(2^m)$ such that

$$\gcd(A, B) = kA + k'B. \tag{2.28}$$

The *extended Euclidean algorithm for polynomials* (EEA) is the same as the Euclidean algorithm for polynomials except that it also keeps track of the 'coefficient' polynomials k, k' above. It does so through the following recurrences.

$$k_j = \begin{cases} 1 & \text{if } j = 0\\ 0 & \text{if } j = 1\\ k_{j-2} - q_{j-1} k_{j-1} & \text{if } j \geq 2 \end{cases} \tag{2.29}$$

and

$$k'_j = \begin{cases} 0 & \text{if } j = 0 \\ 1 & \text{if } j = 1 \\ k'_{j-2} - q_{j-1}k'_{j-1} & \text{if } j \geq 2. \end{cases} \tag{2.30}$$

Claim 2.6.1 *For $0 \leq j \leq m$ we have $r_j = k_j r_0 + k'_j r_1$, where the r_j's are defined as in the Euclidean algorithm for polynomials, and the k_j and the k'_j are defined by the above recurrences.*

> **Proof:** The proof is by induction on j. The claim is clearly true for $j = 0$ and $j = 1$. Now consider $j \geq 2$ and suppose the claim is true for all smaller values of j. Then we have
>
> $$\begin{aligned} r_j &= r_{j-2} - q_{j-1}r_{j-1} & (2.31)\\ &= \left(k_{j-2}r_0 + k'_{j-2}r_1\right) - q_{j-1}\left(k_{j-1}r_0 + k'_{j-1}r_1\right) & (2.32)\\ &= \left(k_{j-2} - q_{j-1}k_{j-1}\right)r_0 + \left(k'_{j-2} - q_{j-1}k'_{j-1}\right)r_1 & (2.33)\\ &= k_j r_0 + k'_j r_1. \quad \square & (2.34) \end{aligned}$$

Since $r_0 = A$ and $r_1 = B$, and since $r_m = \gcd(A, B)$, the values k_m and k'_m generated by the above recurrence are the coefficients in (2.28).

For reference, I write the extended Euclidean algorithm for polynomials in pseudo-code below. The notation $x \leftarrow y$ is intended to mean that we assign the value of y to the variable named x.

EXTENDED EUCLIDEAN ALGORITHM FOR POLYNOMIALS

$A_0 \leftarrow A$

$B_0 \leftarrow B$

$k_0 \leftarrow 1$

$k \leftarrow 0$

$k'_0 \leftarrow 0$

$k' \leftarrow 1$

$q \leftarrow \left\lfloor \frac{A_0}{B_0} \right\rfloor$

$r \leftarrow A_0 - qB_0$

while $r > 0$ **do**

> $temp \leftarrow k'_0 - qk'$
> $k'_0 \leftarrow k'$
> $k' \leftarrow temp$
> $temp \leftarrow k_0 - qk$
> $k_0 \leftarrow k$
> $k \leftarrow temp$
> $A_0 \leftarrow B_0$
> $B_0 \leftarrow r$
> $q \leftarrow \left\lfloor \frac{A_0}{B_0} \right\rfloor$
> $r \leftarrow A_0 - qB_0$

return(r, k, k')

Inverses in GF(2^m) can be computed using the extended Euclidean algorithm for polynomials, as follows. Suppose $f(z)$ is an irreducible polynomial of degree m, and let $C(z)$ be a binary polynomial of degree at most $m-1$. Then $\gcd(C, f) = 1$, and the extended Euclidean algorithm for polynomials finds binary polynomials k and k' such that $kC + k'f = 1$. But this means that $kC \equiv 1 (\text{mod } f)$, and so $k \equiv C^{-1} (\text{mod } f)$. The coefficient k' of f is not needed for the inversion of C, and so we only need to record the coefficient k of C throughout the algorithm.

2.7 Naive implementation of the extended EEA

Let us now turn our attention to reversible implementations of the extended Euclidean algorithm for polynomials for computing the inverse of an element C. The implementations will maintain two ordered pairs (a, A) and (b, B), where A and B record the sequence of remainders in the EEA, and a and b record the updated coefficient of C for each of the past two iterations of the algorithm. We call these ordered pairs *Euclidean pairs*. The algorithm begins with $(a, A) = (1, C)$, and $(b, B) = (0, f)$ (where f is an irreducible polynomial of degree m). Note that $\deg(C) \leq m - 1 < m = \deg(f)$. We will always store the Euclidean pair with the smaller-degree polynomial in the second coordinate first. That is, we store the Euclidean pairs in the order

$$(a, A), (b, B) \tag{2.35}$$

where $\deg(A) < \deg(B)$. We then want to perform long division of B by A, obtaining a quotient polynomial q and a remainder polynomial r satisfying $B = qA - r = qA + r$ (the second equality follows since the field is binary), where q is the quotient polynomial of B/A, which we denote as $q = \lfloor B/A \rfloor$. We will then replace B by $r = B + qA$, and b by $b + qa$. Since $\deg(r) < \deg(A)$, after the above replacement we will have to interchange the Euclidean pairs to maintain the ordering so that the pair with the smaller-degree polynomial in the second coordinate appears first. So one iteration of the algorithm can be written as

$$(a, A), (b, B), 0 \to (b + qa, B + qA), (a, A), q \qquad \text{where } q = \lfloor B/A \rfloor . \tag{2.36}$$

At the beginning of the Euclidean algorithm, we start with $a = 1, b = 0, A = C, B = f$, and so $\deg(A) < \deg(B)$ and $\deg(a) > \deg(b)$. It is easy to see that this condition is preserved in every iteration of the algorithm. This implies that we will have $\left\lfloor \frac{b}{a} \right\rfloor = 0$. So we can write

$$q = \left\lfloor \frac{b + qa}{a} \right\rfloor . \tag{2.37}$$

So, while q is computed from the second coordinates of the Euclidean pairs $(a, A), (b, B)$, it can be uncomputed from the first coordinates of the modified Euclidean pairs $(b + qa, B + qA), (a, A)$. Thus each iteration of the Euclidean algorithm is individually reversible, and

can be written as

$$(a, A), (b, B) \leftrightarrow (b + qa, B + qA), (a, A) \qquad \text{where } q = \lfloor B/A \rfloor. \tag{2.38}$$

This is decomposed into the following three individually reversible steps:

$$\begin{aligned} A, B, 0 &\leftrightarrow A, B + qA, q \\ a, b, q &\leftrightarrow a + qb, b, 0 \\ &\text{SWAP} \end{aligned}$$

where "SWAP" refers to the operation of switching the two Euclidean pairs. Since $\deg(b) < \deg(a + qb)$, the second operation above is simply the reverse of the first operation.

To perform the division $A, B, 0 \leftrightarrow A, B + qA, q$ we can use long division of the binary polynomial B by A. To implement this long division, the basic idea is to shift A all the way to the left (i.e. we shift A left by $m - \deg(A) - 1$ bits). Then we start shifting A to the right one bit at a time, each time conditionally doing a subtraction. For the binary field GF(2^m) this is simplified by virtue of the fact that subtraction is the same as addition, and is achieved by a bitwise XOR operation. This bitwise XOR can be implemented quantumly using CNOT gates, and no ancillary qubits. (Furthermore, these CNOT gates could in principle be performed in parallel, allowing us to do addition in a single step.) Note that in our long divisions we are doing more work than necessary. Often the degree of B will be less than $m - 1$, and so it would not be necessary to shift A all the way to the left (we could just shift it so the most significant bits of A and B line up). For simplicity, in the naive implementation we do not take advantage of this fact, but will do so when we look at an optimized implementation.

2.7.1 Implementing some tools

The long division will require some subroutines, which I will show how to implement in this section. I will show how to implement some basic operations that I will then need to apply conditioned on the value(s) of some other qubit(s). We need to consider the overhead required in making the controlled versions of these operations. Fortunately, by [BBC+95], given a circuit implementing a unitary operation U, we can construct a circuit

for performing a *controlled-U* (that is, U conditioned on a control qubit being in state $|1\rangle$) with no additional ancillary qubits, and a small overhead in running time. Further, we can implement U conditioned on any desired pattern of states of several control qubits (e.g. U may be applied only when a three-qubit control register is in the state $|101\rangle$) with no additional ancillary qubits, and a small overhead in running time.

For the long division, we will need to compute the degree of A. The circuit shown in Figure 2.1 accomplishes this. The (-1) gate decrements the integer value encoded (in binary) in the register. Each of the hollow circles in the figure denotes a 0-*control* (that is, the (-1) operation is applied if all those control qubits are $|0\rangle$). To uncompute the degree, we can simply run the circuit shown in Figure 2.1 backwards.

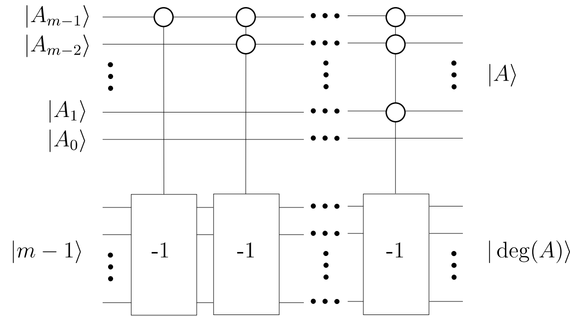


Figure 2.1: A circuit to compute the degree of $A \in GF(2^m)$.

The circuit in Figure 2.1 uses a sequence of m decrementing (-1) gates, each of which is controlled by the values of some of the qubits of $|A\rangle$. These decrementing gates update the value of $\deg(A)$, being computed into a $\lceil \log_2(m-1) \rceil$-qubit register. In Figure 2.2, we show how to implement an incrementing $(+1)$ gate using only one additional ancillary qubit.

The ancillary qubit becomes the most-significant bit of the result. If we only apply the incrementing circuit to integers in the range $[0, \ldots, m-2]$, we know that the ancillary qubit will always be $|0\rangle$ at the output. Decrementing is accomplished by running this circuit backwards, with the ancillary qubit initially set to $|0\rangle$. As long as we apply the decrementing circuit to integers in the range $[1 \ldots m-1]$, we know that the ancillary qubit will always be $|1\rangle$ at the output. So we can reset the ancillary qubit to $|0\rangle$ with a NOT gate

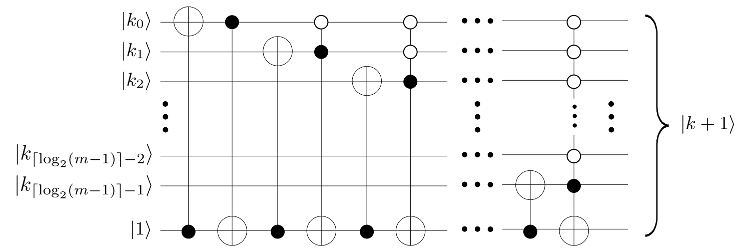
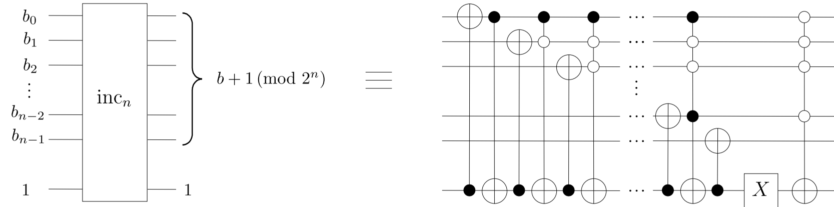


Figure 2.2: A circuit to compute $|k\rangle \leftrightarrow |k+1\rangle$.

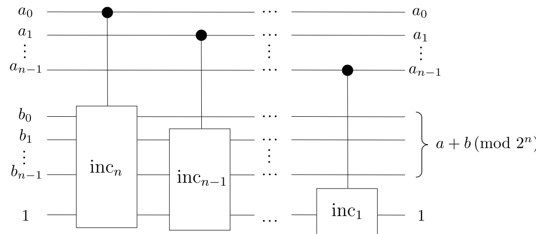
after each decrement gate, and reuse that ancillary qubit for the next decrement gate. The degree of $A \in GF(2^m)$ can be computed using $\lceil\log_2(m-1)\rceil + 1$ qubits (a $\lceil\log_2(m-1)\rceil$-qubit register into which the result is computed and stored, and 1 ancillary qubit shared by the decrementing gates)

Aside: an addition circuit

The incrementing circuit (Figure 2.2) can be used to implement a reversible addition circuit that uses fewer ancillary bits than the circuits reviewed in Section 2.1.2. First the incrementing circuit is modified so that the ancillary qubit is reset to $|1\rangle$ at the end, as shown below.



Then an addition circuit makes use of controlled-incrementing circuits. The controlled incrementing circuit performs the incrementing operation conditioned on a control qubit being in the state 1. The addition circuit is shown below.



By carefully counting the depth of each of the general controlled-NOT operations in the controlled-inc_k circuit, we find

$$|\text{c-inc}_k| = \begin{cases} 1 & \text{if } k = 0 \\ 10 & \text{if } k = 2 \\ 2k^2 + k - 5 & \text{if } k \geq 3. \end{cases}$$

Note that in the above calculation, I assumed that the controlled-inc_k operations will be in the context of a circuit having at leats $2k$ bits in total, so the requirement of the implementation in [BBC+95] is satisfied. This is true for the addition circuit. For $n \geq 3$, the total depth of the addition circuit is

$$\sum_{k=1}^{n} |\text{c-inc}_k| = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{25}{6}n + 8.$$

We also need to implement shifts of our quantum registers. For our purpose it will suffice to implement a cyclic shift. We will make use of the quantum SWAP gate, which can be implemented using 3 CNOT gates and no ancillary qubits, as shown in Figure 2.3.

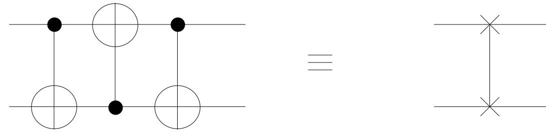


Figure 2.3: The quantum SWAP gate.

A left cyclic shift gate which shifts the state of an n-qubit register to the left cyclically by one qubit is implemented using $n - 1$ SWAP gates, and no ancillary qubits, as shown in Figure 2.4.

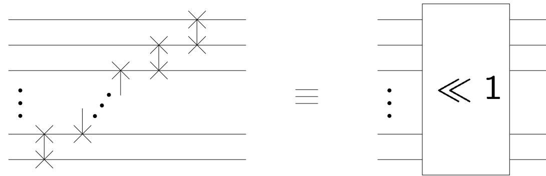


Figure 2.4: A cyclic left shift gate. (The upper qubits in the circuit correspond to those on the left side of the register.)

A left shift of s qubits can be implemented by concatenating s single-qubit left shifts together. Note that right shifts can be performed in an analogous manner. We will also need to implement a shift conditioned on the value contained in a quantum register. That is, a quantum implementation of the operation

$$|\theta\rangle|s\rangle \leftrightarrow |\theta \ll s\rangle|s\rangle. \tag{2.39}$$

The controlled shift operation above is implemented by the circuit shown in Figure 2.5, where k denotes the number of bits in the binary representation of s.

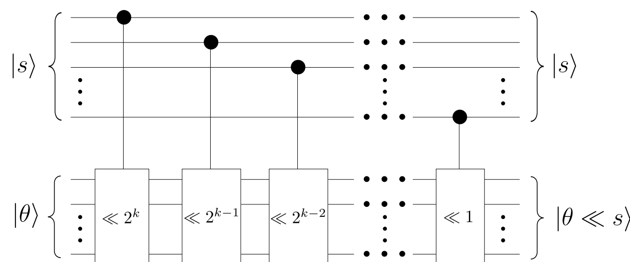


Figure 2.5: A circuit for $|\theta\rangle|s\rangle \leftrightarrow |\theta \ll s\rangle|s\rangle$. Here $k = \log_2 s$, and $\ll 2^k$ is implemented by a sequence of 2^k ($\ll 1$) gates (shown previously).

2.7.2 Long division

Now that we can compute the degrees of polynomials in GF(2^m), and perform shifts of quantum registers, we can state an algorithm to reversibly compute the long division

$$A, B, 0 \leftrightarrow A, B + qA, q \tag{2.40}$$

(note the algorithm requires $\deg(A) \leq \deg(B)$).

Long Division (B divided by A)

(0) Initialize $q = 0$.

(1) Compute $\deg(A)$.

(2) Compute $i = m - \deg(A) - 1$.

(3) Shift A left by $m - \deg(A) - 1$ positions.

(4) While $i \geq 0$ do

- (4.1) If $B_{i+\deg(A)} = 1$, then set $q_i = 1$ and replace B with $B \oplus A$.
- (4.2) Shift A to the right one bit.
- (4.3) $i \leftarrow i - 1$.

(5) Uncompute $\deg(A)$.

At the end of the long division, the register originally containing B will contain $r = qA+B$. Also, the auxiliary counter i will be zeroed, and so can be reused. The conditional setting of $q_i = 1$ in step (4.1) can be accomplished by a CNOT gate, with $|B_{i+\deg(A)}\rangle$ as the control qubit and $|q_i\rangle$ as the target qubit. The operation $|A, B\rangle \leftrightarrow |A, A \oplus B\rangle$ can be accomplished by CNOT gates between the corresponding qubits of A and B. This operation is applied conditioned on $q_i = 1$, and is implemented by using Toffoli gates in place of the CNOT gates, with $|q_i\rangle$ as the additional control qubit.

2.8 The problem of synchronization

Classically, if properly implemented, the extended Euclidean algorithm takes time $O(m^2)$ [MvOV97]. To achieve this, one has to take advantage of the fact that the quotient in the $O(m)$ divisions is usually small, thus we have to use a division algorithm that takes time proportional to the number of subtractions that are needed. In such an efficient implementation, the time each division step takes depends on the input to the algorithm, and so also the time at which we reach the ith step will depend on the input. We want to apply the Euclidean algorithm to a superposition of different inputs and thus we have to "desynchronize" these parallel calculations so that different inputs in superposition can be executing different iterations of the algorithm at any given time[4].

This synchronization problem can be dealt with by applying a general technique of *desynchronization* [PZ03]. I explain desynchronization by way of an example. Suppose a computation C consists of some sequence of three simple reversible operations o_1, o_2 and o_3 (and no other operations). The time taken to perform each of the operations o_1, o_2, o_3 is independent of the input. This means that on a superposition of inputs, the time required to perform the operation o_1 (for example) is the same for all elements in the superposition.

The quantum computation C is some sequence of the operations o_1, o_2 and o_3, in any order, possibly with repetitions. For example, C applied to the input basis state $|x\rangle$ might consist of o_1 applied 4 times, followed by o_2 applied 1 time, followed by o_3 applied 2 times,

[4]To see this, one needs not think in quantum terms; it is enough to think about reversible computation.

followed by o_1 applied 1 time, followed by o_2 applied 3 times. That is,

$$C|x\rangle = o_2 o_2 o_2 \;\; o_1 \;\; o_3 o_3 \;\; o_2 \;\; o_1 o_1 o_1 o_1 |x\rangle. \tag{2.41}$$

The synchronization problem is that for another input basis state $|x'\rangle$ (in a superposition of inputs), the sequence of operations might be different. For example, on $|x'\rangle$ the same computation C might consist of o_1 applied 1 time, followed by o_2 applied 4 times, followed by o_3 applied 1 time, followed by o_1 applied 3 times. That is,

$$C|x'\rangle = o_1 o_1 o_1 \;\; o_3 \;\; o_2 o_2 o_2 o_2 \;\; o_1 |x'\rangle. \tag{2.42}$$

The idea of desynchronization is to have all the computation paths in the superposition cycle through the 3 operations repeatedly, each time allowing the computation to either apply the operation once, or not apply it (wait for the next operation). The cycle is repeated enough times that sufficiently many of the computations in superposition have finished. For the computation C above applied to the two input basis states $|x\rangle$ and $|x'\rangle$, this is illustrated in Figure 2.6. In the figure, the operation being applied at each step is indicated by an $\times$ in the corresponding box.

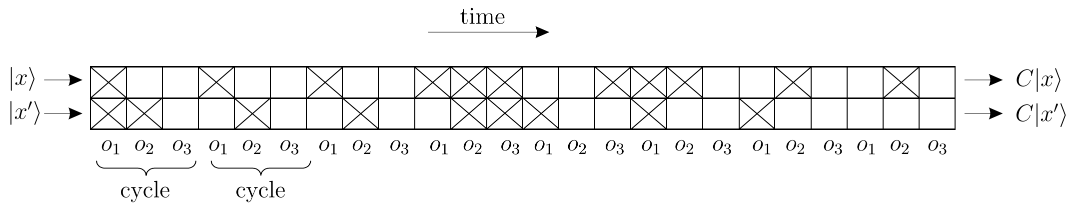


Figure 2.6: Desynchronization example.

I now describe more explicitly how to implement desynchronization. There must be a way for the computation to tell when a series of o_i's is finished and the next one should begin. We want to do this reversibly, so there must be a way to tell both when an o_i is the first in a series, and when it is last in a series. In each o_i we can include a sequence of gates that flips a flag qubit f if o_i is the first in a sequence, and another mechanism that flips f if o_i is the last in a sequence. We also make use of a small "counter" register c to control which operation is scheduled to be applied at the current step. Thus we have a triple x, f, c where x stands for the actual data. We initialize both f and c to 1 to signify that the

first operation will be the first in a sequence of o_1 operations. The physical quantum gate sequence that we apply is

$$\dots\ ac\ o_1'\ ac\ o_3'\ ac\ o_2'\ ac\ o_1'\ ac\ o_3'\ ac\ o_2'\ ac\ o_1'\ |x\rangle \tag{2.43}$$

where the o_i' are the o_i conditioned on $i = c$ and ac stands for "advance counter". These operations act as follows on the triple:

$$\begin{aligned} o_i' : &\quad \text{if } i = c,\ \ x, f, c \ \leftrightarrow\ o_i(x), f \oplus \text{first} \oplus \text{last}, c \\ ac : &\quad x, f, c \ \leftrightarrow\ x, f, (c + f) \bmod 3 \end{aligned}$$

where o_i' does nothing if $i \neq c$, the symbol "$\oplus$" means XOR, and $(c + f) \bmod 3$ is taken from $\{1, 2, 3\}$. In the middle of a sequence of o_i' operations, the flag f is 0, and so the counter doesn't advance. The last in a sequence of o_i' operations will set $f = 1$ and the counter will advance in the next ac step. The first operation of the next series resets f to 0, so that this series can progress.

Although desynchronization can be applied to the individual steps in each iteration of the algorithm, the computations in the superposition will in general finish the extended Euclidean algorithm after different numbers of iterations. For those that finish earlier than others, we cannot simply have them halt and wait for the others to finish (this would result in an implementation that is not reversible). To ensure reversibility, those elements in superposition that finish earlier can increment a small counter at each time step until the other elements in superposition finish. I will call this small counter the "halting counter".

I do not describe in detail how to apply desynchronization to the naive implementation, but instead proceed with a better optimized implementation that will make use of desynchronization.

2.9 An optimized implementation

2.9.1 The implementation

The starting point for an optimized implementation is the observation that the degrees of the polynomials being divided decrease steadily during the extended Euclidean algorithm

for polynomials. In the naive implementation, by shifting A all the way to the left for all the long divisions, we were doing more work than necessary. The optimized implementation will make use of "adaptive" long divisions, whose behaviour is conditioned on the sizes of the arguments.

The other main observation underlying the optimized implementation is that in the naive implementation we were using much more space than necessary to store the Euclidean pairs. In the naive implementation we used a separate m-qubit register for each of A, B, a, b. It turns out that this is twice as much space as is necessary.

Claim 2.9.1 *At every stage of the extended Euclidean algorithm for polynomials we have* $\deg(aB) = m$.

Proof: Initially we have $aB = f$ and so $\deg(aB) = m$, so the claim is true at the first iteration. Each iteration transforms

$$\begin{aligned} a &\to a' = b + qa \\ B &\to B' = A. \end{aligned}$$

So we have

$$\begin{aligned} \deg(a'B') &= \deg((b + qa)A) \\ &= \deg(qaA) \quad (\text{since } \deg(qa) \geq \deg(a) > \deg(b)) \\ &= \deg(q) + \deg(a) + \deg(A) \\ &= \deg(B) - \deg(A) + \deg(a) + \deg(A) \\ &= \deg(aB) \\ &= m \end{aligned}$$

and so the claim is true after each iteration. □

We have the following corollary.

Corollary 2.9.1 *At every stage of the extended Euclidean algorithm for polynomials we have*

$$\deg(a) + \deg(A) \leq m \quad \text{and} \quad \deg(b) + \deg(B) \leq m. \tag{2.44}$$

Proof: Since $\deg(A) < \deg(B)$ we have

$$\deg(a) + \deg(A) = \deg(aA) \leq \deg(aB) = m. \tag{2.45}$$

Similarly, since $\deg(a) > \deg(b)$ we have

$$\deg(b) + \deg(B) = \deg(bB) \leq \deg(aB) = m. \quad \square \tag{2.46}$$

By the corollary, we see that a single m-qubit register will be sufficient to store both a and A, and a second m-qubit register is sufficient to store both b and B. Thus A and a can share a single m-qubit register, and b and B can share a second m-qubit register. This reduces the total space to store A, B, a, b from $4m$ to $2m$. The problem with this approach is that the relative sizes of a and A change from one iteration to the next, and thus so does the boundary between A and a within the single m-qubit register (similarly for b and B). Further, at any iteration, this boundary may be different between elements in superposition. So we need a way to calculate the position of this boundary for each iteration.

First, observe that the boundary between A and a can be at the same position as the boundary between B and b, in any iteration (since $\deg(A) < \deg(B)$). Second, notice that the boundary can be easily determined if we know the degrees of A, B, a, b. It will turn out to be convenient to store A and a in a single register in opposing directions. That is, the most significant bit of A is at one end of the register, and the most significant bit of a is at the extreme other end of the register. Between A and a the register will be padded with zeros. Similarly for B and b. The situation for register sharing is illustrated in Figure 2.7.

From Figure 2.7 it can be seen that the boundary for register-sharing can be determined from $\deg(a)$ or from $\deg(B)$. Our strategy will be to store the degree of each of A, B, a, b at each step, and use either $\deg(a)$ or $\deg(B)$ (depending on what operation we are performing) to determine the boundary. For convenience, we will keep track of the degrees of all of A, B, a and b, requiring 4 separate $\lceil \log_2 m \rceil$-qubit registers.

As before, we focus on implementing the long division

$$A, B, 0 \ \leftrightarrow \ A, B + qA, q. \tag{2.47}$$

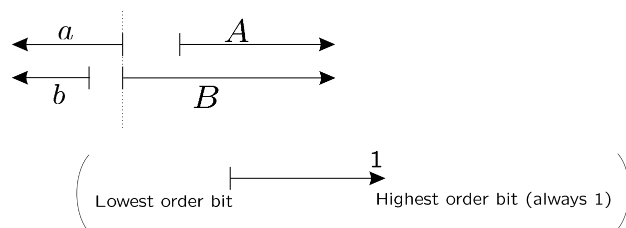


Figure 2.7: The positions of A, B, a, b for register sharing.

The long division algorithm is modified slightly as a result of the new strategy for storing A and B. Note that we do not need to initially shift A all the way towards the high-order end, since the most significant bits of A and B are already in the same position. Instead of shifting A one bit at a time towards the low-order end at each step, we shift B one bit at a time towards the high-order end. At each stage, a new bit of q is first read-out from the high-order bit of B. Then, controlled on the new bit of q (equivalently the high-order bit of B), B is XORed with A (this is the conditional subtraction). Then B is shifted towards the high-order end by 1 bit, and the value of $\deg(B)$ is decremented by 1. Note that no significant bits of B are lost in the shift, because after the conditional XOR operation, we know the high-order bit of B will be 0. After the long division is complete, the remaining operation is to shift off any leading (high-order) zeros in the final value of B, and decrement the value of $\deg(B)$ accordingly. This is done so that the most significant bits of A and B are in corresponding positions for the next iteration. The operations o_1 and o_2 for implementing the long division are as follows:

o_1: (*a*) The high-order bit of B becomes the next bit of q (starting at the high-order bit of q and working down).

(*b*) Conditioned on the new bit of q, B is replaced with $B \oplus A$.

(*c*) B is shifted towards the high-order end by 1 bit, and $\deg(B)$ is decremented by 1.

o_2: B is shifted towards the high-order end by 1 bit, and $\deg(B)$ is decremented by 1.

The first in a sequence of o_1 operations is recognized by the condition $q = 0$. The last in a sequence of o_1 operations is recognized by $\deg(A) = \deg(B)$. When performing the

last in a sequence of o_1 operations, only part (a) is performed (so parts (b) and (c) can be conditioned on the flag qubit). The first in a sequence of o_2 operations is recognized by $\deg(A) = \deg(B)$. The last in a sequence of o_2 operations is recognized when the bit in the high-order "slot" of the register containing B is $|1\rangle$.

The long division algorithm is illustrated by an example. Suppose we have the following:

$$\begin{aligned} A &= z^2 + 1 \qquad (A = 101) \\ B &= z^4 + z^2 + 1 \qquad (B = 10101). \end{aligned}$$

The long division B/A as would be performed by hand is shown in Figure 2.8.

```
         1 0 0
1 0 1 ) 1 0 1 0 1
        1 0 1 0 0
        0 0 0 0 1
        0 1 0 1 0
        0 0 0 0 1
        0 0 1 0 1
        0 0 0 0 1
```

Figure 2.8: Example of long division by hand.

The long division as performed by the algorithm is shown in Figure 2.9. One feature of the algorithm suggested by the example is that the qubits can be spatially arranged so that operations are performed on neighbouring qubits, which might be advantageous for some physical implementations. Note that in the implementation of the shifts (Figure 2.4), the CNOT gates are between adjacent qubits as well.

I have omitted the details of how to condition the steps of the long division on the value that determines the boundary for register sharing. For example, in the implementation of $A, B, 0 \leftrightarrow A, B + qA, q$, the operations on A, B, q will be conditioned on the value in the register containing $\deg(a)$ (from which the boundary position for register sharing can be determined). These details are very complicated, but the techniques for implementing controlled gates in [BBC+95] indicate that it can be done with no ancillary qubits, and a polynomial increase in time.

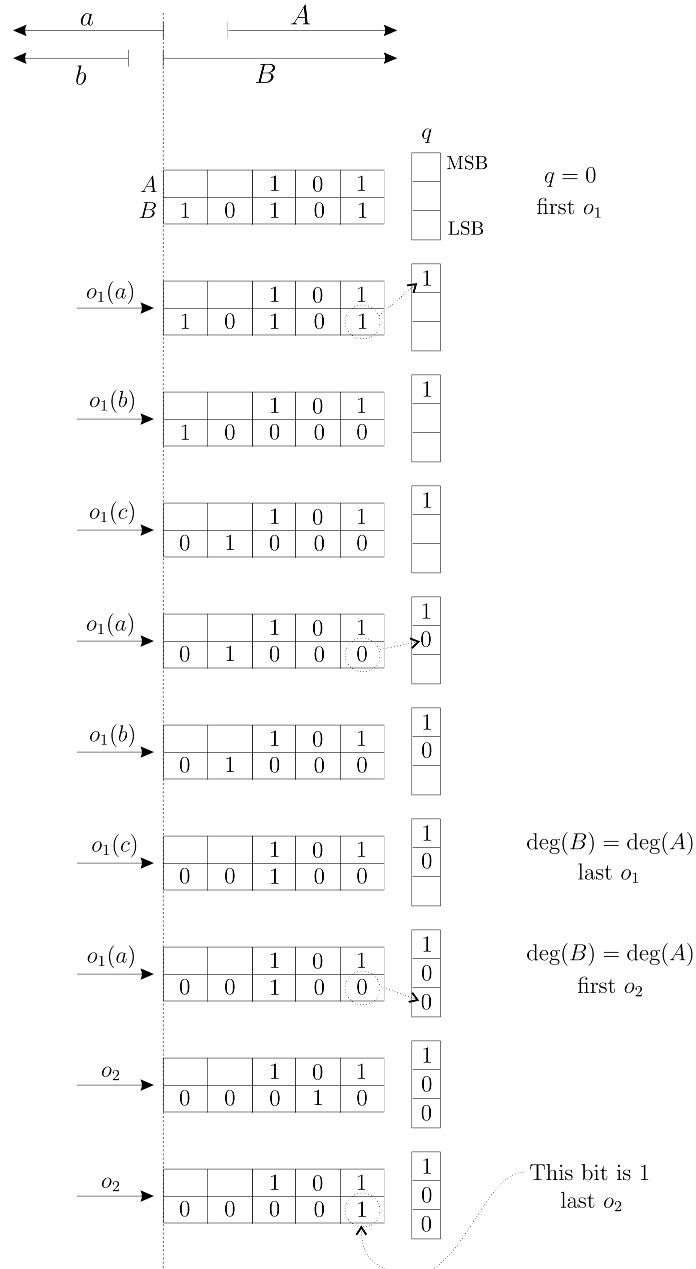


Figure 2.9: Example of optimized implementation of long division. Blank cells implicitly contain the value 0.

2.9.2 Space complexity

We saw in Section 2.5 that the number of qubits required to implement the elliptic curve group operation is determined by the number of qubits required to implement the extended Euclidean algorithm for polynomials. Now we will count the number of qubits required by the optimized implementation.

By using register sharing, the values of A, B, a, b can be stored using $2m$ qubits, and the value of q can be computed into a third m-bit register. The values of $\deg(A)$, $\deg(B)$, $\deg(a)$ and $\deg(b)$ must be initially computed and stored, requiring $4\lceil \log_2 m \rceil + 4$ qubits (as seen in Section 2.7.1). For the desynchronization, we need a flag qubit f, and a 2-qubit counter register c (to index the 4 operations $o_1(a), o_1(b), o_1(c)$, and o_2 used in the desynchronization). Recall that we also need a halting counter, as the computations in the superposition will finish the extended Euclidean algorithm for polynomials after different numbers of iterations. The exact size of this halting counter depends on the exact time complexity of the algorithm. However, as our implementation is clearly polynomial in m, we know that the size of the halting counter will be at most logarithmic in m. We will write H for the number of qubits required for the halting counter, where it is understood that H is $O(\log_2 m)$.

So the space complexity for our implementation of the extended Euclidean algorithm for polynomials, and thus of the elliptic curve group operation for curves over $\mathrm{GF}(2^m)$, is

$$\underbrace{3m}_{A,B,a,b,q} + \underbrace{4\lceil \log_2 m \rceil + 4}_{\deg A, \deg B, \deg a, \deg b} + \underbrace{1+2}_{f,c} + H \tag{2.48}$$

$$= 3m + 4\lceil \log_2 m \rceil + 7 + H. \tag{2.49}$$

The discrete logarithm algorithm requires only one register for elliptic curve points, plus an additional control qubit[5]. An elliptic curve point can be represented by two m-bit field elements, and so the space requirement for the discrete logarithm algorithm is 2 m-bit registers plus a register on which to carry out the EEA. The total is

$$5m + 4\lceil \log_2 m \rceil + 8 + H. \tag{2.50}$$

[5]This is due to a semiclassical implementation due to [GN96].

2.10 Conclusions and future work

I have given a reversible implementation of the extended Euclidean algorithm for polynomials for finding inverses of elements in $GF(2^m)$. This is a requirement for implementing Shor's algorithm for finding discrete logarithms in elliptic curve groups for curves over binary fields (which are used extensively in classical cryptosystems). The ability to compute inverses also solves the problem of reversibly performing multiplication of two variable parameters in $GF(2^m)$ without generating additional junk (or equivalently for uncomputing the junk bits left over by existing methods for multiplying two parameters).

It should be possible to combine in some way the techniques described here and in [PZ03] to give a method of reversibly computing inverses in the more general Galois fields $GF(p^k)$, but the details of such a strategy have yet to be explored.

Chapter 3

Globally controlled quantum arrays

3.1 Background

3.1.1 Quantum cellular automata and globally controlled arrays

Quantum cellular automata (QCA) is an idea that dates back at least to 1983, where it is presented in Richard Feynman's famous paper in which he proposed the idea of a quantum computer. "QCA" is perhaps a misleading choice of terminology, because many of the schemes that are referred to by that name are not actually autonomous. The term "classically controlled quantum cellular automata" (ccQCA) has been used to distinguish these systems from truly autonomous schemes, but that term is perhaps somewhat of an oxymoron. Instead, I will refer to the schemes we consider in this chapter as "globally controlled arrays", and abbreviate this with the acronym "GCA".

Several proposals for implementing globally controlled arrays of qubits using spacially-symmetric lattices of qubits have appeared over the years. In 1993, Lloyd proposed an architecture based on a 1-dimensional lattice of weakly coupled qubits of 3 distinguishable species [Llo93]. The state of the qubits in the lattice is influenced by global "pulses". These pulses affect all qubits in the lattice in uniform manner. For example, a given global pulse may apply a certain unitary operator to every qubit in the lattice whose left and right neighbours are in some specified basis states. Local control is achieved by encoding

the information in the lattice in a clever manner, so that the global pulses have a desired effect only on specific local patterns of states within the lattice. Computation is achieved by applying a suitable sequence of global pulses, controlled by a classical program.

This basic model has been adapted and modified over the years. Benjamin devised an architecture with only 2 distinguishable species, where the couplings do not allow the states of the left and right neighbours to be distinguished (isotropic) [Ben00]. Related architectures have been proposed by Benjamin, Kay and others (e.g. [BBK04]). The various architectures differ in the number of distinguishable species, the manner in which information is encoded in the lattice, the structure of the underlying lattice itself, and the nature of the control pulses that can be applied. There is a common structure exhibited by all these architectures, however, in the way the information and fundamental operations are organized to implement the basic steps of a computation. In practice, the selection of a GCA architecture for a particular application will be an engineering consideration, and the choices may be constrained by the technologies that become available for implementing them.

A GCA is programmed by selecting a sequence of global control pulses. Different GCA architectures will support different kinds of control pulses. The set of control pulses applicable to a particular GCA architecture can be thought of as the *machine language* for that architecture.

Programs for GCA are usually specified to simulate the behaviour of a given quantum circuit. This is practical, because the known quantum algorithms are most commonly expressed and understood with respect to the quantum circuit model. Because of the differences in the machine instructions between GCA architectures, the program to simulate any given quantum circuit may vary greatly between them. This is unfortunate, because the various GCA architectures are organized quite similarly, at a slightly higher level of abstraction. A program for a GCA to simulate a quantum circuit is most often designed and understood by organizing sequences of control pulses into sequences of more sophisticated operations at this higher level of abstraction.

It is desirable to have a unified framework for studying GCA programs, protocols, and complexity results in a manner that is independent of implementation details. This framework

can take the form of an *assembly language* for GCA. Such a langauge will consist of a set of *basic instructions*, at a level of abstraction higher than that of the pulse sequences specific to individual architectures. An algorithm or simulation of a given quantum circuit can be expressed in terms of these basic instructions and will then be applicable to an entire class of GCA architectures. It may also provide a convenient framework for theoretical investigations into the behaviour of GCA architectures in general. Given a candidate architecture for a GCA, it will suffice to show how to implement the basic instructions using the machine language of that architecture. So the GCA assembly language also provides a requirements specification for designing physical systems to implement GCA.

In the following sections, I will define such a language for GCA, and give examples illustrating how the basic instructions can be implemented for specific GCA architectures. I will then consider error correction and fault tolerance for GCA in the context of this framework. I will consider two kinds of approaches, and discuss the advantages and disadvantages of each. The fault-tolerance requirement will motivate the definition of a more powerful GCA allowing for more parallelism, and we will see techniques that can be used to implement this.

3.1.2 Between quantum circuits and simple spin chains

Studies into global control systems have been motivated by the goal of showing that we can do quantum computation using systems for which achieving local control over individual qubits can be very difficult (e.g. quantum computing using nuclear magnetic resonance). For this reason, many of the proposed schemes have used very simple spin chains, consisting of one, two or three distinct species of qubits. We might consider the quantum circuit model to be the science-fiction end of a spectrum of (real and imagined) quantum computing technology, where we have total local control and can address any desired qubit in the circuit. By contrast, we might consider the simple spin-chain models as representing the other end of the spectrum, where we have extremely limited local control. While we know that we can do universal quantum computing at both ends of this spectrum, it is not clear how well we can do fault-tolerant quantum computing in the case of extremely limited local control.

Later in this chapter, when I consider implementing error correction schemes for global control models, I will be exploring the territory between these two extreme ends of the spectrum. That is, I will consider models with various amounts of local control. While implementing some of my schemes is beyond current technology, it should be remembered that implementing quantum circuits is also beyond current technology. Since it is not yet clear what the winning technology for implementing quantum computers will be, it is worth investigating the potential for reliable quantum computing for systems offering different amounts of local control.

3.2 The basic GCA model

Conceptually, the behaviour of the simplest GCA schemes can be organized in terms of a structure that resembles a Turing machine (we will see later how we can extend this structure to achieve certain kinds of parallelism). This can be described in terms of a finite 1-dimensional *array* of *data qubits*, along with some mechanism to act as a *pointer* to address specific data qubits in the array. For some of the schemes described in the literature, the data array is assumed to be infinite. For practical implementations, however, the data array would consist of a finite number of data qubits, say $d_1, \ldots d_N$. The data qubit d_{i-1} is referred to as the *left neighbour* of d_i, and d_{i+1} is the *right neighbour* of d_i. This basic structure is illustrated below.

d_1	d_2	$\cdots$	d_{i-1}	d_i	d_{i+1}	$\cdots$	d_N
				$\uparrow$ pointer			

(3.1)

It should be emphasized that the data qubits will not in general correspond directly to the physical qubits in a lattice implementing a GCA. Each data qubit may be encoded by several physical qubits in the lattice. Some of the lattice qubits may be used to encode the position of the pointer rather than states of the data qubits. The set of physical lattice qubits used to encode the state of a given data qubit may even change during the course of a computation. We use the term "array" to distinguish the list of logical data qubits from the underlying physical "lattice" that may be used to encode this array (and the pointer).

3.2.1 The language SPA

In this section, I will define a langauge of basic instructions that describes the behaviour of a variety of simple GCA architectures. I will refer to this language by the acronym SPA, for "Single Pointer Array".

The position of the pointer in a GCA model is a classical parameter (we do not allow the pointer to be in multiple positions in quantum superposition), and so SPA refers to the machine's state as follows.

$$\boxed{\begin{array}{c}\underline{\textit{State}\text{ for }\texttt{SPA}}\\ \big(\rho, i\big)\end{array}} \tag{3.2}$$

In the above expression, ρ is the density operator for the N-qubit state of the array of data qubits, and i is an integer between 1 and N representing the position of the pointer.

A GCA must have a mechanism for *moving* the pointer to the left or right along the data array so that different data qubits can be addressed. This is provided by the following basic instructions of SPA.

$$\boxed{\begin{array}{l}\texttt{P}_\texttt{L} : \big(\rho, i\big) \mapsto \left(\rho, i-1\right) \quad , \quad 2 \leq i \leq N \\ \\ \texttt{P}_\texttt{R} : \big(\rho, i\big) \mapsto \big(\rho, i+1\big) \quad , \quad 1 \leq i \leq N-1\end{array}} \tag{3.3}$$

Having positioned the pointer over a specific data qubit, we want to have a means of applying a single-qubit unitary gate U to that qubit. So we provide the following basic instruction.

$$\boxed{\texttt{G}^U : \big(\rho, i\big) \mapsto \big(U^{(i)} \rho\, U^{(i)\dagger}, i\big) \quad , \quad 1 \leq i \leq N} \tag{3.4}$$

In the above expression, $U^{(i)}$ is defined (in the computational basis of the data qubits) as

$$U^{(i)} \equiv \big(I^{\otimes i-1}\big) \otimes U \otimes \big(I^{\otimes N-i}\big)\,. \tag{3.5}$$

For the GCA to be capable of universal quantum computing, it will suffice to implement $\texttt{G}^U$ for a set of unitary gates U that is universal for 1-qubit gates.[1]

We also need a means of performing a nontrivial (entangling) 2-qubit gate to the data qubits. I will use the controlled-Z gate, where

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

For SPA, we will specify a controlled-Z operation between the qubit currently addressed by the pointer, and the qubit immediately to its right. The controlled-Z and single-qubit gates together are universal for quantum computing. Some of the schemes presented in the literature give implementations of controlled-NOT gates, but the controlled-Z may be more appropriate because it has the property that it is symmetric with respect to assignment of the target and control qubits, which is convenient for systems exhibiting translational invariance (like GCA). We have the following basic instruction.

$$\boxed{\texttt{CZ} : \big(\rho, i\big) \mapsto \big(cZ^{(i,i+1)} \rho \, cZ^{(i,i+1)\dagger}, i\big) \quad , \quad 1 \le i \le N-1} \tag{3.6}$$

The operator $cZ^{(i,i+1)}$ is defined as

$$cZ^{(i,i+1)} \equiv \big(I^{\otimes i-1}\big) \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \otimes \big(I^{\otimes N-i-1}\big)\,. \tag{3.7}$$

Given a GCA architecture with implementations of the basic instructions $\{\texttt{P}_\texttt{L}, \texttt{P}_\texttt{R}, \texttt{CZ}, \{\texttt{G}^U\}\}$ for a set $\{U\}$ that is universal for 1-qubit gates, the system can efficiently simulate any quantum circuit. In this sense, SPA is a specification for a universal GCA.

In practice, we will also need a way to *load* the input and to *measure* the output. I will not build this into the framework, but assume that a given architecture provides some mechanisms for these operations. For example, one approach may be to exploit the unique geography of the cells at the ends of a finite lattice to successively load and unload bits

[1] That is, so that any 1-qubit gate can be efficiently approximated by a sequence of gates from that set.

from those cells and shift the information along the lattice. However this is accomplished, my focus will be on performing quantum computation on data that is already on the array. The basic instructions of SPA are summarized below.

Basic instructions of SPA

$$
\begin{aligned}
&\mathtt{P_L} : \big(\rho, i\big) \mapsto (\rho, i-1) \quad , \quad 2 \le i \le N \\
&\mathtt{P_R} : \big(\rho, i\big) \mapsto \big(\rho, i+1\big) \quad , \quad 1 \le i \le N-1 \\
&\mathtt{G}^U : \big(\rho, i\big) \mapsto \big(U^{(i)} \rho \, U^{(i)\dagger}, i\big) \quad 1 \le i \le N \\
&\mathtt{CZ} : \big(\rho, i\big) \mapsto \big(CZ^{(i,i+1)} \rho \, CZ^{(i,i+1)\dagger}, i\big) \quad , \quad 1 \le i \le N-1
\end{aligned}
\tag{3.8}
$$

The controlled-NOT gate is a very convenient tool for describing algorithms, and so it will be convenient to define "subroutines" of basic instructions that implement the quantum CNOT gate with the data qubit d_i playing the role of the control qubit, and either d_{i+1} or d_{i-1} playing the role of the target qubit. We will give these subroutines the labels $\mathtt{CNOT_R}$ and $\mathtt{CNOT_L}$ respectively, and these labels should be understood to simply be convenient shorthand expressions for the sequences of basic instructions they represent (i.e. $\mathtt{CNOT_R}$ and $\mathtt{CNOT_L}$ are not strictly part of the SPA language).

$$
\begin{aligned}
\mathtt{CNOT_R} &\equiv \mathtt{P_R} \, , \, \mathtt{G}^H \, , \, \mathtt{P_L} \, , \, \mathtt{CZ} \, , \, \mathtt{P_R} \, , \, \mathtt{G}^H \, , \, \mathtt{P_L} \\
\mathtt{CNOT_L} &\equiv \mathtt{P_L} \, , \, \mathtt{G}^H \, , \, \mathtt{CZ} \, , \, \mathtt{G}^H \, , \, \mathtt{P_R}
\end{aligned}
\tag{3.9}
$$

The instructions in the above expression are read from left to right, so that $\mathtt{CNOT_R}$ is executed first, then $\mathtt{P_R}$, etc. This is different than the algebraic convention in which products of unitary operators are temporally ordered from right to left, but it is a more natural convention for a programming language to read from left to right.

The SWAP operation ($|x_i\rangle|x_{i+1}\rangle \mapsto |x_{i+1}\rangle|x_i\rangle$) on the data qubits is so commonly used in GCA programs, that it is worth giving it a label as a shorthand for the sequence of instructions that implements it. It can be implemented using $\texttt{CNOT}_\texttt{R}$ and $\texttt{CNOT}_\texttt{L}$ as follows.

$$\boxed{\texttt{SWAP} \ \equiv\ \texttt{CNOT}_\texttt{R}\ ,\ \texttt{P}_\texttt{R}\ ,\ \texttt{CNOT}_\texttt{L}\ ,\ \texttt{P}_\texttt{L}\ ,\ \texttt{CNOT}_\texttt{R}.} \tag{3.10}$$

Again, `SWAP` is not a basic instruction, but a shorthand for the sequence of basic instructions that implements it.

When the pointer is at data qubit d_i, the basic instruction `SWAP` has the effect of the quantum SWAP operation on data qubits d_i and d_{i+1}, and the pointer remains in position i. After a `SWAP` sequence, the labels d_i and d_{i+1} are interchanged (so that the label d_i can be taken to refer to the state of the data qubit at *position* i within the array at all times). The `SWAP` sequence maps states as follows.

$$\boxed{\texttt{SWAP} : \big(\rho, i\big) \mapsto \big(\textsc{swap}^{(i,i+1)}\rho\, \textsc{swap}^{(i,i+1)^\dagger}, i\big)} \tag{3.11}$$

The $\textsc{swap}^{(i,i+1)}$ operator referred to above is defined as

$$\textsc{swap}^{(i,i+1)} = \big(I^{\otimes i-1}\big)\begin{pmatrix} 1&0&0&0\\ 0&0&1&0\\ 0&1&0&0\\ 0&0&0&1 \end{pmatrix}\big(I^{\otimes N-i-1}\big). \tag{3.12}$$

3.2.2 Implementations of SPA for some example architectures

I will now illustrate how the basic instructions of `SPA` can be implemented using the low-level hardware pulses of two specific GCA architectures.

3.2.2.1 1-D lattice with 3 species, anisotropic nearest-neighbour coupling

Lloyd's GCA architecture [Llo99] is based on a polymer $A, B, C, A, B, C, \ldots$ forming a repeating chain (1-dimensional lattice) of atoms of three distinct species. The following is a modified version of the scheme originally proposed by Lloyd.

Each atom in the polymer possesses an electron having a well defined transition frequency between the ground and first excited states. This frequency is ω^A, ω^B and ω^C for all atoms of species A, B and C respectively. The ground and first excited states of each atom represent the $|0\rangle$ and $|1\rangle$ states of a single qubit. I will refer to these underlying *lattice qubits* as being of *types* A, B and C, respectively (sometimes I will write "A-qubit" to refer to a lattice qubit of type A).

Adjacent lattice qubits are coupled by local interactions. The effect of the interactions is to shift the energy level, and thus the transition frequency, for each qubit as a function of the energy levels of its immediate neighbours. The transition frequency ω^B of a B-qubit then becomes ω^B_{01} when the A-qubit on its left is in the ground state $|0\rangle$ and the C-qubit on its right is in the excited state $|1\rangle$. The transition frequencies are assumed to be distinct for each combination of qubit-type, and basis states of the left and right neighbours. This means that it is possible to use an electromagnetic pulse to collectively target all (and only) lattice qubits of a given type having left and right neighbours in specified basis states. The effect of such pulses will extend linearly to quantum superpositions of states of the lattice qubits (and their neighbours). For a 1-qubit gate U, we define a lattice operation T^U_{lr}, where $T \in \{A, B, C\}$ and $l, r \in \{0, 1\}$. This operation has the effect of applying the gate U to every lattice qubit of type T that has left neighbour in the basis state $|l\rangle$ and the right neighbour in the basis state $|r\rangle$. For example, A^X_{01} has the effect of applying the X gate to every qubit of type A whose left neighbour is in the state $|0\rangle$ and whose right neighbour is in state $|1\rangle$. We will assume that the hardware allows us to directly implement pulses that implement $T^{V_i}_{lr}$ for a set $\{V_i\}$ of 1-qubit gates that is universal for 1-qubit gates. Then for *any* 1-qubit gate U, T^U_{lr} can be implemented by an appropriate sequence of pulses from $\{T^{V_i}_{lr}\}$. We will therefore sometimes speak loosely and refer to any T^U_{lr} as a *pulse*.

For brevity, we will allow l or r to take a blank symbol "$*$", which means that U will be applied regardless of the state of the corresponding neighbour. This could be achieved by applying two pulses, one for each of the two possible states of the neighbour. For example,

$$A^X_{1*} \equiv A^X_{10}, A^X_{11}. \tag{3.13}$$

Alternatively, a single pulse might be applied that covers both of the required transition frequencies.

In practice, the lattice will have finite length, and we would ultimately have to account for how to treat the qubits at each end of the lattice. For the present discussion, I will ignore this issue. I will suppose that the data is encoded in the interior of some sufficiently long lattice so that we can analyze the effect of the control pulses as though the lattice were infinite in extent. In [Llo99] it is proposed that the unique geography of the qubits at the end of the lattice could be exploited to provide a way to load and measure qubits onto and from the lattice. Another way in which unique behaviour at the ends of a lattice might be exploited is discussed in Section 3.2.3.

The data qubits are encoded in the lattice qubits of type A, and the position of the pointer is encoded in the lattice qubits of types B and C. For the data array in a computational basis state $\rho = |x_1 x_2 \ldots x_N\rangle\langle x_1 x_2 \ldots x_N|$, and the pointer at position i, this situation is encoded as follows (when illustrating a basis state of a lattice in a diagram, I will omit the ket symbols for brevity).

0	0	0	x_1	$\delta_{1,i}$	$\delta_{1,i}$	x_2	$\delta_{2,i}$	$\delta_{2,i}$	$\cdots$	x_N	$\delta_{N,i}$	$\delta_{N,i}$	0	0	0
A	B	C	A	B	C	A	B	C	$\cdots$	A	B	C	A	B	C

(3.14)

In the above expression, the Dirac delta function $\delta_{i,j}$ equals 1 if $i = j$, and equals 0 otherwise. So each position of the data array occupies three adjacent qubits of types A, B, C in the lattice, with the data qubits encoded in the A qubits. All the B, C pairs of qubits are in the state $|0\rangle|0\rangle$, except for the B, C pair to the right of the A qubit containing the data qubit d_i (the location of the pointer), which is $|1\rangle|1\rangle$. For example, when the pointer is at position $i = 1$ in the data array, the lattice state is illustrated below.

0	0	0	x_1	1	1	x_2	0	0	$\cdots$	x_N	0	0	0	0	0
A	B	C	A	B	C	A	B	C	$\cdots$	A	B	C	A	B	C

(3.15)

The pulse sequence $B^X_{1*}, A^X_{*1}, B^X_{1*}$ has the effect (in the computational basis) of exchanging

the state of every A-qubit with its neighbouring B-qubit. Applying this sequence to the lattice state (3.15) above has the following effect.

0	0	0	1	x_1	1	0	x_2	0	$\cdots$	0	x_N	0	0	0	0
A	B	C	A	B	C	A	B	C	$\cdots$	A	B	C	A	B	C

I will write EXCH_{AB} as a shorthand for this pulse sequence[2]. That is,

$$\text{EXCH}_{AB} \equiv B^X_{1*}, A^X_{*1}, B^X_{1*}. \tag{3.16}$$

[2]Here I use EXCH as short for "exchange", to avoid confusion with a logical SWAP operation on the data qubits, that we will implement later.

Verifying correctness of pulse sequences

In some of the literature on GCA schemes, pulse sequences are claimed to have a particular effect, with no formal verification. It is desirable to have a systematic method for verifying the correctness of pulse sequences for implementing the basic instructions of `SPA` (if `SPA` is taken as a requirements specification for GCA, then it is desirable to have a systematic way to verify that a proposed scheme meets the requirements).

Most of the nontrivial pulse sequences we consider will be composed entirely of conditional X and Z pulses. First consider sequences consisting only of X pulses. Such sequences never generate nontrivial quantum superpositions, and have no effect on the relative phase for states already in superposition. In these cases we can restrict our analysis to computational basis states of the lattice, which can be represented as tuples of binary variables (one variable representing the basis state of each qubit in the lattice).

Now suppose we wish to analyze a pulse sequence that consists of both X and Z pulses. These sequences still have the property that they don't generate quantum superpositions, but the Z pulses may affect the phases of lattice qubits. As before, suppose the lattice is in a computational basis state which we represent by a tuple of binary variables. Say we apply a Z pulse to the lattice qubit corresponding to the binary variable a_i. The effect is a phase of $(-1)^{a_i}$ on the state of the lattice (assuming, as we are, that the lattice is in a basis state). To keep track of the phase implied by the Z-pulses in the sequence, we use a binary variable ϕ, where the global phase will be $(-1)^{\phi}$. For example, if we apply some sequence of X-pulses, as well as a Z-pulse on a_i, and a controlled-Z pulse between a_j and a_{j+1}, then we have $\phi = a_i + a_j a_{j+1}$.

Typically a pulse sequence will be intended to have a net effect only on the states of the lattice qubits in the vicinity of the pointer. Because of the translational symmetry of the lattices under consideration, the effect of the pulse sequences is typically uniform across identically encoded segments. We can therefore usually restrict our analysis to a small section of the lattice, and make general conclusions by induction.

Proof of correctness for EXCH_{AB}**:**
The correctness of EXCH_{AB} is trivial to demonstrate, but I give a formal proof here to illustrate the technique. Let a_i, b_i, c_i be the binary values associated with the basis states for three adjacent lattice qubits in an (A, B, C)-triple. We want to show that EXCH_{AB} maps $(a_i\,,\, b_i\,,\, c_i)$ to $(b_i\,,\, a_i\,,\, c_i)$ for all i. We can analyze the effect of each pulse on the state $(a_i\,,\, b_i\,,\, c_i)$ algebraically. For example, $(1+a_i)$ denotes the negation (bit flip) of a_i and $(1+a_i)c_i$ denotes the conjunction of $(1+a_i)$ and c_i (arithmetic is modulo 2). Thus, for example, the pulse B^X_{01} transforms the state $(a_i\,,\, b_i\,,\, c_i)$ to $(a_i\,,\, b_i + (1+a_i)c_i\,,\, c_i)$. For the pulse sequence EXCH_{AB} we have the following.

$$\begin{aligned}(a_i\,,\, b_i\,,\, c_i) &\xrightarrow{B^X_{1*}} (a_i\,,\, b_i + a_i\,,\, c_i)\\ &\xmapsto{A^X_{*1}} (b_i + \cancel{2a_i}\,,\, b_i + a_i\,,\, c_i)\\ &\xmapsto{B^X_{1*}} (b_i\,,\, a_i + \cancel{2b_i}\,,\, c_i)\end{aligned}$$

This shows that on each (A, B, C)-triple, the sequence EXCH_{AB} has the desired effect. This implies the correctness of EXCH_{AB} on the entire lattice. □

Similarly the pulse sequence

$$\text{EXCH}_{BC} \equiv C^X_{1*}, B^X_{*1}, C^X_{1*} \tag{3.17}$$

has the effect of exchanging every B-qubit with its neighbouring C-qubit, and

$$\text{EXCH}_{CA} \equiv A^X_{1*}, C^X_{*1}, A^X_{1*} \tag{3.18}$$

exchanges the state of every C-qubit with its neighbouring A-qubit.

We can also derive a sequence of pulses that shifts each data qubit from the A-type lattice qubit to the B-type lattice qubit immediately to the right, and also moves the "11" encoding the position of the pointer from the BC-pair to the CA-pair immediately to the right. Specifically,

$$\text{RIGHT}_A \equiv B^X_{10}, B^X_{01}, A^X_{01}, A^X_{10} \tag{3.19}$$

has his effect, transforming (3.15) into the following state.

0	0	0	0	x_1	1	1	x_2	0	0	$\cdots$	x_N	0	0	0	0
A	B	C	A	B	C	A	B	C	A	$\cdots$	B	C	A	B	C

Proof of correctness for $\textsc{right}_A$**:**
As a slightly less trivial example illustrating the proof technique, we verify the correctness of $\textsc{right}_A$. Let x_i be the binary value associated with a basis state for the data qubit d_i. Let y_i be the binary value associated with the ith B and C-type lattice qubits (these will both be 1 if the pointer is at position i, and will both be 0 otherwise). Consider seven adjacent lattice qubits of types (C, A, B, C, A, B, C). We want to show that $\textsc{right}_A$ maps $(y_{i-1}, x_i, y_i, y_i, x_{i+1}, y_{i+1}, y_{i+1})$ to $(y_{i-1}, y_{i-1}, x_i, y_i, y_i, x_{i+1}, y_{i+1})$.

$$
\begin{aligned}
& (\underset{C}{y_{i-1}}, \underset{A}{x_i}, \underset{B}{y_i}, \underset{C}{y_i}, \underset{A}{x_{i+1}}, \underset{B}{y_{i+1}}, \underset{C}{y_{i+1}}) \\
\xmapsto{B_{10}^X}\; & (\underset{C}{y_{i-1}}, \underset{A}{x_i}, \underset{B}{y_i + x_i(1+y_i)}, \underset{C}{y_i}, \underset{A}{x_{i+1}}, \underset{B}{y_{i+1} + x_{i+1}(1+y_{i+1})}, \underset{C}{y_{i+1}}) \\
=\; & (\underset{C}{y_{i-1}}, \underset{A}{x_i}, \underset{B}{y_i + x_i + x_i y_i}, \underset{C}{y_i}, \underset{A}{x_{i+1}}, \underset{B}{y_{i+1} + x_{i+1} + x_{i+1} y_{i+1}}, \underset{C}{y_{i+1}}) \\
\xmapsto{B_{01}^X}\; & (\underset{C}{y_{i-1}}, \underset{A}{x_i}, \underset{B}{y_i + x_i + x_i y_i + (1+x_i) y_i}, \underset{C}{y_i}, \underset{A}{x_{i+1}}, \underset{B}{y_{i+1} + x_{i+1} + x_{i+1} y_{i+1} + (1+x_{i+1}) y_{i+1}}, \underset{C}{y_{i+1}}) \\
=\; & (\underset{C}{y_{i-1}}, \underset{A}{x_i}, \underset{B}{\cancel{y_i} + x_i + \cancel{x_i y_i} + \cancel{y_i} + \cancel{x_i y_i}}, \underset{C}{y_i}, \underset{A}{x_{i+1}}, \underset{B}{\cancel{y_{i+1}} + x_{i+1} + \cancel{x_{i+1} y_{i+1}} + \cancel{y_{i+1}} + \cancel{x_{i+1} y_{i+1}}}, \underset{C}{y_{i+1}}) \\
\xmapsto{A_{01}^X}\; & (\underset{C}{y_{i-1}}, \underset{A}{x_i + (1+y_{i-1}) x_i}, \underset{B}{x_i}, \underset{C}{y_i}, \underset{A}{x_{i+1} + (1+y_i) x_{i+1}}, \underset{B}{x_{i+1}}, \underset{C}{y_{i+1}}) \\
=\; & (\underset{C}{y_{i-1}}, \underset{A}{\cancel{x_i} + \cancel{x_i} + y_{i-1} x_i}, \underset{B}{x_i}, \underset{C}{y_i}, \underset{A}{\cancel{x_{i+1}} + \cancel{x_{i+1}} + y_i x_{i+1}}, \underset{B}{x_{i+1}}, \underset{C}{y_{i+1}}) \\
\xmapsto{A_{10}^X}\; & (\underset{C}{y_{i-1}}, \underset{A}{y_{i-1} x_i + y_{i-1}(1+x_i)}, \underset{B}{x_i}, \underset{C}{y_i}, \underset{A}{y_i x_{i+1} + y_i(1+x_{i+1})}, \underset{B}{x_{i+1}}, \underset{C}{y_{i+1}}) \\
=\; & (\underset{C}{y_{i-1}}, \underset{A}{\cancel{y_{i-1} x_i} + y_{i-1} + \cancel{y_{i-1} x_i}}, \underset{B}{x_i}, \underset{C}{y_i}, \underset{A}{\cancel{y_i x_{i+1}} + y_i + \cancel{y_i x_{i+1}}}, \underset{B}{x_{i+1}}, \underset{C}{y_{i+1}}) \qquad \square
\end{aligned}
$$

Similarly the pulse sequence

$$
\textsc{left}_A \equiv C_{10}^X, C_{01}^X, A_{01}^X, A_{10}^X \tag{3.20}
$$

shifts the data qubits from the B-type lattice qubits to the A-type lattice qubits immediately to the left. We have analogous pulse sequences for shifting the data qubits between the B- and C-type lattice qubits:

$$\text{RIGHT}_B \equiv C^X_{10}, C^X_{01}, B^X_{01}, B^X_{10}, \quad (3.21)$$

$$\text{LEFT}_B \equiv A^X_{10}, A^X_{01}, B^X_{01}, B^X_{10}, \quad (3.22)$$

$$\text{RIGHT}_C \equiv A^X_{10}, A^X_{01}, C^X_{01}, C^X_{10}, \quad (3.23)$$

$$\text{LEFT}_C \equiv B^X_{10}, B^X_{01}, C^X_{01}, C^X_{10}. \quad (3.24)$$

Notice that the EXCH, RIGHT, and LEFT sequences are not part the language SPA, but are merely convenient shorthand expressions for useful pulse sequences. We can implement SPA using these sequences as follows:

$$\mathtt{P_L} = \text{EXCH}_{AB}\ ,\ \text{EXCH}_{BC}\ ,\ \text{LEFT}_C\ ,\ \text{LEFT}_B, \quad (3.25)$$

$$\mathtt{P_R} = \text{EXCH}_{CA}\ ,\ \text{EXCH}_{BC}\ ,\ \text{RIGHT}_B\ ,\ \text{RIGHT}_C, \quad (3.26)$$

$$\mathtt{G}^U = A^U_{01}, \quad (3.27)$$

$$\mathtt{CZ} = B^X_{01}\ ,\ \text{EXCH}_{BC}\ ,\ A^Z_{1*}\ ,\ \text{EXCH}_{BC} B^X_{01}. \quad (3.28)$$

We give the proof of correctness for CZ below. Because it is very simple, it provides a good example illustrating the proof technique for pulse sequences that include Z operations.

Proof of correctness for CZ

Consider the segment of the lattice in the vicinity of the pointer, consisting of 6 lattice qubits of types (A, B, C, A, B, C). Suppose the basis state of the segment is initially represented by $(a_i, 1, 1, a_{i+1}, 0, 0)$. We want to show that the pulse sequence for CZ maps

$$(a_i, 1, 1, a_{i+1}, 0, 0)\ ,\ \phi = 0 \ \stackrel{\mathtt{CZ}}{\longmapsto}\ (a_i, 1, 1, a_{i+1}, 0, 0)\ ,\ \phi = a_i a_{i+1}.$$

That is, we want to show that the pulse sequence leaves the basis state unchanged,

and reverses the phase when $a_{i+1} = a_i = 1$. The analysis proceeds as follows:

$$\begin{aligned}
& \underset{A}{(a_i}\,,\underset{B}{1}\,,\underset{C}{1}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = 0 \\
\xrightarrow{B^X_{01}} \; & \underset{A}{(a_i}\,,\underset{B}{a_i}\,,\underset{C}{1}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = 0 \\
\xrightarrow{\text{EXCH}_{BC}} \; & \underset{A}{(a_i}\,,\underset{B}{1}\,,\underset{C}{a_i}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = 0 \\
\xrightarrow{A^Z_{1*}} \; & \underset{A}{(a_i}\,,\underset{B}{1}\,,\underset{C}{a_i}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = a_i a_{i+1} \\
\xrightarrow{\text{EXCH}_{BC}} \; & \underset{A}{(a_i}\,,\underset{B}{a_i}\,,\underset{C}{1}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = a_i a_{i+1} \\
\xrightarrow{B^X_{01}} \; & \underset{A}{(a_i}\,,\underset{B}{1}\,,\underset{C}{1}\,,\underset{A}{a_{i+1}}\,,\underset{B}{0}\,,\underset{C}{0)} \;,\; \phi = a_i a_{i+1} \qquad \square
\end{aligned}$$

It is worth taking a moment to consider how the physical system described in this section could be simulated by `SPA` (i.e. the opposite of what was done above), in order to show that the two schemes are equivalent. Suppose we have access to an array of data qubits and can perform the basic instructions of `SPA`. To simulate the anisotropic ABC-chain, we begin by assigning labels $A, B, C, A, B, C, \ldots$ to the data qubits. Each data qubit simulates one lattice qubit. Then suppose we want to simulate the global control pulse T^U_{lr} (where $T \in \{A, B, C\}$, $l, r \in \{0, 1\}$ and U a 1-qubit unitary gate). First note that with the `SPA` instructions `CZ` and $\mathtt{G}^U$ we can simulate any controlled-U gate between data qubits, and also any controlled-controlled-U operation between data qubits using a standard construction (see [BBC+95]). Each such simulation requires a constant number of `SPA` operations. So to simulate T^U_{lr} we just have to move the pointer along the data array, applying the appropriate controlled-controlled-U operation to every data qubit labeled T, conditioned on the states of its left and right neighbours. Since the pointer has to do this for each T-qubit sequentially, the overhead in time is linear in n, the size of the ABC-chain being simulated. So by simulating the ABC-chain with `SPA`, we only lose no fundamental computational power and only a small amount of computational efficiency.

Fact 3.2.1 *A lattice of qubits configured as an ABC-chain with anisotropic nearest-neighbour*

couplings and controlled by the global pulses described above is equivalent in power to the language ***SPA*** *up to at most a linear overhead in time and space.*

Other physical global control systems can be similarly shown to be roughly equivalent to `SPA`. Note that the linear overhead may be very significant when we consider fault-tolerance, and that is why we are motivated to define a more powerful model (called `MPA`) in Section 3.5.1.

3.2.2.2 1-D lattice with 2 species, isotropic coupling

Lloyd's architecture was adapted by Benjamin [Ben00], who showed that it suffices to restrict the hardware to a polymer with only two distinguishable species A and B. Moreover, the couplings can be such that a qubit in the lattice only feels the *net* effect of the states of its neighbours, and cannot distinguish the states of the left and right neighbours. The "field" of a given qubit in the lattice is defined as the number of immediate neighbours (to the left or right) in the state $|1\rangle$ minus the number in the state $|0\rangle$. Therefore, for computational basis states of the lattice, there are three possible values of the field for any given lattice qubit, 0,-2, and 2 (again ignoring the different behaviour of qubits at the ends of a finite lattice). The pulses that are allowed in this architecture are of the form

$$T_f^U \tag{3.29}$$

where $T \in \{A, B\}$ and $f \in \{-2, 0, 2\}$. For example, the pulse A_2^X applies the unitary X to all qubits of type A that have both neighbours in the state $|1\rangle$ (giving a field of 2). The pulse B_0^H applies the Hadamard gate to all qubits of type B in the lattice that have one neighbour in each of the basis states $|0\rangle$ and $|1\rangle$.

In Benjamin's architecture, the encoding of the data qubits is more complicated than that described in the previous section. Each data qubit is now encoded by four adjacent lattice qubits. Furthermore, each data qubit will not occupy the same 4 lattice qubits during the course of a computation. The lattice will be significantly longer than the data array. Between any pair of data qubits (each taking 4 physical qubits in the lattice) there are 4 physical qubits of padding in the state $|0\rangle^{\otimes 4}$. The lattice contains a sufficient number

($O(N)$) of padding qubits in the state $|0\rangle$ to the left and right of the data array. The entire data array will move (as a single unit) back and forth along this long lattice during a computation. For the data array in a computational basis state $\rho = |x_1 x_2 \ldots x_N\rangle\langle x_1 x_2 \ldots x_N|$ (no pointer location shown), this will be implemented in the lattice as follows.

$\cdots$	0	$\bar{x}_1$	$\bar{x}_1$	x_1	x_1	0	0	0	0	$\bar{x}_2$	$\bar{x}_2$	x_2	x_2	0	0	0	0	$\cdots$	0	0	0	0	$\bar{x}_N$	$\bar{x}_N$	x_N	x_N	0	$\cdots$
$\cdots$	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	$\cdots$	A	B	A	B	A	B	A	$\cdots$

where $\bar{x}_i \equiv 1 + x_i$ ("+" denotes addition modulo 2). The basis state $|x_i\rangle$ of a data qubit is encoded in the lattice by the pattern $|\bar{x}_i\rangle|\bar{x}_i\rangle|x_i\rangle|x_i\rangle$. Specifically, the data qubit $|0\rangle$ is encoded by the pattern $|1\rangle|1\rangle|0\rangle|0\rangle$ and $|1\rangle$ is encoded by $|0\rangle|0\rangle|1\rangle|1\rangle$.

The pointer is represented in the same lattice by the 6-qubit basis state $|1\rangle|1\rangle|0\rangle|0\rangle|1\rangle|1\rangle$ (which Benjamin calls the "control unit", CU), that will only be present at one place in the lattice at any given time. When the data qubits are positioned over 4-tuples $ABAB$ in the lattice, the CU is positioned on a 6-tuple $BABABA$. When the data qubits are positioned as $BABA$ the CU is positioned as $ABABAB$. This relative positioning allows pulses to shuttle the CU along the lattice in a contrary direction to the data qubits, allowing movement of the pointer relative to the data. When positioned over a data qubit, the CU pattern behaves as an XOR-mask that changes the basis states of the lattice qubits encoding that data qubit. The scenario in which the pointer is at position i in the data array is represented as follows.

				$\vert\longleftarrow$		CU			$\longrightarrow\vert$								
				1	1	0	0	1	1								
$\cdots$	0	0	$\bar{x}_i$	x_i	$\bar{x}_i$	x_i	0	1	1	0	$\bar{x}_{i+1}$	$\bar{x}_{i+1}$	x_{i+1}	x_{i+1}	0	0	$\cdots$
$\cdots$	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	$\cdots$

The instructions of SPA are implemented in this architecture as follows.

$$\mathtt{P_L} = A_0^X, B_0^X, A_0^X, B_0^X \tag{3.30}$$

$$\mathtt{P_R} = B_0^X, A_0^X, B_0^X, A_0^X \tag{3.31}$$

$$\mathtt{G}^U = B_2^X, A_2^X, A_0^X, B_2^X A_0^X, B_2^U, A_0^X, B_2^X, A_0^X, A_2^X, B_2^X \tag{3.32}$$

$$\begin{aligned}\mathtt{CZ} = \mathtt{P_R}, &B_0^X, A_0^X, A_2^X, B_2^X, A_2^X, A_0^X, B_0^X, B_2^X, A_0^X, B_0^X, B_2^X, A_2^X, A_0^X, B_0^X, A_2^X, A_0^X, B_2^Z, \\ &A_0^X, A_2^X, B_0^X, A_0^X, A_2^X, B_2^X, B_0^X, A_0^X, B_2^X, B_0^X, A_0^X, A_2^X, B_2^X, A_2^X, A_0^X, B_0^X, \mathtt{P_L}\end{aligned} \tag{3.33}$$

In Appendix A, I give proofs of correctness for two of the pulse sequences. The others can be similarly verified.

3.2.3 Implementation on lattices with a distinguished site

In this section we explore what can be done with regular lattices having the added feature of a *distinguished site*. A distinguished site is a lattice qubit, or local grouping of lattice qubits, that responds differently to control pulses than all the other lattice qubits. Such a lattice can be used to implement a global control scheme with the added feature of *local control* only at the distinguished site. In practice, the distinguished site might be located at the end of a finite lattice, where the unique geography gives the terminal qubit unique properties. Alternatively, a distinguished site might be implemented by coupling a specific lattice qubit with some external qubit of another species.

To implement SPA on such a system, the pointer could be located at the fixed position of the distinguished site, and the data qubits moved back and forth under the fixed pointer. For example, to implement a $\mathtt{P_L}$ operation, the data qubits could all be moved to the right along the lattice. These systems have the advantage that the pointer position does not have to be logically encoded, and so the data qubits can be packed more tightly along the lattice. In the example system we describe below, the data qubits are encoded in one-to-one correspondence with the lattice qubits.

As an example, consider an anisotropic ABC-chain configured as a closed loop. Assume that the loop consists of an odd number of ABC-triples. Suppose an atom of a fourth type, D, is positioned adjacent to some ABC-triple selected (arbitrarily) to be the distinguished

triple. Suppose that we can apply pulses that affect all the ABC-triples uniformly. Also assume that we have pulses that can target *only* the distinguished ABC-triple, by making use of the effect of the proximity of the D atom.

For the present discussion, I will refer to the physical qubits of species A, B, C as "cells" of "types" A, B, C. When I talk about "moving a qubit to a cell", I am referring to a sequence of logical operations (usually nearest-neighbour SWAP operations) that permute the logical states of the physical qubits on the chain. We will need the following claim.

Claim 3.2.1 *For any pair of qubits initially occupying adjacent cells, there exists a sequence of pulses that has the effect of bringing those qubits into adjacent positions in the distinguished triple.*

> **Proof:** The pulse sequence $\text{EXCH}_{AB} \equiv B^X_{1*}, A^X_{*1}, B^X_{1*}$ exchanges the states of the qubits on the A-cells with those on the neighbouring B-cells. The sequence $(\text{EXCH}_{AC}, \text{EXCH}_{AB}, \text{EXCH}_{BC}, \text{EXCH}_{AB})$ has the effect of moving every qubit initially in an A-cell to the A-cell of the next ABC-triple to the left (counterclockwise). It also moves every qubit initially in a C-cell to the C-cell of the next ABC-triple to the right (clockwise). It leaves the qubits on the B-cells fixed. By permuting the labels of the species we have similar sequences for moving the qubits in the A- and B-cells, while keeping the qubits in the C-cells fixed. Suppose we have a pair of qubits (b_i, c_i) in adjacent B- and C-cells, that we wish to move into adjacent positions in the distinguished triple. First we apply the sequence that moves the qubits in the A- and B-cells (keeping the qubits in the C-cells fixed) until b_i is in the B-cell of the distinguished triple. Then we apply the sequence that moves the qubits in the A- and C-cells (keeping the qubits in the B-cells fixed), until c_i is in the C-cell of the distinguished triple (beside b_i). Similar procedures will bring any pairs of adjacent qubits into adjacent positions in the distinguished triple. □

From the claim, it follows that we can implement a nearest-neighbour SWAP operation between any pair of adjacent qubits on the lattice. First we move the pair to the distinguished triple, and apply a sequence of pulses to implement the SWAP operation only on

those qubits in the distinguished triple. Then move all the qubits back to their corresponding original positions (respecting the swapped pair). Now it follows that we can implement an arbitrary permutation of the states of the individual qubits on the lattice (by a suitable sequence of nearest-neighbour transpositions). In particular, this allows us to implement a cyclic rotation of the lattice, so that we can implement the $\mathtt{P_L}$ and $\mathtt{P_R}$ operations of SPA. The other operations can be implemented directly, by applying pulses that target only the distinguished triple.

3.2.4 SPA programs to simulate quantum circuits

The utility of the language SPA is that we can use it to write programs for simulating quantum circuits with a GCA, independent of the details of the implementation of a particular GCA system. For any specific physical architecture implementing SPA, these programs can be "compiled" to give the specific pulse sequence to run the circuit simulation.

When presented with a quantum circuit diagram, the circuit can be first be rewritten to make translation into an SPA program straightforward. This rewriting results in an equivalent, sequential, nearest-neighbour circuit that uses only single-qubit, controlled-NOT, controlled-Z and SWAP gates (we can allow controlled-NOT and SWAP because in Section 3.2.1 we have already seen subroutines for implementing these operations using the basic instructions of SPA). The first step in rewriting is to replace any multi-qubit gates other than controlled-Z, controlled-NOT and SWAP with an equivalent subcircuit consisting of single-qubit, controlled-Z, controlled-NOT and SWAP gates. Consider a controlled-U gate for any one-qubit gate U. Since we can write any such U as $U = e^{i\alpha}AXBXC$ for single-qubit gates A, B, C satisfying $ABC = I$, this gives a method for writing the controlled-U in terms of single qubit gates and controlled-NOT gates. For circuits containing gates controlled by multiple qubits, we can use techniques described in [BBC+95] to rewrite these.

The second step in rewriting is to take the circuit resulting from the first rewriting step, and serialize it. That is, whenever multiple gates in the circuit are applied in parallel, separate them into discrete time-steps. The final step is to write an equivalent nearest-neighbour circuit. This is done as follows. Wherever a two-qubit gate appears between two

nonadjacent qubits, we use a sequence of SWAP gates to bring the control and target qubits into adjacent positions. Then apply the required two-qubit gate, and finally perform the reverse sequence of SWAP operations to move the qubits back to their original positions. After the rewriting, the programmer could optionally look for ways to optimize the resulting circuit (without violating the serial or nearest-neighbour constraints).

The rewriting sequence will be illustrated for the circuit fragment shown in Figure 3.1.

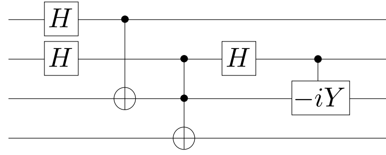


Figure 3.1: A fragment of a circuit to be simulated by `SPA`.

In the circuit of Figure 3.1, $-iY$ denotes the Pauli Y operation multiplied by the scalar $-i$. In the first step of rewriting, the circuit is replaced with an equivalent one consisting only of single-qubit, controlled-NOT and controlled-Z gates (note that $-iY = XZ$). The resulting circuit is shown in Figure 3.2.

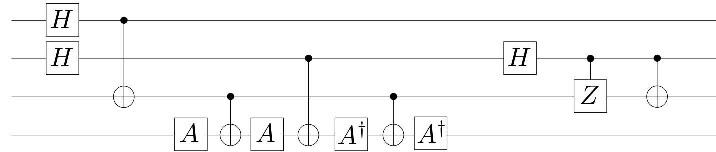


Figure 3.2: A circuit equivalent to Figure 3.1, using only single-qubit, controlled-NOT and controlled-Z gates. Above, $A = \cos(\pi/8)|0\rangle\langle 0| + \sin(\pi/8)|0\rangle\langle 1| + \cos(\pi/8)|1\rangle\langle 0| - \sin(\pi/8)|1\rangle\langle 1|$.

Next, the circuit in Figure 3.2 is serialized, by temporally separating any gates being applied in parallel. The result is shown in Figure 3.3. Then a nearest-neighbour version of the circuit is constructed, resulting in the circuit shown in Figure 3.4.

Finally, we can perform some optimizations on the circuit of Figure 3.4. The first observation is that in two places we have used the straightforward recipe using SWAP gates to perform a distance-2 controlled-NOT gate (a controlled-NOT where the control qubit is two

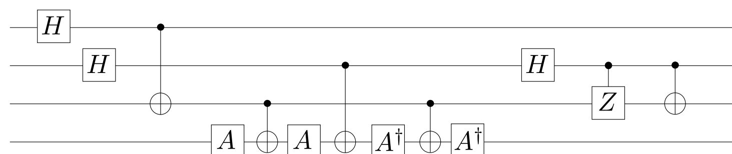


Figure 3.3: The circuit in Figure 3.2 rewritten with no gates acting in parallel.

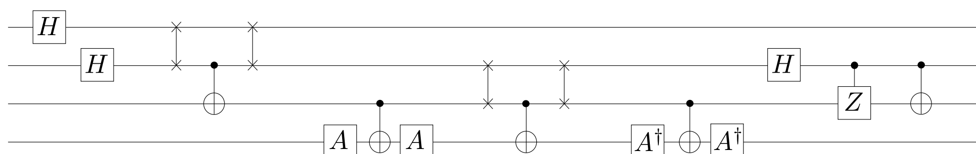


Figure 3.4: A nearest-neighbour version of the circuit shown in Figure 3.3.

qubits away from the target; that is, the control and target have one qubit in between them). The SWAP gate can be represented by 3 nearest-neighbour controlled-NOT gates, and this is analogous to how we have implemented the `SWAP` instruction in `SPA` (see Section 3.2.1). So the distance-2 controlled-NOT constructions use a total of 7 nearest-neighbour controlled-NOT gates. In general, a distance-k controlled-NOT gate implemented by this approach requires $6(k-1)+1$ nearest-neighbour controlled-NOT gates. There is a more efficient construction that implements a distance-k controlled-NOT gate using $4(k-1)$ nearest-neighbour controlled-NOT gates. This construction is illustrated in Figure 3.5 for $k=4$.

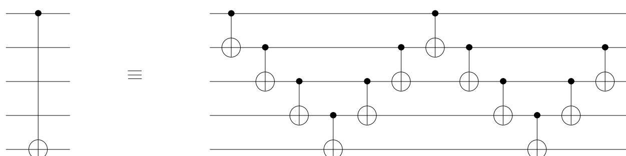


Figure 3.5: A construction for implementing a distance-4 CNOT gate using 12 nearest-neighbour CNOT gates. This construction generalizes naturally for distance-k CNOT gates.

Suppose we want to implement a distance-k CNOT gate on a GCA, with d_i as the control

qubit and d_{i+k} as the target qubit (that is, the target qubit is k positions to the right of the control qubit). Assuming the pointer is initially at data qubit d_i, the following SPA program accomplishes this.

SPA program to implement a distance-k CNOT to the right

1. $\mathtt{CNOT_R}$
2. do $\{\mathtt{P_R}, \mathtt{CNOT_R}\}$ $k-1$ times
3. do $\{\mathtt{P_L}, \mathtt{CNOT_R}\}$ $k-1$ times
4. do $\{\mathtt{P_R}, \mathtt{CNOT_R}\}$ $k-1$ times
5. do $\{\mathtt{P_L}, \mathtt{CNOT_R}\}$ $k-2$ times
6. $\mathtt{P_L}$.

We can write $\mathtt{CNOT_R}(k)$ as a shorthand for the above subroutine in SPA programs. We have a similar program for doing distance-k controlled-NOT gates to the left (which we can denote by the shorthand $\mathtt{CNOT_L}(k)$).

SPA program to implement a distance-k CNOT to the left

1. $\mathtt{CNOT_L}$
2. do $\{\mathtt{P_L}, \mathtt{CNOT_L}\}$ $k-1$ times
3. do $\{\mathtt{P_R}, \mathtt{CNOT_L}\}$ $k-1$ times
4. do $\{\mathtt{P_L}, \mathtt{CNOT_L}\}$ $k-1$ times
5. do $\{\mathtt{P_R}, \mathtt{CNOT_L}\}$ $k-2$ times
6. $\mathtt{P_R}$.

After replacing the distance-2 controlled-NOT implementations, the resulting circuit appears as in Figure 3.6.

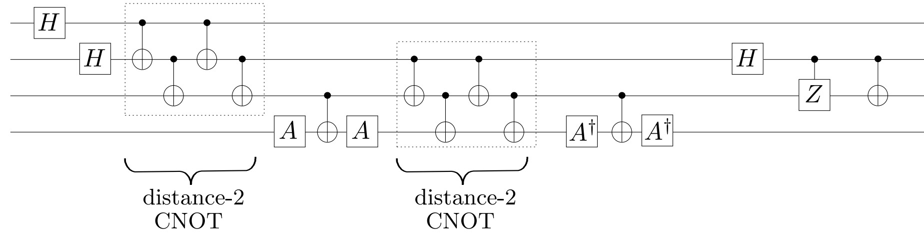


Figure 3.6: Some optimizations applied to the circuit constructed in Figure 3.4.

Now we can write a program in SPA that directly simulates the circuit of Figure 3.6 as follows. We assume the active pointer is initially addressing data qubit d_1. At the end of the following program the pointer will be positioned at data qubit d_3.

SPA program to simulate circuit in Figure 3.6

$\mathtt{G}^H$, $\mathtt{P_R}$, $\mathtt{G}^H$, $\mathtt{P_L}$, $\mathtt{CNOT_R}(2)$, $\mathtt{P_R}$, $\mathtt{P_R}$, $\mathtt{P_R}$, $\mathtt{G}_A$, $\mathtt{P_L}$, $\mathtt{CNOT_R}$, $\mathtt{P_R}$, $\mathtt{G}^A$, $\mathtt{P_L}$, $\mathtt{P_L}$, $\mathtt{CNOT_R}(2)$, $\mathtt{P_R}$, $\mathtt{P_R}$, $\mathtt{G}^{A^\dagger}$, $\mathtt{P_L}$, $\mathtt{CNOT_R}$, $\mathtt{P_R}$, $\mathtt{G}^{A^\dagger}$, $\mathtt{P_L}$, $\mathtt{P_L}$, $\mathtt{G}^H$, CZ , $\mathtt{CNOT_R}$.

Using the rewriting rules described above, it should be straightforward to create a naive compiler for generating SPA programs to simulate quantum circuits (given some description of the circuits). I described the optimization of the distance-k CNOT implementations as an example to illustrate that creating a *good* compiler may be harder. Such a compiler should ideally be capable of applying optimization techniques to generate reasonably efficient SPA programs.

3.2.5 GCA, QCA, and error correction

I have chosen terminology to distinguish GCA from other models of quantum cellular automata, and so we should take a moment to discuss the relationship between these models. Often the term QCA is used to refer collectively to all such models (including GCA), but there is an important distinction. In a true QCA model there is no external control (global or local); that is, the machines are “autonomous”. They are typically organized as regular chains or lattices of qubits linked by local couplings as in GCA, but for QCA there is a fixed

transition rule that specifies the next state for a lattice qubit, based on the current states of its neighbours.

Given a GCA and a program (that is, a fixed sequence of control pulses), the behaviour of the system can be simulated by a QCA. The basic approach is to encode the program in some of the lattice qubits. A transition rule is defined that has the effect of simulating the sequence of global pulses (the program) on the data. In practice, this could be quite complicated. The QCA would have to keep some kind of clock to keep track of the current time-step, and the transition function will have to effectively check the clock, and apply the correct operation at each time-step.

We can simulate the behaviour of any given (1-dimensional) QCA with an `SPA` program by simply moving the pointer back and forth across the array, visiting every data qubit. For each data qubit, we use the (fixed) sequence of basic instructions that will simulate the effect of the QCA transition function on that data qubit. To simulate the behaviour of a higher dimensional QCA model will be more involved, but is only matter of working out the details, as we know `SPA` is capable of universal quantum computation.

It is useful to view GCA as a QCA with additional power. Whereas a QCA has a spacially and temporally uniform transition function, the global control pulses of a GCA are spacially (but not temporally) uniform transition functions. That is, a GCA is like a QCA with a set of possible transition functions, and at each step we get to choose which transition function to apply. Viewing things in this way, it seems that performing reliable computation with a QCA should be at least as hard as performing reliable computation with a GCA.

Given a method for doing reliable computation with a GCA, suppose we want to simulate this with a QCA. As we mentioned above, the basic approach is to encode the program and some mechanism to act as a clock in the state of the QCA. Now, however, we have to account for the fact that the noise can affect the state of the clock. For a GCA we would typically assume that the noise only affects the states of the data qubits in the array, and that the choices we make about which pulses to apply at each time-step (and the ordering of the time-steps) are immune to errors (or at least can be implemented reliably in the classical computer controlling the pulse application). When we simulate these mechanisms by error-prone QCA states, the reliability may be lost. To reliably implement a QCA performing

a simulation like this, we would need to have some means of protecting the state of the clock. Furthermore, this would have to be accomplished using the spacially and temporally symmetric transition function of the QCA.

Gács has given techniques for performing fault-tolerant (classical) computation with classical cellular automata [Gac86]. For reliable QCA models, we might look to expand or modify the techniques in [Gac86]. These techniques are very sophisticated and make use of cellular automata in which each cell has a very large number of states. It seems likely that one should be able to exploit global control and devise simpler solutions for GCA.

3.3 Two approaches to error correction for GCA

As with any computing model, if one wishes to implement a GCA in hardware, the possibility of errors must be considered. A rich theory of quantum error correction has been developed, and quantum error correcting codes for quantum circuits have been designed. Using these codes, arbitrarily long computations in the quantum circuit model can in theory be performed reliably under physically realistic error models. It is not clear how these quantum error correcting codes can be applied to GCA, however. Errors affecting the states of the physical qubits in the underlying lattice may not map to errors in the encoded data qubits in a natural way. Physical errors may also affect the integrity of the pointer in a way that is difficult to control.

Various techniques have been proposed for performing error correction for different GCA models. In terms of the framework described in Section 3.2.1, it is convenient to divide error correction techniques for GCA into two distinct categories.

The first category consists of techniques for *implementation-level error correction* that make the GCA more robust. In terms of the language described in Section 3.2.1, these techniques are implemented at the lowest level, to implement the basic instructions more robustly in the presence of errors that may occur on individual lattice qubits. A shortcoming of these techniques is that scaling the codes generally requires redesigning hardware, as we will see later.

A second category of error correction techniques for GCA is *data-level error correction*,

which protect against errors at the level of the data qubits encoded on the lattice. These techniques are implemented by programs that run on the fixed GCA hardware. The language SPA provides a convenient tool for describing such techniques. Existing quantum error correcting codes for quantum circuits provide candidates for data-level error correction in GCA. These codes have the advantages that they are well studied and are inherently scalable (e.g. through concatenation). This approach has the shortcoming that it will only operate at the level of abstraction of the data qubits and the pointer. If errors at the implementation-level corrupt the encoding, leading to a physical lattice state that is not a valid encoding of a data array and pointer state, then data-level error correction will not function properly. We examine data-level error correction in Section 3.6.

Recall our discussion of lattices with a distinguished site, from Section 3.2.3. Implementation-level error correction for these systems has the advantage that we don't have to worry about error correcting the pointer, since it is realized by physical means, and not a logical encoding. A drawback of these schemes is that a single distinguished site will not provide an opportunity for parallelism. As I will discuss further in Section 3.5, parallelism is vital if we want to achieve fault-tolerance. Parallelism could be achieved by using multiple distinguished sites, but the main motivation for studying globally controlled arrays is precisely the difficulty of implementing physical systems with many distinguished sites where local control is available.

It is unclear how fully fault-tolerant GCA computing can be achieved in practice. By the observations in the preceding paragraphs, it seems that a mixture of implementation-level and data-level error correction techniques will be necessary. Some work (e.g [Kay05], [Kay07]) has been done on hybrid techniques for specific GCA architectures (e.g. by using implementation-level approaches to correct the pointer state, and data-level approaches for the data qubits), but it is not clear that these techniques will satisfy the two requirements of (1) being fully scalable, and (2) not being restricted to an artificial error model (i.e. can deal with the low-level physical qubit errors in the underlying lattice).

In Section 3.4 I will explore some techniques for implementation-level error correction and in Section 3.6 I will consider data-level error correction for GCA.

3.4 Implementation-level error correction

3.4.1 Dissipative pulses—removing unwanted entropy

Errors add entropy to a quantum system. Correction of these errors requires some mechanism for removing this entropy. In the circuit model this can be achieved by performing carefully designed "syndrome measurements", or by providing ancillary qubits that can be initialized to a fixed state (usually $|0\rangle$) when required. For GCA, the second approach is preferable (the ability to perform localized measurements of lattice qubits may not be provided by a global control scheme). To allow a GCA to perform error correction, we assume the hardware provides a "dissipative pulse" that performs the RESET operation $|x\rangle \mapsto |0\rangle$. Previous schemes ([Llo99], [BBK04]) have made use of a pulse that forces all lattice qubits of a given species to the ground state $|0\rangle$, conditioned on the states of their neighbours. I will show that, for a particular class of implementation-level approaches, it suffices to use an unconditional dissipative pulse that forces all lattice qubits of a given species to $|0\rangle$, regardless of the states of their neighbours (i.e. the RESET operation). It may be that the unconditional dissipative pulse is easier to implement in practice, for some schemes.

For implementations using a polymer and electromagnetic pulses as described in Section 3.2.2, the following is a standard approach to implementing a RESET operation. The idea is for each lattice qubit to have a second excited state $|2\rangle$ that rapidly decays to the ground state $|0\rangle$. Then applying a pulse of a suitable frequency to excite qubits of a given species to the state $|2\rangle$ will have the effect of setting these qubits to $|0\rangle$. The problem of resetting qubits to the state $|0\rangle$ by algorithmic techniques is the subject of Chapter 4.

One has to take great care when applying the RESET operation in GCA schemes. By conservation laws in physics, when energy (or entropy) is dissipated from a lattice qubit, it will be absorbed by the environment. The RESET operation acts on the qubit and the environment as follows:

$$|0\rangle|E\rangle \xmapsto{\text{RESET}} |0\rangle|E_0\rangle \tag{3.34}$$

$$|1\rangle|E\rangle \xmapsto{\text{RESET}} |0\rangle|E_1\rangle. \tag{3.35}$$

In other words, the final state of the environment depends on whether the state was reset

from $|1\rangle$ or left unchanged in $|0\rangle$. Suppose we have some multiple-qubit state

$$|x\rangle = \alpha|0\rangle|x_0\rangle + \beta|1\rangle|x_1\rangle.$$

Ideally, we might hope that if we applied a RESET to the first qubit of $|x\rangle$, the result would be

$$\alpha|0\rangle|x_0\rangle + \beta|0\rangle|x_1\rangle = |0\rangle \left(\alpha|x_0\rangle + \beta|x_1\rangle\right).$$

Unfortunately, because of the interaction with the environment, the result will actually be

$$\begin{aligned} |x\rangle|E\rangle \quad &\xmapsto{\text{RESET}} \alpha|0\rangle|x_0\rangle|E_0\rangle + \beta|0\rangle|x_1\rangle|E_1\rangle \\ &= |0\rangle \left(\alpha|x_0\rangle|E_0\rangle + \beta|x_1\rangle|E_1\rangle\right) \end{aligned}$$

and the state of the remaining part of $|x\rangle$ becomes coupled with the environment.

When we apply the RESET operation during error correction, we can avoid the above pitfall by taking care only to reset states containing the *error syndrome*, and not resetting any state containing information about the logical state of the qubits we are trying to protect from errors. Suppose we use an error correcting code, and encode the logical state $|0\rangle$ by the codeword $|C_0\rangle$ and the state $|1\rangle$ by the codeword $|C_1\rangle$. Suppose some error ε transforms $|C_0\rangle$ to $|C_0'\rangle$ and transforms $|C_1\rangle$ to $|C_1'\rangle$. A *syndrome computation* performs

$$|C_i'\rangle|0\rangle \mapsto |C_i'\rangle|S\rangle,$$

where the syndrome S contains enough information to identify the error ε. Note that S is independent of i; that is, the syndrome does not contain any information about the original codeword. After the syndrome computation, an error correction operation is controlled by $|S\rangle$ to reverse the effect of ε.

Suppose we initially have some encoded state $\alpha|C_0\rangle + \beta|C_1\rangle$ and some error occurs, transforming the state to $\alpha|C_0'\rangle + \beta|C_1'\rangle$. The syndrome computation maps

$$\left(\alpha|C_0'\rangle + \beta|C_1'\rangle\right)|0\rangle \mapsto \left(\alpha|C_0'\rangle + \beta|C_1'\rangle\right)|S\rangle.$$

Then the error correction step is performed, mapping

$$\left(\alpha|C_0'\rangle + \beta|C_1'\rangle\right)|S\rangle \mapsto \left(\alpha|C_0\rangle + \beta|C_1\rangle\right)|S\rangle.$$

Finally, we reset the syndrome, so that the ancillary register can be used again for a syndrome computation the next time we do error correction. Accounting for the interaction with the environment, the RESET operation performs

$$(\alpha|C_0\rangle + \beta|C_1\rangle)\,|S\rangle|E\rangle \xmapsto{\text{RESET}} (\alpha|C_0\rangle + \beta|C_1\rangle)\,|0\rangle|E'\rangle.$$

The environment is modified by the erasure of the syndrome[3], but the codeword is not coupled with the environment, and the quantum information remains intact.

I will denote a reset pulse targeting all lattice qubits of type T by T^{RESET}.

3.4.2 A bit-flip code for a GCA memory

When we begin to study error correction for GCA models, it is convenient to initially consider a GCA that does nothing; that is, a "GCA memory", having no pointer. A GCA memory can be made more reliable by using an error correcting code and some sequence of control pulses to perform error correction. Later, in Section 3.4.6 we will discuss methods for reintroducing the pointer, to turn a reliable GCA memory into a reliable implementation of SPA.

In this section, we consider the error model in which each lattice qubit independently suffers a "bit flip" with some fixed probability p. A bit flip is equivalent to a NOT gate on the lattice qubit. A simple code that can help protect qubits against bit-flip errors is a *three-qubit code*, for which the logical qubit state $|d_i\rangle = \alpha|0\rangle + \beta|1\rangle$ is encoded as the three-qubit state $|\hat{d_i}\rangle = \alpha|000\rangle + \beta|111\rangle$.

Consider Lloyd's GCA implementation [Llo99], described in Section 3.2.2.1 (1-D lattice, three species, anisotropic coupling). The three-qubit code can be implemented on this lattice by encoding each data qubit redundantly in an (A, B, C)-triple. A data qubit in the basis state $|x_i\rangle$, $x_i \in \{0, 1\}$, is encoded in the lattice as follows.

$\cdots$	x_i	x_i	x_i	$\cdots$
$\cdots$	A	B	C	$\cdots$

[3] In Section 4.4.3 I discuss this problem in the context of algorithmic cooling using a heat bath.

The encoded basis states could be loaded onto the lattice directly or, assuming the unencoded data qubits are loaded onto the A-qubits, we could perform the encoding with pulses. A nearest-neighbour quantum circuit that performs the encoding for the three-qubit code is shown in Figure 3.7 below.

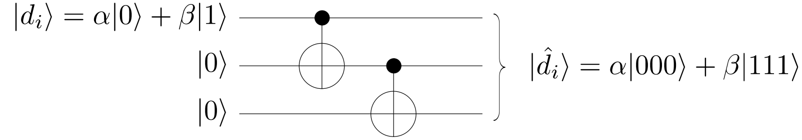


Figure 3.7: A nearest-neighbour quantum circuit implementing the encoding for the three-qubit code.

The circuit in Figure 3.7 is equivalent to the following sequence of pulses:

$$B^X_{1*}, C^X_{1*}.$$

Decoding is performed by running the circuit of Figure 3.7 backwards, which is equivalent to the following sequence of pulses:

$$C^X_{1*}, B^X_{1*}.$$

Error correction is accomplished by a sequence of pulses that performs a *majority-vote* in the computational basis across each (A, B, C)-triple. The majority-vote is accomplished in two stages: first the value of the majority (0 or 1) is computed into the B-qubit. At the same time, the parities of the bits originally in the pairs (A, B) and (B, C) are encoded into the A and C qubits. These parities form the error syndrome. A remarkable property of the 3-bit code is that the error syndrome can be computed "in-place", without the need for ancillary bits. Next the A and C qubits (containing the syndrome) are zeroed with dissipative pulses, and the majority value is encoded back into these qubits from the B-qubit. The pulse sequence is

$$A^X_{*1}\ ,\ C^X_{1*}\ ,\ B^X_{11}\ ,\ A^{\text{RESET}}\ ,\ C^{\text{RESET}}\ ,\ A^X_{*1}\ ,\ C^X_{1*}. \tag{3.36}$$

The above pulse sequence performs the error correction on all (A, B, C)-triples in parallel, and so performs the error correction for all the data qubits in the array simultaneously.

Proof of correctness for majority-vote sequence:
The majority value of three bits a_i, b_i, c_i can be expressed as $M_i = a_i b_i + b_i c_i + a_i c_i$. For an (A, B, C)-section of the lattice in the state (a_i, b_i, c_i), then, we want to show that the majority-vote sequence transforms this state to (M_i, M_i, M_i).

$$
\begin{aligned}
(a_i\,, b_i\,, c_i) &\xmapsto{A^X_{*1}} (a_i + b_i, b_i, c_i) \\
&\xmapsto{C^X_{1*}} (a_i + b_i, b_i, c_i + b_i) \\
&\xmapsto{B^X_{11}} (a_i + b_i, b_i + (a_i + b_i)(c_i + b_i), c_i + b_i) \\
&\quad = (a_i + b_i, M_i, c_i + b_i) \\
&\xmapsto{A^{\text{RESET}}} (0, M_i, c_i + b_i) \\
&\xmapsto{C^{\text{RESET}}} (0, M_i, 0) \\
&\xmapsto{A^X_{*1}} (M_i, M_i, 0) \\
&\xmapsto{C^X_{*1}} (M_i, M_i, M_i) \qquad \square
\end{aligned}
$$

Note that in the RESET steps, the values reset were $a_i + b_i$ and $c_i + b_i$. These *parity bits* form the error syndrome, and together they determine the location of a bit-flip error. These parities are the same whether the bit-flip error occurred on the codeword $|000\rangle$ or on the codeword $|111\rangle$, and so they carry no information about the identity of the codeword, and can be safely reset.

The above scheme will only correct a single bit-flip error within each block of three lattice qubits. This reduces the effective probability of bit-flip errors from $O(p)$ to $O(p^2)$ (but only between operations, as discussed above). This error rate could be further reduced by expanding the hardware. For example, an error rate of $O(p^4)$ could be achieved by a scheme analogous to that above using a lattice having 9 distinct species. Scaling the code in this way to achieve an effective bit-flip error-rate of ε would require increasing the number of distinguishable species polylogarithmically in $\frac{1}{\varepsilon}$.

In the following section, we expand the scheme described here to protect against bit-flip and phase-flip errors (which implies protection against arbitrary single-qubit errors).

3.4.3 A 9-qubit code for a GCA memory

In 1995 Peter Shor invented a 9-qubit code for quantum circuits by concatenating a 3-qubit code for correcting bit flips with a 3-qubit code for correcting phase-flip errors [Sho95]. A bit flip corresponds to a quantum X gate, and a phase flip corresponds to a Z gate. Because any single-qubit operation can be written as a linear combination of $\{I, X, Z, XZ\}$, Shor's 9-qubit code can correct an arbitrary single-qubit error within each codeword. Here we show how this code can be implemented in hardware for a GCA.

The 3-qubit phase-flip code is identical to the bit-flip code, except that data qubits are encoded in the Hadamard basis so that $\alpha|0\rangle+\beta|1\rangle$ is encoded as $\alpha|{+}{+}{+}\rangle+\beta|{-}{-}{-}\rangle$, where

$$|+\rangle = \tfrac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big) \tag{3.37}$$

$$|-\rangle = \tfrac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big). \tag{3.38}$$

In the Hadamard basis, phase flips resemble bit flips ($Z|+\rangle = |-\rangle$). So the phase-flip code can be implemented just as the bit-flip code, except with respect to the Hadamard basis.

Consider protecting a quantum memory (i.e. with no accommodation for a pointer) from phase-flip errors, using the ABC-chain described in Section 3.2.2.1. We encode the data qubits as

$\cdots$	y_i	y_i	y_i	$\cdots$
$\cdots$	A	B	C	$\cdots$

where $|y_i\rangle = H|x_i\rangle$. Then correction of phase-flip errors is accomplished by first applying a sequence of pulses that implements the Hadamard gate on every qubit in the lattice, and then performing majority-voting within the triplets as in the previous section. Finally we return to the Hadamard basis with more Hadamard pulses. The error correction sequence is

$$A^H_{**}, B^H_{**}, C^H_{**}, A^X_{*1}\ ,\ C^X_{1*}\ ,\ B^X_{11}\ ,\ A^{\text{RESET}}\ ,\ C^{\text{RESET}}\ ,\ A^X_{*1}\ ,\ C^X_{1*}, A^H_{**}, B^H_{**}, C^H_{**}. \tag{3.39}$$

For the 9-qubit code, we first encode every qubit using the phase-flip code, and then encode each of the three qubits in that code using the bit-flip code. The result is the following

encoding.

$$|0\rangle \equiv \frac{1}{2\sqrt{2}}\Big(\underset{q_1q_2q_3}{|000\rangle} + \underset{q_1q_2q_3}{|111\rangle}\Big)\Big(\underset{q_4q_5q_6}{|000\rangle} + \underset{q_4q_5q_6}{|111\rangle}\Big)\Big(\underset{q_7q_8q_9}{|000\rangle} + \underset{q_7q_8q_9}{|111\rangle}\Big) \quad (3.40)$$

$$|1\rangle \equiv \frac{1}{2\sqrt{2}}\Big(\underset{q_1q_2q_3}{|000\rangle} - \underset{q_1q_2q_3}{|111\rangle}\Big)\Big(\underset{q_4q_5q_6}{|000\rangle} - \underset{q_4q_5q_6}{|111\rangle}\Big)\Big(\underset{q_7q_8q_9}{|000\rangle} - \underset{q_7q_8q_9}{|111\rangle}\Big) \quad (3.41)$$

In the above expressions I have labeled the physical qubits with $\{q_1, q_2, \ldots, q_9\}$. A GCA hardware scheme convenient for implementing this code for a quantum memory consists of a 2-D lattice of 9 distinct species, arranged as follows.

A	B	C	A	B	C	A	B	C	$\cdots$	A	B	C
D	E	F	D	E	F	D	E	F	$\cdots$	D	E	F
G	H	J	G	H	J	G	H	J	$\cdots$	G	H	J

(3.42)

The qubits in the above lattice are coupled in rows and columns. That is, each A-qubit is coupled to the C to its left, the B to its right, and the D below. Pulses of the form $A_{l,r,b}$ are used for the A-qubits. Similarly, each D-qubit is coupled to the F-qubit to its left, the E to its right, the A above, and the G below. Pulses of the form $D_{l,r,a,b}$ are used for the D-qubits. Pulses for the lattice qubits of the other types are defined accordingly.

We arrange each 9-qubit codeword on a 3×3 section of the lattice as follows.

${q_1}_A$	${q_2}_B$	${q_3}_C$
${q_4}_D$	${q_5}_E$	${q_6}_F$
${q_7}_G$	${q_8}_H$	${q_9}_J$

(3.43)

If we assume that the hardware only allows us to directly load computational basis states onto the lattice, it will be necessary to perform the encoding for the 9-qubit code using an appropriate sequence of pulses. Assume the logical data qubits have been loaded onto the A-qubits of the lattice. A quantum circuit that performs encoding for the Shor code is shown in Figure 3.8.

The circuit in Figure 3.8 is equivalent to the following pulse sequence:

$$D^X_{**1*}\,,\; G^X_{**1*}\,,\; A^H_{***}\,,\; D^H_{****}\,,\; G^H_{***}\,,\; B^X_{1**}\,,\; C^X_{1**}\,,\; E^X_{1***}\,,\; F^X_{1***}\,,\; H^X_{1***}\,,\; J^X_{1***}. \quad (3.44)$$

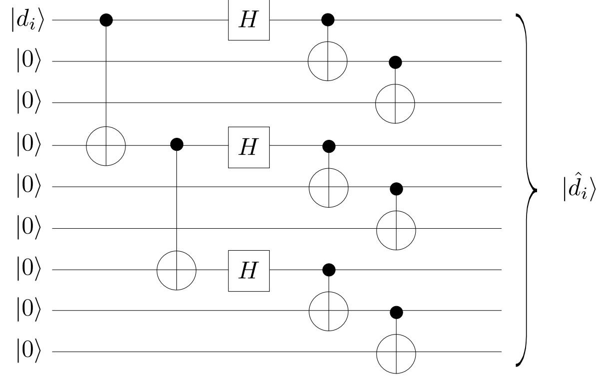


Figure 3.8: A quantum circuit performing encoding for the Shor code.

To decode, we use the reverse pulse sequence:

$$J^X_{1***}\,,\ H^X_{1***}\,,\ F^X_{1***}\,,\ E^X_{1***}\,,\ C^X_{1**}\,,\ B^X_{1**}\,,\ G^H_{***}\,,\ D^H_{****}\,,\ A^H_{***}\,,\ G^X_{**1*}\,,\ D^X_{**1*}. \tag{3.45}$$

We can correct a single bit-flip error, phase-flip error or both a bit-flip and a phase-flip error, as follows. First we perform the correction procedure for bit-flip errors within each of the three blocks of three qubits of the codeword. This is done using the majority-vote scheme discussed in section 3.4.2, across each row. The pulse sequence is

$$\begin{aligned} &A^X_{*1*}\,,\ C^X_{1**}\,,\ B^X_{11*}\,,\ A^{\text{RESET}}\,,\ C^{\text{RESET}}\,,\ A^X_{*1*}\,,\ C^X_{1**}\,,\\ &D^X_{*1**}\,,\ F^X_{1***}\,,\ E^X_{11**}\,,\ D^{\text{RESET}}\,,\ F^{\text{RESET}}\,,\ D^X_{*1**}\,,\ F^X_{1***}\,,\\ &G^X_{*1*}\,,\ J^X_{1**}\,,\ H^X_{11*}\,,\ G^{\text{RESET}}\,,\ J^{\text{RESET}}\,,\ G^X_{*1*}\,,\ J^X_{1**}. \end{aligned} \tag{3.46}$$

After this procedure, providing at most one bit-flip error had occurred in the codeword, we are left with a 9-qubit state of the form

$$\tfrac{1}{2\sqrt{2}}\left(\underset{q_1q_2q_3}{|0\,0\,0\rangle}\pm\underset{q_1q_2q_3}{|1\,1\,1\rangle}\right)\left(\underset{q_4q_5q_6}{|0\,0\,0\rangle}\pm\underset{q_4q_5q_6}{|1\,1\,1\rangle}\right)\left(\underset{q_7q_8q_9}{|0\,0\,0\rangle}\pm\underset{q_7q_8q_9}{|1\,1\,1\rangle}\right). \tag{3.47}$$

To correct a phase-flip error, we must implement a procedure for performing a majority-vote on the $\pm$ signs within the three blocks. This is accomplished by first implementing an appropriate transformation (similar to the Hadamard for the 3-qubit phase-flip code as

discussed above) and then doing the standard majority-vote procedure across the qubits in the resulting blocks of three bits. The transformation we need to implement is one that maps each block of three qubits in (3.47) according to

$$\begin{aligned} \tfrac{1}{\sqrt{2}}\left(|000\rangle + |111\rangle\right) &\mapsto |000\rangle \\ \tfrac{1}{\sqrt{2}}\left(|000\rangle - |111\rangle\right) &\mapsto |111\rangle. \end{aligned} \tag{3.48}$$

That is, we want to implement the transformation

$$\begin{aligned} \tfrac{1}{2\sqrt{2}} \left(|\underset{q_1q_2q_3}{0\,0\,0}\rangle + (-1)^{\phi_1} |\underset{q_1q_2q_3}{1\,1\,1}\rangle \right) \left(|\underset{q_4q_5q_6}{0\,0\,0}\rangle + (-1)^{\phi_2} |\underset{q_4q_5q_6}{1\,1\,1}\rangle \right) \left(|\underset{q_7q_8q_9}{0\,0\,0}\rangle + (-1)^{\phi_3} |\underset{q_7q_8q_9}{1\,1\,1}\rangle \right) \\ \mapsto |\underset{q_1}{\phi_1}\underset{q_2}{\phi_1}\underset{q_3}{\phi_1}\rangle |\underset{q_4}{\phi_2}\underset{q_5}{\phi_2}\underset{q_6}{\phi_2}\rangle |\underset{q_7}{\phi_3}\underset{q_8}{\phi_3}\underset{q_9}{\phi_3}\rangle. \end{aligned} \tag{3.49}$$

For example, suppose after performing the majority-vote to correct a bit-flip error, the 9-qubit state is

$$\tfrac{1}{2\sqrt{2}} \left(|\underset{q_1q_2q_3}{0\,0\,0}\rangle + |\underset{q_1q_2q_3}{1\,1\,1}\rangle \right) \left(|\underset{q_4q_5q_6}{0\,0\,0}\rangle - |\underset{q_4q_5q_6}{1\,1\,1}\rangle \right) \left(|\underset{q_7q_8q_9}{0\,0\,0}\rangle + |\underset{q_7q_8q_9}{1\,1\,1}\rangle \right) \tag{3.50}$$

(which would be the result if a phase-flip error had occurred on any of qubits 4,5 or 6). Then after applying the transformation (3.49), the resulting state is

$$\left(|\underset{q_1q_2q_3}{0\,0\,0}\rangle \right) \left(|\underset{q_4q_5q_6}{1\,1\,1}\rangle \right) \left(|\underset{q_7q_8q_9}{0\,0\,0}\rangle \right). \tag{3.51}$$

To correct the phase-flip error, we now do a majority-vote between the corresponding bits of each of the three blocks; that is, between bits 1,4,7, between bits 2,5,8, and between bits 3,6,9.[4] Since these triplets are lined up in columns in our lattice, this majority-vote can be accomplished using the following pulse sequence (compare with sequence (3.36) discussed earlier):

$$\begin{aligned} &A^X_{**1}\,,\ G^X_{**1}\,,\ D^X_{**11}\,,\ A^{\text{RESET}}\,,\ G^{\text{RESET}}\,,\ A^X_{**1}\,,\ G^X_{**1} \\ &B^X_{**1}\,,\ H^X_{**1}\,,\ E^X_{**11}\,,\ B^{\text{RESET}}\,,\ H^{\text{RESET}}\,,\ B^X_{**1}\,,\ H^X_{**1} \\ &C^X_{**1}\,,\ J^X_{**1}\,,\ F^X_{**11}\,,\ C^{\text{RESET}}\,,\ J^{\text{RESET}}\,,\ C^X_{**1}\,,\ J^X_{**1}. \end{aligned} \tag{3.52}$$

[4] Alternatively, we could just do the majority-vote across one of these triplets (say the first), and then copy the resulting value to the other qubits.

After applying this pulse sequence to the state (3.51), the result is the state

$$\left(\underset{q_1 q_2 q_3}{|0\,0\,0\rangle}\right)\left(\underset{q_4 q_5 q_6}{|0\,0\,0\rangle}\right)\left(\underset{q_7 q_8 q_9}{|0\,0\,0\rangle}\right). \tag{3.53}$$

Then we apply the inverse of the transformation (3.49) to get the corrected codeword

$$\tfrac{1}{2\sqrt{2}}\left(\underset{q_1 q_2 q_3}{|0\,0\,0\rangle}+\underset{q_1 q_2 q_3}{|1\,1\,1\rangle}\right)\left(\underset{q_4 q_5 q_6}{|0\,0\,0\rangle}+\underset{q_4 q_5 q_6}{|1\,1\,1\rangle}\right)\left(\underset{q_7 q_8 q_9}{|0\,0\,0\rangle}+\underset{q_7 q_8 q_9}{|1\,1\,1\rangle}\right). \tag{3.54}$$

It remains to show how to implement the transformation (3.49) on the lattice. The following pulse sequence accomplishes this.

$$\begin{aligned}
&C^X_{1**}\ ,\ B^X_{1**}\ ,\ A^H_{***}\ ,\ B^X_{1**}\ ,\ C^X_{1**}\ ,\\
&F^X_{1***}\ ,\ E^X_{1***}\ ,\ D^H_{****}\ ,\ E^X_{1***}\ ,\ F^X_{1***}\ ,\\
&J^X_{1**}\ ,\ H^X_{1**}\ ,\ G^H_{***}\ ,\ H^X_{1**}\ ,\ J^X_{1**}.
\end{aligned} \tag{3.55}$$

Proof of correctness for pulse sequence implementing (3.49):
Consider the first block of three qubits, q_1, q_2, q_3 (of types A, B, C respectively) in (3.49). We show that the pulse sequence C^X_{1**} , B^X_{1**} , A^H_{***} , B^X_{1**} , C^X_{1**} has the effect of transforming this block as

$$\tfrac{1}{2\sqrt{2}}\left(\underset{q_1 q_2 q_3}{|0\,0\,0\rangle}+(-1)^{\phi_1}\underset{q_1 q_2 q_3}{|1\,1\,1\rangle}\right)\mapsto \underset{q_1\ q_2\ q_3}{|\phi_1\phi_1\phi_1\rangle}.$$

The rest of the sequence implements the required transformations on the remaining two blocks analogously.

$$\begin{aligned}
\tfrac{1}{\sqrt{2}}\big(|000\rangle+(-1)^{\phi_1}|111\rangle\big) &\xmapsto{C^X_{1**}} \tfrac{1}{\sqrt{2}}\big(|000\rangle+(-1)^{\phi_1}|110\rangle\big)\\
&\xmapsto{B^X_{1**}} \tfrac{1}{2\sqrt{2}}\big(|000\rangle+(-1)^{\phi_1}|100\rangle\big)\\
&\quad= \tfrac{1}{\sqrt{2}}\big(|0\rangle+(-1)^{\phi_1}|1\rangle\big)|0\rangle|0\rangle\\
&\xmapsto{A^H_{***}} |\phi_1\rangle|0\rangle|0\rangle\\
&\xmapsto{B^X_{1**}} |\phi_1\rangle|\phi_1\rangle|0\rangle\\
&\xmapsto{C^X_{1**}} |\phi_1\rangle|\phi_1\rangle|\phi_1\rangle \qquad \square
\end{aligned}$$

3.4.4 A 1-dimensional implementation of the 9-qubit code

In the previous section, we arranged the 9 qubits into a 2-dimensional lattice. This spacial arrangement is convenient, given the structure of the 9-qubit code. The same scheme could be implemented on a 1-dimensional lattice structured as follows.

A	B	C	D	E	F	G	H	J	A	B	C	D	E	F	G	H	J	$\cdots$

(3.56)

The error correction operations can be implemented exactly as described in the previous section, except that now we will have to add suitable EXCH pulse sequences to permute the lattice qubits so that those that we previously had in columns move into adjacent positions when we want to correct phase-flip errors.

An important consideration is that because a 1-dimensional implementation requires many more EXCH pulses during the error correction operations, there will be a greater opportunity for errors to occur during the execution of these operations. However, the above scheme is not a fault-tolerant error correction procedure and so we have already implicitly assumed that the error correction operations (pulses) are not themselves prone to errors.

3.4.5 Scaling the 9-qubit code

To achieve better error rates in the quantum circuit model, codes can be concatenated at multiple levels by recursively encoding each of the qubits within a codeword. To concatenate an n-qubit codeword to k levels requires n^k qubits. Concatenation may not be a suitable approach for implementation-level error correction in a GCA, however. If we wanted a k-level concatenation of the 9-qubit code as implemented in the GCA described in the previous section, we would need a much larger lattice having 9^k distinguishable species. This may very quickly exceed the capabilities of available hardware.

In this section I propose an alternative method of scaling the 9-qubit code in finer gradations than can be achieved through concatenation. Specifically, I show how it could be generalized to a $(2n + 1)^2$-qubit code for any $n \geq 1$, requiring a lattice with $(2n + 1)^2$ distinguishable species. This code will be able to correct up to n single-qubit errors.

Letting $N = (2n + 1)$, the basic idea is to encode the logical basis states in N^2-qubit codewords according to

$$\begin{aligned}|0\rangle \equiv \tfrac{1}{2^{\frac{N}{2}}}\Big(|\underset{q_{1,1}}{0}\underset{q_{1,2}}{0}\dots\underset{q_{1,N}}{0}\rangle + |\underset{q_{1,1}}{1}\underset{q_{1,2}}{1}\dots\underset{q_{1,N}}{1}\rangle\Big)\Big(|\underset{q_{2,1}}{0}\underset{q_{2,2}}{0}\dots\underset{q_{2,N}}{0}\rangle + |\underset{q_{2,1}}{1}\underset{q_{2,2,}}{1}\dots\underset{q_{2,N}}{1}\rangle\Big)\dots \\ \dots\Big(|\underset{q_{N,1}}{0}\underset{q_{N,2}}{0}\dots\underset{q_{N,N}}{0}\rangle + |\underset{q_{N,1}}{1}\underset{q_{n,2}}{1}\dots\underset{q_{N,N}}{1}\rangle\Big)\end{aligned} \quad (3.57)$$

$$\begin{aligned}|1\rangle \equiv \tfrac{1}{2^{\frac{n}{2}}}\Big(|\underset{q_{1,1}}{0}\underset{q_{1,2}}{0}\dots\underset{q_{1,N}}{0}\rangle - |\underset{q_{1,1}}{1}\underset{q_{1,2}}{1}\dots\underset{q_{1,N}}{1}\rangle\Big)\Big(|\underset{q_{2,1}}{0}\underset{q_{2,2}}{0}\dots\underset{q_{2,N}}{0}\rangle - |\underset{q_{2,1}}{1}\underset{q_{2,2,}}{1}\dots\underset{q_{2,N}}{1}\rangle\Big)\dots \\ \dots\Big(|\underset{q_{N,1}}{0}\underset{q_{N,2}}{0}\dots\underset{q_{N,N}}{0}\rangle - |\underset{q_{N,1}}{1}\underset{q_{N,2}}{1}\dots\underset{q_{N,N}}{1}\rangle\Big).\end{aligned} \quad (3.58)$$

We arrange the N^2-qubits into a 2-D lattice with the same number of distinguishable species as follows.

$$\begin{array}{|c|c|c|c|}\hline q_{1,1} & q_{1,2} & \cdots & q_{1,N} \\ \hline q_{2,1} & q_{2,2} & \cdots & q_{2,N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline q_{N,1} & q_{N,2} & \cdots & q_{N,N} \\ \hline \end{array} \quad (3.59)$$

The encoding can be performed by a circuit analogous to that in Figure 3.8, which can be simulated by a suitable pulse sequence. Decoding can be achieved by the reverse pulse sequence.

To perform error correction, the first step is to implement the majority-vote across each row $(q_{i,1}, \dots, q_{i,N})$ (for $1 \le i \le N$) to correct bit-flip errors. Then we implement a transformation that maps each row of N qubits according to

$$\begin{aligned}\tfrac{1}{\sqrt{2}}\big(|00\dots0\rangle + |11\dots1\rangle\big) &\mapsto |00\dots0\rangle \\ \tfrac{1}{\sqrt{2}}\big(|00\dots0\rangle - |11\dots1\rangle\big) &\mapsto |11\dots1\rangle.\end{aligned} \quad (3.60)$$

Following this transformation, we perform the majority-vote across qubits $(q_{1,j}, \dots, q_{N,j})$ (for $1 \le j \le N$) to correct phase-flip errors. Finally, the inverse of the transformation (3.60) is applied.

The transformation (3.60) can be implemented in a manner directly analogous to that used in the previous section for the 9-qubit code. The majority-vote requires more careful consideration. We will discuss the procedure in terms of a 1-dimensional lattice, consisting of N distinct species $(T_1), \ldots, (T_N)$ (the procedure can easily be adapted to apply to any row or column of the lattice (3.59)). Pulses of the form $(T_i)^U_{lr}$ are used to apply U to the qubit of type (T_i) whenever the left and right neighbours are in the basis states $|l\rangle$ and $|r\rangle$ respectively. Assume the lattice is in some basis state $|y_1\rangle|y_2\rangle \ldots |y_N\rangle$, as shown below.

y_1	y_2	y_3	y_4	$\cdots$	y_N
(T_1)	(T_2)	(T_3)	(T_4)	$\cdots$	(T_N)

Consider $y_1, y_2, \ldots y_N$ to be logical bits that we wish to perform the majority-vote on (transforming the n lattice qubits to the basis state $|00\ldots0\rangle$ or $|11\ldots1\rangle$ depending on whether the majority of $\{y_1, y_2, \ldots y_N\}$ is 0 or 1).

One approach is to select a triple $\{y_i, y_j, y_k\}$ for i, j, k selected randomly from $\{1, 2, \ldots, n\}$, and then perform the majority-vote on this triple. The (classical) control program chooses the random i, j, k, and then generates the required EXCH pulses to bring y_i, y_j, y_k into adjacent positions on the lattice. The majority-vote is then implemented for this triple by a pulse sequence analogous to (3.36). Repeating this procedure a sufficient number of times gives a probabilistic method for the majority-vote. After the last majority-vote on a triple, the resulting value is taken to be the majority of all n bits. The remaining bits are then zeroed with the RESET operation, and the majority is copied into them by a suitable sequence of EXCH and $(T_m)^X_{lr}$ pulses.

I will now show that it may be possible to implement the majority-vote step for the above code deterministically, without resorting to the random selection of triples, and without requiring any additional ancillary qubits. Recall that this must be done in a manner so that our applications of the RESET operation do not cause any unwanted coupling with the environment. Consider computing the majority value M of $N = 2n + 1$ bits, as well as a syndrome S indicating those positions where the value of the bit does not agree with the majority. The majority M can be stored in a single bit. As long as at most n bit-flip

errors occurred, the number of possible syndromes is

$$\sum_{i=0}^{n} \binom{2n+1}{i} = 2^{2n}.$$

So the syndrome can be stored in $2n$ bits. This means that the majority value and the syndrome can together be stored in $N = 2n + 1$ bits. Given an N-bit codeword that has been subjected to at most n bit-flip errors resulting in the state $|x\rangle$, the map

$$P : |x\rangle \leftrightarrow |M, S\rangle$$

is a permutation of the N bit strings. If we can implement this permutation, then we can safely reset the syndrome $|S\rangle$ (which is independent of M), and then re-encode the majority value M to obtain the original (corrected) codeword.

Each N-tuple is encoded on a lattice segment containing N distinct species. The lattice segment can be thought of as an N-qubit quantum computer on which we have local control. So if we can show that there exists a quantum circuit that will implement the permutation P without using any ancillary qubits, then we can implement the error correction step on the lattice.

By methods in [BBC+95] (see Cor. 7.6 of that paper) we can implement any controlled-NOT gate with an arbitrary pattern of $(N-1)$ control qubits, on an N-qubit circuit with $O(N^2)$ CNOT and single-qubit gates. Each such gate implements a permutation of the N-bit strings that exchanges two strings differing in one bit, and leaves the remaining strings alone. By a simple inductive argument, we can then implement any permutation that exchanges any desired pair of strings, and thus that we can implement any desired permutation of the N-bit strings on an N-qubit circuit. In general the circuits produced by this approach will have exponential depth, but it may be that for suitable encodings of the syndromes S, an efficient circuit can be found for the required permutation P.

Following the strategy outlined in Section 3.4.4 we can implement the scaled codes on a 1-dimensional lattice, although appropriate EXCH operations are needed to shuttle the lattice qubits into the positions required for error correction (and this will effect the error threshold as mentioned in Section 3.4.4). This observation leads us to the following.

Fact 3.4.1 *Given a 1-D lattice configured as a repeating chain of $(2n+1)^2$ distinguishable qubit species sharing anisotropic nearest-neighbour couplings, we can implement an error correcting code for a* GCA *memory that protects against n single-qubit errors within each block of $(2n+1)^2$ lattice qubits.*

3.4.6 From a robust GCA memory to a robust implementation of SPA

3.4.6.1 For the 3-bit code

In the previous sections we considered reliable implementations of GCA memory; we encoded the data qubits but gave no provision for implementing a pointer. In this section we consider ways to extend our implementations of GCA memory to provide reliable implementations of SPA.

Consider the 3-bit code we saw in Section 3.4.2. One possibility for introducing the pointer is to form a 2-dimensional lattice by adding a second 1-dimensional chain coupled to the first. The second chain could be another ABC-chain, or it could have a simpler structure. It could consist of just a single species D, each coupled to the A-, B- or C-qubit beside it. The D-qubits do not have to be coupled to each other. The pointer is represented on the D qubits, as illustrated below (where the pointer is shown to be addressing the ith data qubit).

A	B	C	A	B	C	$\cdots$	A	B	C	A	B	C	$\cdots$	A	B	C	A	B	C
0	0	0	x_1	x_1	x_1	$\cdots$	x_{i-1}	x_{i-1}	x_{i-1}	x_i	x_i	x_i	$\cdots$	x_N	x_N	x_N	0	0	0
0	0	0	0	0	0	$\cdots$	0	0	0	1	1	1	$\cdots$	0	0	0	0	0	0
D	D	D	D	D	D	$\cdots$	D	D	D	D	D	D	$\cdots$	D	D	D	D	D	D

pointer

We suppose the hardware now provides pulses of the form T^U_{lrc}, where $T \in \{A, B, C\}$, $l, r, c \in \{0, 1, *\}$ and U is a 1-qubit unitary. This pulse has the effect of applying U to all qubits of type T whose left and right neighbours are in states l and r respectively, and whose D-type neighbour in the parallel chain is in the state c. We also allow the

(unconditional) reset pulse T^{RESET} targeting qubits of type T. For the D qubits in the lower chain, the hardware provides pulses D_c^U. Here, $c \in \{0, 1, *\}$ represents the state of the A, B or C neighbour in the parallel chain.

Error correction for the data array can be performed exactly as before with the sequence

$$A^X_{*1*}\ ,\ C^X_{1**}\ ,\ B^X_{11*}\ ,\ A^{\text{RESET}}\ ,\ C^{\text{RESET}}\ ,\ A^X_{*1*}\ ,\ C^X_{1**}, \tag{3.61}$$

which is directly analogous to Sequence (3.36). To correct the state of the pointer, we simply exchange the D-chain with the ABC-chain by the following pulse sequence.

$$D^X_1\ ,\ A^X_{**1}\ ,\ B^X_{**1}\ ,\ C^X_{**1}\ ,\ D^X_1. \tag{3.62}$$

Since the pointer is encoded in the manner of the bit-flip code (triplet repetition), we can use the same pulse sequence as above to correct errors on the pointer state, which now occupies the ABC-chain.

To move the pointer left and right along the D-chain, we use the same trick, and swap the state of the D-chain with the state of the ABC-chain, and move the pointer along the ABC-chain before swapping it back down to the D-chain.

We can implement `SPA` by this scheme as follows.

$$\mathtt{P_L} = D^X_1\ ,\ A^X_{**1}\ ,\ B^X_{**1}\ ,\ C^X_{**1}\ ,\ D^X_1\ ,\ C^X_{10*}\ ,\ C^X_{01*}\ ,\ B^X_{10*}\ ,\ B^X_{01*}\ ,\ A^X_{10*}\ ,\ A^X_{01*}\ , \tag{3.63}$$

$$D^X_1\ ,\ A^X_{**1}\ ,\ B^X_{**1}\ ,\ C^X_{**1}\ ,\ D^X_1 \tag{3.64}$$

$$\mathtt{P_R} = D^X_1\ ,\ A^X_{**1}\ ,\ B^X_{**1}\ ,\ C^X_{**1}\ ,\ D^X_1\ ,\ A^X_{01*}\ ,\ A^X_{10*}\ ,\ B^X_{01*}\ ,\ B^X_{10*}\ ,\ C^X_{01*}\ ,\ C^X_{10*}\ , \tag{3.65}$$

$$D^X_1\ ,\ A^X_{**1}\ ,\ B^X_{**1}\ ,\ C^X_{**1}\ ,\ D^X_1 \tag{3.66}$$

$$\mathtt{G}^U = C^X_{1*1}\ ,\ B^X_{1*1}\ ,\ A^U_{**1}\ ,\ B^X_{1*1}\ ,\ C^X_{1*1} \tag{3.67}$$

$$\mathtt{CZ} = D^X_1\ ,\ B^X_{**1}\ ,\ D^X_1\ ,\ C^Z_{11*}\ ,\ B^X_{**1}\ ,\ D^X_1 \tag{3.68}$$

To implement the `CZ` operation, I used the fact that for the 3-bit code it suffices to implement a controlled-Z gate between any pair of qubits, one qubit from the first codeword, and one qubit from the second codeword. We swapped the B-qubits with the D-qubits below, and then controlled the Z operation on the C-qubits, conditioned on the A-qubit to its right and the B-qubit to its left (which now contains the pointer bit).

After every SPA instruction, error correction can be applied as described above (for both the data and the pointer). Notice that the data array and pointer are only protected from bit-flip errors *between* instructions of SPA. During the execution of a sequence of pulses for a basic instruction, the encoding is not maintained and the system is unprotected from errors (as before, this is not a fault-tolerant construction).

3.4.6.2 For the 9-qubit code and scaled versions

In Section 3.4.3 we saw how to implement the 9-qubit code for protecting a quantum memory in a 2-dimensional lattice against arbitrary single-qubit errors. It is a natural extension of the 3-bit code implementation discussed in Section 3.4.2. This scheme can be extended to implement a pointer, in a manner analogous to what was done for the 3-bit code above. We could add a chain of qubits of a tenth species, K, and couple this chain to the (G, H, J)-row of the 9-qubit code implementation. As before, the K qubits do not have to be coupled to each other. This setup is illustrated below.

A	B	C	A	B	C	A	B	C	$\cdots$	A	B	C
D	E	F	D	E	F	D	E	F	$\cdots$	D	E	F
G	H	J	G	H	J	G	H	J	$\cdots$	G	H	J
K	K	K	K	K	K	K	K	K	$\cdots$	K	K	K

(3.69)

By the observation in Section 3.4.4, we could alternatively arrange the system as follows.

A	B	C	D	E	F	G	H	J	A	B	C	D	E	F	G	H	J	$\cdots$
K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	$\cdots$

(3.70)

The description of the implementation of SPA is somewhat more compact for the arrangement (3.69), so we will examine that one in detail. We redefine the pulses for the (G, H, J) qubits to account for the state of the K-qubits coupled to them, for example $G_{l,r,a,b}$. Pulses for the K-qubits are of the form K_a since they are only coupled to the qubits in the row above them.

The data qubits are encoded on the $(A, B, \ldots J)$-qubits using the Shor code, as described in Section 3.4.3. The pointer is encoded on the K-qubits. The pointer can be shuttled

around in a manner analogous to what we did in Section 3.4.6.1, by swapping the row of K-qubits with the (G, H, J)-row above it, and shuttling along that row before swapping back down to the K-row. Since the pointer is encoded using a 3-bit repetition code, we can correct bit-flip errors on the pointer just as we did in Section 3.4.6.1, by swapping to the (G, H, J)-row and doing the majority-vote there.

Using the above scheme, we can implement SPA as follows.

$$
\begin{aligned}
\mathtt{P_L} = {} & K_1^X\ ,\ G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ K_1^X\ ,\ J_{10**}^X\ ,\ J_{01**}^X\ ,\ H_{10**}^X\ ,\ H_{01**}^X\ ,\ G_{10**}^X\ ,\ G_{01**}^X\ , \\
& K_1^X\ ,\ G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ K_1^X \\
\mathtt{P_R} = {} & K_1^X\ ,\ G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ K_1^X\ ,\ G_{01**}^X\ ,\ G_{10**}^X\ ,\ H_{01**}^X\ ,\ H_{10**}^X\ ,\ J_{01**}^X\ ,\ J_{10**}^X\ , \\
& K_1^X\ ,\ G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ K_1^X \\
\mathtt{G}^U = {} & A_{*1*}^X\ ,\ B_{*1*}^X\ ,\ D_{*1**}^X\ ,\ E_{*1**}^X\ ,\ G_{*1**}^X\ ,\ H_{*1**}^X\ , \\
& C_{***}^H\ ,\ F_{****}^H\ ,\ J_{****}^H\ ,\ C_{**1}^X\ ,\ F_{***1}^X\ ,\ A_{****}^U\ , \\
& F_{***1}^X\ ,\ C_{**1}^X\ ,\ J_{****}^H\ ,\ F_{****}^H\ ,\ C_{***}^H\ ,\ H_{*1**}^X\ , \\
& G_{*1**}^X\ ,\ E_{*1**}^X\ ,\ D_{*1**}^X\ ,\ B_{*1*}^X\ ,\ A_{*1*}^X \\
\mathtt{CZ} = {} & G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ \textsc{exch}_{DG}\ ,\ \textsc{exch}_{EH}\ ,\ \textsc{exch}_{FJ} G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ , \\
& \textsc{exch}_{AD}\ ,\ \textsc{exch}_{BE}\ ,\ \textsc{exch}_{CF}\ ,\ \textsc{exch}_{DG}\ ,\ \textsc{exch}_{EH}\ ,\ \textsc{exch}_{FJ}\ , \\
& G_{***1}^X\ ,\ H_{***1}^X\ ,\ J_{***1}^X\ ,\ \textsc{exch}_{DG}\ ,\ \textsc{exch}_{EH}\ ,\ \textsc{exch}_{FJ}\ , \\
& \textsc{exch}_{AD}\ ,\ \textsc{exch}_{BE}\ ,\ \textsc{exch}_{CF}\ ,\ \textsc{exch}_{DG}\ ,\ \textsc{exch}_{EH}\ ,\ \textsc{exch}_{FJ}
\end{aligned}
$$

To implement the $\mathtt{G}^U$ operation we decoded the codeword into the J-qubit, and then controlled the U operation on the K-qubit below it, and then re-encoded. To implement the CZ instruction we used the fact that for the Shor code the Z operation is implemented by applying the X gate to each of the 9 qubits in the codeword. Conditioned on the pointer in the K-qubits, first X is applied to the qubits in the (G, H, J) row. Then the rows are permuted by EXCH pulse sequences and the process is repeated until each qubit in the codeword has had X applied, conditioned on the pointer.

The above strategy generalizes in a straightforward manner to give an implementation of SPA protected by the scaled code described in Section 3.4.5. When implemented in the

1-D manner described in Section 3.4.4, this scheme demonstrates the following (which is an extension of Fact 3.4.1).

Fact 3.4.2 *Given a 1-D lattice configured as a repeating chain of $(2n+1)^2$ distinguishable qubit species sharing anisotropic nearest-neighbour couplings, where each lattice qubit is additionally coupled to a separate qubit of a $((2n+1)^2+1)^{th}$ distinguishable species, we can implement* `SPA` *with an error correcting code that protects against n single-qubit errors within each block of $(2n+1)^2$ lattice qubits.*

3.4.7 A general construction for implementation-level codes on lattices

An implementation-level error correction scheme for a GCA system can be designed based on any quantum error correcting code. Consider a code having m-qubit codewords. Suppose the error correction operation requires a ancillary qubits. A quantum memory can be implemented on a lattice having $k = (m+a)$ distinguishable species of lattice qubits. Each k-tuple of lattice qubits can be treated as a small quantum computer which can perform the error correction operation for a single codeword.

Given a reliable implementation of a quantum memory in a GCA system using a suitable error correcting code, a scheme for implementing `SPA` can be designed in the following way. Suppose the GCA memory is implemented on a lattice having k distinguishable species, say $A_1, \ldots, A_k$. A pointer can be implemented by adding a $(k+1)^{\text{st}}$ distinguishable species A_{k+1}, and coupling an A_{k+1}-qubit to each and every lattice qubit of species A_1 through A_k. The $\mathtt{P_L}$ and $\mathtt{P_R}$ operations can be implemented by swapping the A_1-qubits with their neighbouring $A_1 \ldots A_k$ qubits, shuttling the pointer state down to the next A_1 qubit (left or right), and then swapping the pointer states back to the A_{k+1}-qubits. For specific architectures, there may certainly be more efficient schemes than this.

3.5 GCA models with parallelism

Suppose we have implemented SPA, perhaps using a scheme like the ones described in the previous sections. A natural question is whether we can achieve fault tolerance by implementing scalable error correcting codes directly using the basic instructions of SPA (this is the data-level error correction that was mentioned earlier, and will be explored at greater length in Section 3.6). We can immediately see that fault-tolerant programs can *not* be expressed using SPA, because that scheme does not provide enough parallelism (SPA provides no parallelism at all). It has been shown [ABO97] that for fault tolerance we require better than a logarithmic degree of parallelism (that is, at must be possible to perform gates on at least a logarithmic number of qubits simultaneously). The basic problem is that without parallelism, errors may occur faster than they can be corrected.

In the next subsection I will define a class of GCA models with higher degrees of parallelism. The basic idea is to allow the GCA to use multiple pointers simultaneously, and allow subsets of these pointers to be activated and deactivated at specific times during a computation. Suppose we want to implement the well known 5-qubit code to protect the data qubits in a GCA. Parallelism is required when we want to do an error correction step. An identical error correction operation would be applied to each 5-qubit codeword simultaneously. In a GCA this can be achieved by activating a uniformly spaced set of pointers, one for each encoded data qubit. Then global pulses will affect each of these pointers in the same way, so that an identical error correction circuit is simulated on each codeword. After the error correction operation, all the pointers except for one are deactivated, and the single remaining active pointer is used to direct the next stage of the computation. To achieve fault tolerance using this approach, we will need to be able to simulate circuits that use concatenated versions of the error correcting code, and we will explore this problem later.

3.5.1 The language MPA(k)

Our first parallel GCA model will be described by a language that I will call MPA(k). The parameter k is an integer specifying the degree of parallelism, which in practice might be dictated by size of an error correcting code that we wish to use. The state of such a GCA

at any time is given as follows.

$$\begin{array}{|c|}\hline \underline{\textit{State} \text{ for } \texttt{MPA}(k)} \\ (\rho, m, b, o) \\ \hline \end{array} \tag{3.71}$$

As before, ρ is the global state of the data array. For simplicity we will assume that the number N of data qubits is a multiple of k (this is natural if we are proposing to divide the N data qubits into blocks (codewords) of k data qubits each). The parameter m gives the *mode* of operation: $m = 0$ for *single-pointer mode* (in which there is only a single active pointer), and $m = 1$ for *multi-pointer mode* (in which there is an active pointer in every block of k data qubits). For single-pointer mode, the parameter $b \in \{0, \dots, N/k - 1\}$ indicates the *block* in which the active pointer currently resides (blocks are numbered from 0), and the parameter $o \in \{1, \dots, k - 1\}$ is the *offset* within this block (offsets numbered from 1). That is, the single active pointer is at data qubit $kb + o$. For multi-pointer mode, b indicates the block in which the single pointer resided just prior to the change to multi-pointer mode (that is, the last block occupied by a single active pointer), and o indicates the offset of each of the N/k active pointers. That is, in multi-pointer mode there are active pointers at data qubits $o, o + k, o + 2k, \dots, o + (N - 1)k$.

For activating and deactivating the pointers, we have the following basic instruction in $\texttt{MPA}(k)$.

$$\boxed{\texttt{PNTR} : (\rho, m, b, 1) \mapsto (\rho, m \oplus 1, b, 1)} \tag{3.72}$$

If the instruction $\texttt{PNTR}$ is called from a state (ρ, m, b, o) with $o \neq 1$, the first step could be to execute a sequence of $\texttt{P}_\texttt{L}$ instructions to move the pointer(s) to the offset $o = 1$. Then $\texttt{PNTR}$ will map $(\rho, 0, b, 1)$ to $(\rho, 1, b, 1)$ and conversely. The idea is that the pointers are moved to positions $1, k + 1, 2k + 1, \dots$ (in the case of a single pointer it is moved to position 1),

and then PNTR toggles between the single and multiple pointer modes. When toggling to multi-pointer mode, the state records the block in which the single pointer resided before the switch (this is important so that the single data pointer does not have to travel too far to resume its previous location after we switch back to single pointer mode following an error correction cycle).

The basic instructions $\mathtt{P_L}$ and $\mathtt{P_R}$ now have the effect of moving *every active pointer* one place to the left or right respectively.

$$\boxed{\begin{aligned} \mathtt{P_L} &: \big(\rho, m, b, o\big) \mapsto \big(\rho, m, b - \delta_{o,1}, o - 1 + (k-1)\delta_{o,1}\big) \\ \mathtt{P_R} &: \big(\rho, m, b, o\big) \mapsto \big(\rho, m, b + \delta_{o,k-1}, o + 1 - (k-1)\delta_{o,k-1}\big) \end{aligned}} \tag{3.73}$$

The basic instruction $\mathtt{G}^U$ has the effect of applying the 1-qubit gate U to every data qubit currently addressed by an active pointer.

$$\boxed{\mathtt{G}^U : \big(\rho, m, b, o\big) \mapsto \big(U^{(m,b,o)} \rho \, U^{(m,b,o)\dagger}, m, b, o\big)} \tag{3.74}$$

The operator $U^{(m,b,o)}$ appearing above is defined as follows.

$$U^{(m,b,o)} \equiv \begin{cases} (I^{\otimes kb+o-1}) \otimes U \otimes (I^{\otimes N-kb-o}) \text{ if } m = 0, \\ (I^{\otimes o-1}) \otimes U \otimes (I^{\otimes k-1}) \otimes U \otimes \cdots \otimes (I^{\otimes k-1}) \otimes U \otimes (I^{\otimes N-o}) \text{ if } m = 1. \end{cases} \tag{3.75}$$

We will also allow pulses of the form $\mathtt{G}^{\text{RESET}}$, by taking U to be the (non-unitary) reset-to-0 operation. Note that this basic instruction might be implemented similarly to $\mathtt{G}^U$ using the dissipative pulses described in Section 3.4.1. These basic operations will be required for implementing data-level error correction.

The CZ instruction also behaves as before, except now with respect to the data qubits at

every position indexed by an active pointer. Additionally, there is a constraint that $k > 1$, otherwise pointers at adjacent locations would conflict.

$$\boxed{\texttt{CZ} : \big(\rho, m, b, o\big) \mapsto \big(cZ^{(m,b,o,o+1)}\rho\, cZ^{(m,b,o,o+1)^\dagger}, m, b, o\big)} \tag{3.76}$$

The operator $cZ^{(m,b,o,o+1)}$ is defined as follows.

$$cZ^{(m,b,o,o+1)} \equiv \begin{cases} \big(I^{\otimes kb+o-1}\big) \otimes \left(\begin{smallmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&-1 \end{smallmatrix}\right) \otimes \big(I^{\otimes N-kb-o-1}\big) \text{ if } m=0 \\ \big(I^{\otimes o-1}\big) \otimes \left(\begin{smallmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&-1 \end{smallmatrix}\right) \otimes \big(I^{\otimes k-2}\big) \otimes \left(\begin{smallmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&-1 \end{smallmatrix}\right) \otimes \big(I^{\otimes k-2}\big) \otimes \dots \\ \qquad \dots \otimes \big(I^{\otimes k-2}\big) \otimes \left(\begin{smallmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&-1 \end{smallmatrix}\right) \otimes \big(I^{\otimes N-o-1}\big) \text{ if } m=1. \end{cases} \tag{3.77}$$

We summarize the basic instructions for $\texttt{MPA}(k)$ below.

$$\boxed{\begin{aligned} &\underline{\text{Basic instructions of } \texttt{MPA}(k)} \\ &\texttt{PNTR} : (\rho, m, b, 1) \mapsto (\rho, m \oplus 1, b, 1) \\ &\texttt{P}_\texttt{L} : \big(\rho, m, b, o\big) \mapsto \big(\rho, m, b - \delta_{o,1}, o - 1 + (k-1)\delta_{o,1}\big) \\ &\texttt{P}_\texttt{R} : \big(\rho, m, b, o\big) \mapsto \big(\rho, m, b + \delta_{o,k-1}, o + 1 - (k-1)\delta_{o,k-1}\big) \\ &\texttt{G}^U : \big(\rho, m, b, o\big) \mapsto \big(U^{(m,b,o)}\rho\, U^{(m,b,o)^\dagger}, m, b, o\big) \\ &\texttt{CZ} : \big(\rho, m, b, o\big) \mapsto \big(cZ^{(m,b,o,o+1)}\rho\, cZ^{(m,b,o,o+1)^\dagger}, m, b, o\big) \end{aligned}} \tag{3.78}$$

3.5.2 An approach to implementing MPA(k)

In this section I describe an approach for implementing MPA(k) with the ABC-chain described in Section 3.2.2.1. The main idea is one that appeared in [BBK04], and that is to use locations in the lattice called "switching stations" to selectively activate and deactivate the pointers.

The lattice is organized into sections. Some of the sections contain the encoding of a data qubit ρ_i (encoded by an appropriate state on that section of the lattice) and the presence/absense of an active pointer addressing that data qubit. Other sections contain the encoding of switching stations (SS), and still other sections are "dummy switching stations" ($\overline{SS}$). The dummy switching stations are the same length as the switching stations, but contain all $|0\rangle$ qubits and so do not behave like switching stations. Their purpose is to ensure that the data qubits are evenly spaced, which is a convenience (more complicated schemes could be devised that do not require dummy switching stations). There is a switching station immediately to the left of each of the data qubits $d_1, d_{k+1}, \ldots, d_{N-k+1}$. There is a dummy switching station immediately to the left of all other d_i sections.

For $k = 3$, the arrangement of these lattice sections is illustrated below. Each section depicted here is composed of a fixed number of lattice qubits.

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline SS & d_1 & \overline{SS} & d_2 & \overline{SS} & d_3 & SS & d_4 & \overline{SS} & d_5 & \overline{SS} & d_6 & SS & \ldots & SS & d_{N-2} & \overline{SS} & d_{N-1} & \overline{SS} & d_N \\ \hline \end{array} \tag{3.79}$$

Suppose the machine is in the single-pointer mode, the pointer is at data qubit $kb + o$, and we now wish to enter multi-pointer mode for an error correction cycle (that is, we want to implement the PNTR basic instruction). The procedure is to move the single pointer into the nearest SS and then to "mark" that SS (so that we can later reactivate the single pointer in this location). Then a suitable pulse sequence spawns active pointers inside all of the SS (but not the $\overline{SS}$) and these pointers move out of the SS to address the data qubits $d_1, d_{k+1}, \ldots, d_{N-k+1}$. Now we are in multi-pointer mode, and the active pointers can move within their blocks to direct error correction operations on each block in parallel.

After the error correction cycle, every pointer is moved back into the SS from which it came. Then a sequence of pulses will have the effect of deactivating the pointer in every SS *except* the one that was previously marked as the last single-pointer position. This leaves only one active pointer in the marked SS, which can then be moved back to its original position within that block (in a constant number of steps). Using the above scheme, we can always do a round of error correction after a constant number of single-pointer operations.

I now describe a way to implement the various lattice sections on the ABC-chain described in Section 3.2.2.1. A section of the lattice containing a data qubit in a basis state $d_i = |x_i\rangle$ is formatted as follows.

$$\boxed{d_i} = \underset{A}{\boxed{x_i}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}}\underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}} \tag{3.80}$$

In the case that d_i is currently being addressed by an active pointer, we use the same encoding that was used in Section 3.2.2.1, putting the B- and C-qubits to the right of x_i in the basis state $|1\rangle$.

$$\underset{\uparrow}{\boxed{d_i}} = \underset{A}{\boxed{x_i}}\underset{B}{\boxed{1}}\underset{C}{\boxed{1}}\underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}} \tag{3.81}$$

The switching stations and dummy switching stations are formatted as follows.

$$\boxed{SS} = \underset{A}{\boxed{1}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}}\underset{A}{\boxed{1}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}}\underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}} \tag{3.82}$$

$$\boxed{\overline{SS}} = \underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}}\underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}}\underset{A}{\boxed{0}}\underset{B}{\boxed{0}}\underset{C}{\boxed{0}} \tag{3.83}$$

When the pointer is “parked” in a switching station, the encoding is as follows.

$$\begin{array}{c} \boxed{SS} \\ \uparrow \end{array} = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} & \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} & \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \end{array} \tag{3.84}$$

Notice that, with respect to each (A, B, C)-triple, the encoding given above is consistent with that described in Section 3.2.2.1. That is, in the computational basis, each triple contains a $|0\rangle$ or a $|1\rangle$ in the A-qubit, followed by either $|00\rangle$ or $|11\rangle$ in the B, C-qubits. This means that we can use some of the pulse sequences already derived in Section 3.2.2.1 as primitives for implementing basic operations in this scheme.

Consider moving the pointer(s) to the left or right. In Section 3.2.2.1 we saw a pulse sequence for moving the pointer left or right one (A, B, C)-triple. To move the pointer between two adjacent data qubits (which are separated by a SS or $\overline{SS}$), we use this sequence repeatedly to move to the left or right by 5 (A, B, C)-triples. Single-qubit operations $\mathtt{G}^U$ are implemented exactly as in Section 3.2.2.1. For `CZ` instructions between non-neighbouring data qubits, we first move the pair of data qubits into adjacent positions by a sequence of `SWAP` instructions, which can be implemented using $\mathtt{G}^U$ and `CZ` instructions between neighbouring data qubits.

To implement the `PNTR` operations, first suppose there is a single active pointer, and we wish to enter multi-pointer mode. The first step is to move the active pointer to the nearest SS, so that the SS-section of the lattice is configured as in (3.84).

The next step is to mark that switching station. This is accomplished by moving the pointer to the next BC pair to the right, and then using an A^X_{01} pulse. This transforms that SS-section of the lattice into the following configuration.

$$\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} & \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} & \multicolumn{1}{c}{A} & \multicolumn{1}{c}{B} & \multicolumn{1}{c}{C} \end{array} \tag{3.85}$$

At this point the (A, B, C, A)-sequence composing the first 4 lattice qubits of every SS contains the pattern $1, 0, 0, 1$. The (A, B, C, A)-sequence in the middle of every SS contains the pattern $1, 0, 0, 0$, except for the marked SS, in which these middle qubits contain the pattern $1, 1, 1, 1$. The SS's are the *only* locations in the lattice containing an (A, B, C, A)-

sequence configured with a 1 on both of the A qubits. We exploit this fact to implement a sequence that will deactivate the pointer currently residing in the middle (B, C)-pair of the marked switching station, and will activate new pointers in the leftmost (B, C)-pairs of all switching stations (including the marked one). To achieve this, we want to implement the following transformation on all (A, B, C, A)-sequences in the lattice (where a_i, b_i, c_i, a_{i+1} are the binary values describing basis states of the 4 adjacent qubits of types A, B, C, A, and the addition and multiplication are performed mod 2).

$$(a_i\ ,\ b_i\ ,\ c_i\ ,\ a_{i+1}) \mapsto (a_i\ ,\ b_i + a_i a_{i+1}\ ,\ c_i + a_i a_{i+1}\ ,\ a_{i+1}) \tag{3.86}$$

Notice that the above transformation has the desired effect on the lattice states within the SS's, and has no effect at any other position in the lattice (including the $\overline{SS}$'s). It can be implemented with a pulse sequence as follows.

$$\begin{aligned}
(a_i\ ,\ b_i\ ,\ c_i\ ,\ a_{i+1}) &\xmapsto{C^X_{*1}} (a_i\ ,\ b_i\ ,\ c_i + a_{i+1}\ ,\ a_{i+1}) && (3.87)\\
&\xmapsto{B^X_{11}} (a_i\ ,\ b_i + a_i(c_i + a_{i+1})\ ,\ c_i + a_{i+1}\ ,\ a_{i+1}) && (3.88)\\
&\xmapsto{C^X_{*1}} (a_i\ ,\ b_i + a_i(c_i + a_{i+1})\ ,\ c_i\ ,\ a_{i+1}) && (3.89)\\
&\xmapsto{B^X_{11}} (a_i\ ,\ b_i + a_i(c_i + a_{i+1}) + c_i a_i\ ,\ c_i\ ,\ a_{i+1}) && (3.90)\\
&\xmapsto{B^X_{11}} (a_i\ ,\ b_i + a_i(c_i + a_{i+1}) + c_i a_i\ ,\ c_i\ ,\ a_{i+1}) && (3.91)\\
&= (a_i\ ,\ b_i + a_i a_{i+1}\ ,\ c_i\ ,\ a_{i+1}) && (3.92)\\
&\xmapsto{B^X_{1*}} (a_i\ ,\ b_i + a_i a_{i+1} + a_i\ ,\ c_i\ ,\ a_{i+1}) && (3.93)\\
&\xmapsto{C^X_{11}} (a_i\ ,\ b_i + a_i a_{i+1} + a_i\ ,\ c_i + (b_i + a_i a_{i+1} + a_i)a_{i+1}\ ,\ a_{i+1}) && (3.94)\\
&\xmapsto{B^X_{1*}} (a_i\ ,\ b_i + a_i a_{i+1}\ ,\ c_i + (b_i + a_i a_{i+1} + a_i)a_{i+1}\ ,\ a_{i+1}) && (3.95)\\
&\xmapsto{C^X_{11}} (a_i\ ,\ b_i + a_i a_{i+1}\ , && (3.96)\\
&\qquad c_i + (b_i + a_i a_{i+1} + a_i)a_{i+1} + (b_i + a_i a_{i+1})a_{i+1}\ ,\ a_{i+1}) && (3.97)\\
&= (a_i\ ,\ b_i + a_i a_{i+1}\ ,\ c + a_i a_{i+1}\ ,\ a_{i+1}) && (3.98)
\end{aligned}$$

Noting that $(B^X_{11}, B^X_{1*}) = B^X_{10}$, the above pulse sequence can be abbreviated to:

$$C^X_{*1}\ ,\ B^X_{11}\ ,\ C^X_{*1}\ ,\ B^X_{10}\ ,\ C^X_{11}\ ,\ B^X_{1*}\ ,\ C^X_{11}. \tag{3.99}$$

After using the above sequence to enter multiple-pointer mode, each SS will have a pointer encoded in its leftmost (B, C)-pair. These pointers can now be moved around and can direct operations on the data blocks (codewords) in parallel. When we wish to return to the single-pointer mode, the pointers are returned to the leftmost (B, C)-pair in each SS. Then applying the above sequence again will deactivate all the pointers in these positions, and at the same time reactivate the single pointer in the middle (B, C)-pair of the marked SS. Before moving off, the single pointer can be used to "unmark" the SS.

In Appendix B, I describe the strategy proposed by Benjamin for implementing switching stations for the architecture described in [Ben00].

3.6 Data-level error correction

3.6.1 The general approach

Data-level error correction using MPA(k) can be accomplished by simulating a quantum circuit that uses one of the standard quantum error correcting codes (e.g. Steane's 7-qubit code). During an error correction step in such a circuit, all of the codewords are corrected in parallel, each by an identical procedure. Since the codewords are formed by regular groupings of qubits, the regularly spaced pointers of MPA(k) can be employed.

A circuit using m-qubit codewords at one level of concatenation could be simulated by MPA(k) in the following manner. First we rearrange the circuit so that parallelism is only exploited during an error correction cycle. So, except for error correction cycles, the circuit can be simulated with only a single active pointer (exactly as in SPA). For error correction cycles, the circuit will perform identical correction operations on each codeword. Suppose an error correction operation for each m-qubit codeword uses a ancillary qubits, and let $k = m + a$. Then we can simulate the error correction cycle by a program in MPA(k) as follows. First use a PNTR instruction to activate one pointer for each block of k data

qubits (each block contains a codeword, plus a ancillary data qubits for the error correction operation). Then we use G^{RESET} instructions to reset the ancillary data qubits to $|0\rangle$, and then use the appropriate instructions to simulate the error correcting circuit operations (this happens for each codeword in parallel). After the error correction cycle, we use a `PNTR` instruction to deactivate all but a single pointer, in preparation to simulate the next part of the circuit.

3.6.2 Example: the Steane code

Suppose we wish to simulate a quantum circuit that uses the 7-qubit Steane code [Ste96]. We will divide the data qubits into blocks, each block containing 7 qubits for a codeword, and an additional 3 ancillary qubits for performing the error correction operation. Note that at the beginning of every error correction cycle these three qubits will have to be reset to $|0\rangle$. Figure 3.9 shows a quantum circuit for performing the encoding operation for the Steane code, and Figure 3.10 shows a quantum circuit for performing error correction.

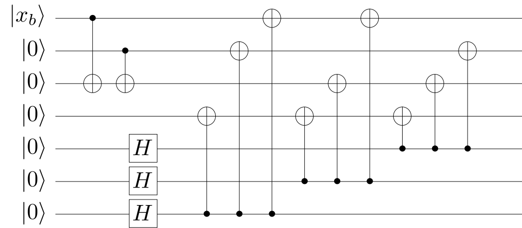


Figure 3.9: A circuit for performing encoding for the Steane code.

Suppose the data array has been initially loaded so that the data qubits (not yet in Steane codewords) are located at positions $10b+1$, for each block b of 10 data qubits (and suppose all other data qubits are set to $|0\rangle$). If we put the GCA into multi-pointer mode ($m = 1$) and the pointer within each block b is located at position $10b + 1$, then the following `MPA`(10) program implements the encoding operation, computing the Steane codeword for each logical qubit into data qubits $10b + 1, 10b + 2, \ldots, 10b + 7$ for each block b.

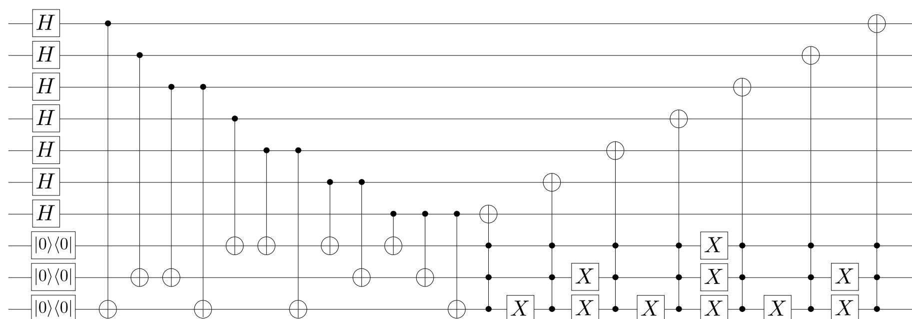


Figure 3.10: A circuit for performing error correction for the Steane code.

MPA(10) program for encoding for the Steane code

$\mathtt{CNOT}_R(2)$, $\mathtt{P_R}$, $\mathtt{CNOT}_R$, $\mathtt{P_R}$, $\mathtt{P_R}$, $\mathtt{P_R}$, $\mathtt{G}^H$, $\mathtt{P_R}$, $\mathtt{G}^H$, $\mathtt{P_R}$, $\mathtt{G}^H$, $\mathtt{CNOT_L}(3)$, $\mathtt{CNOT_L}(5)$, $\mathtt{CNOT_L}(6)$, $\mathtt{P_L}$, $\mathtt{CNOT}_L(2)$, $\mathtt{CNOT_L}(3)$, $\mathtt{CNOT_L}(5)$, $\mathtt{P_L}$, $\mathtt{CNOT_L}$, $\mathtt{CNOT_L}(2)$, $\mathtt{CNOT_L}(3)$, $\mathtt{P_L}$, $\mathtt{P_L}$, $\mathtt{P_L}$, $\mathtt{P_L}$.

In the above program, $\mathtt{CNOT_R}(k)$ and $\mathtt{CNOT_L}(k)$ are shorthand expressions for routines to implement distance-k CNOT gates as described in Section 3.2.4.

To find a program to simulate the error correction circuit (Figure 3.10), we can proceed with the method described in Section 3.2.4. The first part of the program would consist of a sequence of seven $\mathtt{G}^H$ and $\mathtt{P_R}$ instructions, followed by three $\mathtt{G}^{\text{RESET}}$ and $\mathtt{P_R}$ instructions. Then a suitable sequence of pointer-movement and $\mathtt{CNOT_R}(k)$ routines will be used for the second stage of the circuit. The trickiest part of the rewriting procedure will be the triple-controlled NOT gates in the last stage the circuit. Methods from [BBC+95] can be employed for these. The resulting MPA (10) program will be quite lengthy.

Although MPA (10) programs for simulating even modest circuits can become quite long, they have the nice property of expressing GCA programs in a manner that is abstracted from the underlying architecture used to implement them.

3.6.3 Code-concatenation requires more sophisticated parallelism

A fault-tolerant quantum circuit may use an m-qubit code concatenated to some number of levels, say l. A codeword at level i (for any $1 \leq i \leq l$) consists of m codewords at level $i-1$. Suppose that the error correction operation for the basic code requires a ancillary qubits. In a fault-tolerant scheme using concatenated codes, the error correction operation at level i would typically use a ancillary codewords from level $(i-1)$.

Let $k = m + a$. Implementing the error correction operation for each level-i codeword in parallel will require an active pointer for every k^i data qubits in the array. For example, at the second level of concatenation, each block consists of k^2 data qubits. To correct errors at the second level we must have an active qubit for each such block of data qubits. To implement a fault-tolerant scheme for data-level error correction by simulating a circuit using concatenated codes will therefore require a GCA architecture that can support the activation and deactivation of pointers for every k^i data qubits, for *each* i in the range $1 \leq i \leq l$. In principle, such a system might be implemented by techniques such as those suggested in [BBK04], employing some complicated scheme of "labeled switching stations". To my knowledge, no detailed scheme has been described for doing this on a particular architecture, and it does not seem clear that a workable scheme exists.

3.7 Unresolved problems

A weakness of data-level error correction in GCA is particularly evident for schemes like that of [Ben00], for which each data qubit is encoded by several lattice qubits. Suppose we use data-level error correction to simulate a fault-tolerant circuit that protects against arbitrary single-qubit errors. For architectures like that of [Ben00], these correspond to errors on the states of the *data qubits* in the array, which are themselves encoded by several physical qubits in the underlying lattice. Suppose each data qubit is represented by k physical qubits in the underlying lattice. The state space $\mathcal{H}_p$ for the block of k physical qubits has dimension 2^k. A GCA scheme is designed so that the state of the k physical qubits will be constrained to a 2-dimensional subspace $\mathcal{H}_c$, the *computation subspace.* For example, in [Ben00], each data qubit is represented on $k = 4$ adjacent lattice qubits. The

computation subspace of the 2^4-dimensional physical state space is span$\{|1100\rangle, |0011\rangle\}$.

When we simulate an error correcting circuit that is designed to correct arbitrary errors on single qubits, in the GCA model these correspond to errors on the state in $\mathcal{H}_c$. That is, they are errors that transform a data qubit state ρ_i to a new data qubit state ρ'_i. For the error correction circuit being simulated, the assumption is that ρ_i and ρ'_i both describe states in the computation space $\mathcal{H}_c$. But for the GCA, ρ_i and ρ'_i actually describe the states of k physical qubits which live in the larger space $\mathcal{H}_p$. A more realistic error model would account for the possibility of individual errors on one or more of the k physical qubits. The problem is that such errors may not leave the state of the k qubits in the computation space $\mathcal{H}_c$. If the corrupted state ρ' is in $\mathcal{H}_p \setminus \mathcal{H}_c$, the error correction circuit simulation will not be effective.

For other architectures, like that of [Llo99] or architectures using a distinguished site, each data qubit is encoded by a single lattice qubit. This approach resolves the problem described above. However, other problems remain. For such architectures, the data qubits are typically separated on the lattice by some number of qubits of padding in the state $|0\rangle$. Also, for fault tolerance we will need to encode several pointers as well as some mechanism like labeled switching stations for activating and deactivating these pointers (as discussed in Section 3.6.3). Data-level error correction *assumes* the stability of these pointers, switching stations and padding, and cannot itself be used to achieve stability for them. One approach that has been proposed ([Kay07]) for stabilizing these begins with the observation that they are implicitly *classical* states. For pointers and switching stations, most of the time (that is, except during operations for moving a pointer, or applying 2-qubit gates) they can be assumed to reside on qubits of species distinct from those containing the data (e.g. the B- and C-qubits in [Llo99]). So we could periodically apply pulses to implement measurements of these in the computational basis. This is itself an implementation-level approach to error correction for the pointers and switching stations. The problem is that there will still be a fixed minimum time between successive applications of these measurements (dictated by the time required to move the pointer past a data qubit, or to perform a 2-bit gate). During this fixed time, there will be some fixed probability that a bit within a pointer, a (labeled) switching station or a bit of padding will become flipped. The implementation-level technique will not reduce this fixed probability.

One solution that has been proposed for the pointers [Kay07] is to encode them with some classical redundancy. However, doing so will make the pointers larger thus increasing the fixed time mentioned above during which errors may occur. No such scheme has been completely specified, and it is not clear whether one could be designed in a way that allows us to arbitrarily reduce the probability of an unrecoverable error on any pointer, switching station, or padding qubit (all of which would be required for full fault tolerance).

3.8 Conclusions and future work

I have described two kinds of approaches that can be taken for implementing error correction in quantum global control schemes. One approach implements codes at the physical level, giving a more robust realization of the GCA model. These techniques are always specific to a particular GCA architecture, and scaling them to achieve reduced error rates generally requires redesigning the architecture. They are effective for protecting lattice states against errors on physical qubits, but the difficulty in scaling these techniques poses a problem for fault tolerance.

The second approach that I described was to simulate fault-tolerant quantum circuits by a GCA program. I proposed a language for programming such simulations in a manner that is abstracted from the details of how the GCA scheme is implemented in hardware. These approaches have the strength that they are naturally scalable and are independent of the underlying physical architecture, but the weakness that they are not effective in general at protecting against errors on the physical lattice qubits.

To achieve fault tolerance under a realistic error model for GCA seems to require some new technique, or some novel way of combining the techniques described in this chapter. This is an important direction for future work. Other directions are to consider ways of making more robust implementations of `MPA`(k), perhaps using an approach similar to the one described in Section 3.4.6.2. Also, complete and detailed implementations of some variant of `MPA`(k) for supporting logical code-concatenation would be nice, including optimized schemes for implementing the labeled switching station concept.

Another direction for future work is to develop more systematic techniques for optimizing

GCA programs for simulating given quantum circuits.

Chapter 4

Cooling algorithms based on the 3-bit majority

4.1 Background

Consider a probabilistic bit that equals 0 with probability p. Define the *bias* of the bit to be

$$\mathcal{B} = p - (1-p) = 2p - 1,$$

which is the difference between the probability that the bit equals 0 and the probability that the bit equals 1. (The symbol "ε" is usually used to denote the bias in the literature on algorithmic cooling; I prefer to reserve this symbol for error rates.) For quantum bits represented by nuclear spins (such as in quantum computing with nuclear magnetic resonance), the bias corresponds to the spin "polarization". The problem addressed by algorithmic cooling is the following. Given some number of bits initially having a common bias $\mathcal{B}_{\mathrm{i}} > 0$, distill out some smaller number of bits having greater bias. This should be accomplished without the need for any pure ancillary bits initialized to 0, since preparing such initialized bits is the problem to be solved. Also, we should assume that we cannot perform projective measurements.

Algorithmic cooling has significant relevance to quantum computing, because for physical

systems like nuclear spins controlled using nuclear magnetic resonance (NMR), obtaining a pure initial state can be very challenging. It is this fact that has motivated recent research on the implementation of algorithmic cooling in NMR quantum computers, as well as theoretical investigations of the efficiency and performance of cooling algorithms.

Algorithmic cooling in the context of NMR quantum computation first appeared in [SV99]. The authors presented a method for implementing *reversible polarization compression* (RPC). The idea of RPC is to use reversible logic to implement a permutation on the (classical) states of n bits, so that the bias of some of the bits is increased, while the bias of others is decreased. Unfortunately RPC is theoretically limited by Shannon's Bound, which says that the entropy of a closed system cannot decrease [1]. According to Shannon's bound, an RPC operation on 3 bits with initial bias $\mathcal{B}_{\mathrm{i}}$ cannot yield a bit with bias higher than $1.5\mathcal{B}_{\mathrm{i}}$ [RMBL07].

An alternative algorithm was proposed in [BMR+02] to enable cooling below the Shannon bound. The idea is to use a second register of bits that quickly *relax* to the initial bias $\mathcal{B}_{\mathrm{i}}$. These are called the *relaxation bits*, and the bits on which we perform the RPC operation are referred to as the *compression bits*. First, RPC is used to increase the bias of some of the compression bits, while decreasing the bias of the other compression bits. Then the hotter compression bits (i.e. those having decreased bias) are swapped with the relaxation bits, where they will quickly relax back to the initial bias $\mathcal{B}_{\mathrm{i}}$. Repeating this procedure effectively pumps heat out of the some of the compression bits, cooling them to a bias much higher than $\mathcal{B}_{\mathrm{i}}$. This system is analogous to a kitchen refrigerator, where the relaxation bits

[1] The bias can be viewed as a measure of how random the state of a bit is. For more general random variables, the notion of randomness is captured by the *Shannon entropy*. Consider a binary string X of n probabilistic bits, each identically distributed with a probability p of being in the state 0. The entropy of of the string is defined to be $H(X) = -\sum_{\sigma\in\{0,1\}^n} p(\sigma)\log_2 p(\sigma)$, where $p(\sigma)$ is the probability of the state of the string being σ. Roughly speaking, the basic goal of data compression is to transform the bitstring X into a new bitstring Y_1Y_2, where Y_1 is some k-bit substring having entropy $H(X)$, and Y_2 is an $(2^n - k)$-bit substring having entropy 0. Since all the entropy is in the string Y_1, Y_2 is the substring that will be "cooled" by the data compression (these are the bits we are interested in for polarization compression). So we want to make k as small as possible, to yield the greatest number of cooled bits. A famous theorem in information theory, due to Shannon [Sha49], implies that k must be larger than the entropy, $H(X)$.

behave like the radiator on the back of the refrigerator, dumping the heat taken from the refrigerator compartment out into the surrounding environment. This approach is often referred to as "heat-bath algorithmic cooling", and the relaxation bits are often referred to as the "heat bath".

Another approach to heat-bath algorithmic cooling was introduced in [FLMR04]. Their algorithm has a simpler analysis than the algorithm in [BMR+02], and gives a better bound on the size of molecule required to cool a single bit.

In [SMW05], the physical limits of heat-bath cooling are explored. In their analysis, the assumption is that the basic operations can be implemented perfectly, without errors. Even given this assumption, the authors show that if the heat bath temperature is above a certain threshold, no cooling procedure can initialize the system sufficiently for quantum computation.[2] A heat-bath cooling algorithm called the "partner pairing algorithm" (PPA) is introduced to derive bounds on the best possible performance of algorithmic cooling with a heat bath. The PPA performs better than the previous algorithms, but it is unclear whether implementing the required permutations will be realistic in practice. In this chapter, I will focus on cooling algorithms based on repeated application of simple 2- or 3-bit RPC steps. Note that when restricted to 2 or 3 bits, the PPA actually performs the same operation as RPC.

4.2 Architecture

To be useful for NMR quantum computing, we should implement cooling algorithms on a register of quantum bits all having some initial bias $\mathcal{B}_{\mathrm{i}}$, without access to any clean ancillary bits. Further, we should be careful about how much local control we assume is directly provided by the system. In [SV98], four primitive computational operations are proposed as being supported by NMR quantum computers. For implementing the cooling algorithm, the first two of these suffice:

o_1) Cyclically shift the n bits left or right one position.

[2]Specifically, if $\mathcal{B} < 2^k$, then starting with k bits we cannot get a sufficiently-cooled single bit.

o_2) Apply an arbitrary two-bit operation to the first two bits (i.e. to the bits under a fixed "tape-head").

To implement the two operations, [SV98] suggest using a repeating polymer like the ABC-chains used for global control schemes (e.g. [Llo93]). The chain could be configured as a closed loop. To mark the position of the "first two bits" of the chain (for operation o_2), an atom of a fourth type, D, is positioned adjacent to the chain, in the desired location.

Notice that a system supporting operations o_1 and o_2 above can be rephrased in terms of a fixed tape containing the bit-string, and a moving head that can be positioned over any pair of adjacent tape cells. In [SV99] an architecture is proposed that uses a repeating polymer with 8 species to implement a system having four such tapes. A rather complicated scheme for implementing the cooling algorithm is described for this four-tape machine.

The cooling algorithms use (classical reversible) 3-bit operations: generalized Toffoli gates (from which controlled-SWAP operations can be implemented).[3] Without access to ancillary bits, the Toffoli cannot be implemented by classical 2-bit gates (CNOT and NOT gates)[4] This can be seen by noting that CNOT and NOT gates on a 3-bit register generate the group of even permutations of the states, whereas the Toffoli implements an odd permutation. The Toffoli can be implemented without ancilla *if* we also have access to arbitrary single-qubit quantum gates [BBC+95]. So to implement the cooling algorithms using operations o_1 and o_2 would require that some of these be inherently quantum operations. An error analysis of the cooling algorithms is greatly simplified if we assume it has a classical implementation, however. Fortunately, ABC-chains naturally support generalized Toffoli operations directly, since the transition frequency of one species will be affected by the states of the neighbouring bits of two other species.

It is worth revisiting the idea put forth in [SV98], to use an ABC-chain. I propose an alternative set of operations that should be supported (these are sufficient for cooling, although obviously not for universal quantum computing):

[3]By "generalized Toffoli" I mean any 3-bit gate that applies a NOT operation to one of the bits conditioned on a specific pattern of the basis states of the other two bits.

[4]By "classical", I mean gates that do not generate nontrivial superpositions given basis states as inputs, and that do not affect the phase.

o'_1) Move any three bits into adjacent positions under a fixed "tape head" (which covers three bits).

o'_2) Apply any generalized Toffoli or CNOT operation to the bits under the tape head.

Using the scheme described in Section 3.2.3, o'_1 and o'_2 can be implemented on an ABC-chain which is configured as a closed loop.[5] An atom of a fourth type, D is positioned adjacent to some ABC-triple selected (arbitrarily) to be the position of the tape head.

The cooling algorithms work by moving some bits under the tape head and applying a basic (2-bit or 3-bit) RPC step. The resulting cooler bits are then moved to one side of the array (tape), while the hotter bits are moved to the other side. The RPC step is repeated to cool several bits, and then recursively applied to these cooled bits.

4.3 The reversible polarization compression step

Assume that the initial configuration is some string of bits, each of which is (independently) in state 0 with some probability $p > 0$. Equivalently, the bits all have an identical bias $\mathcal{B} > 0$ before applying the polarization compression step. The assumption of independence (i.e. a binomial distribution on the strings) is required for the analysis.[6] Algorithmic cooling only amplifies an existing bias and hence the initial bias $\mathcal{B}$ must be positive.

The basic idea behind RPC is to implement a permutation that maps strings with low Hamming weight (i.e. having many 0's) to strings having a long prefix of 0's. Because it will be useful to implement cooling algorithms on systems for which we don't have arbitrary local control, we will construct RPC permutations based on basic "RPC steps". An RPC step will be a permutation on the states of a small number of bits (2 or 3 in the examples I consider). The overall system will be cooled by recursively applying the basic RPC step

[5] We could alternatively use a linear configuration, but would then have to be careful about the behaviour at the ends of the chain. One approach would be to have the chain be long enough so that the bits of interest are sufficiently far into the interior of the chain that the behaviour the ends is irrelevant. Alternatively, the bit at one end of the chain could serve as the position of the tape-head.

[6] In [SV99] it is suggested that by performing an initial permutation of the bits we can limit our reliance on the assumption of independence.

to all the bits. If we apply the RPC step to disjoint pairs or triples of bits at each stage, the assumption of independence will hold throughout.

In the following sections, I will examine candidates for the RPC step and discuss how they may be implemented.

4.3.1 The 2-bit RPC step

The algorithms described in [SV99] and [BMR+02] both use a very simple 2-bit operation for the basic RPC step. The operation begins with a CNOT gate. Suppose the CNOT is applied to two bits initially having some positive bias $\mathcal{B}$. After the CNOT, the target bit is 0 if both bits were originally equal, and is 1 if both bits were originally different. In the case that they were both the same, the control bit has an amplified bias after the CNOT. So, conditioned on the outcome of the target bit, the control bit is either accepted as a new bit with higher bias and is subsequently moved to the colder side of the array by a sequence of controlled-SWAP operations, or it is rejected and subsequently moved to the warmer side of the array. For specificity, I will refer to this 2-bit RPC step as “2BC”.

Suppose the values of the control and target bits before the CNOT are b_c and b_t respectively. Then, after the CNOT, the value of the target bit is $b_c + b_t$. The control bit is accepted if and only if this value equals 0. The probability that $b_c = 0$, given that $b_c + b_t = 0$, is

$$\frac{P(b_c = 0 \wedge b_t = 0)}{P(b_c + b_t = 0)} \tag{4.1}$$

$$= \frac{1}{2} + \frac{\mathcal{B}}{1 + \mathcal{B}^2} \tag{4.2}$$

and so in this case the bias of the control bit is

$$\mathcal{B}' = \frac{2\mathcal{B}}{1 + \mathcal{B}^2}. \tag{4.3}$$

The probability that the control bit is accepted equals the probability that $b_c + b_t = 0$, which is

$$\frac{1 + \mathcal{B}^2}{2}. \tag{4.4}$$

If the control bit is rejected, it has bias 0. To achieve the polarization compression, the CNOT must be followed by an operation that selects the accepted bits to be retained. This

is accomplished for the 2BC step by controlled-SWAP operations that move the bit to the left or right according to whether it was accepted or rejected.

A cooling algorithm can work by recursive application of the 2BC step across many bits having an initial bias $\mathcal{B}_{\mathrm{i}}$. First some of the bits will be cooled by one application of 2BC, while others are warmed. The cooled bits will be moved away from the warmed bits, and then cooled further by another application of 2BC, and so on. The total number of starting bits required is determined by the depth of recursion required to obtain a single bit cooled to the desired target bias.

4.3.2 The 3-bit RPC step

The algorithm described in [FLMR04] uses a 3-bit reversible polarization compression step (3BC). This RPC step is implemented by a permutation of the basis states of a 3-bit register, and has the effect of increasing the bias of the one of the bits, while decreasing the bias of the other two. Experimental demonstration of the 3-bit RPC step has been conducted using NMR [BMR+05]. The implementation of the 3BC operation given in [FLMR04] uses a CNOT gate followed by a controlled-SWAP gate. Recall from our discussion in Section 4.2 that we are assuming that the bits have already been moved onto an *ABC*-triple under the "tape head", and that we can implement any reversible 3-bit (classical) operation on them. The quantum circuit model is a convenient paradigm for describing the operations[7]. Note that the controlled-SWAP can be implemented by generalized Toffoli operations, as shown in Figure 4.1. (Approaches for implementing such generalized Toffoli gates on *ABC*-chains are described for example in [Llo93] and [Ben00].)

The permutation implemented by the circuit in Figure 4.1 results in the majority value of the three bits (before the operation) being computed into bit A. Since we are only interested in the final bias of bit A, we can use any permutation that has this effect. In fact, the following claim says that such a permutation is the best choice for a 3-bit RPC

[7] Current NMR experiments in algorithmic cooling [BMR+05] do not implement the 3-bit permutation through a decomposition into a sequence of gates such as we consider here, but rather use a more direct method. This direct method is not scalable in the number of bits over which the majority is being computed.

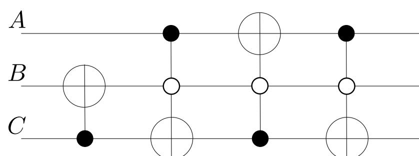


Figure 4.1: A circuit for the 3BC step using CNOT gates and generalized Toffoli gates. The small solid black circles refer to a 1-control (i.e. the gate is conditioned on that bit being in the state 1), and the solid white circles refer to a 0-control.

step.

Claim 4.3.1 *Suppose we have a register of n bits independently having identical bias $\mathcal{B} > 0$, where n is odd. Suppose we want to implement a permutation that has the effect of increasing the bias of the first bit as much as possible. Then the best choice is a permutation that computes the majority value of the n bits into the first bit.*

> **Proof:** Since each bit has bias $\mathcal{B} > 0$, each bit is independently 0 with probability $p > \frac{1}{2}$. An optimal permutation for increasing the bias of the first bit will be one that maps the $\frac{2^n}{2}$ most likely strings to strings having a 0 in the first bit. The $\frac{2^n}{2}$ most likely strings are precisely those having at least $\lceil \frac{n}{2} \rceil$ bits in the state 0. $\square$

The circuit is shown in Figure 4.2 is an alternative implementation of the 3-bit majority, which is simpler in terms of Toffoli and CNOT operations. I will henceforth refer to the operation implemented by this circuit as 3BC. Note that the circuit of Figure 4.2 implements a different permutation than that implemented by the circuit of Figure 4.1, but the effect on bit A (i.e. after tracing-out bits B and C) is the same for both circuits (assuming the input bits are independently distributed).

Since the Toffoli and CNOT operations are classical, we can analyze the behaviour of the 3BC circuit entirely in the computational basis. In the following, I will restrict the analysis in terms of classical bits.

Consider the effect on the bias of bit A after applying the circuit of Figure 4.2. The majority value is computed into bit A. Suppose initially the bias of each of the three

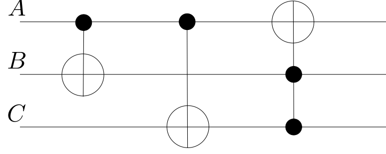


Figure 4.2: An alternative circuit for computing the majority of three bits that can be used for the 3BC operation.

bits is $\mathcal{B}$. So the probability for each bit equaling 0 is initially $(1+\mathcal{B})/2$. After the 3BC operation, the probability that bit A (which now equals the majority of the initial values of A, B, C) equals 0 is

$$p^{(A)} = \left(\frac{1+\mathcal{B}}{2}\right)^3 + 3\left(\frac{1+\mathcal{B}}{2}\right)^2\left(\frac{1-\mathcal{B}}{2}\right) \tag{4.5}$$

$$= \frac{1}{4}(2 + 3\mathcal{B} - \mathcal{B}^3). \tag{4.6}$$

So the bias of bit A after the 3BC operation is

$$\mathcal{B}' = 2p^{(A)} - 1 \tag{4.7}$$

$$= \frac{3}{2}\mathcal{B} - \frac{1}{2}\mathcal{B}^3. \tag{4.8}$$

4.3.3 Equivalence between the 2BC and 3BC operations

Recall that 2BC is a CNOT followed by controlled-SWAP operations that move the control bit (of the CNOT) to the left or right, conditioned on the state of the target bit. The CNOT itself has no effect on the bias of the control bit. It is the value of the target bit after the CNOT that provides some information about the state of the control bit. In the case that the target bit equals 0, the control bit is more likely to be 0, and hence has a greater bias. So the 2BC step is really a method for gaining some information about which bits are more likely to be 0, and moving these off to one side. After a single application of 2BC on two bits having equal bias, we may or may not be left with a bit having increased bias.

The 3BC step, on the other hand, deterministically increases the bias of the third bit at the expense of decreasing the bias of the other two. Every time we apply the 3BC step to three

bits having equal bias, we are certain to be left with a bit whose bias has been increased. This property makes the analysis of algorithms based on 3BC somewhat simpler than those based on 2BC. The analysis of the 2BC-based heat-bath algorithm in [BMR+02] relies on the law of large numbers, and gives a worse bound than does the analysis of the 3BC-based algorithm of [FLMR04].

In the algorithms of [SV99] and [BMR+02], the CNOT of the 2BC step is always followed by a controlled-SWAP operation. An important observation is that the CNOT followed by a controlled-SWAP actually computes the 3-bit majority (indeed this is the way the 3BC step was implemented in [FLMR04]). Specifically, suppose we first apply a CNOT between bits in states b_1 and b_2 (with b_1 as the control bit), and then apply a controlled-SWAP between b_1 and a third bit in state c, controlled on the target bit of the CNOT being 0. The final state of c is

$$b_1c + b_2c + b_1b_2 \tag{4.9}$$

which is the majority of b_1, b_2, c. So if we explicitly account for the extra target bit of the controlled-SWAP operation, the 2BC step is equivalent to the 3BC step.

This suggests an equivalence between the early algorithms described in terms of a 2BC operation and algorithms phrased in terms of a 3-bit majority-vote (3BC). For this reason, in the following I will restrict attention to algorithms based on the 3BC operation.

4.4 Efficiency

4.4.1 The simple recursive algorithm

We will analyze the efficiency of a simple algorithm that recursively partitions the string of bits into triplets and applies 3BC to these triplets. After each 3BC step (say on bits A, B, C), the B and C bits which become heated are discarded. Thus at each level of recursion the total number of bits is reduced by a factor of 3, and the remaining bits' bias is increased from $\mathcal{B}$ to a new value

$$\mathcal{B}' = \frac{3}{2}\mathcal{B} - \frac{1}{2}\mathcal{B}^3. \tag{4.10}$$

To simplify the analysis we will approximate $\mathcal{B}'$ by

$$\mathcal{B}' \approx \frac{3}{2}\mathcal{B}. \tag{4.11}$$

After k levels of recursion the bias is increased to

$$\mathcal{B}_k \approx \left(\frac{3}{2}\right)^k \mathcal{B}. \tag{4.12}$$

This gives us an estimate on the number of levels of recursion k required to achieve some target bias $\mathcal{B}_t < 1$ on a single bit.

$$k \approx \log_{3/2}\left(\frac{\mathcal{B}_t}{\mathcal{B}}\right). \tag{4.13}$$

Therefore the total number of bits starting at bias $\mathcal{B}$ required to obtain one bit with a target bias of $\mathcal{B}_t$ is 3^k, which is polynomial in $\mathcal{B}_t$. For example, suppose we start with a bias of $\mathcal{B} = 10^{-5}$ (see [M05]). Then the number of bits required to yield a single bit with bias 0.1 is about 6.9×10^{10}, and the number required to yield a bit with bias 0.9999 is about 3.5×10^{13}.

This has only been an analysis of the space complexity. To obtain a good estimate of the time complexity, we would have to specify the computational model more precisely, and account for the time required to shuttle the states around as required by the architecture and the algorithm.

4.4.2 Algorithms using a heat bath

There are many ways in which the recursive algorithm might be modified to take advantage of a *heat bath*, which is a mechanism by which a heated bit can be exchanged for a fresh bit having initial bias $\mathcal{B}_\mathrm{i}$ (taken from the environment). For a rough analysis, I ignore the details of how the heat-bath contact will be implemented, and assume we can apply an operation that resets a bit's bias to $\mathcal{B}_\mathrm{i}$ on-demand (this may be an unrealistically optimistic assumption, since it is likely that the heat-bath exists on specific physical qubits, and that states will have to be shuttled to this location before they can be reset).

One approach to using the heat bath in a 3BC algorithm is the following. First apply the 3BC step as in the simple recursive algorithm. At this point we have $\lfloor n/3 \rfloor$ bits cooled to $\mathcal{B}'$. Now, instead of discarding the $\lfloor 2n/3 \rfloor$ bits that were heated in this process, send them to the heat bath to return them to bias $\mathcal{B}_{\mathrm{i}}$. Then partition these $\lfloor 2n/3 \rfloor$ bits into triples, and apply the 3BC step to them. This yields another $\lfloor 2n/9 \rfloor$ bits of bias $\mathcal{B}'$. Repeat this process until there are fewer than 3 bits left having bias less than $\mathcal{B}'$ (there will always be exactly 2 bits left over). Now we have $n-2$ bits cooled to bias $\mathcal{B}'$ and we can proceed to the next level of recursion. As before, the number of levels of recursion k required to achieve a bit having some target bias $\mathcal{B}_t < 1$ is

$$k \approx \log_{3/2}\left(\frac{\mathcal{B}_t}{\mathcal{B}_{\mathrm{i}}}\right). \tag{4.14}$$

This time, however, a logarithmic amount additional work is done for each level of recursion. By taking this extra time, we save on the total number of bits required. After each level of recursion an additional 2 bits are discarded. So the total number of bits required to obtain one bit cooled to $\mathcal{B}_t$ by this method is $2k$, which is polylogarithmic in $\mathcal{B}_t$. As before, suppose we start with a bias of $\mathcal{B}_{\mathrm{i}} = 10^{-5}$. Now the number of bits required to yield a single bit with bias 0.1 is about 46, and the number required to yield a bit with bias 0.9999 is about 57.

Another approach to using the heat bath is described in [SMW07]. Their algorithm repeatedly applies the 3BC step to three bits having bias values $\mathcal{B}_{j-2}, \mathcal{B}_{j-1}$ and $\mathcal{B}_j$. This requires a more careful analysis. Consider applying 3BC to three bits b_{j-2}, b_{j-1}, b_j having initial bias values $\mathcal{B}_{j-2}, \mathcal{B}_{j-1}$ and $\mathcal{B}_j$ respectively, where the majority is computed into the third bit b_j. The resulting bias of the third bit is

$$\mathcal{B}'_j = \frac{\mathcal{B}_{j-2} + \mathcal{B}_{j-1} + \mathcal{B}_j - \mathcal{B}_{j-2}\mathcal{B}_{j-1}\mathcal{B}_j}{2}. \tag{4.15}$$

Now suppose the first two bits are sent to the heat bath, and then run back through the cooling procedure to regain bias values of $\mathcal{B}_{j-2}$ and $\mathcal{B}_{j-1}$. Then 3BC is applied again (on the same three bits, except this time the third bit starts with bias $\mathcal{B}'_j$). If this process is repeated several times, the bias of the third bit reaches a steady-state value of

$$\frac{\mathcal{B}_{j-2} + \mathcal{B}_{j-1}}{1 + \mathcal{B}_{j-2}\mathcal{B}_{j-1}}. \tag{4.16}$$

The algorithm described by [SMW07] is based on this process. Suppose the algorithm has built up an array of $k > 3$ cooled bits $b_1, b_2 \ldots, b_k$ having bias values $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k$ in that order, where $\mathcal{B}_j = \frac{\mathcal{B}_{j-2}+\mathcal{B}_{j-1}}{1+\mathcal{B}_{j-2}\mathcal{B}_{j-1}}$ for each $3 < j < k$. Then, in the next stage of the algorithm, a new bit b_{k+1} is introduced having the heat-bath bias $\mathcal{B}_0$. The 3BC procedure is applied to the three bits b_{k-1}, b_k, b_{k+1} repeatedly, where between each application the algorithm is recursively repeated to re-establish the bias values $\mathcal{B}_{k-1}, \mathcal{B}_k$ on bits b_{k-1}, b_k. Repeating this process several times the bias of bit b_{k+1} will reach the steady state value $\mathcal{B}_{k+1} = \frac{\mathcal{B}_{k-1}+\mathcal{B}_k}{1+\mathcal{B}_{k-1}\mathcal{B}_k}$. Starting with n bits of bias $\mathcal{B}_{\mathrm{i}}$, the algorithm of [SMW07] starts at the left side of the register ($k = 2$), and repeatedly performs the above operation for increasing values of k. The algorithm achieves one bit of bias approximately $\mathcal{B}_n = \mathcal{B}_{\mathrm{i}} F(n)$, where $F(n)$ is the nth Fibonacci number. This is even better than the simple recursive heat-bath method described previously. Starting with a bias of $\mathcal{B}_{\mathrm{i}} = 10^{-5}$, the number of bits required for this method to yield a single bit with bias 0.1 is about 20, and the number required to yield a bit with bias 0.9999 is about 28. There is a polynomial cost in time incurred by the recursive re-cooling of bits from the point of heat-bath contact at the left end of the chain.

For the heat-bath algorithms I have described, after a 3BC operation the two bits that have become heated by this operation are both sent to the heat bath. In the early stages of an algorithm, this would be sensible, because the 3BC operation will have warmed those two bits to bias values less than the initial bias $\mathcal{B}_{\mathrm{i}}$. Towards the end of the algorithm, however, 3BC will be applied to triples of bits that are all very cold, and the bits that become heated may still have bias considerably higher than $\mathcal{B}_{\mathrm{i}}$. In this case, sending these bits to the heat bath would not be the right thing to do. To analyze the performance of the algorithms, however, it is extremely convenient to assume we always do so. If we do not send the two heated bits back to the heat bath after a 3BC application, the bits' values are no longer described by independent probability distributions, and bias values are no longer well-defined. It is convenient to model the process of a 3BC application followed by sending the two heated bits to the heat bath as a single operation, as follows.

Definition 4.4.1 *Consider three bits b_1, b_2, b_3 having bias values $\mathcal{B}_1 \le \mathcal{B}_2 \le \mathcal{B}_3$ respectively. Define* $3\mathrm{BC}_{\mathrm{hb}}$ *as the 3-bit majority on b_1, b_2, b_3 (where the majority is computed*

into b_3) followed by sending b_1 and b_2 to the heat bath. The bias values of the three bits after this operation are $\mathcal{B}_\mathrm{i}, \mathcal{B}_\mathrm{i}, \frac{\mathcal{B}_1+\mathcal{B}_2+\mathcal{B}_3-\mathcal{B}_1\mathcal{B}_2\mathcal{B}_3}{2}$ respectively.

The heat-bath algorithms described above can both be expressed as a sequence of operations $(3\mathrm{BC}_\mathrm{hb}, P_1, 3\mathrm{BC}_\mathrm{hb}, P_2, 3\mathrm{BC}_\mathrm{hb}, P_3, \ldots)$ where each $3\mathrm{BC}_\mathrm{hb}$ is applied to three bits in some specific positions (e.g. under a tape head), and each P_i is some permutation of the positions of the bits in the string. The following claim shows that the algorithm of [SMW07] is the best such algorithm (this is not claimed in [SMW07]).

Claim 4.4.1 *Consider a string of bits each having initial bias value $\mathcal{B}_i$. Let $\mathcal{A}$ be any cooling algorithm described by a sequence of operations*

$$3\mathrm{BC}_\mathrm{hb}, P_1, 3\mathrm{BC}_\mathrm{hb}, P_2, 3\mathrm{BC}_\mathrm{hb}, P_3, \ldots$$

where each $3\mathrm{BC}_\mathrm{hb}$ is applied to three bits in some specific positions (e.g. under a "tape head"), and each P_i is some permutation of the positions of the bits in the string. At any stage of the algorithm, suppose we arrange the bits in a nondecreasing order of their bias values $\mathcal{B}_1, \ldots \mathcal{B}_N$. Then we have $B_j \leq \mathcal{B}_\mathrm{i} F_j$ for all $1 \leq j \leq N$, where F_j is the jth Fibonacci number.

Proof: The proof is by induction. The claim is initially true (before starting the algorithm) by assumption. Since the only operation allowed in $\mathcal{A}$ that changes the bias values is the $3\mathrm{BC}_\mathrm{hb}$ operation, it suffices to show that after an arbitrary $3\mathrm{BC}_\mathrm{hb}$ operation the claim is still true. Suppose the ordered bias values before the 3BC operation are

$$\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_N.$$

Then suppose the $3\mathrm{BC}_\mathrm{hb}$ operation is applied to any three bits, suppose those having bias values $\mathcal{B}_k, \mathcal{B}_l$ and $\mathcal{B}_m$, where $k < l < m$. We assume that after the 3BC operation the value of $\mathcal{B}_m$ is not decreased. This is a safe assumption, because otherwise algorithm $\mathcal{A}$ would have done just as well not to apply that 3BC operation.

After the 3BC_{hb} operation, the new bias of the bit originally indexed by m is

$$\mathcal{B}'_r = \frac{\mathcal{B}_k + \mathcal{B}_l + \mathcal{B}_m - \mathcal{B}_k\mathcal{B}_l\mathcal{B}_m}{2},$$

where r is an index not less than m. By assumption, we have $\mathcal{B}_m \leq \mathcal{B}_{\text{i}}F_m$, $\mathcal{B}_l \leq \mathcal{B}_{\text{i}}F_{m-1}$ and $\mathcal{B}_k \leq \mathcal{B}_{\text{i}}F_{m-2}$. Since $F_m = F_{m-1} + F_{m-2}$ by definition, we have

$$\mathcal{B}'_r \leq \mathcal{B}_{\text{i}}F_m. \tag{4.17}$$

Suppose the reordered bias values are $\mathcal{B}'_j$, for $1 \leq j \leq N$. Consider the following ranges for the index j.

$$1 \leq j \leq 2: \quad \mathcal{B}'_j = \mathcal{B}_{\text{i}} \leq \mathcal{B}_{\text{i}}F_j. \tag{4.18}$$

$$3 \leq j \leq k+1: \quad \mathcal{B}'_j = \mathcal{B}_{j-2} \leq \mathcal{B}_j \leq \mathcal{B}_{\text{i}}F_j. \tag{4.19}$$

$$k+2 \leq j \leq l: \quad \mathcal{B}'_j = \mathcal{B}_{j-1} \leq \mathcal{B}_j \leq \mathcal{B}_{\text{i}}F_j. \tag{4.20}$$

$$l+1 \leq j \leq m-1: \quad \mathcal{B}'_j = \mathcal{B}_j \leq \mathcal{B}_{\text{i}}F_j. \tag{4.21}$$

$$m \leq j \leq r: \quad \mathcal{B}'_j \leq \mathcal{B}'_r \leq \mathcal{B}_{\text{i}}F_m \leq \mathcal{B}_{\text{i}}F_j. \tag{4.22}$$

$$r+1 \leq j \leq N: \quad \mathcal{B}'_j = \mathcal{B}_j \leq \mathcal{B}_{\text{i}}F_j. \tag{4.23}$$

$$\tag{4.24}$$

This completes the proof. □

4.4.3 Accounting for the heat bath as a computational resource

The heat bath is typically modeled by a process whereby a hot bit is magically replaced by a fresh bit having the initial bias $\mathcal{B}_{\text{i}}$. Usually we would make some assumption about where the heat-bath contact occurs, for example requiring that only the bit on the end of a chain can be replaced with a fresh bit.

From a complexity theory point of view, the heat bath is a resource that should be accounted for. For modeling the physics of the situation it might be very convenient to draw a conceptual boundary between the system we are trying to cool and the heat bath, which for all practical purposes might be extremely large. Continuing our previous analogy

between heat-bath cooling and a kitchen refrigerator, if we put the refrigerator in a large enough room, we won't have to account for the fact that the room itself is gradually heated by the radiator on the back of the fridge. While heat-bath techniques appear to drastically reduce the number of bits required to achieve a target bias, it should be recognized that this hasn't come for free. The extra bits ultimately come from the heat bath. In practice, it may be very reasonable to assume we get these bits for free, since we don't have to exercise control over the heat bath the way we do with the bits directly involved in the algorithm.

4.5 Accounting for errors in an analysis of cooling

In the following sections I investigate the performance of cooling algorithms when errors can occur in the RPC step. The bounds I will derive will apply to cooling algorithms that are based on recursive application of the 3BC step, where the step is always applied to 3 bits that have been previously cooled to equal bias values. In Section 4.8 I discuss how the same approach can be applied to analyze more general algorithms based on the 3BC step. I do not account for errors that might occur between applications of the RPC steps, such as when bits are being shuttled around, or placed in an external heat bath. For this reason the bounds apply quite generally, independent of implementation details and low-level algorithmic details.

The most general way to analyze the effect of errors on a quantum circuit is to examine the effect of the errors on the density matrix of the state as it evolves through the circuit. As we observed above, the 3BC step can be implemented by classical operations, and can be analyzed entirely in the computational basis. I therefore perform the error analysis in a classical setting.

Suppose we implement the RPC operation in a system subject to errors described by a set of error patterns $\{S_j\}$. The error pattern is a record of what errors actually occurred. For each error pattern S_j we can analyze the effect by considering a new circuit containing the original RPC circuit as well as the error operations that occurred. We can then find the probability p_j that the cooled bit would be in state 0 after applying this new circuit. The

probability that the cooled bit equals 0 for the overall process is

$$p = \sum_j p_j \Pr(S_j) \tag{4.25}$$

where $\Pr(S_j)$ is the probability that error pattern S_j occurs. The new bias of the cooled bit after the process is then

$$\mathcal{B}' = 2p - 1. \tag{4.26}$$

Equivalently, we could compute the new bias $\mathcal{B}'_j$ of the cooled bit resulting from application of the RPC step for each error pattern S_j, and take a weighted sum of these bias values (weighted by the probabilities $\Pr(S_j)$):

$$\mathcal{B}' = \sum_j \mathcal{B}'_j \Pr(S_j). \tag{4.27}$$

After obtaining the new bias $\mathcal{B}'$ of the cooled bit for the overall process, we can obtain theoretical limits on the performance of the cooling algorithm by analyzing the condition

$$\mathcal{B}' > \mathcal{B} \tag{4.28}$$

where $\mathcal{B}$ is the bias before the RPC and error channel were applied (this simply says that the bias should be greater after application of the 3BC step). In practice, to analyze the inequality

$$\mathcal{B}' - \mathcal{B} > 0 \tag{4.29}$$

we study the expression $\mathcal{B}' - \mathcal{B}$, which for the error models we consider will be a quadratic or cubic polynomial in $\mathcal{B}$ (and also a function of the error rates). By studying the roots of this polynomial we can find ranges of values for the error rates for which inequality (4.29) has solutions $\mathcal{B} > 0$, and also obtain the maximum value of $\mathcal{B}$ which is a solution (this maximum value will be the maximum bias achievable by the RPC step for the given error rates).

4.6 The symmetric bit-flip channel

The first error model we will consider is the symmetric bit-flip model, under which a bit's value is flipped with probability $\varepsilon < \frac{1}{2}$ ("symmetric" in this context means that the probability of a bit-flip error is independent of the initial state of the bit).

If the bit-flip channel is applied to a bit initially having bias $\mathcal{B}$, the result is a bit with bias $-\mathcal{B}$.

4.6.1 3BC followed by a symmetric bit-flip error

Now consider the case in which a bit-flip error can occur after the 3BC step has been performed (and errors do not occur between application of the gates in Figure 4.2).

There are two error patterns. Pattern S_1 represents the case where a bit flip does not occur. In this case, the final bias of bit A is

$$\mathcal{B}_1' = \frac{3}{2}\mathcal{B} - \frac{1}{2}\mathcal{B}^3 \tag{4.30}$$

as we found in Section 4.3.2 (equation (4.8)). The error pattern S_2 represents the case where the bit flip occurs on the newly biased bit. In this case, the bias is negated, and so the new bias is

$$\mathcal{B}_2' = -\frac{3}{2}\mathcal{B} + \frac{1}{2}\mathcal{B}^3 \tag{4.31}$$

So the new bias of A for the overall process is

$$\begin{aligned} \mathcal{B}' &= (1-\varepsilon)\mathcal{B}_1' + \varepsilon\mathcal{B}_2' && (4.32)\\ &= \left(\frac{3}{2}\mathcal{B} - \frac{1}{2}\mathcal{B}^3\right)(1-2\varepsilon). && (4.33) \end{aligned}$$

Then the condition that $\mathcal{B}' > \mathcal{B}$ gives

$$-(1-2\varepsilon)\mathcal{B}^2 - 6\varepsilon + 1 > 0 \tag{4.34}$$

which leads to

$$\begin{aligned} \varepsilon &< \frac{1}{2} - \frac{1}{3-\mathcal{B}^2} && (4.35)\\ &< \frac{1}{6}. && (4.36) \end{aligned}$$

So for this simple error model $\varepsilon_{\text{th}} = 1/6$ is an error threshold beyond which the 3BC procedure can have no positive effect on the bias (regardless of how low the initial bias

is). For a fixed error rate $\varepsilon < \varepsilon_{\text{th}}$ a bound on the maximum bias that will be achievable is obtained by solving for $\mathcal{B}$ in (4.34)):

$$\mathcal{B} < \sqrt{\frac{1-6\varepsilon}{1-2\varepsilon}} = \mathcal{B}_{\text{lim}}. \tag{4.37}$$

Approximating the expression to second order gives

$$\mathcal{B}_{\text{lim}} \approx 1 - 2\varepsilon - 6\varepsilon^2. \tag{4.38}$$

Once the bias exceeds $\mathcal{B}_{\text{lim}}$, the 3BC procedure will no longer be effective in increasing the bias, and the algorithm will yield no further improvement. So $\mathcal{B}_{\text{lim}}$ represents the limit of the bias that can be achieved by any cooling algorithm that is based on the 3BC step, under this error model.

For error rates ε below 1%, the approximate value in (4.38) is within 0.01% of the value in (4.37).

4.6.2 Symmetric bit-flip errors during application of 3BC

I will now do a more careful analysis, accounting for the possibility of errors occurring during the application of the 3BC step. Consider independent bit-flip errors on each bit with probability ε, where the errors can occur at each time step; that is, immediately after the application of any gate in the circuit of Figure 4.2 (equivalently after the application of each o_2' operation). This is only one possible decomposition of the majority-vote operation into a sequence of basic operations, but it serves to illustrate the technique for analysis. A similar analysis can easily be conducted given an alternative decomposition of the majority-vote into a sequence of basic operations.

There are 9 possible sites for bit-flip errors in Figure 4.2, but two of these can be ignored (errors on the B or C bits after the final Toffoli have no effect on the final bias of the A bit). Figure 4.3 illustrates the circuit including the possible error operations. The binary variables e_i shown on the circuit are taken to be "1" if a bit-flip error occurs in that location, and "0" otherwise.

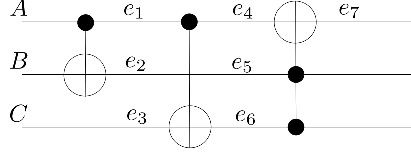


Figure 4.3: The majority circuit with relevant error positions shown. The binary variables e_i are taken to have the value 1 if a bit-flip error occurred in the relevant location (otherwise $e_i = 0$).

Suppose the value of the (A, B, C) register is initially (a, b, c), where a, b, and c are the binary values of the three bits. Analyzing the circuit in Figure 4.3, the final value of the A bit is found to be

$$a + e_1 + e_4 + e_7 + (a + b + e_2 + e_5)(a + c + e_1 + e_3 + e_6) \bmod 2. \tag{4.39}$$

Since the errors occur independently with probability ε at each position, the probability associated with each error pattern $S_i = (e_1, e_2, \ldots, e_7)$ (where $i = \sum_{k=0}^{6} e_k 2^k$ indexes the possible patterns) can be evaluated as

$$\Pr(S_i) = \varepsilon^{e_1+e_2+e_3+e_4+e_5+e_6+e_7}(1-\varepsilon)^{\bar{e}_1+\bar{e}_2+\bar{e}_3+\bar{e}_4+\bar{e}_5+\bar{e}_6+\bar{e}_7} \tag{4.40}$$

where $\bar{x} \equiv 1 + x \bmod 2$. Initially, the probability that each bit a, b or c equals 0 is $p = \frac{\mathcal{B}+1}{2}$. So the tuple $(a, b, c, e_1, \ldots, e_7)$ describes the situation where the register was initially in the state (a, b, c) and the error described by $(e_1, e_2, \ldots, e_7)$ occurred, and this happens with probability

$$\Pr(a, b, c, e_1, \ldots, e_7) \equiv (1-p)^{a+b+c} p^{\bar{a}+\bar{b}+\bar{c}} \varepsilon^{e_1+e_2+e_3+e_4+e_5+e_6+e_7}(1-\varepsilon)^{\bar{e_1}+\bar{e_2}+\bar{e_3}+\bar{e_4}+\bar{e_5}+\bar{e_6}+\bar{e_7}}. \tag{4.41}$$

Let $p^{(A)}$ be the probability that the final value of A for the overall process equals 0. The value of $p^{(A)}$ is obtained by adding the probabilities $\Pr(a, b, c, e_1, \ldots, e_7)$ over all those tuples $(a, b, c, e_1, \ldots, e_7)$ for which the value of (4.39) equals 0. The new bias of A is then determined as

$$\mathcal{B}' = 2p^{(A)} - 1. \tag{4.42}$$

This value is

$$\mathcal{B}' = (2\varepsilon - 1)^3 \left[1 + 4\varepsilon^2(\varepsilon - 1) - 4p\varepsilon(6\varepsilon^2 - 8\varepsilon + 3) - 2p^2(2p - 3)(2\varepsilon - 1)^3\right] \tag{4.43}$$

which can be expressed in terms of the original bias by substituting $p = \frac{\mathcal{B}+1}{2}$:

$$\mathcal{B}' = \frac{1}{2}\mathcal{B}(1-2\varepsilon)^3\left(3 - 6\varepsilon + 4\varepsilon^2 - \mathcal{B}^2(1-2\varepsilon)^3\right). \tag{4.44}$$

Now the condition $\mathcal{B}' > \mathcal{B}$ leads to

$$-2 + (1-2\varepsilon)^3\left(3 - 6\varepsilon + 4\varepsilon^2 - \mathcal{B}^2(1-2\varepsilon)^3\right) > 0. \tag{4.45}$$

The expression on the left side of (4.45) represents the improvement in the bias. It decreases monotonically as $\mathcal{B}$ increases from 0, and so an upper bound can be obtained by setting $\mathcal{B} = 0$. Then, by studying the real roots of the resulting polynomial in ε, we can determine the range of values for which the improvement is positive. By numerical computation, the threshold is found to be

$$\varepsilon < 0.048592 \equiv \varepsilon_{\text{th}}. \tag{4.46}$$

For a fixed $\varepsilon < \varepsilon_{\text{th}}$, inequality (4.45) also gives a bound on the maximum bias achievable by the 3BC step under the given error model:

$$\mathcal{B} < \frac{\sqrt{1 - 24\varepsilon + 76\varepsilon^2 - 120\varepsilon^3 + 96\varepsilon^4 - 32\varepsilon^5}}{(1-2\varepsilon)^3} \equiv \mathcal{B}_{\text{lim}}. \tag{4.47}$$

For small values of ε, we can approximate (4.47) to second order:

$$\mathcal{B}_{\text{lim}} \approx 1 - 6\varepsilon - 82\varepsilon^2. \tag{4.48}$$

For error rates ε below 1%, the approximate value in (4.48) is within 0.1% of the value in (4.47).

4.7 Debiasing errors

Now consider a more general error model for a classical bit. Under this error model, called the *asymmetric bit-flip channel*, a bit transforms from 0 to 1 with some probability ε_0, and transforms from 1 to 0 with some probability ε_1.

A fixed-point probability distribution for the asymmetric bit-flip channel is

$$p[0] = \frac{\varepsilon_1}{\varepsilon_0 + \varepsilon_1} \tag{4.49}$$

$$p[1] = \frac{\varepsilon_0}{\varepsilon_0 + \varepsilon_1}. \tag{4.50}$$

If left to evolve for under the asymmetric channel, a bit will eventually settle to a bias value of

$$\mathcal{B}_{\text{steady}} = \frac{\varepsilon_1 - \varepsilon_0}{\varepsilon_0 + \varepsilon_1}. \tag{4.51}$$

The rate at which the bias approaches this fixed point is related to $(\varepsilon_0 + \varepsilon_1)$.

It will be convenient to make a couple of assumptions about the error rates. First, we will assume that errors cause the system to tend back to the initial bias $\mathcal{B}_{\text{i}}$ (which would likely be the same as, or close to, the bias of the heat bath for cooling algorithms that use this device). That is,

$$\mathcal{B}_{\text{steady}} = \mathcal{B}_{\text{i}}. \tag{4.52}$$

In other words, errors cause a partial debiasing of the cooled bits (ideally, this will happen very slowly, and so the value for the sum of the error rates, $(\varepsilon_0 + \varepsilon_1)$, will be small). In the following, I will refer to this type of asymmetric bit-flip error as a *debiasing error*.

Since $\mathcal{B}_{\text{i}} > 0$, we have

$$\varepsilon_1 - \varepsilon_0 > 0. \tag{4.53}$$

We will also assume that the error rates ε_0 and ε_1 are both less than $\frac{1}{2}$. In this case we have

$$\varepsilon_1 - \varepsilon_0 < \mathcal{B}_{\text{i}}. \tag{4.54}$$

Since we assumed that the bias of the bit being cooled starts at $\mathcal{B}_{\text{i}}$ and is thereafter nondecreasing, we can say that at any stage of the algorithm we have

$$\varepsilon_1 - \varepsilon_0 < \mathcal{B} \tag{4.55}$$

where $\mathcal{B}$ is the current bias of the bits that the RPC step is being applied to.

Consider what happens to a bit initially having some bias $\mathcal{B}$ when we apply the asymmetric bit-flip channel once. A simple calculation shows the resulting bias to be

$$\mathcal{B}' = \mathcal{B}(1 - (\varepsilon_0 + \varepsilon_1)) + (\varepsilon_1 - \varepsilon_0). \tag{4.56}$$

In the following analysis, it will be convenient to make a change of variables, letting

$$s \equiv \varepsilon_0 + \varepsilon_1 \text{ , and} \tag{4.57}$$

$$d \equiv \varepsilon_1 - \varepsilon_0. \tag{4.58}$$

Then our assumptions are $s < 1$, and $0 < d < \mathcal{B}$, and equation (4.56) becomes

$$\mathcal{B}' = \mathcal{B}(1 - s) + d. \tag{4.59}$$

Notice that $d < s$ is also an obvious condition.

Consider the special case of the symmetric bit-flip channel. In this case $\mathcal{B}_{\text{steady}} = 0$, and so $\mathcal{B}_{\text{steady}} < \mathcal{B}_{\text{i}}$. This is why we obtained positive threshold error rates for the RPC step to increase the bias. Now, under our assumption $\mathcal{B}_{\text{steady}} = \mathcal{B}_{\text{i}}$, we will not obtain such a threshold. Even with high error rates (fast debiasing) the RPC step will increase the bias above $\mathcal{B}_{\text{i}}$ by some positive amount.

It is still important to analyze the effect of these errors on the RPC step, because they will imply a limiting value on the highest bias achievable. The RPC step tends to increase the bias away from the value $\mathcal{B}_{\text{i}} = d/s$, while the errors tend to force the bias back towards $\mathcal{B}_{\text{i}}$. The maximum achievable value of $\mathcal{B}$ will be determined by d and s, or equivalently, by $\mathcal{B}_{\text{i}}$ and s. The parameter s can be seen as a measure of the rate at which the errors force the bias towards the initial value $\mathcal{B}_{\text{i}}$. Thus the maximum achievable bias is limited by the initial bias, and by the rate at which errors cause the system to tend back to the initial bias.

4.7.1 3BC followed by a debiasing error

Consider the scenario in which a debiasing error may occur immediately after the 3BC operation. The bound obtained here will apply regardless of how the 3BC step is implemented. Assuming all three bits initially start with bias $\mathcal{B}$, the bias of bit A after the

process (the 3BC circuit followed by a debiasing error) is

$$\mathcal{B}' = \left(\frac{3}{2}\mathcal{B} - \frac{1}{2}\mathcal{B}^3\right)(1-s) + d. \tag{4.60}$$

The condition that $B' > B$ leads to

$$\mathcal{B}^3(s-1) + \mathcal{B}(1-3s) + 2d > 0. \tag{4.61}$$

For values of $s < 1/3$ (recall the threshold condition $\varepsilon < 1/6$ we obtained in Section 4.6.1) and for $d < s$, the cubic polynomial on the left-hand side of (4.61) has one positive real root (and the value of this root will be less than 1). A positive value of $\mathcal{B}$ will satisfy inequality (4.61) only if it is less than the value of this root. That is,

$$\mathcal{B} < \frac{i\left(-3(\sqrt{3}-i)(s-1)(3s-1) + (\sqrt{3}+i)(-27d(s-1)^2 + \sqrt{729d^2(s-1)^4 + (-3+12s-9s^2)^3})^{\frac{2}{3}}\right)}{6(s-1)\left(-27d(s-1)^2 + \sqrt{729d^2(s-1)^4 + (-3+12s-9s^2)^3}\right)^{\frac{1}{3}}}. \tag{4.62}$$

The appearance of nonreal numbers in (4.62) is unavoidable[8]. To second order in d and s, (4.62) gives

$$\mathcal{B}_{\text{lim}} \approx 1 - s + d - \frac{3}{2}s^2 - \frac{3}{2}d^2 + 3ds. \tag{4.63}$$

In the symmetric case, the bound (4.63) agrees with the bound (4.38) which we found in Section 4.6.1.

In terms of s and $\mathcal{B}_{\text{i}}$, (4.63) is

$$\mathcal{B}_{\text{lim}} \approx 1 - s - \frac{3}{2}s^2 + \mathcal{B}_{\text{i}}s + 3\mathcal{B}_{\text{i}}s^2 - \frac{3}{2}\mathcal{B}_{\text{i}}^2 s^2. \tag{4.64}$$

For error rates less than 1%, the approximate value (4.64) agrees with the actual value to within 10^{-5}.

[8]This is *Casus Irreducibilis*: in certain cases, any expression for the roots of a cubic polynomial in terms of radicals must involve nonreal expressions, even if all the roots are real.

4.7.2 Debiasing errors during application of 3BC

We will now consider the error model in which debiasing errors can occur at each time step (i.e. immediately after the application of any gate in the circuit of Figure 4.2, or equivalently after each o_2' operation). The analysis is performed similarly to what we did in Section 4.6.2, by considering the probability associated with each binary tuple $(a, b, c, e_1, \ldots, e_7)$ for which the resulting value of bit A equals 0. For the asymmetric model, by tracing through the circuit of Figure 4.2, we find that equation (4.41) generalizes to

$$\Pr(a, b, c, e_1, \ldots, e_7) \equiv (1-p)^{a+b+c} p^{\bar{a}+\bar{b}+\bar{c}} \left(\varepsilon_0^{\sum_{i=1}^{7} \bar{\phi}_i e_i}\right) \left((1-\varepsilon_0)^{\sum_{i=1}^{7} \bar{\phi}_i \bar{e}_i}\right) \left(\varepsilon_1^{\sum_{i=1}^{7} \phi_i e_i}\right) \left((1-\varepsilon_1)^{\sum_{i=1}^{7} \phi_i \bar{e}_i}\right) \tag{4.65}$$

where $\bar{x} \equiv (1 + x \bmod 2)$ and

$$\phi_1 = a \tag{4.66}$$
$$\phi_2 = a + b \bmod 2 \tag{4.67}$$
$$\phi_3 = c \tag{4.68}$$
$$\phi_4 = \phi_1 + e_1 \bmod 2 \tag{4.69}$$
$$\phi_5 = \phi_2 + e_2 \bmod 2 \tag{4.70}$$
$$\phi_6 = \phi_3 + e_3 + \phi_4 \bmod 2 \tag{4.71}$$
$$\phi_7 = \phi_4 + e_4 + (\phi_5 + e_5)(\phi_6 + e_6) \bmod 2. \tag{4.72}$$

Again we can sum the probabilities $\Pr(a, b, c, e_1, \ldots, e_7)$ over those tuples for which the final value of bit A (given by equation (4.39)) equals 0, and compute the new bias. The new bias, approximated to second order in s and d, is

$$\mathcal{B}' \approx \frac{1}{2}\left((5d + 4d^2 - 6sd) + (3 - 12s + 19s^2 - d^2 + 4ds)\mathcal{B} + d\mathcal{B}^2 + (-1 + 6s - 15s^2)\mathcal{B}^3\right). \tag{4.73}$$

Then the condition $\mathcal{B}' > \mathcal{B}$ gives

$$(5d + 4d^2 - 6sd) + (1 - 12s + 19s^2 - d^2 + 4ds)\mathcal{B} + d\mathcal{B}^2 + (-1 + 6s - 15s^2)\mathcal{B}^3 > 0. \tag{4.74}$$

For values of $s \lesssim 0.04$ (recall the threshold condition we obtained in Section 4.6.2) and for $d < s$, the cubic polynomial on the left-hand side of (4.74) has one positive real root. A

positive value of $\mathcal{B}$ will satisfy inequality (4.74) only if it is not greater than the value of this root, which is (to second order in s and d)

$$\mathcal{B}_{\text{lim}} \approx 1 - 3s + 3d - 9d^2 - \frac{41}{2}s^2 + 32ds. \tag{4.75}$$

In the symmetric case, the bound (4.75) agrees with the bound (4.48) that we obtained in Section 4.6.2. In terms of s and $\mathcal{B}_{\text{i}}$ we have,

$$\mathcal{B}_{\text{lim}} \approx 1 - 3s - \frac{41}{2}s^2 + 3\mathcal{B}_{\text{i}}s + 32\mathcal{B}_{\text{i}}s^2 - 9\mathcal{B}_{\text{i}}^2 s^2. \tag{4.76}$$

For error rates less than 1%, the approximate value (4.64) agrees with the actual value (4.62) to within 10^{-4}.

4.8 More general algorithms based on 3BC

In all of the above error analysis, we have assumed that the 3BC step is applied to three bits having identical bias at each stage of the algorithm. Recall in Section 4.4 it was mentioned that an algorithm proposed in [SMW07] is structured somewhat differently, and applies the 3BC step to three bits having different bias values $\mathcal{B}_{j-2}$, $\mathcal{B}_{j-1}$ and $\mathcal{B}_j$. We can still learn something by performing the previous analysis assuming all three bits have bias $\max(\mathcal{B}_{j-2}, \mathcal{B}_{j-1}, \mathcal{B}_j)$, but it is worth briefly considering how we could analyze this more general scenario directly. Consider applying the debiasing error channel with error parameters ε_0 and ε_1 immediately after the 3BC step is applied. In this case, the bias of the third bit after the process is

$$\frac{\mathcal{B}_{j-2} + \mathcal{B}_{j-1} + \mathcal{B}_j - \mathcal{B}_{j-2}\mathcal{B}_{j-1}\mathcal{B}_j}{2}(1-s) + d \tag{4.77}$$

(recall $s = \varepsilon_0 + \varepsilon_1$ and $d = \varepsilon_1 - \varepsilon_0$). As in Section 4.7, we assume that the error parameters satisfy $s < 1$, $d > 0$ and d is less than each of $\mathcal{B}_{j-2}$, $\mathcal{B}_{j-1}$ and $\mathcal{B}_j$. We also assume that $\frac{d}{s}$ is less than each of $\mathcal{B}_{j-2}$, $\mathcal{B}_{j-1}$ and $\mathcal{B}_j$ so that the errors are indeed pushing the system towards a lower bias.

Suppose we proceed as in [SMW07] and send the first two bits back to the heat bath, re-cool them up to bias values $\mathcal{B}_{j-2}$ and $\mathcal{B}_{j-1}$, and again apply 3BC. Without errors, we

mentioned previously that by repeating this process several times the third bit reaches a steady-state bias value of

$$\frac{\mathcal{B}_{j-2} + \mathcal{B}_{j-1}}{1 + \mathcal{B}_{j-2}\mathcal{B}_{j-1}}. \tag{4.78}$$

With the debiasing error channel being applied after every application of 3BC, this steady-state bias value is reduced to

$$\frac{(\mathcal{B}_{j-2} + \mathcal{B}_{j-1})(1-s) + 2d}{1 + \mathcal{B}_{j-2}\mathcal{B}_{j-1}(1-s) + s}. \tag{4.79}$$

Equations (4.77) and (4.79) can be used to analyze more general algorithms based on repeated application of 3BC, including the algorithm proposed in [SMW07], under the effect of debiasing errors that may occur after each application. We could similarly decompose the 3BC step into a suitable sequence of discrete operations, and proceed as we have done above to analyze the effect of errors that may occur after each discrete step.

4.9 Conclusions and other considerations

I have studied the performance of cooling algorithms that use the 3-bit majority as the compression step (e.g. [FLMR04], [SMW07]) and argued that previously discovered algorithms (e.g. [SV99], [BMR+02]) can be recast in this way. I have proven the optimality of the best such algorithm (operating in a restricted setting) for obtaining one cold bit with the fewest possible number of initially mixed bits. An error analysis of these algorithms has been conducted, first under a simple error model (symmetric bit-flip errors), and then under a more realistic model of debiasing. Since the implementations of the RPC steps are inherently classical (states do not leave the computational basis), it is reasonable to restrict attention to these classical error models. In each case, I first derived some bounds assuming that errors may occur immediately after the RPC step. Since this may be taken as a best-case scenario, these bounds apply regardless of the implementation. I also derived bounds assuming that the 3BC cooling step is implemented by a sequence of physical operations that simulate a sequence of CNOT and Toffoli gates (i.e. a sequence of o_2' operations). Specifically, I considered the simplest such arrangement for implementing the 3BC step, shown in Figure 4.2. The results are summarized below (approximated to second order).

Error Model	**Threshold**	**Maximum achievable bias**
Symmetric bit-flip after 3BC	$\varepsilon < \frac{1}{6}$	$1 - 2\varepsilon - 6\varepsilon^2$
Symmetric bit-flip during 3BC	$\varepsilon \lesssim 0.048592$	$1 - 6\varepsilon - 82\varepsilon^2$
debiasing error after 3BC	N/A	$1 - s - \frac{3}{2}s^2 + \mathcal{B}_\mathrm{i}s + 3\mathcal{B}_\mathrm{i}s^2 - \frac{3}{2}\mathcal{B}_\mathrm{i}^2 s^2$
debiasing error during 3BC	N/A	$1 - 3s - \frac{41}{2}s^2 + 3\mathcal{B}_\mathrm{i}s + 32\mathcal{B}_\mathrm{i}s^2 - 9\mathcal{B}_\mathrm{i}^2 s^2$

Given a specific low-level implementation of a cooling algorithm, specified as a sequence of pulses applied to an ABC-chain or some other suitable hardware, a detailed error analysis could be conducted in a manner similar to the approach I have taken here. For specific cooling algorithms it will also be interesting to analyze the effects of errors occurring between applications of the RPC step (for example, while the bits are being permuted to move the required bits into position for the next application of the cooling step). By studying the time-complexity of a specific algorithm implemented on a specific architecture, we can determine the balance between the rate at which the algorithm increases the bias, and the rate at which debiasing errors are causing the bias to decrease.

Cooling algorithms can be built from basic steps other than the 3-bit majority. For those that have "classical" implementations (that is, can be built from some sequence of generalized Toffoli gates), the approach I have taken here could be employed to conduct a similar error analysis. For basic RPC steps operating on more than 3 bits, this analysis would require examining higher-order polynomials, and may have to be done numerically.

For RPC steps that are implemented "quantumly" (i.e. using gates that force states to leave the computational basis), more general quantum error models will have to be considered, and a different approach to the error analysis will be required.

Appendix A

Proofs of correctness for sequences in Section 3.2.2.2

Proof of correctness for $\mathtt{P_R}$:
Consider a section of the lattice consisting of 18 lattice qubits encoding the data qubits d_i and d_{i+1}, with the pointer positioned at data qubit d_i. Let x_i be the binary value associated with a basis state of the data qubit d_i. Then we have

$$(\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{0},\underset{B}{1},\underset{A}{1},\underset{B}{0},\underset{A}{1+x_{i+1}},\underset{B}{1+x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{0}).$$

Note that evaluation of the state after each pulse requires evaluation of the states of the cells to the left and right of the segment. Because the lattice is encoded according to a known repeating structure, we can deduce these states as required. The effect

of the pulse sequence $\mathtt{P_R}$ is as follows.

$$(\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{0},\underset{B}{1},\underset{A}{1},\underset{B}{0},\underset{A}{1+x_{i+1}},\underset{B}{1+x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{0})$$

$$\xmapsto{B_0^X} (\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{1+x_i},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{1+x_i},\underset{B}{1},\underset{A}{0},\underset{B}{0},\underset{A}{1},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{0},\underset{A}{0},\underset{B}{0})$$

$$\xmapsto{A_0^X} (\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i},\underset{A}{x_i},\underset{B}{x_i},\underset{A}{0},\underset{B}{1},\underset{A}{1},\underset{B}{0},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0})$$

$$\xmapsto{B_0^X} (\underset{A}{0},\underset{B}{1+x_i},\underset{A}{1+x_i},\underset{B}{x_i},\underset{A}{x_i},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{1},\underset{A}{0},\underset{B}{0},$$
$$\underset{A}{0},\underset{B}{1+x_{i+2}})$$

$$\xmapsto{A_0^X} (\underset{A}{1+x_i},\underset{B}{1+x_i},\underset{A}{x_i},\underset{B}{x_i},\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{1},\underset{A}{1},\underset{B}{0},,$$
$$\underset{A}{1+x_{i+2}},\underset{B}{1+x_{i+2}})$$

and so the resulting state of the lattice segment encodes the pointer positioned at data qubit x_{i+1}. (Notice that the entire data array has moved to the left along the lattice in the process). $\square$

Proof of correctness for CZ:

Consider a section of the lattice consisting of 16 lattice qubits encoding the data qubits d_{i-1} and d_i, with the pointer initially at data qubit d_i. Let x_i be the binary value associated with a basis state of the data qubit d_i. Initially, the state is as follows:

$$(\underset{A}{1+x_i},\underset{B}{1+x_i},\underset{A}{x_i},\underset{B}{x_i},\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{1+x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{1},\underset{A}{1},\underset{B}{0})$$

Let the phase associated with a basis state be $(-1)^{\phi}$, and suppose initially we have $\phi = 0$. The goal is to show that `CZ` Leaves a basis state unaffected, but introduces

the phase $\phi = x_i x_{i+1}$. The effect of the pulse sequence `CZ` is as follows.

$$(\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}) \ , \quad \phi = 0$$

$$\xmapsto{\mathtt{P_R}} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}) \ , \quad \phi = 0$$

$$\xmapsto{B_0^X} (\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{1+x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{1}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1}, \underset{B}{x_{i+2}}) \ , \quad \phi = 0$$

$$\xmapsto{A_0^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+2}}, \underset{B}{x_{i+2}}) \ , \quad \phi = 0$$

$$\xmapsto{A_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+2}}, \underset{B}{x_{i+2}}) \ , \quad \phi = 0$$

$$\xmapsto{B_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{1}, \underset{B}{1+x_{i+2}}, \underset{A}{1+x_{i+2}}, \underset{B}{1}) \ , \quad \phi = 0$$

$$\xmapsto{A_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}+x_{i+2}+x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+2}},$$
$$\underset{A}{0}, \underset{B}{1}) \ , \quad \phi = 0$$

$$\xmapsto{A_0^X} (\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{1+x_{i+1}}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+2}}, \underset{A}{x_{i+2}}, \underset{B}{1}) \ , \quad \phi = 0$$

$$\xmapsto{B_0^X} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{x_{i+1}}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+1}x_{i+2}},$$
$$\underset{A}{x_{i+2}}, \underset{B}{1+x_{i+2}}) \ , \quad \phi = 0$$

$$\xmapsto{B_2^X} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{x_{i+1}}, \underset{A}{1}, \underset{B}{1}, \underset{A}{1}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1}, \underset{A}{x_{i+2}}, \underset{B}{1+x_{i+2}}) \ ,$$
$$\phi = 0$$

$$\xmapsto{A_0^X} (\underset{A}{0}, \underset{B}{1+x_i}, \underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{1}, \underset{A}{0}, \underset{B}{1+x_{i+2}}),$$
$$\phi = 0$$

$$\xmapsto{B_0^X} (\underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}+x_{i+1}x_{i+2}},$$
$$\underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{0}, \underset{B}{0}) \ , \quad \phi = 0$$

$$\xmapsto{B_2^X} (\underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i+x_{i+1}+x_ix_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}},$$
$$\underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{0}, \underset{B}{0}) \ , \quad \phi = 0$$

$$\xmapsto{A_2^X} (\underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i+x_{i+1}+x_ix_{i+1}}, \underset{B}{x_i+x_{i+1}+x_ix_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{1}, \underset{B}{1+x_{i+1}},$$
$$\underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{0}, \underset{B}{0}) \ , \quad \phi = 0$$

$$\begin{aligned}\xmapsto{A_0^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{1+x_i},\underset{A}{1+x_i},\underset{B}{x_i+x_{i+1}+x_ix_{i+1}},\underset{A}{x_i+x_ix_{i+1}},\underset{B}{0},\underset{A}{1},\underset{B}{1+x_{i+1}},\underset{A}{1},\underset{B}{1+x_{i+1}},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}},\underset{A}{x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{0})\quad,\quad \phi=0\end{aligned}$$

$$\begin{aligned}\xmapsto{B_0^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i+x_{i+1}},\underset{A}{x_i+x_ix_{i+1}},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{1},\underset{B}{1+x_{i+1}},\underset{A}{1},\underset{B}{0},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}})\quad,\quad \phi=0\end{aligned}$$

$$\begin{aligned}\xmapsto{A_2^X}(&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i+x_{i+1}},\underset{A}{1+x_{i+1}+x_ix_{i+1}},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{x_i+x_{i+1}+x_ix_{i+1}},\underset{B}{1+x_{i+1}},\\ &\underset{A}{1},\underset{B}{0},\ ,\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}})\quad,\quad \phi=0\end{aligned}$$

$$\begin{aligned}\xmapsto{A_0^X}\ (&\underset{A}{x_{i-1}},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{x_{i+1}},\underset{B}{1+x_i+x_{i+1}},\underset{A}{1},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{0},\underset{B}{1+x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{0},\underset{A}{0},\underset{B}{x_{i+1}},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}x_{i+2}},x_{i+1}+x_{i+1}x_{i+2})\quad,\quad \phi=0\end{aligned}$$

$$\begin{aligned}\xmapsto{B_2^Z}\ (&\underset{A}{x_{i-1}},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{x_{i+1}},\underset{B}{1+x_i+x_{i+1}},\underset{A}{1},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{0},\underset{B}{1+x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{0},\underset{A}{0},\underset{B}{x_{i+1}},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}x_{i+2}},x_{i+1}+x_{i+1}x_{i+2})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\begin{aligned}\xmapsto{A_0^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i+x_{i+1}},\underset{A}{1+x_{i+1}+x_ix_{i+1}},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{x_i+x_{i+1}+x_ix_{i+1}},\underset{B}{1+x_{i+1}},\\ &\underset{A}{1},\underset{B}{0},\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{0},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\begin{aligned}\xmapsto{A_2^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i+x_{i+1}},\underset{A}{x_i+x_ix_{i+1}},\underset{B}{1+x_i+x_ix_{i+1}},\underset{A}{1},\underset{B}{1+x_{i+1}},\underset{A}{1},\underset{B}{0},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}},\underset{A}{x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\begin{aligned}\xmapsto{B_0^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{0},\underset{B}{1+x_i},\underset{A}{1+x_i},\underset{B}{x_i+x_{i+1}+x_ix_{i+1}},\underset{A}{x_i+x_ix_{i+1}},\underset{B}{0},\underset{A}{1},\underset{B}{1+x_{i+1}},\underset{A}{1},\underset{B}{1+x_{i+1}},\\ &\underset{A}{x_{i+1}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}},\underset{A}{x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{0})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\begin{aligned}\xmapsto{A_0^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i},\underset{A}{x_i+x_{i+1}+x_ix_{i+1}},\underset{B}{x_i+x_{i+1}+x_ix_{i+1}},\underset{A}{x_{i+1}},\underset{B}{0},\underset{A}{x_{i+1}},\underset{B}{1+x_{i+1}},\underset{A}{1},\underset{B}{1+x_{i+1}},\\ &\underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}},\underset{A}{0},\underset{B}{0})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\begin{aligned}\xmapsto{A_2^X}\ (&\underset{A}{0},\underset{B}{0},\underset{A}{1+x_i},\underset{B}{1+x_i},\underset{A}{x_i},\underset{B}{x_i+x_{i+1}+x_ix_{i+1}},\underset{A}{x_{i+1}},\underset{B}{0},\underset{A}{x_{i+1}},\underset{B}{1+x_{i+1}},\underset{A}{x_{i+1}},\underset{B}{1+x_{i+1}},\\ &\underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}},\underset{B}{x_{i+1}+x_{i+1}x_{i+2}},\underset{A}{0},\underset{B}{0})\quad,\quad \phi=x_ix_{i+1}\end{aligned}$$

$$\xmapsto{B_2^X} (\underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{0}, \underset{B}{0}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{B_0^X} (\underset{A}{0}, \underset{B}{1+x_i}, \underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}+x_{i+1}x_{i+2}}, \underset{B}{1}, \underset{A}{0}, \underset{B}{1+x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{A_0^X} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{x_{i+1}}, \underset{A}{1}, \underset{B}{1}, \underset{A}{1}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1}, \underset{A}{x_{i+2}}, \underset{B}{1+x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{B_2^X} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{x_{i+1}}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1}, \underset{B}{x_{i+1}+x_{i+1}x_{i+2}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+1}x_{i+2}}, \underset{A}{x_{i+2}}, \underset{B}{1+x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{B_0^X} (\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{1+x_{i+1}}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+2}}, \underset{A}{x_{i+2}}, \underset{B}{1}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{A_0^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}+x_{i+2}+x_{i+1}x_{i+2}}, \underset{B}{1+x_{i+2}}, \underset{A}{0}, \underset{B}{1}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{A_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{0}, \underset{A}{x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{1}, \underset{B}{1+x_{i+2}}, \underset{A}{1+x_{i+2}}, \underset{B}{1}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{B_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+2}}, \underset{B}{x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{A_2^X} (\underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+2}}, \underset{B}{x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{A_0^X} (\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{x_i}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{1+x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{1}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1}, \underset{B}{x_{i+2}}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{B_0^X} (\underset{A}{1+x_i}, \underset{B}{1+x_i}, \underset{A}{x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{1+x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}) \quad , \quad \phi = x_i x_{i+1}$$

$$\xmapsto{\mathtt{P_L}} (\underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{1+x_i}, \underset{B}{x_i}, \underset{A}{0}, \underset{B}{1}, \underset{A}{1}, \underset{B}{0}, \underset{A}{1+x_{i+1}}, \underset{B}{1+x_{i+1}}, \underset{A}{x_{i+1}}, \underset{B}{x_{i+1}}, \underset{A}{0}, \underset{B}{0}, \underset{A}{0}, \underset{B}{0}) \quad , \quad \phi = x_i x_{i+1}$$

It can similarly be verified that there is no net effect on qubits encoded in other parts of the data array. □.

Appendix B

Implementing switching stations for the architecture of Section 3.2.2.2

I briefly describe the approach proposed in [BBK04] to implementing switching stations for the GCA architecture that was outlined in Section 3.2.2.2. Recall from Section 3.2.2.2 that the pointer is implemented by a “control unit” which is the specific pattern of states $|1\rangle|1\rangle|0\rangle|0\rangle|1\rangle|1\rangle$ within the lattice. A switching station for this scheme could be encoded by another pattern of states, illustrated below.

			← switching station												← CU								
$\cdots$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1	0	0	$\cdots$
$\cdots$	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	$\cdots$

(B.1)

The switching station is canonically positioned starting on the A qubits, as are the data qubits. This means that the switching station will move along the lattice in parallel with the data qubits, and so maintain its relative position. The control unit moves contrary to the data qubits and switching station. The control unit can be made to move transparently through the switching station (just as it moves transparently through data qubits). For the lattice segment (B.1), to move the control unit into the switching station and *deactivate*

the control unit, the following pulse sequence can be applied.

$$A_0^X, B_0^X, B_2^X, A_0^X, B_0^X, A_0^X, A_2^X, B_0^X, A_0^X, B_0^X, A_2^X, A_0^X, B_0^X. \tag{B.2}$$

After this sequence the lattice segment (B.1) is transformed to the following.

···	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	···
···	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	···

(B.3)

The control unit has been "absorbed" into the switching station, forming the pattern $|1\rangle|1\rangle|1\rangle|1\rangle$, which is positioned starting on the A qubits. It will then move in parallel with the encoded data qubits, and so always be in the same relative position. Therefore its ability to act as a control unit is disabled. To re-activate the control unit, the reverse pulse sequence of (B.2) is used.

A method for "marking" these switching stations, as in Section 3.5.2 is not given in [BBK04]. They propose the idea that the switching stations could be "labeled" by distinct patterns of states, but no specific labeling strategies are proposed (it is unclear how they could be implemented).

Bibliography

[ABO97] D. Aharonov, M. Ben-Or. "Fault tolerant quantum computation with constant error". *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC'97)*, 1997. *Also arXiv e-print quant-ph/9611025.*

[ADH97] L. Adleman, J, Demarrais, M.D. Huang. "Quantum computability". *SIAM Journal on Computing*, 26(5):1524-1540, 1997.

[ADK+04] D. Aharonov, W. van Dam, J. Kempe, Z. Landau, S. Lloyd, and O. Regev. "Adiabatic Quantum Computation is Equivalent to Standard Quantum Computation." *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, 42-51, 2004.

[AHU74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. "The Design and Analysis of Computer Algorithms". *Addison-Wesley*, Reading, Massachusetts, 1974.

[AKN98] D. Aharonov, A. Kitaev, N. Nisan. "Quantum circuits with mixed stats". *Proceedings of the 31st Annual ACM Symposium on the Theory of Computation (STOC'98)*, 20-30, 1998.

[BBBV97] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, Umesh Vazirani. "Strengths and weaknesses of quantum computing". *SIAM Journal on Computing*, 26:1510-1523, 1997.

[BBC+95] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John Smolin, Harald Weinfurter. "Ele-

mentary gates for quantum computation". *Physical Review A*, 52(5):3457-3467, 1995. *Also arXiv e-print quant-ph/9503016*.

[BBF03] Stephane Beauregard, Gilles Brassard, Jose Manuel Fernandez, "Quantum Arithmetic on Galois Fields", Quant-ph/0301163.

[BBK04] A. Bririd, S. Benjamin, A. Kay. "Quantum Error Correction in Globally Controlled Arrays". quant-ph/0308113.

[BCD+96] David Beckman, Amalavoyal Chari, Srikrishna Devabhaktuni, John Preskill. "Efficient Networks for Quantum Factoring". Quant-ph/9602016.

[Bea03] S. Beauregard. "Circuit for Shor's algorithm using $2n + 3$ qubits". Quant-ph/0205095.

[Ben00] S. Benjamin. Physical Review A **61**, 020301 (2000).

[Ben73] Charles H. Bennett. "Logical Reversibility of Computation". *IBM Journal of Research and Development*, 17:525-532, November 1973.

[Ben89] Charles H. Bennett. "Time/space trade-offs for reversible computing". *SIAM Journal on Computing*, 18(4):766-776, 1989.

[BMP+99] P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, Farokh Vatan. "On Universal and Fault Tolerant Quantum Computing". *arXiv e-print quant-ph/9906054*, 1999.

[BMR+02] P.O. Boykin, Tal Mor, Vwani Roychowdhury, F.Vatan and R. Vrijen, "Algorithmic cooling and scaleable NMR quantum computers", *Proc. Natl. Acad. Sci.,* 99(6):3388-3393,2002.

[BMR+05] J. Baugh, O. Moussa, C.A. Ryan, A. Nayak, R. Laflamme. "Experimental implementation of heat-bath algorithmic cooling using solid-state nuclear magnetic resonance". *Nature* Vol. 438, No. 7076, 470-473, 2005.

[Buh96] H. Buhrman. "A short note on Shor's factoring algorithm". *SIGACT News*, 27(1):89-90, 1996.

[CEH+99] Richard Cleve, Artur Ekert, Leah Henderson, Chiara Macchiavello, Michele Mosca. “On quantum algorithms”. *Complexity*, 4:33-42,1999.

[CEMM98] Richard Cleve, Artur Ekert, Chiara Macchiavello, Michele Mosca, “Quantum Algorithms Revisited”. *Proceedings of the Royal Society of London A*, 454:339-354, 1998.

[Cop94] Don Coppersmith. “An approximate fourier transform useful in quantum factoring”. *Research report, IBM*, 1994.

[CS96] A.R.Calderbank, P.W. Shor. “Good quantum error-correcting codes exist”. *Physical Review A*, 54:1098-1105, 1996.

[Deu85] David Deutsch. “Quantum theory, the Church-Turing principle and the universal quantum computer”. *Proceedings of the Royal Society of London A*, 400:97-117, 1985.

[Deu89] David Deutsch. “Quantum computational networks”. *Proceedings of the Royal Society of London A*, 425:73-90,1989.

[DiV95] David DiVincenzo. “Two-bit gates are universal for quantum computation”. *Physical Review A*, 51(2):1015-1022,1995.

[DJ92] David Deutsch, Richard Josza. “Rapid solution of problems by quantum computation”. *Proceedings of the Royal Society of London A*, 439:553-558, 1992.

[Dra00] T. Draper. “Addition on a Quantum Computer”. quant-ph/0008033.

[EPR35] A. Einstein, B. Podolsky, and N. Rosen, “Can quantum-mechanical description of reality be considered complete?” *Physical Review* 47:777-780, 1935.

[Fey65] Richard P. Feynmann. “The Feynmann Lectures on Physics, Volume III: Quantum Mechanics”. *Addison-Wesley*, 1965.

[Fey82] Richard Feynman, “Simulating Physics with Computers”. *International Journal of Theortical Physics*, 21(6,7):467-488, 1982.

[FIPS] FIPS 186-2, Federal Information Processing Standards, http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf.

[FLMR04] Jose M. Fernandez, Seth Lloyd, Tal Mor, Vwani Roychowdhury, "Algorithmic Cooling of Spins: A Practicable Method for Increasing Polarization" *quant-ph/0401135*.

[FT06] Joseph Fitzsimons, Jason Twamley, "Globally controlled quantum wires for perfect qubit transport, mirroring and quantum computing". quant-ph/0601120.

[Gac86] P. Gács. "Reliable computation with cellular automata". *Journal of Computer System Science* **32** (1986), 15-78.

[GN96] R.B. Griffiths, C.S. Niu. "Semi-classical fourier transform for quantum computation". *Physical Review Letters*, 3228-3231, 1996.

[Got98] Daniel Gottesman. "A theory of fault-tolerant quantum computation". *Physical Review A*, 57:127-137, 1998. *also arXiv e-print quant-ph/9702029*

[HMP+98] Huelga, S.F., Macchiavello, C., Pellizzari, T., Ekert, A., Plenio, M.B. and Cirac J.I. 1997, e-print quant-ph/9707014.

[HMV03] D. Hankerson, A. Menezes and S. Vanstone, "Guide to Elliptic Curve Cryptography", Springer-Verlag, 2003.

[Kay05] A. Kay, "Error Correcting the Control Unit in Global Control Schemes", *arXiv e-print quant-ph/0504197*.

[Kay07] A. Kay, "Deriving a Fault-Tolerant Threshold for a Global Control Scheme", *arXiv e-print quant-ph/0702239*.

[Kaye05] Phillip Kaye. "Optimized Quantum Implementation of Elliptic Curve Airthmetic over Binary Fields". *Quantum Information and Computation*, 5(6):474-491, 2005.

[Kaye07] Phillip Kaye. "Cooling algorithms based on the 3-bit majority ". *Quantum Information Processing*, To appear: Issue 5, Vol 6, 2007. *arXiv e-print quant-ph/0703194v3*.

[Kit97] A. Y. Kitaev. "Quantum computations: algorithms and error correction". *Russ. Math. Surv.*, 52(6):1191-1249, 1998.

[KLM07] P. Kaye, R. Laflamme, M. Mosca. "An Introduction to Quantum Computing". *Oxford University Press*, Oxford. 2007.

[KLZ97] Emanuel Knill, Raymond Laflamme, Wojciech, Zurek. "Resilient quantum computation: Error models and thresholds". Technical report, 1997. *Also arXiv e-print quant-ph/9702058.*

[KM01] P. Kaye, M. Mosca 2001. "Quantum Networks for Concentrating Entanglement" On the quant-ph archive, report no. 0101009.

[KM02] P. Kaye, M. Mosca. "Quantum networks for generating arbitrary quantum states". *Proceedings of the International Conference on Quantum Information, OSA CD-ROM (Optical Society of America, Washington, D.C., 2002)*, PB28.

[Llo93] S. Lloyd. "A potentially realisable quantum computer". Science **261** 1569; see also Science **263** 695 (1994).

[Llo99] S. Lloyd. "Programming Pulse Driven Quantum Computers". Quant-ph/9912086.

[LMR08] R. Laflamme, O. Moussa, Colm Ryan. *Personal communication.*

[LTV98] M. Li, J. Tromp, P. Vitanyi. "Reversible simulation of irreversible computation". *Physica D*, 120:168-176, 1998.

[M05] O. Moussa, "On Heath-Bath Algorithmic Cooling and its Implementation in Solid-State NMR". MMath thesis, University of Waterloo.

[ME99] Michele Mosca, Artur Ekert. "The hidden subgroup problem and eigenvalue estimation on a quantum computer". *Lecture Notes in Computer Science*, Volume 1509, 1999. *Also arXiv e-print quant-ph/9903071.*

[Mos99] Michele Mosca, "Quantum Computer Algorithms". *D.Phil. Dissertation.* Wolfson College, University of Oxford, 1999.

[MvOV97] Alfred J. Menezes, Paul C. Van oorschot, Scott A. Vanstone. “Handbook of Applied Cryptography”. *CRC Press*, London, 1997.

[MZ04] M. Mosca, C. Zalka. “Exact quantum Fourier transforms and discrete logarithm algorithms”. *International Journal of Quantum Information*, Vol. 2, No. 1 (2004) 91-100.

[NC00] Michael Nielson, Isaac Chuang . “Quantum Computation and Quantum Information”. 2000, Cambridge University Press.

[PP05] S. Parker, M.B. Plenio. “Efficient factorization with a single pure qubit and log N mixed qubits”. Phys. Rev. Lett. 85, 3049 (2000).

[Pre] John Preskill. *Lecture notes.* *Available at http://www.theory.caltech.edu/% 7Epreskill/ph219/index.html# lecture.*

[PZ03] Christof Zalka, John Proos. “Shor’s discrete logarithm quantum algorithm for elliptic curves”, QIC Vol. 3 No. 4, pp 317-344 (2003), also quant-ph/0301141.

[RMBL07] C.A. Ryan, O. Moussa, J. Baugh, and R. Laflamme. “A spin based heat engine: multiple rounds of algorithmic cooling”, arXiv:0706.2853.

[Sha49] Claude E. Shannon. “A Mathematical Theory of Communication”, Urbana, IL. University of Illinois Press, 1949 (reprinted 1998).

[SFW05] D.J. Shepherd, T. Franz, R.F. Werner. “A universally programmable quantum cellular automata”. *quant-ph/0512058.*

[Sho94] Peter Shor. “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 124-134, 1994.

[Sho95] Peter Shor. “Scheme for reducing decoherence in quantum computer memory”. *Phys. Rev. A*, 52:2493, 1995.

[Sho96] Peter Shor. “Fault-tolerant quantum computation”. *Proceedings of the* 37th *Annual Symposium on Fundamentals of Computer Science*, 56-65, IEEE Press, Los Alimitos, CA, 1996.

[Sho97] P. Shor, “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer” *SIAM J. Computing*, 26:1484-1509, 1997.

[SMW05] L.J. Schulman, T. Mor, Y,Weinstein, “Physical limits of heat-bath algorithmic cooling”. *Physical Review Letters* 94:120501, 2005.

[SMW07] L.J. Schulman, T. Mor, Y,Weinstein, “Physical limits of heat-bath algorithmic cooling”. *SIAM Journal on Computing*, Vol. 36, Issue 6, 1729-1747.

[Ste96] A.M. Steane. “Error correcting codes in quantum theory”. *Physical Review Letters*, 77:793-797, 1996.

[SV98] L.J. Schulman, U. Vazirani, “Molecular scale heat engines and scaleable quantum computation”, *Proc. 31st ACM STOC (Symp. Theory of Computing)* 322-329.

[SV99] L.J. Schulman, U. Vazirani, “Scaleable NMR Quantum Computation” (preliminary draft), *quant-ph.9804060*.

[VBE95] V. Vedral, A. Barenco, A. Ekert, “Quantum networks for elementary arithmetic operations”, *Phys. Rev. A*, **54**, 147.

[Yao93] Andrew Chi-Chih Yao. “Quantum Circuit Complexity”. *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, 352-361, Los Alamitos, California, 1993. *Institute of Electrical and Electronic Engineers Computer Society Press.*

[Zal98] Christof Zalka. “Fast versions of Shor’s quantum factoring algorithm”. *Technical report 9806084, Los Alamos Archive*, 1998.