



Practical Implementation of a Quantum Algorithm for the Solution of Systems of Linear Systems of Equations

A thesis submitted to the
FACULTY OF APPLIED SCIENCES
DELFT INSTITUTE OF APPLIED MATHEMATICS

by

Otmar Ubbens

in partial fulfilment of the requirements for the degree of

BACHELOR OF SCIENCE

in

APPLIED PHYSICS

and

APPLIED MATHEMATICS

Delft, the Netherlands

July 2019



Practical Implementation of a Quantum Algorithm for the Solution of Systems of Linear Systems of Equations

OTMAR UBBENS

Delft University of Technology

Supervisors

DR. M. MÖLLER

DR. C.G. ALMUDEVER

Other committee members

DR. W.G.M. GROENEVELT

DR. L.M.K. VANDERSYPEN

Delft, the Netherlands

July 17, 2019

An electronic version of this thesis is available at

<http://repository.tudelft.nl/>.

Abstract

With the rapid development of Quantum Computers (QC) and QC Simulators, there will be an increased demand for functioning Quantum Algorithms in the near future. Some of the most ubiquitously useful algorithms are solvers for linear systems of equations. Since the conception of the Quantum Linear Solver Algorithm (QLSA) by Harrow, Hassidim and Lloyd (HHL) in 2009, many improvements have been made, although a generic implementation for arbitrary matrices and vectors is still not available. In this thesis a variant of the HHL QLSA is studied, and the open challenges are investigated. Solutions for two of the challenges, namely the Eigenvalue Inversion subroutine and the Higher-Order Ancilla Rotation subroutine, are discussed. As part of the thesis project, these subroutines have been implemented in the QX Quantum Computer Simulator, and the subroutines are combined to form a complete Quantum Linear Solver (QLS), with the restraint that the implementation for the vector and Hamiltonian of the matrix must be provided by the user. A proof-of-concept QLS by Cao et al. is also implemented in the QX simulator, and using the implementation of the vector and Hamiltonian of Cao et al. the complete solver is tested. In the process of this thesis, a framework for basic Quantum Arithmetic is built providing three variants of Integer Adders, two variants of Integer Subtracters, one Integer Multiplier and one Integer Divider. In addition, gates not natively available in the QX simulator are implemented, and a number of improvements and extensions of algorithms presented in the literature are given, making the described algorithms function on the QX simulator and extending features.

Contents

Introduction	1
1 Introduction to Quantum Computing	2
1.1 Basics of Quantum Computing	2
1.1.1 Single Qubit	2
1.1.2 Multiple Qubits	4
1.2 Quantum Circuits	5
1.3 QX Implementation	7
1.3.1 Building gates	7
1.3.2 Python code generator	9
2 Quantum Subroutines	11
2.1 Quantum Fourier Transform	11
2.2 Quantum Phase Estimation	13
2.3 QX Implementation	15
3 Integer Arithmetic on a Quantum Computer	18
3.1 Integer Addition	18
3.1.1 Draper Adder	19
3.1.2 Cuccaro Adder	21
3.1.3 Muños-Coreas Adder	23
3.1.4 Discussion on the adders	24
3.2 Integer Subtraction	25
3.2.1 General Approach	25
3.2.2 Specific algorithm for the Draper Adder	26
3.3 Integer Multiplication	26
3.3.1 Additive Multipliers	26
3.3.2 Alternative Integer Multiplication Algorithms	29
3.4 Integer Division	30
3.4.1 Adapted Thapliyal Algorithm	30
3.5 QX implementation	36
4 Quantum Linear Solver Algorithm	38
4.1 General problem statement	38
4.2 Classical alternative	39
4.3 Literature review	39
4.4 The HHL algorithm	40
4.4.1 Introduction	40
4.4.2 Realisation of the HHL algorithm	40
4.4.3 In depth discussion	42
4.4.4 Implementation	42
5 Eigenvalue Estimation	44
5.1 The Quantum Phase Estimation Algorithm	44
5.2 The Eigenvalue Estimation Algorithm	45
5.3 Hamiltonian Simulation	47
5.3.1 Classical Implementation	47
5.3.2 Quantum Implementations	47
6 Eigenvalue Inversion	49
6.1 Classical Approaches	49
6.2 Methods for Computing the Reciprocal of a number	49
6.2.1 Powers-of-Two Algorithm	50
6.2.2 Newton-Raphson Algorithm	50
6.2.3 Thapliyal Inversion Algorithm	54

6.3	Comparison of the algorithms	55
6.4	QX Implementation	55
7	Ancilla Rotation	57
7.1	Cao implementation	57
7.2	Proposed extensions	58
7.2.1	Third order approximations	58
7.2.2	Higher order approximations	61
7.2.3	Error analysis	64
7.3	QX Implementation	64
8	Quantum Linear Solver Algorithm	67
8.1	Baseline circuit by Cao et al.	67
8.1.1	Overview	67
8.1.2	Implementation	68
8.1.3	Results	70
8.2	General circuit using Cao matrix	73
9	Conclusions and Future Work	77
	List of references	81
A	Controlled gates	82
B	Cao Eigenvalue Inversion Algorithm	85
B.1	The Rotation subroutine	86
B.2	The Hamiltonian Subroutine	86
B.3	Results	87

Introduction

As the development of Quantum Computers continues, the field of Quantum Computing inches closer to real world applications. The release of the first fully-integrated commercial Quantum Computer by IBM in early 2019, making use of a 20 qubit processor, is a first step to the so-called Noisy Intermediate-Scale Quantum (NISQ) era of Quantum Computing, where Quantum Computers may transcend the capabilities of classical computers for the first time [1].

It is up to the field of Quantum Computation to be prepared and have the algorithms ready to be implemented once the hardware allows for it. Quantum Computer Simulators play a vital role in testing the feasibility of algorithms, and with ongoing projects such as LibKet¹ that allow for platform-independent high-level implementations of Quantum Algorithms, direct benchmarks between platforms may soon be possible.

One of the most ubiquitously useful algorithms is that of the solver of systems of linear equations. In many fields of research, especially those concerned with Quantum Physics and Simulations, it is an essential part of computation. The proposition of the Quantum Linear Solver Algorithm (QLSA) by Harrow, Hassidim and Lloyd (HHL) in 2009 promises a Linear Solver Algorithm that runs exponentially faster than those on classical computers. However, a general QLSA subroutine for generic matrices and vectors is not yet available.

The aim of this thesis is therefore to implement at least some of the parts of the HHL Quantum Linear Solver Algorithm in the QX Quantum Computer Simulator [2]. To this purpose, the algorithm including its improvements will be studied. In addition, its challenges will be investigated. They include the implementation of Hamiltonian Simulation, Vector Implementation, Eigenvalue Inversion and Ancilla Rotation. Of these challenges, a number will be analysed further, solved and combined to form a working prototype Quantum Linear Solver that is as complete as possible.

Of the four main challenges, the Eigenvalue Inversion subroutine and Ancilla Rotation subroutine are fully solved using multiple alternative methods. The result is a Quantum Linear Solver (QLS) able to solve any matrix and vector, given that the implementation of the vector and Hamiltonian of the matrix are provided by the user. This solver and its subroutines are implemented in the QX Quantum Computer Simulator, and a prototype solver for a specific Hamiltonian and vector implementation is verified.

Besides the solver, a complete framework for Integer Arithmetic is developed to facilitate in solving the Eigenvalue Inversion subroutine and Ancilla Rotation subroutine. The framework consists of multiple variants of Integer Adders and Subtracters, a Multiplier and an Integer Division algorithm. Additionally, multiple gates that are by default unavailable in the QX simulator are implemented.

The thesis is structured as follows. In Chapter 1, the basic concepts of Quantum Computation are explained. The concepts are used in the remainder of the thesis to construct Quantum Circuits. The first applications are in Chapters 2 and 3, where, respectively, a number of common Quantum Algorithms and Quantum Arithmetic are presented. In Chapter 4, the HHL Quantum Linear Solver is introduced and an overview of its subroutines is given. In the three subsequent Chapters 5, 6 and 7 the three main subroutines, that is, Eigenvalue Estimation, Eigenvalue Inversion and Ancilla Rotation are worked out in detail, respectively. The latter two are solved in multiple ways, verified in the QX simulator, and compared to one another. Chapter 8 combines the subroutines to form a complete proof-of-concept Quantum Linear Solver, and implements and compares it with another proof-of-concept QLS. Finally, Chapter 9 gives a summary of the main findings of the thesis, as well as conclusions, discussion and recommendations for future research.

This thesis is written in partial fulfilment of the requirements for the degree of Bachelor of Science in Applied Physics and Applied Mathematics.

¹Soon to be released open-source at gitlab.com/mmoelle1/LibKet.

1. Introduction to Quantum Computing

Quantum Computing is the culmination of Computer Science on one hand, and Quantum Mechanics on the other. In Computer Science bits are used to encode information and computations are performed by using logical operations such as ‘AND’ and ‘OR’. In Quantum Computing a similar system is built using quantum bits or *qubits* instead of bits, to which quantum gates are applied to change their states and thus perform computations [3]. These qubits allow for the use of effects known from Quantum Mechanics, such as entanglement and superposition, which are not available in classical computers. An alternative to gate-based Quantum Computing, called Quantum Annealing [4], is available, but it is not discussed in this thesis.

1.1 Basics of Quantum Computing

1.1.1 Single Qubit

In Quantum Computing, two energy levels are used to get a binary system. These energy levels are indicated as $|0\rangle$ and $|1\rangle$, respectively. The states are written according to the *bra-ket* notation introduced in 1939 by Dirac [5]. A qubit cannot only be in one of the two pure states $|0\rangle$ and $|1\rangle$, but also in a so-called *superposition* of the two states. This can be written as a special linear combination of the pure states $|0\rangle$ and $|1\rangle$ as follows,

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \quad (1.1)$$

with coefficients $\alpha, \beta \in \mathbb{C}$ and the normalisation requirement $|\alpha|^2 + |\beta|^2 = 1$. When measuring the $|\psi\rangle$ state, the result is a binary value ‘0’ or ‘1’, with respective probabilities $|\alpha|^2$ and $|\beta|^2$, and the qubit collapses into one of the two possible basis states $|0\rangle$ or $|1\rangle$. The $|\psi\rangle$ state is fully characterised by the two complex amplitudes α and β , hence $|\psi\rangle$ can equivalently be written as the coefficient vector of these values with regard to the pure state basis,

$$|0\rangle \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad (1.2)$$

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \equiv \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \quad (1.3)$$

With this notation, the normalisation requirement can be rewritten as an inner product,

$$\langle\psi|\psi\rangle = [\alpha^\dagger \quad \beta^\dagger] \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha^\dagger \alpha + \beta^\dagger \beta = |\alpha|^2 + |\beta|^2 = 1, \quad (1.4)$$

where $\langle\psi|$ and $x^\dagger \in \mathbb{C}$ are defined as the complex conjugates of $|\psi\rangle$ and $x \in \mathbb{C}$, respectively. The basis vectors chosen for $|0\rangle$ and $|1\rangle$ follow the orthogonality requirement $\langle i|j\rangle = \delta_{ij}$, as $\langle 0|1\rangle = \langle 1|0\rangle = 0$. Here, δ_{ij} is defined as the Kronecker delta. From the normalisation conditions in Equation (1.4), it is seen that the most general allowed state is of the form

$$|\psi\rangle = e^{i\delta} \left(\cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\gamma} \sin\left(\frac{\theta}{2}\right) |1\rangle \right) \quad (1.5)$$

with $\theta \in [0, \pi]$ and $\gamma \in [0, 2\pi)$. The value for $\delta \in [0, 2\pi)$ can be chosen arbitrarily, since it is immeasurable and does not have any physical significance. It contributes equally to both $|0\rangle$ and $|1\rangle$, and drops out in any inner product. The rewrite of the general qubit state in Equation (1.5) allows for spherical representation of all possible single qubit states. The representation, called the Bloch sphere, is shown in Figure 1.1. The states on the axes are known as the Clifford states.

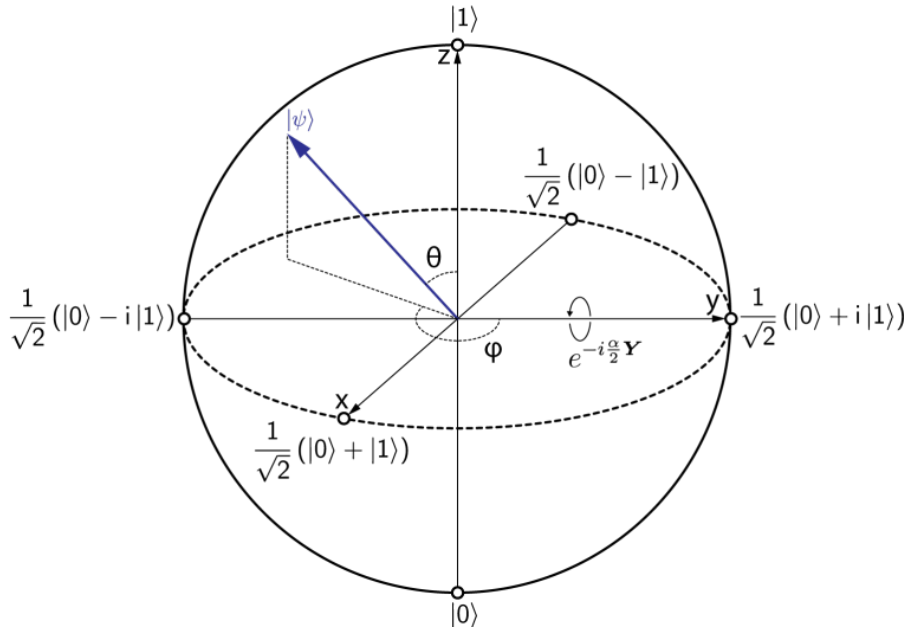


Figure 1.1: The Clifford states in the Bloch sphere [6].

Similar to classical computers, operations are used to perform computations. Among the operations are so-called *gates*. An example of a gate that can also be found in a classical computer, is the X gate (or NOT gate), which negates its input: $|0\rangle \xrightarrow{X} |1\rangle$ and $|1\rangle \xrightarrow{X} |0\rangle$. Gates work linearly with superpositions, meaning that each single qubit operation can be written as a two-by-two matrix that acts on the coefficient matrix of a state. For the example of the X gate, the matrix reads

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \quad (1.6)$$

Multiplication of this matrix with an input qubit state yields the the output of the gate applied to the qubit. Application of the X gate to the general state given in Equation (1.1) for example yields the output state

$$X|\psi\rangle \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} \equiv \beta|0\rangle + \alpha|1\rangle. \quad (1.7)$$

The result of an operation should again be a legitimate quantum state, i.e. the normalisation condition in Equation (1.4) should again hold after the transformation. Operations which obey this condition are the so-called unitary operators [7]. An operator U is called unitary if and only if it satisfies the condition $UU^\dagger = U^\dagger U = I$, with I the identity matrix. An important property of a unitary matrix U is that all of its eigenvalues $\lambda \in \mathbb{C}$ obey $|\lambda| = 1$. This implies that all unitary matrices are invertible. Since any Quantum Gate is of this form, it has the implicit consequence that Quantum Computing is inherently reversible.

Every single qubit gate can be defined as a rotation around a specific axis, where the axis is defined as in the Bloch sphere in Figure 1.1. A basis of three gates which allows for the construction of any rotation is the set of rotations around the x -, y - and z -axis. These gates are defined as,

$$R_x(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}. \quad (1.8)$$

The X gate for example can be interpreted as a rotation of π around the x -axis. Indeed, $R_x(\pi) = e^{-\pi/2} X$, which confirms the statement (global phase is immeasurable and hence unimportant). Examples of other single-qubit gates used in this thesis are shown in Table 1.1. Since the decompositions of the gates are not a focus in this thesis, they will not be discussed.

Name	Symbol	Matrix
Identity	\boxed{I}	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Pauli-X	\boxed{X}	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y	\boxed{Y}	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z	\boxed{Z}	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard	\boxed{H}	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase	\boxed{S}	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$	\boxed{T}	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
$\sqrt{\text{NOT}}$	\boxed{V}	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$

Table 1.1: Examples of single-qubit gates used in this thesis.

1.1.2 Multiple Qubits

To design practical quantum algorithms, it is necessary to work and interact with multiple qubits. To show what happens when several qubits are combined, first the case of a classical two bit system is examined. The possible classical states are 0 and 1 for each bit, giving the four possible states 00, 01, 10 or 11 for two bits. A quantum system composed of two qubits, say $|\psi_1\rangle = \alpha_1|0\rangle_1 + \beta_1|1\rangle_1$ and $|\psi_2\rangle = \alpha_2|0\rangle_2 + \beta_2|1\rangle_2$, is described jointly as $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$, where the “ \otimes ” symbol, called the Kronecker product, represents that the qubits form one system. The notation is often shortened to $|\psi\rangle = |\psi_1\psi_2\rangle$. The possible pure states for the system of two qubits are $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, and the most general two qubit state can be written as a superposition of these pure states,

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle, \quad (1.9)$$

with the complex amplitudes $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, and the normalisation condition $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$. It is possible that the complex amplitudes allow the total state to be written as two separate qubits, i.e. as $|\psi\rangle = (\alpha_1|0\rangle_1 + \beta_1|1\rangle_1) \otimes (\alpha_2|0\rangle_2 + \beta_2|1\rangle_2)$. If however this is not the case, it is said that the qubits are *entangled*, a feature that has no analogy in classical computers.

Comparable to the single qubit case, the complex amplitudes α through δ in Equation (1.9) fully define the quantum state, and thus the system can equivalently be written as the coefficient vector,

$$|\psi\rangle \equiv \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}, \quad (1.10)$$

thereby assuming the canonical ordering, which is with the least significant bit combination on top, i.e.,

$$|00\rangle \equiv \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle \equiv \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad |10\rangle \equiv \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad |11\rangle \equiv \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \quad (1.11)$$

For a quantum system comprised of $n \in \mathbb{N}$ qubits, the size of the coefficient vector is $N = 2^n$.

Operations can be applied to multiple qubits. Generally, any gate acting on an n qubit system can be represented by a 2^n -by- 2^n unitary matrix, on the grounds that the normalisation condition should again hold. Of special interest are controlled gates which will be discussed in the remainder of this section.

A Controlled Quantum Gate applies a gate to the second qubit (“target”) only when the first qubit (“control”) is in the $|1\rangle$ state. This operation is denoted as $C(A)$, and it performs the following transformation, $|0\rangle_1 \otimes |\psi\rangle_2 \xrightarrow{C(A)} |0\rangle_1 \otimes |\psi\rangle_2$ and $|1\rangle_1 \otimes |\psi\rangle_2 \xrightarrow{C(A)} |1\rangle_1 \otimes A|\psi\rangle_2$. The matrix representation and symbol of this operation are,

$$C(A) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & A_{11} & A_{12} \\ 0 & 0 & A_{21} & A_{22} \end{bmatrix}, \quad \begin{array}{c} \bullet \\ | \\ \boxed{A} \end{array}, \quad (1.12)$$

where A_{11} , A_{21} , A_{12} and A_{22} are the four elements of the matrix representation of A . In the symbolic representation, the block with the A represents the gate applied to the target qubit, and the dot represents the control qubit. A well known example of a controlled gate is the $C(X)$ gate, or CNOT gate, which has its own specific symbol. The matrix representation and symbol are,

$$\text{CNOT} = C(X) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \begin{array}{c} \bullet \\ | \\ \oplus \end{array}. \quad (1.13)$$

The \oplus symbol represents the application of the X gate to the target qubit, and the dot represents the control qubit.

To illustrate the application of gates to higher numbers of qubits, one of the most ubiquitous gates for three qubits is examined, which is the TOFFOLI gate. This gate operates comparable to the CNOT gate, as it is also a controlled- X gate. The difference lies in the fact that there are two controlling qubits instead of one. Consider a three qubit system in which the first two qubits are taken as controls, and the third qubit as target. The TOFFOLI gate is defined such that the third qubit is flipped only when both other qubits are in the $|1\rangle$ state, i.e. $|110\rangle \xrightarrow{\text{TOFFOLI}} |111\rangle$ and $|111\rangle \xrightarrow{\text{TOFFOLI}} |110\rangle$, whereas nothing happens for all 6 other states. The matrix representation of the TOFFOLI gate is,

$$\text{TOFFOLI} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \quad (1.14)$$

Finally completeness is briefly discussed. It can be proved that using a limited set of basis gates, any gate of any size can be constructed up to arbitrary precision. An example of a set of universal gates are the Hadamard, phase, CNOT and $\pi/8$ gates [3]. However, the known process generally takes an exponentially number of gates. One of the practical challenges of Quantum Computing is to find efficient decompositions of quantum gates in terms of the aforementioned quantum gates.

1.2 Quantum Circuits

Qubits and gates as discussed in the previous section are used to construct Quantum Circuits. Any quantum algorithm can be described as a quantum circuit. An example of a general Quantum

Circuit is displayed in Figure 1.2. On the basis of this circuit the notation for Quantum Circuits is explained.

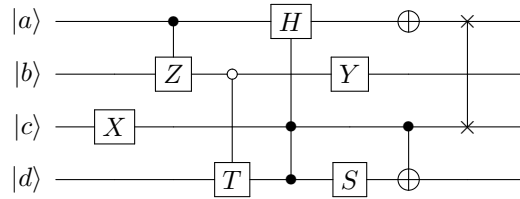


Figure 1.2: Example of a quantum circuit.

In a quantum circuit, the different qubits are displayed on the vertical axis. By default, each line represents a single qubit. Time is displayed horizontally, from left to right. A block represents a gate, where the content inside the block denotes its type.

A controlled gate operates on two or more qubits. The qubit that acts as a control is shown with a dot. A black dot is used when the gate is applied to the target qubit when the control qubit is in the $|1\rangle$ state. It is also possible for the gate to be applied specifically when the controlling qubit is in the $|0\rangle$ state. In that case a hollow dot is shown. A controlled gate can have multiple qubits that act as control.

When two gates are applied to different qubits, they can be performed simultaneously. The gates are then shown below one another. The \oplus symbol is an alternative representation of the X gate, causing the negation of the qubit states it is applied to. When two qubits are desired to switch their states, a SWAP gate is applied which is denoted by two connected crosses.

A brief note should be made about Computer Science. Algorithms are often designed around the ability to function for any number of qubits as input size and other input parameters. Consider for example an algorithm that add two numbers together. Larger numbers as input will require more qubits to store the values, and the process will require more gates. The terms commonly used to describe the required numbers of qubit and gates are the *width* and *depth* of a circuit. The width refers to the number of qubits necessary to perform the algorithm, while the circuit depth refers to the number of time steps required to apply all the gates. The number of time steps is by default the same as the number of gates, unless multiple gates can be applied simultaneously. Then, the circuit depth is lower than the number of gates. Depending on the algorithm, the circuit width and depth may either only slightly increase depending on input parameters, or can increase substantially. To formalise the phenomenon, the Landau “Big O” notation is used [3]. A function $f(n)$ is called $\mathcal{O}(g(n))$ for some function $g(n)$ if and only if some $n_0 \in \mathbb{N}$ and $c \in \mathbb{R}$ exist, such that $f(n) \leq cg(n)$ for all $n \in \mathbb{N}$ with $n \geq n_0$. That is,

$$f(n) = \mathcal{O}(g(n)) \iff \left[\exists c \in \mathbb{R} : \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq n_0 \implies f(n) \leq cg(n) \right]. \quad (1.15)$$

Consider for example the function $3n^2 + 5n - 3$. This function scales as $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, but not as $\mathcal{O}(n)$. In general, an algorithm is called feasible when for inputs of size n the circuit depth and size scale as $\mathcal{O}(p(n))$ for some polynomial function $p(n)$, i.e. when the circuit width and depth contain no exponentially growing terms.

The gates that are included in the calculation of the circuit width and depth can have considerable impact on its outcome. Not every gate is directly implementable, but may instead require multiple gates or even extra qubits to be implemented. Especially at hardware level there are only a limited number of operations available, but also Quantum Simulators generally only offer a subset of all possible Quantum Gates. Since the practical implementation of Quantum Algorithms is the main focus of this thesis instead of thorough analyses on circuit width and depth, not much attention is spent on these types of analyses.

1.3 QX Implementation

The implementations of all circuits in this thesis are performed in the QX simulator, version 2.6.0 [2]. This simulator offers a limited number of gates, specifically: H , X , Y , Z , $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$, S , T , T^\dagger , CNOT, TOFFOLI, SWAP, $C(Z)$ and $C(R_k)$. The definition of the R_k gate will follow in the section on the Quantum Fourier Transform. The QX simulator reads quantum circuits in the form of the Common Quantum Assembly language (cQASM) Version 1.0 [8]. In this section two subjects are discussed. First an overview of the challenge of building gates that are not among the default gates in QX and some implementations. Secondly is the procedure in which algorithms are transformed into cQASM code in this thesis, which is through a cQASM code generator specifically built in Python.

1.3.1 Building gates

Due to the limited number of available gates in the QX simulator, desired operations may not be directly implementable. As was mentioned before, it is possible to approximate each gate up to arbitrary precision using only a limited set of gates [3]. However, the process generally takes exponentially many gates, which is not acceptable for an efficient practical implementation. Whether a gate is able to be approximated efficiently depends on the gate, but a general analysis is beyond the scope of this thesis. The interested reader is referred to [3]. The procedures to construct a controlled gate and a multi-qubit controlled gate will be of use in this thesis, and will be discussed in this section. First the construction of the controlled- U gate for any gate U is discussed.

It can be proven that for any single qubit gate U , there are always three single-qubit gates A , B and C , such that $ABC = I$, and $AXBXC = U$, up to a phase difference $e^{i\alpha}$ between the outputs for $|0\rangle$ and $|1\rangle$, see [3]. Therefore, a controlled- U gate can be constructed using the circuit in Figure 1.3. The gates A , B and C are case dependent. The algorithms to approximate these single-qubit gates are not discussed in this thesis, and generally take an exponential circuit depth.

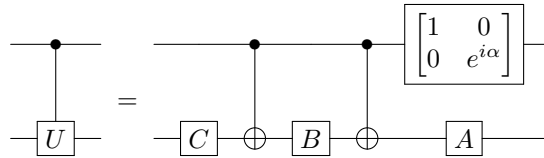


Figure 1.3: General decomposition of a controlled gate into single gates and CNOT gates

Secondly, the construction of the doubly-controlled- U gate is examined. There is no general process as for the single controlled gate, but a procedure is available when the controlled-square-root-of- U gate V is available, that is, $V^2 = U$. The circuit for this method is shown in Figure 1.4. The $C(V)$ gates are again not necessarily available directly in the QX simulator, but they may be approximated using the method described in Figure 1.3.

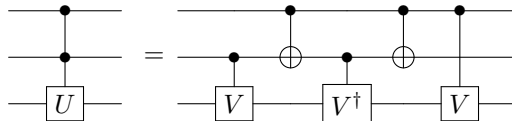
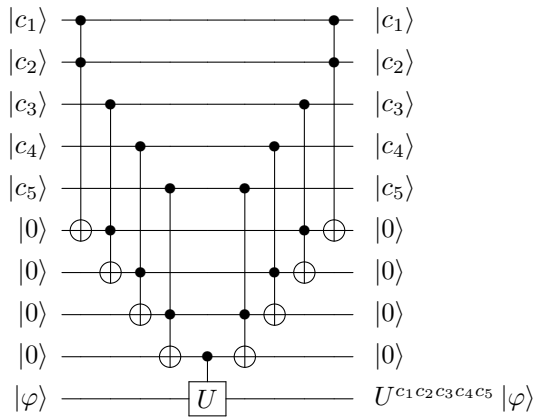


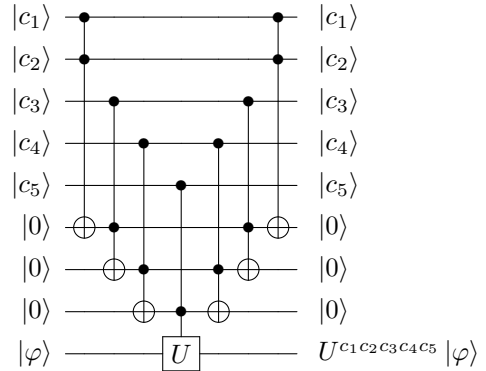
Figure 1.4: Doubly-controlled- U gate. V is defined such that $V^2 = U$.

For a more general method to build a doubly-controlled gate, or to construct a multi-qubit controlled gate, a new concept is introduced. In the following circuits, extra work-qubits are required in the process of implementing the gates. These work-qubits will be referred to as *ancilla qubits* or *ancillae*. These are qubits which are required in the process to perform the operation, but do not provide any useful information at the beginning or end. Preferably the ancillae start out at a

simple state (usually $|0\rangle$), and after the operation are returned to that state. The circuits to make a fivefold-controlled- U gate from either a singly-controlled- U gate or a doubly-controlled- U gate are shown in Figure 1.5. It is seen that $n - 1 = 4$ and $n - 2 = 3$ ancillae are required in the circuits respectively.



(a) Circuit for a fivefold-controlled- U gate, using a singly-controlled- U gate. The process requires $n - 1 = 4$ ancilla qubits.



(b) Circuit for a fivefold-controlled- U gate, using a doubly-controlled- U gate. The process requires $n - 2 = 3$ ancilla qubits.

Figure 1.5: Circuits for a fivefold-controlled- U gate.

Many algorithms require some number of ancilla qubits. Not always however are they returned to their initial state, in which case the output states are referred to as *garbage output*. An algorithm to clear the garbage output exists, at the cost of more ancillae. Consider an operation U shown in Figure 1.6a. In this operation a new notation is introduced; namely the notation for multiple qubits on one line. This is denoted with a slash crossing the line. It provides shorthand notations for larger operations, and it is fully case dependent what is precisely meant with any operation involving multiple qubits per line. In this case, it is meant that the operation takes register $|r\rangle$ and ancillae $|a\rangle$ as input, and transforms these respective registers into a desired output register $|d\rangle$ and garbage-states $|g\rangle$. Instead of the garbage outputs, it is desirable to receive the ancillae state back in their original state, in order to reuse them for other operations. A method to clear the garbage register is shown in Figure 1.6b. It requires as many extra ancillae as there are in the $|d\rangle$ register, initialised to $|0\rangle$. The output $|d\rangle$ can then be ‘copied’ into these qubits using the subroutine shown in Figure 1.7. The word ‘copy’ for the copying subroutine is considered dangerous to use in the context of quantum mechanics, as the No Cloning Theorem [3] states that a quantum state cannot be duplicated; a duplicate will always be entangled with the original state. In the case of the copy subroutine this is precisely desired, and hence the term ‘copy’ is safe to use in this context. A downside of the garbage removal process besides the extra ancillae, is that more than twice as many gates are required in order to undo U and to copy the $|d\rangle$ state.



Figure 1.6: Method to avoid garbage production in a circuit. The registers $|a\rangle$, $|r\rangle$, $|g\rangle$ and $|d\rangle$ are respectively the ancillae, the register, garbage output and desired output. The operation U^\dagger is defined as the inverse of operation U .

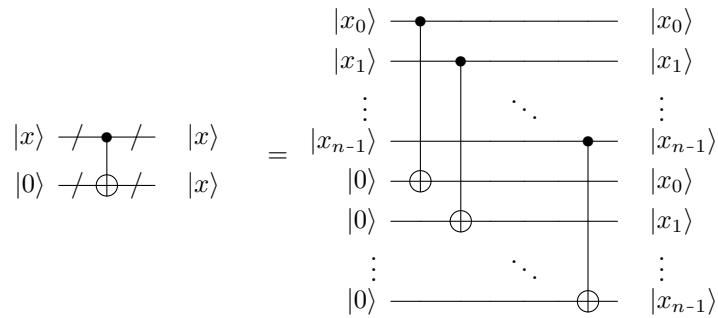


Figure 1.7: Subroutine to ‘copy’ a quantum state into another register

1.3.2 Python code generator

The QX simulator simulates circuits written in the cQASM language, which solely supports syntax to perform gates. A file written in the cQASM language can be fed into the simulator. The simulator returns the output states of the qubits in a command window. An example of cQASM code that adds two 4 qubit values using the Cuccaro Adder algorithm [9] is shown in Figure 1.8a. To aid the creation of complex circuits, a cQASM code generator is desirable. During the writing of this thesis, an OpenQL compiler that takes a high-level language and outputs cQASM code was in development, but it was not yet available in the course of this thesis. Therefore, a code generator for cQASM was built in Python [10]. The tool contains three classes: Quantum Gates, Quantum Subroutines and Quantum Functions. Each of the classes builds upon the previous, and the Quantum Circuit class outputs a circuit in the cQASM language when ran. A Python code example is shown in Figure 1.8b. It is the code that outputs the cQASM file in Figure 1.8a, and can the cQASM code for a Cuccaro Adder of any input size. Additionally, it runs the circuit in the QX Simulator, and retrieves the output state. Every circuit presented in this thesis has been implemented and tested in the QX simulator using the Python code generator, unless stated otherwise. Due to the multiple thousand lines of Python code and tens of thousands of lines of QX code, the code is not included in this thesis. It can instead be found in the GitHub repository https://github.com/Otmar/BEP_Quantum. All code is run through the `RunQX.py` file. In the file, there is a list of booleans that determine which algorithms are ran, including input parameters for most algorithms. The results shown in the thesis are often screen captures of the outputs of this file.


```

from cQASM import *

def MAJ(qna, qnb, qnc):
    gates = []
    gates += [Qgate("cx", qna, qnb)]
    gates += [Qgate("cx", qna, qnc)]
    gates += [Qgate("toffoli", qnc, qnb, qna)]
    return gates

def UMA(qna, qnb, qnc):
    gates = []
    gates += [Qgate("toffoli", qnc, qnb, qna)]
    gates += [Qgate("cx", qna, qnc)]
    gates += [Qgate("cx", qnc, qnb)]
    return gates

class ADD(Qsubroutine):
    def __init__(self, n=1, qubitnames=None, qubitnamesb=None, qubitnamesc=None, qubitnamez=None, do_overflow=True):
        if type(do_overflow) is not type(True):
            raise TypeError("'do_overflow' must be of type Boolean, not '{}'.format(type(do_overflow))")

        qna = buildnames(n, qubitnamesa, "a")
        qnb = buildnames(n, qubitnamesb, "b")
        if qubitnamesc is None:
            qnc = "c"
        else:
            qnc = qubitnamesc
        if do_overflow:
            if qubitnamez is None:
                qnz = "z"
            else:
                qnz = qubitnamez
        else:
            qnz = None

        self.qubitnamesa = qna
        self.qubitnamesb = qnb
        self.qubitnamec = qnc
        if do_overflow:
            self.qubitnamez = qnz

        gates = []

        gates += MAJ(qna[0], qnb[0], qnc)
        for i in range(1, n):
            gates += MAJ(qna[i], qnb[i], qna[i-1])
        if do_overflow:
            gates += [Qgate("cx", qna[-1], qnz)]
        for i in range(n-1, 0, -1):
            gates += UMA(qna[i], qnb[i], qna[i-1])
        gates += MAJ(qna[0], qnb[0], qnc)

        super().__init__(name="add", gates=gates)

class ADDcircuit(Qfunction):
    def __init__(self, inp_a="0", inp_b="0", do_overflow=True):
        name = "Cuccaro Quantum Adder"
        na = len(inp_a)
        nb = len(inp_b)
        n = max(na, nb)
        inp_a = (n-na)*"0" + inp_a
        inp_b = (n-nb)*"0" + inp_b
        qubits = 2*n + 2
        addsubroutine = ADD(n=n, do_overflow=do_overflow)
        qna = addsubroutine.qubitnamesa
        qnb = addsubroutine.qubitnamesb
        qnc = addsubroutine.qubitnamec
        if do_overflow:
            qnz = addsubroutine.qubitnamez
        else:
            qnz = "z"

        if not isinstance(inp_a, str):
            raise TypeError("input must be of type string")
        if not isinstance(inp_b, str):
            raise TypeError("input must be of type string")

        initgates = []
        initgates += [Qgate("map", "q0", qnc)]
        for i in range(n):
            initgates += [Qgate("map", "q"+str(2*i+1), qnb[i])]
            initgates += [Qgate("map", "q"+str(2*i+2), qna[i])]
        initgates += [Qgate("map", "q" + str(2*n + 1), qnz)]
        for i in range(n):
            if inp_a[-i-1] == "1":
                initgates += [Qgate("x", qna[i])]
        for i in range(n):
            if inp_b[-i-1] == "1":
                initgates += [Qgate("x", qnb[i])]
        initgates += [Qgate("display")]

        initsubroutine = Qsubroutine(name="init", gates=initgates)

        resultgates = [Qgate("measure"), Qgate("display")]
        resultsubroutine = Qsubroutine(name="result", gates=resultgates)

        subroutines = [initsubroutine, addsubroutine, resultsubroutine]
        super().__init__(name=name, qubits=qubits, subroutines=subroutines)

if __name__() == "__main__":
    '''Runs the Cuccaro Ripple-Carry Adder'''

    # Inputs for the values
    inp_a = "0010"
    inp_b = "0001"
    na = len(inp_a)
    nb = len(inp_b)
    n = max(na, nb)

    # Write the circuits to a file
    f = open(path + "adder_cuccaro.qc", "w")
    f.write(str(AdderCuccaro.ADDcircuit(inp_a=inp_a, inp_b=inp_b)))
    f.close()

    # Run the circuits in the QX simulator and retrieve the results
    res_add_cuc = runQX('adder_cuccaro', n_tot + 2, return_res=True)
    outp_a_plus_b_cuc = res_add_cuc[-1:-2]

    # Show the results
    print("\nAdder Cuccaro:\ninput a = {} = {} \input b = {} = {} \noutput a+b = {} = {}".format(
        (n-na)*" " + inp_a, int(inp_a, 2),
        (n-nb)*" " + inp_b, int(inp_b, 2),
        outp_a_plus_b_cuc, int(outp_a_plus_b_cuc, 2)))

```

(a) QX code for a four qubit Cuccaro Adder.

(b) Python code for generating a general Cuccaro Adder.

Figure 1.8: QX and Python code for a Cuccaro Adder [9].

2. Quantum Subroutines

With the basics of Quantum Computation introduced in the previous chapter, quantum algorithms can be constructed, with the aim to build algorithms that work ‘faster’ than their classical counterparts. The thinking is that the possible superposition of states can be used to process multiple input configurations in parallel. One of the main difficulties in building efficient quantum algorithms lies in the fact that any information stored in the complex amplitudes of the superimposed states cannot be accessed. As soon as a state is measured, it collapses back into one of the pure states. Algorithms have to be designed around this challenge. Only a relatively small number of core algorithms have been discovered so far that give a speedup compared to classical computers. Two of the most well known of these algorithms are Grover’s Search [11] and the Quantum Fourier Transform [12]. In this chapter the Quantum Fourier Transform is focused on, as it is a key element in the algorithm for solving systems of linear equations. More specifically, the Quantum Linear Solver Algorithm applies the Quantum Phase Estimation Algorithm [13], which relies on the Quantum Fourier Transform. The Quantum Phase Estimation Algorithm will therefore also be discussed in depth in this chapter. The explanations for both algorithms are adapted from [3]. The challenges in the QX implementation of the algorithms will also be discussed in this chapter.

2.1 Quantum Fourier Transform

The Quantum Fourier Transform is the Quantum equivalent of the classical Discrete Fourier Transform, or more precisely, its inverse. On a series of values x_0, \dots, x_{N-1} the Discrete Fourier Transform is defined as the operation $x_0, \dots, x_{N-1} \rightarrow y_0, \dots, y_{N-1}$, with [14],

$$y_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N}. \quad (2.1)$$

This operation will now be rewritten as an operation on qubits $|0\rangle, \dots, |N-1\rangle$. The Quantum Fourier Transform on one of the qubits $|j\rangle$ is defined as,

$$|j\rangle \xrightarrow{\text{QFT}} \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k / N} |k\rangle, \quad (2.2)$$

with $j = 0, 1, \dots, N-1$. For a superposition of the $|0\rangle, \dots, |N-1\rangle$ states with respective amplitudes x_0, \dots, x_{N-1} , the Quantum Fourier Transform is then defined as,

$$\sum_{j=0}^{N-1} x_j |j\rangle \xrightarrow{\text{QFT}} \sum_{k=0}^{N-1} y_k |k\rangle, \quad (2.3)$$

where the amplitudes y_k are the discrete Fourier transform of the amplitudes x_j as defined in Equation (2.1).

The value for N assumed to be a power of two $N = 2^n$ for $n \in \mathbb{N}$, and the basis $|0\rangle, \dots, |N-1\rangle$ to be the basis states of an n qubit quantum computer, ordered from least significant to most significant, i.e. $|0\rangle \equiv |0 \dots 000\rangle$, $|1\rangle \equiv |0 \dots 001\rangle$, $|2\rangle \equiv |0 \dots 010\rangle$, etc. until $|N-1\rangle \equiv |1 \dots 111\rangle$. More generally, each of the states $|j\rangle$ is written as,

$$j = j_1 j_2 \dots j_n = j_1 2^{n-1} + j_2 2^{n-2} + \dots + j_n 2^0, \quad (2.4)$$

for all $j = 0, 1, \dots, N-1$ and $j_k \in \{0, 1\}$ for $k = 1, \dots, n$. The following notation and approximation will be used for values j between zero and one,

$$\begin{aligned} j &= 0.j_\ell j_{\ell+1} \dots j_m \dots \\ &\approx 0.j_\ell j_{\ell+1} \dots j_m \\ &\equiv j_\ell / 2^1 + j_{\ell+1} / 2^2 + \dots + j_m / 2^{m-\ell+1} + \dots, \end{aligned} \quad (2.5)$$

for some $\ell, m \in \mathbb{Z}$ and $j_k \in \{0, 1\}$ for $k \in \mathbb{Z}$. Using this notation, the Quantum Fourier Transform of a single basis state as defined in Equation (2.2) can be rewritten into the following product representation,

$$\begin{aligned} |j_1 j_2 \dots j_n\rangle &\rightarrow \frac{1}{2^{n/2}} \bigotimes_{\ell=1}^n \left[|0\rangle + e^{2\pi i j 2^{-\ell}} |1\rangle \right] \\ &= \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i \cdot (0 \cdot j_n)} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i \cdot (0 \cdot j_{n-1} j_n)} |1\rangle \right) \otimes \dots \otimes \left(|0\rangle + e^{2\pi i \cdot (0 \cdot j_1 j_2 \dots j_n)} |1\rangle \right). \end{aligned} \quad (2.6)$$

For the rewrite from the first to the second line, the fact is used that the binary representation of $j = j_1 j_2 \dots j_n$ with a power of two $2^{-\ell}$ is written as $j \cdot 2^{-\ell} = j_1 \dots j_{n-\ell} j_{n-\ell+1} \dots j_n$, and that $e^{2\pi i k} = 1 = e^{2\pi i 0}$ for any $k \in \mathbb{Z}$. Therefore it is found that $e^{2\pi i j 2^{-\ell}} = e^{2\pi i j_1 \dots j_{n-\ell} j_{n-\ell+1} \dots j_n} = e^{2\pi i 0 \cdot j_{n-\ell+1} \dots j_n}$. The desired operation can be summarised into the circuit as shown in Figure 2.1.

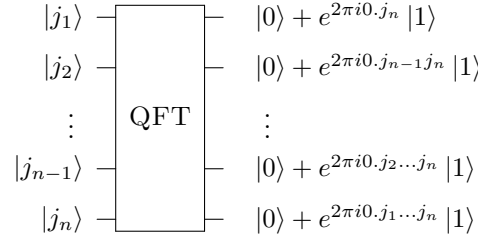


Figure 2.1: Desired operation of the Quantum Fourier Transform.

To perform the Quantum Fourier Transform in a Quantum Circuit, the unitary transformation R_k is introduced for $k \in \mathbb{N}$,

$$R_k \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i / 2^k} \end{bmatrix}. \quad (2.7)$$

On a general state $\alpha |0\rangle + \beta |1\rangle$, the gate has the effect $\alpha |0\rangle + \beta |1\rangle \xrightarrow{R_k} \alpha |0\rangle + \beta e^{2\pi i / 2^k} |1\rangle$. The R_k gates are used to build the Quantum Fourier Transform circuit. The main part of the circuit is shown in Figure 2.2. When compared to the desired operation shown in Figure 2.1, it is seen that the output of the circuit in Figure 2.2 is upside down.

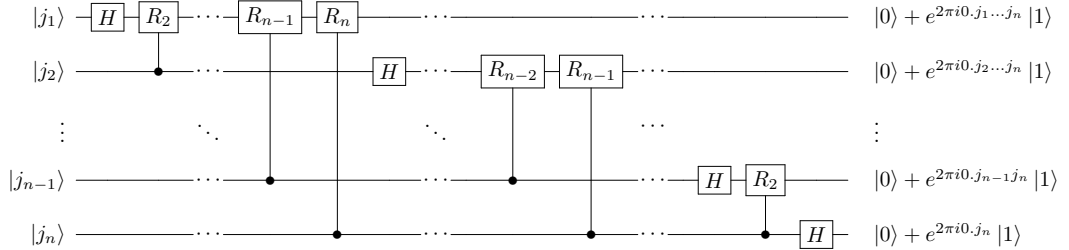


Figure 2.2: Circuit for the Quantum Fourier Transform.

In the circuit, the normalisation $1/2^{n/2}$ is ignored. The effects of the gates will now be examined one by one, to show that the output is indeed that of Equation (2.6). The input state is assumed to be a single value $|j\rangle$, which implies that the values j_k are deterministic. Now consider only the first qubit $|j_1\rangle$. If $j_1 = 0$, then the first H gate transforms the state to $|0\rangle + |1\rangle$, and otherwise to $|0\rangle - |1\rangle$, meaning that

$$\begin{aligned} |j_1\rangle &\xrightarrow{H} |0\rangle + (-1)^{j_1} |1\rangle \\ &= |0\rangle + e^{\pi i j_1} |1\rangle \\ &= |0\rangle + e^{2\pi i \cdot (0 \cdot j_1)} |1\rangle. \end{aligned} \quad (2.8)$$

Again the normalisation is ignored for clarity. Now the effects of the controlled- R_2 gate are examined. If $|j_2\rangle = |0\rangle$, then nothing is changed to the state. If on the other hand $|j_2\rangle = |1\rangle$, then the amplitude of the $|1\rangle$ state is multiplied by $e^{2\pi i/2^2} = e^{2\pi i \cdot 0.01}$. Therefore the total effect of the controlled- R_2 gate on the $|1\rangle$ state is $e^{2\pi i \cdot 0.j_2}$. Since $0.j_1 + 0.0j_2 = 0.j_1j_2$, it is seen that the total state of the first qubit after the controlled- R_2 gate becomes

$$|0\rangle + e^{2\pi i \cdot (0.j_1)} |1\rangle \xrightarrow{C(R_2)} |0\rangle + e^{2\pi i \cdot (0.j_1j_2)} |1\rangle. \quad (2.9)$$

The rewriting steps applied to the R_2 gate can be repeated for all other R_k gates applied to $|j_1\rangle$ to show that the final state for the first qubit becomes $|0\rangle + e^{2\pi i \cdot 0.j_1j_2\dots j_n} |1\rangle$. Moreover, the same method can be repeated to show that the output states are indeed $|0\rangle + e^{2\pi i \cdot 0.j_k\dots j_n} |1\rangle$ for qubit $|j_k\rangle$. The remaining issue is that the outcome of the circuit in Figure 2.2 is upside down. To this end, $\lfloor n/2 \rfloor$ SWAP gates are needed to get the output of the circuit as shown to output the actual Quantum Fourier Transform: one to swap $|j_1\rangle$ and $|j_n\rangle$; one to swap $|j_2\rangle$ and $|j_{n-1}\rangle$; etcetera.

Due to the reversibility of quantum circuits, the Inverse Quantum Fourier Transform directly follows from the Quantum Fourier Transform discussed above: the order of all gates is to be reversed, and all gates are to be replaced by their adjoints. The adjoint of the Hadamard gate is itself, since $HH = I$. The adjoint of the R_k gate for $k \in \mathbb{N}$ is named the R_{-k} gate, and it is defined as,

$$R_{-k} \equiv \begin{bmatrix} 1 & 0 \\ 0 & -e^{2\pi i/2^k} \end{bmatrix}. \quad (2.10)$$

It is indeed seen that $R_k R_{-k} = I$, and thus the adjoint of the R_k gate is the R_{-k} gate. The main part of the Inverse Quantum Fourier Transform circuit is shown in Figure 2.3. Again the swap gates are left out, meaning that the input is upside down. The same reversal subroutine as for the forward Quantum Fourier Transform can be used, which should now be applied before the circuit instead of after it.

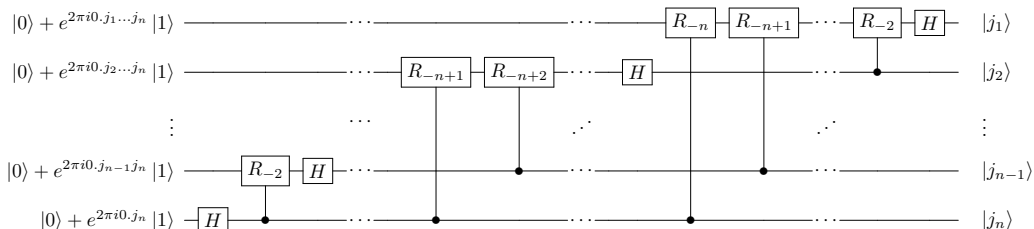


Figure 2.3: Circuit for the Inverse Quantum Fourier Transform.

A difficulty with the Quantum Fourier Transform is that all information of the transform is stored in the complex amplitudes of the output states. Therefore if the output is measured, it collapses back into a single state which destroys all information. Hence, the Quantum Fourier Transform cannot be used as a standalone algorithm, but can instead solely be used as a subroutine in a larger algorithm. In larger circuits the quantum Fourier transform and Inverse Quantum Fourier Transform are often denoted by QFT and QFT † , respectively.

2.2 Quantum Phase Estimation

Consider a unitary operator U which has an eigenvector $|u\rangle$ with eigenvalue $e^{2\pi i\varphi}$, where φ is unknown. It is assumed that φ lies in $[0, 1)$, since any value outside of this interval is mapped onto it. The Phase Estimation Algorithm [13] approximates the value φ , through the efficient construction of the Quantum Fourier Transform of φ which is then transformed into an approximation of φ using the Inverse Quantum Fourier Transform. To find the Quantum Fourier Transform of φ , the Quantum Phase Estimation algorithm uses (1) a memory register with the $|u\rangle$ state, (2) a work register of n qubits initialised at the $|00\dots 0\rangle$ state, and (3) an operator that efficiently performs

a controlled- U^{2^k} operation for any $k \in \mathbb{N}$. That is, an operator that can efficiently perform the U gate 2^k times.

It is important to understand the effect of the controlled- U^{2^k} operation when applied to $|u\rangle$. The effect is a global phase shift of $(e^{2\pi i\varphi})^{2^k} = e^{2\pi i\varphi 2^k}$ when the control qubit is in the $|1\rangle$ state, and no effect when the control qubit is in the $|0\rangle$ state. Therefore, if the control qubit is in the general state $\alpha|0\rangle + \beta|1\rangle$, the effect is

$$\left(\alpha|0\rangle + \beta|1\rangle\right) |u\rangle \xrightarrow{C(U^{2^k})} \left(\alpha|0\rangle + e^{2\pi i\varphi 2^k} \beta|1\rangle\right) |u\rangle. \quad (2.11)$$

This exponential is rewritten to a form that can be related to the Quantum Fourier Transform. Since $\varphi \in [0, 1)$, the binary representation of φ can be written as and approximated to

$$\begin{aligned} \varphi &= 0.\varphi_1\varphi_2\dots\varphi_n\varphi_{n+1}\dots \\ &\approx 0.\varphi_1\varphi_2\dots\varphi_n \\ &\equiv \tilde{\varphi}, \end{aligned} \quad (2.12)$$

with φ_j either 0 or 1. Through this approximation $\varphi \approx \tilde{\varphi}$, $2^k\varphi$ in turn can be approximated to $2^k\tilde{\varphi} = \varphi_1\dots\varphi_k.\varphi_{k+1}\dots\varphi_n$, similar to how multiplication with some power of ten moves the decimal point in basic arithmetic. In the section on the Quantum Fourier Transform, it was stated that $e^{2\pi ik} = 1$ for any integer $k \in \mathbb{Z}$. Application of these equalities shows that the right hand side of Equation (2.11) can be rewritten to

$$\left(\alpha|0\rangle + e^{2\pi i\varphi 2^k} \beta|1\rangle\right) |u\rangle \approx \left(\alpha|0\rangle + e^{2\pi i.(0.\varphi_{k+1}\varphi_{k+2}\dots\varphi_n)} \beta|1\rangle\right) |u\rangle. \quad (2.13)$$

Since $|u\rangle$ is an eigenvector of U , it remains unchanged in the process. Hence, it will be omitted from further calculations.

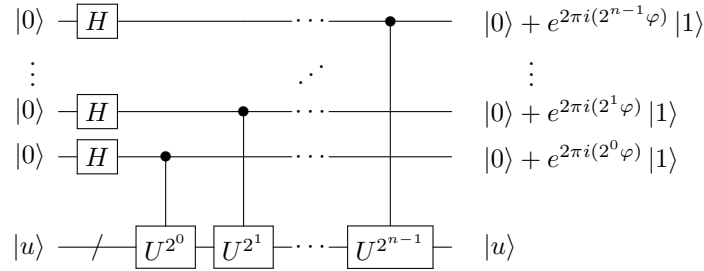


Figure 2.4: First part of the Phase Estimation Algorithm.

The first part of the Phase Estimation Algorithm, which is the part that constructs the Quantum Fourier Transform of φ , is shown in Figure 2.4. In it, first a Hadamard transform is performed on the work register. That is, to each qubit in the work register a Hadamard gate is applied, which yields a state where each qubit in the work register is in an equal superposition of $|0\rangle$ and $|1\rangle$,

$$|00\dots 0\rangle \xrightarrow{H^{\otimes n}} \frac{1}{2^{n/2}} \left(|0\rangle + |1\rangle\right) \otimes \left(|0\rangle + |1\rangle\right) \otimes \dots \otimes \left(|0\rangle + |1\rangle\right). \quad (2.14)$$

The second step is to apply a sequence of n U^{2^k} operations to the memory, controlled by the work qubits. Specifically, for each $k = 1, 2, \dots, n$, a $U^{2^{n-k}}$ gate is applied to $|u\rangle$, controlled by the k -th work qubit. From Equation (2.11) it is seen that these gates perform the operation,

$$\begin{aligned} &\frac{1}{2^{n/2}} \left(|0\rangle + |1\rangle\right) \otimes \left(|0\rangle + |1\rangle\right) \otimes \dots \otimes \left(|0\rangle + |1\rangle\right) \\ &\xrightarrow{(U^{2^{n-k}})^{\otimes n}} \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i\varphi 2^{n-1}} |1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i\varphi 2^{n-2}} |1\rangle\right) \otimes \dots \otimes \left(|0\rangle + e^{2\pi i\varphi 2^0} |1\rangle\right) \\ &\approx \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i.(0.\varphi_n)} |1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i.(0.\varphi_{n-1}\varphi_n)} |1\rangle\right) \otimes \dots \otimes \left(|0\rangle + e^{2\pi i.(0.\varphi_1\varphi_2\dots\varphi_n)} |1\rangle\right). \end{aligned} \quad (2.15)$$

The approximation is performed using Equation (2.13). A comparison with Equation (2.6) shows that the approximation is indeed the Quantum Fourier Transform of $\tilde{\varphi} \approx \varphi$, and hence the application of the Inverse Quantum Fourier Transform on the work register will result in an approximation $\tilde{\varphi}$ of φ in the work register. The quantum circuit depicted in Figure 2.5 shows the complete Quantum Phase Estimation algorithm.

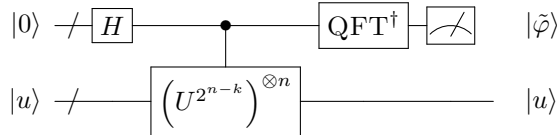


Figure 2.5: Complete Phase Estimation Algorithm. The part of the Hadamard gate and controlled- $(U^{2^{n-k}})^{\otimes n}$ gate is shorthand for the circuit described in Figure 2.4, and the Inverse Quantum Fourier Transform is defined as the opposite of the operation in Figure 2.1.

The net process of the Phase Estimation Algorithm can be summarised as follows,

$$|0\rangle |u\rangle \xrightarrow{\text{Phase Est.}} |\tilde{\varphi}\rangle |u\rangle, \quad (2.16)$$

with $|\tilde{\varphi}\rangle = |\varphi_1\rangle |\varphi_2\rangle \cdots |\varphi_n\rangle$. The work register can be measured to find $\tilde{\varphi}$.

2.3 QX Implementation

In QX, only a restricted form of the controlled- R_k gate is available, in which k cannot be chosen manually. The restricted controlled- R_k gate is called the C_r gate. It applies a controlled- R_k gate, but the value of k depends on the relative position of the qubits: for a control qubit and target qubit that differ in position by $\ell \in \mathbb{Z} \setminus \{0\}$ qubits, the controlled- $R_{|\ell|+1}$ gate is applied. An example for the application of the R_2 gate and R_3 gate is shown in Figure 2.6a and Figure 2.6b.

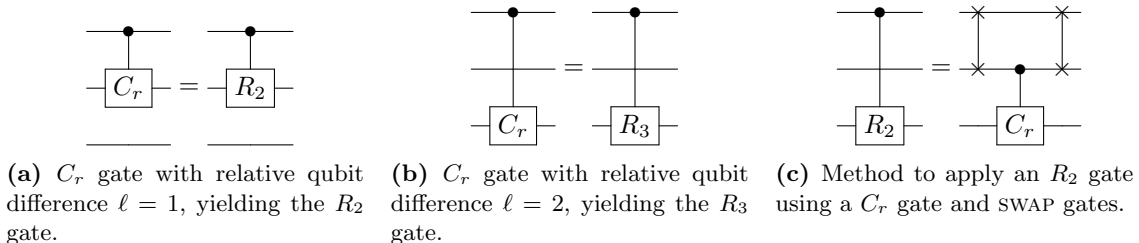


Figure 2.6: QX implementations of the R_k gate using C_r gates.

The C_r gate is optimised for the forward QFT, but it hinders the implementation of other algorithms, since qubit swaps might be necessary to bring the respective qubits in the correct relative positions. An example for the application of the R_2 gate is shown in Figure 2.6c. Moreover, the QX simulator has no native support for the R_{-k} gate. In [15], a method is described to construct one from positive R_k gates. This is accomplished using the equality $e^{-2\pi i/2^k} = e^{(2^k-2)\pi i/2^k} = e^{2\pi i/2} + e^{2\pi i/4} + \cdots + e^{2\pi i/2^k}$, which shows that the R_{-k} gate is equal to the product of the gates R_1 up to R_k , i.e.,

$$-R_{-k} = -R_1 - R_2 \cdots - R_k. \quad (2.17)$$

Again, in the QX simulator the k in the R_k gates is dependent on the relative position of the two qubits, and hence qubit swaps are required in between gates. Consider for example the process for the application of the controlled- R_{-4} gate, which is shown in Figure 2.7.

In some application it is desirable to have an R_k gate not only controlled by a single qubit, but by two control qubits. For example, when using a control qubit to control whether a QFT is performed. The method described in Figure 1.4 can be used to this end. The process makes use

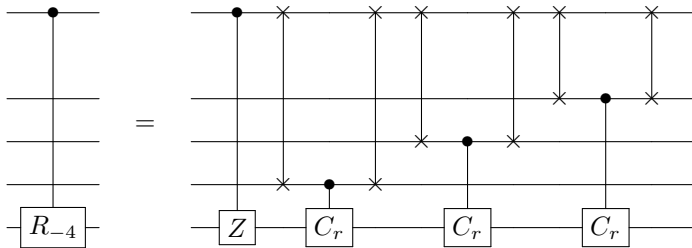


Figure 2.7: QX implementation for a R_{-k} gate for $k = 4$.

of controlled (inverted-)square-root-of- R_k gates, which are controlled- $R_{\pm(k+1)}$ gates. The circuit applying this method is shown in Figure 2.8a. However, the circuit requires the application of R_{-k-1} gates, which have to be built using the cumbersome methods shown in Figure 2.7. Moreover, when only a limited number of qubits is used, not enough qubits may be available to create the necessary qubit distance to apply an R_{k+1} gate. If the use of a single extra ancilla qubit is not an issue, then a second method is available, based on the method shown in Figure 1.5a. The circuit for this second method is shown in Figure 2.8b. It does not require the decomposition of the R_k gate, and thereby reducing the circuit depth considerably.

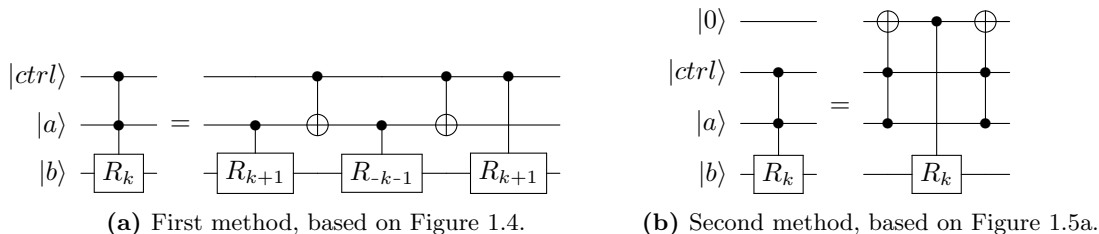


Figure 2.8: QX implementations for a doubly-controlled- R_k gate.

When the Quantum Fourier Transform is made controlled, it is also necessary to have a controlled- H gate. Its implementation is not trivial, and is shown in Appendix A on page 82.

To validate the QX implementation of the Quantum Fourier Transform, a test circuit was constructed. In this circuit, first the QFT was performed, and directly afterwards the Inverse QFT. If the implementation is correct, then the output of the register should be identical to the input. An example of the circuit and the QX output for the three qubit input $|101\rangle$ is shown in Figure 2.9.

It is seen that the output indeed matches the input. The output was validated for all four-qubit inputs $|0000\rangle$ through $|1111\rangle$ and several other values with either higher or lower numbers of qubits, which suggests that the QFT implementation and its inverse indeed function as expected. This will be further verified in the next chapter, where the QFT will be used as subroutine for multiple algorithms, which would be impossible if the QFT was implemented incorrectly.

The Phase Estimation Algorithm is not tested in this chapter, as it depends on an operator U and vector v . As these are not available at this stage, the algorithm will only be implemented when used as a subroutine in a Quantum Linear Solver.

```

#function: Back and forth Quantum Fourier Transform

qubits 3

.init
x q0
x q2
display

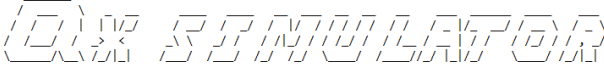
.qft
h q0
cr q1,q0
cr q2,q0
h q1
cr q2,q1
h q2

.display
display

.iqft
h q2
cr q2,q1
cz q2,q1
h q1
cr q2,q0
swap q2,q1
cr q1,q0
swap q2,q1
cz q2,q0
cr q1,q0
cz q1,q0
h q0

.result
measure
display

```



version 0.1 beta - QuTech - 2016 - report bugs and suggestions to: nader.khammassi@gmail.com

```

[+] loading circuit from 'Circuits/qftiqft.qc' ...
[-] loading quantum_code file 'Circuits/qftiqft.qc'...
[+] code loaded successfully.
[+] creating quantum register of 3 qubits...
-----[quantum state]-----
(+1.000000,+0.000000) |101> +
-----
[>>] measurement averaging (ground state) : | +0.000000 | +0.000000 | +0.000000 |
[>>] measurement prediction: | 1 | 0 | 1 |
-----
[>>] measurement register : | 0 | 0 | 0 |
-----
-----[quantum state]-----
(+1.000000,+0.000000) |101> +
-----
[>>] measurement averaging (ground state) : | +0.000000 | +0.000000 | +0.000000 |
[>>] measurement prediction: | 1 | 0 | 1 |
-----
[>>] measurement register : | 1 | 0 | 1 |
-----

```

(a) cQASM Code.

(b) QX ouptut.

Figure 2.9: Back and forth QFT using the three qubit input $|101\rangle$.

3. Integer Arithmetic on a Quantum Computer

In multiple parts of the work done in this thesis it will be necessary to perform basic integer arithmetic, i.e. Integer Addition, Integer Subtraction, Integer Multiplication and Integer Division. In this chapter, multiple algorithms are discussed to perform those operations. More specifically, algorithms for *positive* integers are discussed. The algorithms for subtraction, multiplication and division all depend on addition algorithms, so the algorithms for addition will be described first.

3.1 Integer Addition

In this section, three alternative algorithms to add positive integers on a quantum computer will be described, each named by their respective developers: the Draper [16], Cuccaro [9] and Muños-Coreas [17] algorithms. The Draper algorithm exploits the advantages of quantum effects, by using the Quantum Fourier Transform in the algorithm. The Cuccaro and Muños-Coreas adders on the other hand more closely resemble classical algorithms, as they solely use X , CNOT and TOFFOLI gates. Each algorithm has its own benefits and downsides, which will be discussed after all three algorithms have been introduced.

Before continuing to the actual algorithms, a general overview of the desired operation of an adder is given. The aim of an adder is to construct the sum $a + b$ of two numbers a and b , which in this case are positive integers. The numbers a and b are each saved in a quantum register as a binary number. For example, the number $a = 13$ is stored as $|a\rangle = |1101\rangle$, since $1101_2 = 13_{10}$. Here, the subscripts 2 and 10 depict that the value is to be interpreted as a binary or decimal number, respectively. More generally, if a is saved in an n qubit register, then it is referred to as $a = a_{n-1}a_{n-2}\dots a_0$, where a_{n-1} is the most significant bit, and a_0 the least significant. The notation is chosen such that the index of each bit indicates the power of two it represents, i.e. $a = a_{n-1}a_{n-2}\dots a_0 = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \dots + a_0 \cdot 2^0$. The same notation is used for b . Depending on the adder, the $|b\rangle$ register may or may not be the same size as $|a\rangle$.

In order to use as few qubits as possible, it is desirable to save the sum $a + b$ in one of the two already existing registers. The convention will be used that the $|b\rangle$ register is transformed into the sum $a + b$, while a 's register is left untouched by the algorithm. The desired operation performed by an adder is therefore

$$|a\rangle |b\rangle \xrightarrow{\text{ADD}} |a\rangle |a + b\rangle. \quad (3.1)$$

It is necessary to assume that b 's register is at least as large as that of a in order to fit the sum of both numbers. When the sum register is taken the same size as the largest of the two numbers, problems may still occur. Namely, the sum $a + b$ may require one additional qubit to save compared to a and b separately (e.g. $100_2 + 100_2 = 1000_2$). This extra qubit is called the *overflow* qubit, and it depends on the situation whether the overflow is implemented. If left out, the numbers and sum are called *two's complement* [18]. Then, when the highest value is reached, the register rolls back to 0, e.g. $1111_2 + 0001_2 = 0000_2$ for 4 qubit two's complement numbers, instead of the normal $1111_2 + 0001_2 = 10000_2$. One might also want to be able to control the addition operation, meaning that an external qubit controls whether the addition is performed or not. The inputs and outputs of the most complete adder algorithm, i.e. with both an overflow and a control, are as shown in Figure 3.1.

Another important remark is that it will generally be assumed that a and b are integers. Calculation would not change in any way when placing the decimal point halfway through the numbers, as long as it is in the same place for both numbers. Otherwise padding with zeroes is necessary. This thesis will however not go into detail on non-integer values.

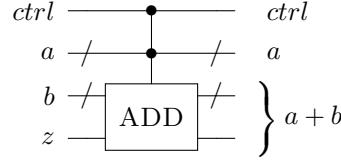


Figure 3.1: Inputs and outputs of a quantum addition circuit implementing both an overflow qubit z and a control $ctrl$.

3.1.1 Draper Adder

One of the earlier algorithms for additions was the Quantum Fourier Transform Adder algorithm introduced by Draper in [16]. The complete algorithm is shown in Figure 3.2. The most significant qubits are as mentioned those with the *highest* index, e.g. $a = a_{n-1}a_{n-2}\dots a_0$. This is precisely the other way around compared to the explanation of the QFT.

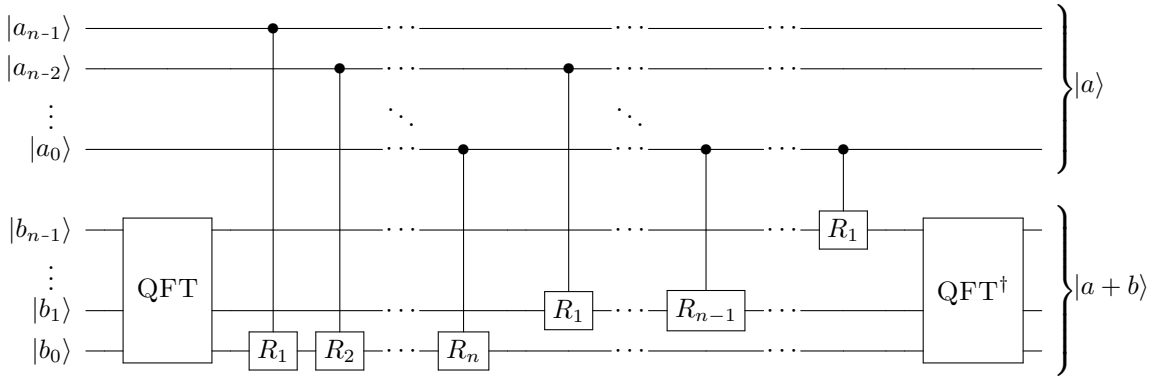


Figure 3.2: Complete Draper Adder.

The algorithm will now be analysed step by step to show that its output is indeed that of Equation (3.1). First, the complete QFT (i.e. including SWAP subroutine) as shown in Figure 2.1 is applied to register $|b\rangle$. This results in the states for $|b\rangle$ shown in Equation (2.6),

$$|b\rangle = |b_{n-1}\rangle |b_{n-2}\rangle \dots |b_0\rangle \xrightarrow{\text{QFT}} \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i \cdot (0.b_0)} |1\rangle \right) \left(|0\rangle + e^{2\pi i \cdot (0.b_1 b_0)} |1\rangle \right) \dots \left(|0\rangle + e^{2\pi i \cdot (0.b_{n-1} b_{n-2} \dots b_0)} |1\rangle \right). \quad (3.2)$$

The final qubit of the $|b\rangle$ register ($|b_0\rangle$) is now examined, in order to visualise the effect of the controlled- $R_1, -R_2, \dots, -R_n$ gates on its state. In the explanation of the Quantum Fourier Transform it was explained that an R_k gate controlled by qubit $a_{n-\ell}$ applied to the state $\alpha |0\rangle + \beta |1\rangle$ has the effect $\alpha |0\rangle + \beta |1\rangle \xrightarrow{R_k} \alpha |0\rangle + e^{2\pi i \cdot 0.0\dots 0 a_{n-\ell}} \beta |1\rangle$, with $\ell - 1$ zeroes between the decimal point and $a_{n-\ell}$. The effect of the controlled- $R_1, -R_2, \dots, -R_n$ gates $|b_0\rangle$ is thus,

$$\begin{aligned} \left(|0\rangle + e^{2\pi i \cdot (0.b_{n-1} b_{n-2} \dots b_0)} |1\rangle \right) &\xrightarrow{R_k^{\otimes n}} \left(|0\rangle + \left[e^{2\pi i \cdot (0.a_{n-1})} e^{2\pi i \cdot (0.0 a_{n-2} \dots 0 a_0)} \right] e^{2\pi i \cdot (0.b_{n-1} b_{n-2} \dots b_0)} |1\rangle \right) \\ &= \left(|0\rangle + e^{2\pi i \cdot (0.a_{n-1} a_{n-2} \dots a_0)} e^{2\pi i \cdot (0.b_{n-1} b_{n-2} \dots b_0)} |1\rangle \right) \\ &= \left(|0\rangle + e^{2\pi i \cdot (0.a_{n-1} a_{n-2} \dots a_0 + 0.b_{n-1} b_{n-2} \dots b_0)} |1\rangle \right). \end{aligned} \quad (3.3)$$

In the same way it can be seen that the effect of the controlled- $R_1, -R_2, \dots, -R_{n-1}$ gates on qubit $|b_1\rangle$ is,

$$\begin{aligned} (|0\rangle + e^{2\pi i 0.b_{n-2}b_{n-3}\dots b_0} |1\rangle) &\xrightarrow{R_k^{\otimes n-1}} (|0\rangle + [e^{2\pi i 0.a_{n-2}} e^{2\pi i 0.0a_{n-3}} \dots e^{2\pi i 0.0\dots 0a_0}] e^{2\pi i 0.b_{n-2}b_{n-3}\dots b_0} |1\rangle) \\ &= (|0\rangle + e^{2\pi i 0.a_{n-2}a_{n-3}\dots a_0} e^{2\pi i 0.b_{n-2}b_{n-3}\dots b_0} |1\rangle) \\ &= \left(|0\rangle + e^{2\pi i (0.a_{n-2}a_{n-3}\dots a_0 + 0.b_{n-2}b_{n-3}\dots b_0)} |1\rangle \right). \end{aligned} \quad (3.4)$$

The steps can be repeated for all other qubits and R_k gates, yielding a total state of the $|b\rangle$ after all R_k gates of,

$$\begin{aligned} &\frac{1}{2^{n/2}} (|0\rangle + e^{2\pi i 0.b_0} |1\rangle) (|0\rangle + e^{2\pi i 0.b_1b_0} |1\rangle) \dots (|0\rangle + e^{2\pi i 0.b_{n-1}b_{n-2}\dots b_0} |1\rangle) \\ &\xrightarrow{R_k^{\otimes n(n+1)/2}} \frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i (0.a_0+0.b_0)} |1\rangle \right) \left(|0\rangle + e^{2\pi i (0.a_1a_0+0.b_1b_0)} |1\rangle \right) \dots \left(|0\rangle + e^{2\pi i (0.a_{n-1}a_{n-2}\dots a_0+0.b_{n-1}b_{n-2}\dots b_0)} |1\rangle \right). \end{aligned} \quad (3.5)$$

This is precisely the Quantum Fourier Transform of $a + b$, which shows that after the application of the Inverse Quantum Fourier Transform subroutine to the b -register indeed the result $|a + b\rangle$ is found in it.

Since the controlled- R_k gates are diagonal matrices (i.e. their only non-zero values lie on the diagonal), it is seen that the the different R_k gates commute, as all diagonal matrices commute. This implies that the order of the R_k gates in the circuit in Figure 3.2 is irrelevant. Moreover, multiple of the R_k gates can even be performed simultaneously. This allows for the rewrite of the circuit shown in Figure 3.3, which has a greatly reduced circuit depth compared to the initial circuit shown in Figure 3.2.

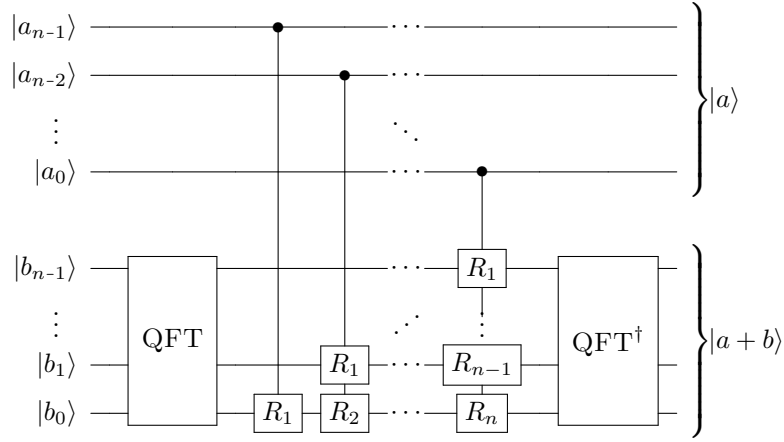


Figure 3.3: Rewrite of the complete Draper Adder optimised for circuit depth.

Now consider a value a saved in a register that is m qubits smaller than the $|b\rangle$ register, i.e.

$$\begin{aligned} a &= 0_{n-1}0_{n-2} \dots 0_{n-m}a_{n-m-1}a_{n-m-2} \dots a_0 \\ &= a_{n-m-1}a_{n-m-2} \dots a_0. \end{aligned} \quad (3.6)$$

Then the first m columns of R_k gates as shown in Figure 3.3 can be omitted, since the a_ℓ for $\ell \geq n - m$ are always zero. The value of a can thus indeed be saved in a smaller register than b , and the circuit can be rewritten to the form shown in Figure 3.4.

The Draper adder does not natively support overflow. This thesis proposes a small extension to the Draper adder that effectively implements an overflow qubit. If the register containing $|b\rangle$ is defined one qubit larger than necessary and if the most significant is left zero, then the method for

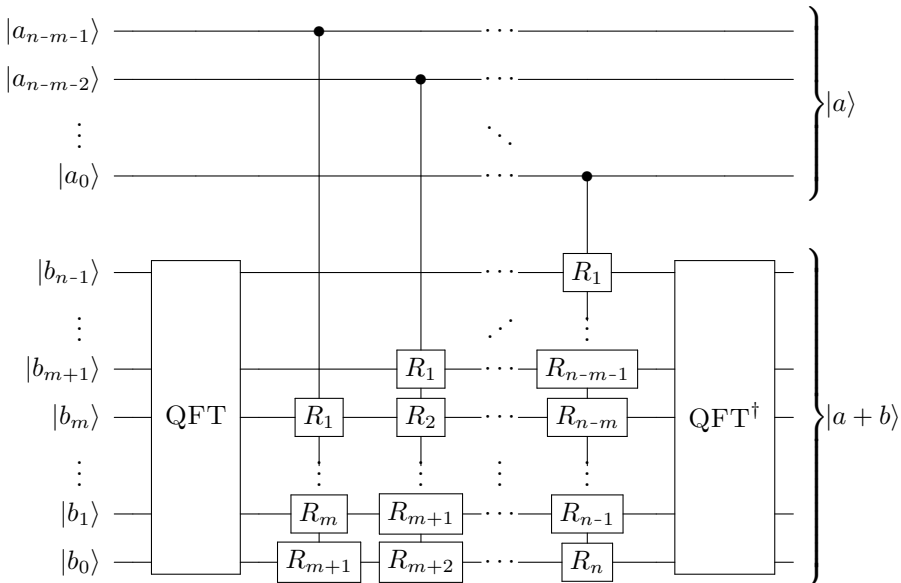


Figure 3.4: Draper adder for a saved in a register m qubits smaller than b .

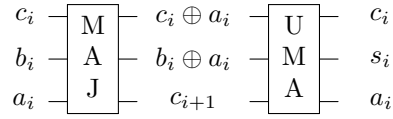
addition of two qubits of different sizes (Figure 3.4) can be applied: if m is taken as $m = 1$ and n is changed to $n \rightarrow n + 1$, then the resulting circuit precisely outputs the sum $a + b$ with overflow in the b register.

3.1.2 Cuccaro Adder

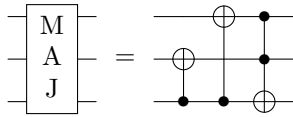
The Cuccaro Addition Algorithm was first described in [9]. This algorithm resembles a classical algorithm, as it does not exploit any quantum-exclusive effects. The Cuccaro adder algorithm is instead based on a classical integer adder algorithm, the so-called *ripple-carry* adder [9]. In the Cuccaro algorithm the numbers are saved in registers of the same size, i.e. as $a = a_{n-1}a_{n-2} \cdots a_1a_0$ and $b = b_{n-1}b_{n-2} \cdots b_1b_0$ for some $n \in \mathbb{N}$.

The Cuccaro Adder algorithm consists of multiple subroutines, which partially solve the addition problem for one significant bit at a time. The first subroutine starts at the least significant bits a_0 and b_0 . The subroutine determines whether a_0 and b_0 are both in the $|1\rangle$ state, in which case the partial sum exceeds the length of one bit, since $1_2 + 1_2 = 10_2$. In that case, the resulting $|1\rangle$ qubit is communicated to the next significance level as a so called *carry bit*, denoted as c_1 . The next subroutine takes the carry bit c_1 and the next two bits a_1 and b_1 as inputs, and determines the carry bit for the next level, c_2 , etcetera. This process is repeated until the most significant qubits a_{n-1} and b_{n-1} , at which level the carry bit is equal to the overflow. The information of the carry bits can now be used to calculate the actual values of $s \equiv a + b$ for each significant bit, which are $s_i = a_i \otimes b_i \oplus c_i$. Here, the “ \oplus ” symbol depicts the XOR operation. In the Cuccaro Quantum Ripple Carry Adder the desired in- and outputs for a single significance level are shown in Figure 3.5a. The circuits for the MAJ and UMA subroutines are defined in Figures 3.5b and 3.5c. Their names stand for *MAJority add* and *UnMAjority Add*, respectively.

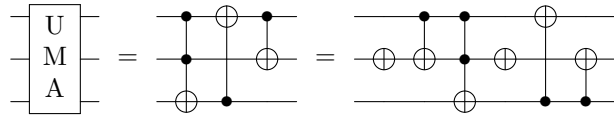
Using the MAJ and UMA subroutines, a full adder can be built using the circuit shown in Figure 3.6. Note that an extra ancilla qubit $|c_0\rangle$ initialised at $|0\rangle$ is necessary to perform the first MAJ and last UMA subroutine in the algorithm. It is possible to omit the overflow qubit $|z\rangle$. In that case the single CNOT in Figure 3.6 is left out, and the sum becomes two’s complement. This also allows for the removal of the last two gates in the last MAJ subroutine and the first two gates of the first UMA subroutine, since these gates now cancel out. A visual difference compared to the Draper adder is the order of the qubits, with the $|a\rangle$ and $|b\rangle$ registers interwoven instead of separated. Since the order of the qubits is physically unimportant, this does not have any impact and is solely done for ease of explanation.



(a) Desired in- and output of a significance level step in the Cuccaro Quantum Ripple Carry Adder.



(b) MAJ subroutine.



(c) UMA subroutine, including rewrite.

Figure 3.5: The MAJ and UMA subroutines used in the Cuccaro Quantum Ripple Carry Adder.

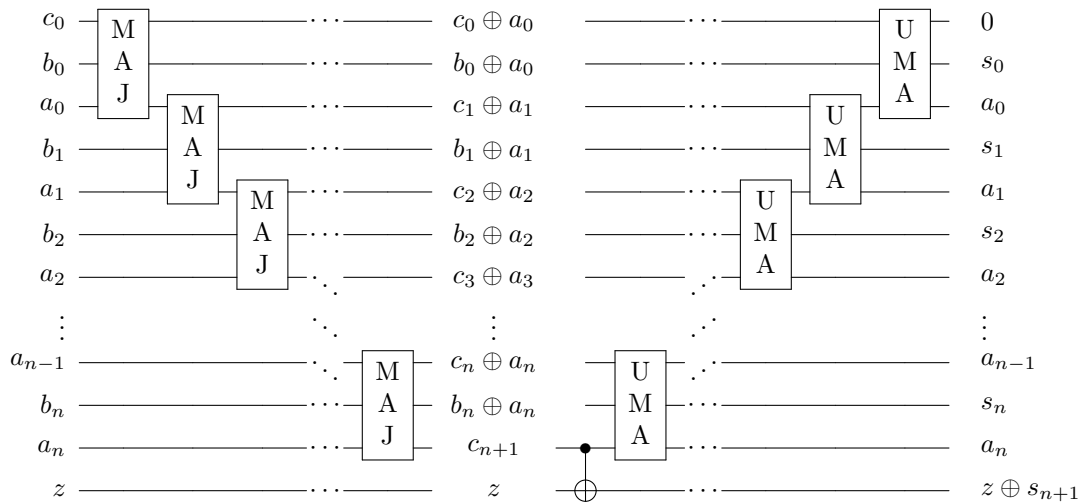


Figure 3.6: Full Cuccaro Adder circuit, with the MAJ and UMA subroutines as defined in Figures 3.5b and 3.5c.

In the Cuccaro paper [9], the circuit is compacted using the rewrite of the UMA subroutine as defined in Figure 3.5c. This rewrite allows for more parallel executions of gates. An example of the rewrite for $n = 5$ is shown in Figure 3.7.

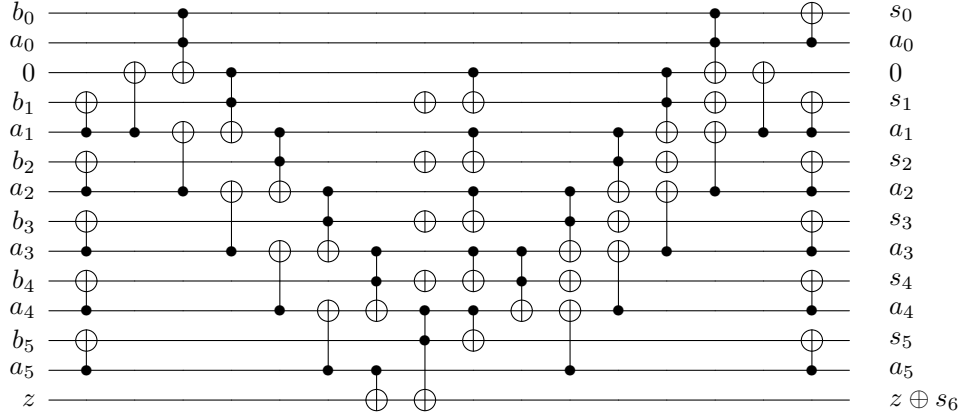


Figure 3.7: Compact parallel rewrite of the Cuccaro Adder as defined in Figure 3.6.

The Cuccaro paper does not elaborate on how to extend the circuit to make the addition controlled by an external qubit. This thesis proposes a minor extension to the Cuccaro Integer adder to add this feature. The non-compact version of the algorithm is transformed into a controlled circuit by changing out the first CNOT in each MAJ subroutine for a TOFFOLI gate also controlled by the control qubit, and the same for the last CNOT in every UMA subroutine. These controlled MAJ and UMA subroutines are shown in Figure 3.8. When the control qubit is in the $|0\rangle$ state, these new TOFFOLI gates are not performed. The algorithm now has no effect, since the MAJ and UMA subroutines are now effectively each other's complements, as TOFFOLI gates and CNOT gates are their own adjoints. When an overflow qubit is used, the single CNOT gate should also be replaced by a TOFFOLI gate additionally controlled by the control qubit.

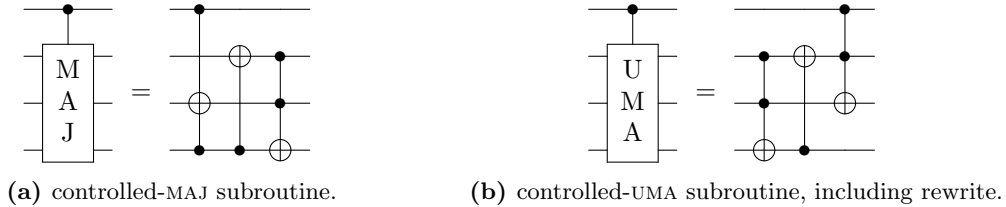


Figure 3.8: The controlled-MAJ and controlled-UMA subroutines used in the controlled Cuccaro Quantum Ripple Carry Adder.

To the author's best knowledge, there is no extension for the compacted form of the Cuccaro Adder to make it controlled. The methods which can be implemented cause the parallelisation to be undone, meaning that the benefit of the rewrite is lost entirely. Therefore, when a controlled version of the Cuccaro adder is desired, the default form is used.

3.1.3 Muñoz-Coreas Adder

A more recent implementation of a Quantum Ripple Carry Adder is described by Muñoz-Coreas et al. in [17]. An example of a Muñoz-Coreas adder circuit for $n = 5$ qubit numbers is shown in Figure 3.9.

The complete Muñoz-Coreas adder requires the same number of gates as the uncompressed Cuccaro adder, and also requires an ancilla qubit, but it has two adaptations with benefits over the

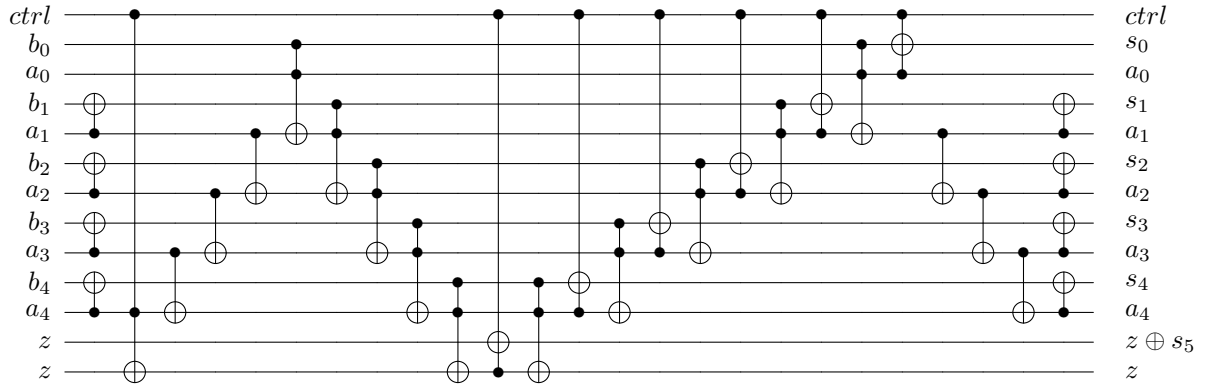


Figure 3.9: Circuit for the complete Muñoz-Coreas Adder for $n = 5$ qubit numbers.

Cuccaro adder. These extensions are that the full adder can be transformed such that it requires no extra ancilla qubit when either no overflow, or no control is needed. The circuits for these adaptations are shown in Figure 3.10 and Figure 3.11 respectively. The adapted circuit in Figure 3.10 was proposed in [18], while the adapted circuit in Figure 3.11 is the outcome of this thesis' work. The adapted circuits will be beneficial in the algorithm used for integer division.

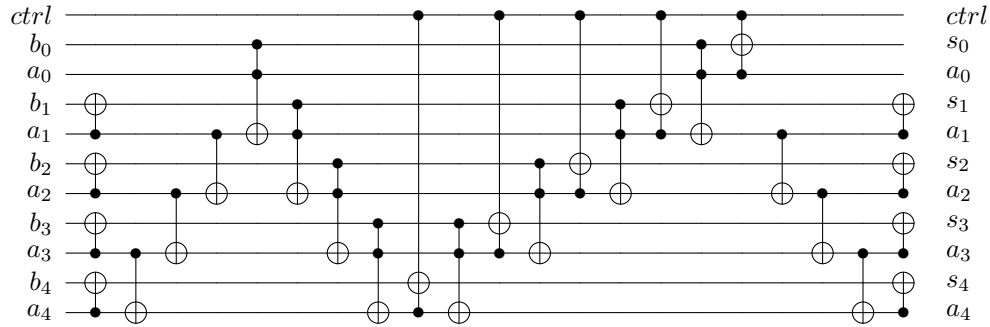


Figure 3.10: Circuit for the Muñoz-Coreas Adder for $n = 5$ qubit numbers, without overflow.

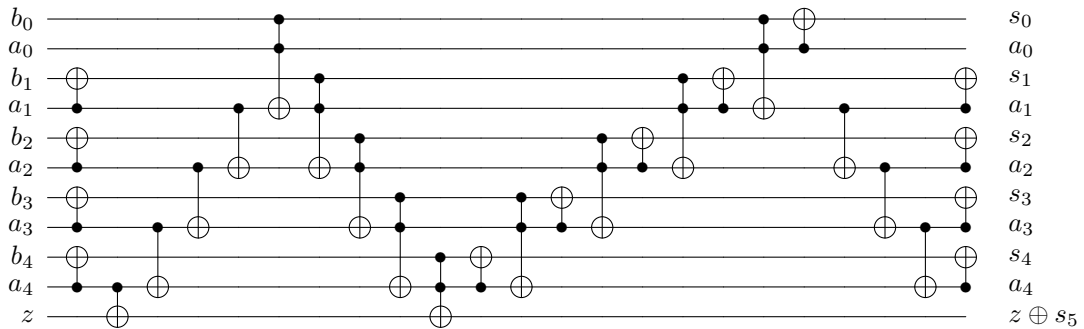


Figure 3.11: Circuit for the Muñoz-Coreas Adder for $n = 5$ qubit numbers, without control.

3.1.4 Discussion on the adders

The three adders introduced in this chapter each have their benefits and downsides, and hence the choice of the 'optimal' adder depends on the application. The main properties of the adders and their variations are listed in Table 3.1.

If the goal of the adder is to add a small number to another larger number, then the Draper Adder is the only efficient option. Besides, it is the only "fully featured" (both controlled and

Category	Draper Default	Cuccaro		Muñoz-Coreas		
		Default	Compact	Default	No control	No overflow
Number of gates ¹	$\frac{3}{2}n(n-1)$	$6n+1$	$9n-8$	$7n-4$	$7n-6$	$7n-8$
Circuit Depth	n^2	$6n+1$ ⁽⁴⁾	$2n+2$	$5n$	$5n-2$	$5n-4$
Number of ancillae	0 ⁽²⁾	1	1	1	0	0
Controlled	Yes ⁽²⁾	Yes	No	Yes	No	Yes
Overflow	Yes ⁽³⁾	Yes	Yes	Yes	Yes	No
Unequal register size	Yes	No	No	No	No	No
Only basic gates	No	Yes	Yes	Yes	Yes	Yes

Table 3.1: Comparison between the different adders.

⁽¹⁾: In the required number of gates, no distinction is made in difficulty to implement gates. This may give a skewed view of the numbers, but is satisfactory for our purposes and outside the scope of this thesis.

⁽²⁾: When making the Draper Adder controlled, doubly-controlled- R_k gates are required, possibly making an ancilla qubit necessary. See the section on QX implementation for details.

⁽³⁾: The overflow can be implemented using an unequal register size, as described in the Draper Adder section.

⁽⁴⁾: Some form of parallelisation is possible with the default Cuccaro Adder, but it becomes impossible when making the adder controlled.

with overflow) adder that does not inherently require an ancilla qubit. However, it is also the only adder with a circuit depth of $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n)$. Moreover, the adder is built from R_k gates, which are very expensive when implemented in the QX simulator due to its lack of a user-defined k in the R_k gates. When doubly-controlled, the R_k gates practically even require an ancilla qubit to efficiently be implemented on the QX simulator. See the section on QX implementation in the previous chapter for more details. All together this makes the Draper Adder far from ideal in practical situations, despite its otherwise desirable features.

The differences between the Cuccaro and Muñoz-Coreas Adders are more subtle; which adder is preferred is strongly case dependent. It should be based on whether parallel gates can be used, whether the operation should be controlled and/or have an overflow. The decision criteria can then be based on what is most important: fewer required qubits or a smaller circuit depth. It will depend on those criteria which adder is the most suited for the situation.

3.2 Integer Subtraction

Any Quantum Integer Adder can be transformed into a Quantum Integer Subtractor through one of two possible methods. In this section, the two methods will be discussed. The Draper Adder additionally has a specific method to be transformed into a subtracter, which will be discussed separately.

3.2.1 General Approach

In this section two methods are shown to transform any adder performing the operation $|a\rangle|b\rangle \rightarrow |a\rangle|a+b\rangle$ for $a, b \in \mathbb{N}$ into a subtracter. The first approach creates the operation $|a\rangle|b\rangle \rightarrow |a\rangle|a-b\rangle$, the second one $|a\rangle|b\rangle \rightarrow |a\rangle|b-a\rangle$. The $b-a$ circuit was introduced by Thapliyal in [19], whereas the $a-b$ circuit is the outcome of this thesis.

To begin with, the method to construct the $b-a$ subtracter is discussed. To this purpose the operation \bar{a} is defined as the bitwise inverse of a , i.e. each bit of a is flipped. For example, $\overline{11010} = 00101$. As was noted in [19], using this operation and an adder the value $b-a$ can be calculated using the equality

$$b-a = \overline{(\bar{b}+a)}. \quad (3.7)$$

In order to understand why this is correct, the algebraic consequences of the \bar{a} operation are examined. For any n -bit number $a \in \mathbb{N}$, the equality $a+\bar{a} = 11 \cdots 1 = 2^n - 1$ holds. Consequently,

$\bar{a} = 2^n - 1 - a$. This second equality allows for the rewrite of the right hand side of Equation (3.7) to indeed find its left hand side,

$$\begin{aligned} \overline{\overline{b+a}} &= 2^n - 1 - [(2^n - 1 - b) + a] \\ &= 2^n - 1 - 2^n + 1 + b - a \\ &= b - a. \end{aligned} \quad (3.8)$$

The equality is used to build the subtracter circuit depicted in Figure 3.12a. The \oplus symbol represents that an X gate is applied to each qubit in the register. The same calculation as in Equation (3.8) can be performed to find that

$$\overline{\overline{a+b}} = a - b, \quad (3.9)$$

which shows how to construct $a - b$. In this case however, the two inversions do not occur in the same register, so attention needs to be paid when making the operation controlled. The method for a controlled subtracter is shown in Figure 3.12b. The CNOT symbol represents that a CNOT gate is applied to each qubit in the register. When the subtraction is not controlled, the CNOT gates are replaced by X gates.

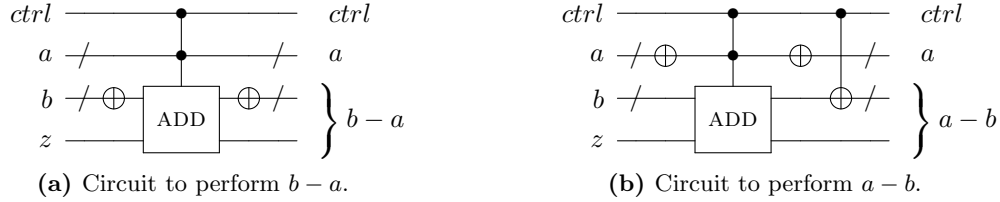


Figure 3.12: General methods to convert an addition circuit into a subtraction circuit.

It may appear contraintuitive that the the overflow qubit is excluded from the inversions in Figure 3.12. In the $b - a$ case, inverting the qubit before and after the addition has no effect, since no other qubits depend on the overflow qubit and therefore the X gates cancel out. In the case of $a - b$, the overflow qubit is only flipped when $2^n - 1 - b < a$, which is precisely also when $b > a$. Hence, the operation is already performed correctly when no CNOT is applied; a CNOT gate would thus precisely yield an incorrect result.

3.2.2 Specific algorithm for the Draper Adder

The Draper Subtracter conversion is also able to perform either the operation $b - a$ or $a - b$. The $b - a$ operation was introduced in [15], whereas the $a - b$ method is introduced in this thesis. Firstly the former is discussed. In the Draper Adder, a is added to b due to the positive rotation of the R_k gates. This implies that if the same R_k gates between the QFT's are applied, only with negative rotation instead, then the state right before the inverse QFT becomes

$$\frac{1}{2^{n/2}} (|0\rangle + e^{2\pi i(0.b_0 - 0.a_0)} |1\rangle) (|0\rangle + e^{2\pi i(0.b_1 b_0 - 0.a_1 a_0)} |1\rangle) \dots (|0\rangle + e^{2\pi i(0.b_{n-1} b_{n-2} \dots b_0 - 0.a_{n-1} a_{n-2} \dots a_0)} |1\rangle). \quad (3.10)$$

This is exactly the requested Quantum Fourier Transform of $b - a$. Note that the negative rotations are exactly obtained when the R_{-k} gates introduced in the previous chapter are used instead of R_k gates. With these gates, the subtraction circuit can be built as shown in Figure 3.13.

If it is instead desired to calculate $a - b$, the signs in all R_k gates should be flipped, including those in the (Inverse) Quantum Fourier Transforms.

3.3 Integer Multiplication

3.3.1 Additive Multipliers

Multiplication of two numbers can always be rewritten as a number of additions. As an example, $123 \cdot 5$ can be written as $100 \cdot 5 + 2 \cdot 10 \cdot 5 + 3 \cdot 5 = 615$. A variation on this method is used in

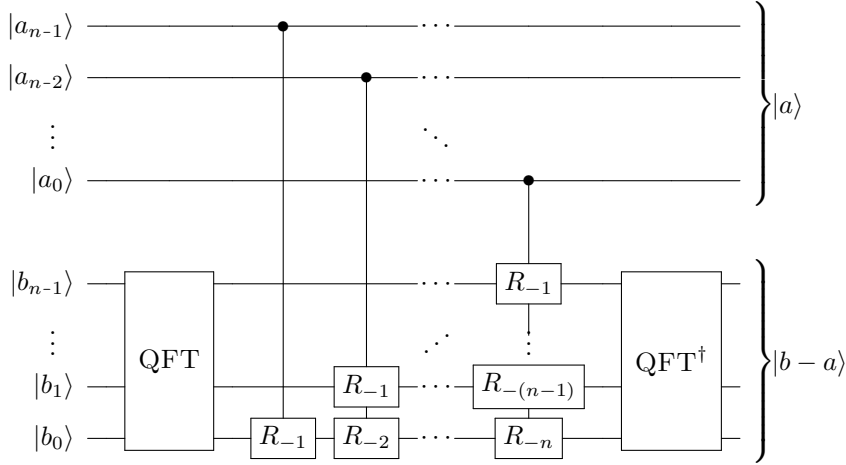


Figure 3.13: Complete Draper Subtractor

the implementation of the Integer Multiplication Algorithm proposed by Vedral in [20], and it was slightly modified by Nguyen in [21]. Again, in this thesis only positive integers are considered.

3.3.1.1 Vedral Multiplier

Consider two positive integers $a, b \in \mathbb{N}$, of m and n qubits in length, respectively. The goal of the Vedral Multiplier is to compute the product $a \cdot b$, which will be an $m + n$ bit integer. In order to accommodate the numbers, three registers $|a\rangle$, $|b\rangle$ and $|c\rangle$ are used of sizes m , n and $m + n$, respectively. The algorithm, which will be referred to as the MUL subroutine in subsequent sections, performs the operation

$$|a\rangle |b\rangle |0\rangle \xrightarrow{\text{MUL}} |a\rangle |b\rangle |ab\rangle. \quad (3.11)$$

The product $a \cdot b$ will now be rewritten in a form that is suitable to be performed only by performing additions on binary numbers. The integer a is written as $a = a_{m-1} \cdots a_1 a_0 \equiv a_{m-1} 2^{m-1} + \cdots + a_1 2^1 + a_0 2^0$, and b is written as $b = b_{n-1} \cdots b_1 b_0 \equiv b_{n-1} 2^{n-1} + \cdots + b_1 2^1 + b_0 2^0$. Note that multiplication by 2^ℓ in binary effectively adds ℓ zeroes at the end of the number it is multiplied with, e.g. $101 \cdot 2^3 = 101000$. The product $a \cdot b$ can therefore be rewritten as follows,

$$\begin{aligned} a \cdot b &= (a_{m-1} 2^{m-1} + \cdots + a_1 2^1 + a_0 2^0) \cdot b \\ &= a_{m-1} \cdot (2^{m-1} \cdot b) + \cdots + a_1 \cdot (2^1 \cdot b) + a_0 \cdot (2^0 \cdot b) \\ &= a_{m-1} \cdot (b_{n-1} \cdots b_1 b_0 00 \cdots 0) + \cdots + a_1 \cdot (b_{n-1} \cdots b_1 b_0 0) + a_0 \cdot (b_{n-1} \cdots b_1 b_0). \end{aligned} \quad (3.12)$$

The number of zeroes after b_0 at the start of the last line is equal to $m - 1$. The equation shows that the product $a \cdot b$ is equivalent to m controlled additions: first, of b controlled by a_0 , then of b shifted by one bit controlled by a_1 etcetera, until b shifted by $m - 1$ bits controlled by a_{m-1} .

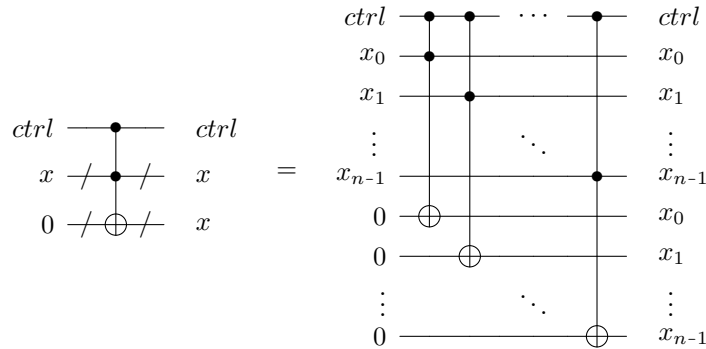


Figure 3.14: Subroutine to ‘copy’ a binary state into another register, controlled by a single qubit.

This particular order of additions is chosen deliberately. Namely, it was assumed that register $|c\rangle$ is initialised as $|0\rangle$, which means that adding b to it only requires a controlled ‘copying’ subroutine to duplicate b into $|c_{n-1} \cdots c_1 c_0\rangle$. The controlled-‘copy’ subroutine is an adaptation of the ‘copy’ subroutine shown in Figure 1.7, and is shown in Figure 3.14. For the next steps of adding $2^1 b$ through $2^{m-1} b$ however, ADD subroutines are required, since the product register is not empty anymore. The complete circuit is shown in Figure 3.15.

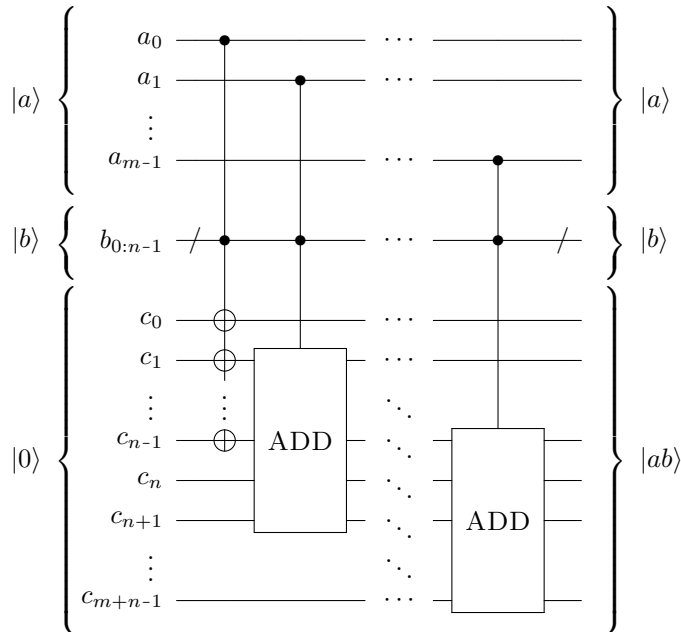


Figure 3.15: Verdral Multiplication Algorithm circuit.

It is noteworthy that the ADD subroutines cover only a part of the qubits of the product register. This is allowed because when summing a binary number ending in k zeroes (e.g. 101000), the addition has no impact on those k least significant bits of the outcome (3 in the case of the example, e.g. $101000 + 011 = 101011$). Additionally, since the addition steps are ordered from small to large, the most significant qubits of register $|c\rangle$ are always zero, meaning that any overflow can only reach one qubit further than the added $2^k b$. For example, after the copying subroutine the most significant m qubits of the product register are zero, i.e. the number in the product register is smaller than 2^n . The value $2^1 \cdot b$ that is then possibly added in the first ADD-subroutine has a maximum value of $2^{n+1} - 1$, implying that the sum can never exceed $2^{n+2} - 1$, and thus only a single overflow at qubit c_{n+1} is necessary to save the sum. The last $m - 1$ qubits of the product register therefore remain zero after this subroutine, and the method can be repeated for the next ADD subroutine.

3.3.1.2 Nguyen Extension

In some situations it is beneficial to apply a sum and product at the same time, i.e. the goal is to not only calculate the product $a \cdot b$, but also add some value c at the same time, resulting in the final desired answer of $c + ab$. The Verdral algorithm just discussed can easily be transformed to have those desired in- and outputs. The adaptation will be referred to as MULADD, and has the following effect,

$$|a\rangle |b\rangle |c\rangle \xrightarrow{\text{MULADD}} |a\rangle |b\rangle |c + ab\rangle. \quad (3.13)$$

There are two possible situations, which require different circuits. The distinction lies in whether $c < 2^n$. In other words, whether c can (like b) be written as an n qubit number $c_{n-1} \cdots c_1 c_0$ or not. If it is the case, then the only change necessary is that the copying subroutine needs to be replaced by an ADD subroutine. The ADD subroutine should add $b \cdot 2^0$ to the product register, controlled

by a_0 . This change results in a circuit that can deal with the addition of c to the product ab . The complete circuit is shown in Figure 3.16. The algorithm is again allowed to have the sums only partially cover the $|c\rangle$ register, since all qubits $|c_i\rangle$ with $i \geq n$ start out at zero, meaning that after the first step only qubits $|c_0\rangle, |c_1\rangle, \dots, |c_n\rangle$ can have non-zero values, after which the algorithm can continue as described before.

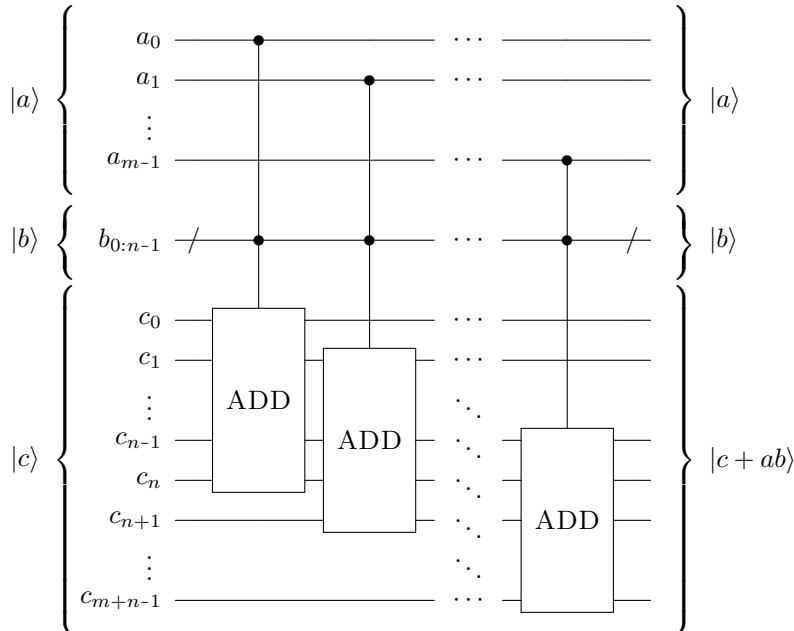


Figure 3.16: Nguyen Multiplication adaptation capable of dealing with the addition of $c < 2^n$.

For the second case, where $c \geq 2^n$, an adaptation of the above algorithm is possible that allows a c up to a value of $2^{m+n} - 1$ to be added alongside the multiplication. In this case, all c_i in the product register can be non-zero, which means that the assumptions that the higher order qubits in the product register can no longer be assumed zero, and hence any addition $2^k b$ can now affect the more significant qubits of c previously unaffected. It is now also possible for the sum of c and ab to exceed $2^{m+n} - 1$, meaning that an extra overflow qubit is necessary. To accommodate these new requirements, the additions themselves can remain the same, but the addition subroutines now need to be able to deal with the changes in the most significant qubits of the c register. The circuit that can deal with these changes is shown in Figure 3.17. In this case, the register for b is smaller than the sub-registers of register c to which b is being added. To circumvent the need for multiple ancilla qubits, the Draper QFT Adder is required to perform these additions.

It may also be desired not to add but to subtract the product ab from c . The circuit for this operation, called the MULSUB operation, is proposed in this thesis and has the following effect,

$$|a\rangle |b\rangle |c\rangle \xrightarrow{\text{MULSUB}} |a\rangle |b\rangle |c - ab\rangle. \quad (3.14)$$

To change a MULADD operation into a MULSUB subroutine, only the ADD subroutines need to be replaced by SUB subroutines. However, only the second method can be used, since we now always either have a c larger than ab (meaning non-zero high significance qubits), or an overflow of the register will occur, which will give an erroneous result with the first method in both cases.

3.3.2 Alternative Integer Multiplication Algorithms

The Vedral multiplier has a circuit depth of $\mathcal{O}(n^2)$ [20]. There are multipliers with circuit depths of $\mathcal{O}(n^{1.5})$ using Karatsuba's algorithm ("Divide and conquer") [22, 23, 24]. These methods do require extra ancillae. The goal of this thesis is to get a fully functioning QLSA implementation.

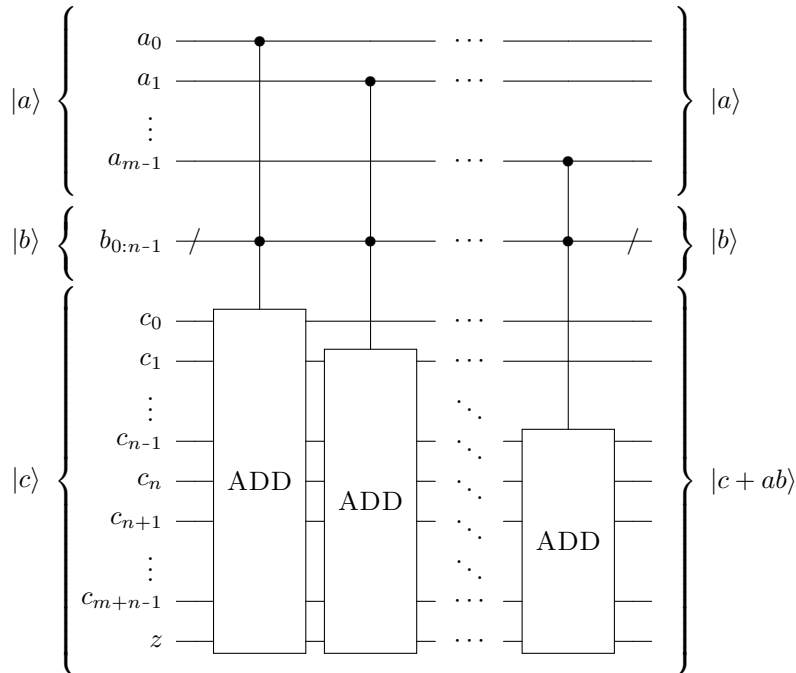


Figure 3.17: Complete Straightforward Multiplication Algorithm adaptation capable of dealing with the addition of any $c < 2^{m+n}$.

The tuning of individual components, such as a more efficient Integer Multiplication Algorithm, is therefore not a primary goal of this thesis, and therefore left for future research.

3.4 Integer Division

The final operation of the basic integer arithmetic is Integer Division, where the goal is to find the quotient a/b of two integers $a, b \in \mathbb{N}$. The classical concept is less straightforward than the previous algorithms, since it is not necessary for the outcome of the division a/b to be an integer. Consider for example $7/4 = 1.75 \notin \mathbb{N}$. The outcome to integer division a/b is therefore defined differently, as two integers q and r , such that $a = q \cdot b + r$ and $r \in \{0, 1, \dots, b-1\}$. q is called the quotient and r the remainder. Thapliyal et al. describe an Integer Division Algorithm in [18] and [19] which solves the problem without requiring any ancillae. However, attempts to implement the algorithm as described in [18] and [19] resulted in different outcomes compared to the descriptions. The quotient and remainder ended up in each others registers, and the algorithm failed for $b > 100 \dots 00_2 = 2^{n-1}$. An adaptation of the algorithm is proposed in this thesis which works experimentally, and which requires no extra qubits or gates compared to the Thapliyal algorithm. Although the interpretation of the steps performed in the adaptation is completely different from that in [18] and [19], the changes to the circuits in the algorithm are only minor.

3.4.1 Adapted Thapliyal Algorithm

3.4.1.1 Overview

For any two n -qubit integers a and b , the adapted Thapliyal Integer Division Algorithm aims to find the n -qubit integers q and r , such that $a = q \cdot b + r$. For the algorithm $3n$ qubits are required. Due to the increased complexity of the integer division algorithm compared to the other algorithms described in this chapter, explicit register names are used. The $3n$ qubits are ordered into three registers N , Q and D , each of size n . Specifically, the $3n$ qubits are labelled as

$$|\psi_{3n-1} \dots \psi_{2n} \psi_{2n-1} \dots \psi_n \psi_{n-1} \dots \psi_0\rangle \equiv |D_{n-1} \dots D_0 Q_{n-1} \dots Q_0 N_{n-1} \dots N_0\rangle, \quad (3.15)$$

with $|\psi_i\rangle$ the qubits. The registers N and D are initialised to $|a\rangle_N$ and $|b\rangle_D$, the binary representations of a and b , respectively. Register Q is initialised as $|0\rangle_Q$, and is transformed into $|q\rangle_Q$, the binary representation of the desired quotient q . During the algorithm, register D is left untouched, while register N is transformed into $|r\rangle_N$, the binary representation of the remainder r . The total operation of the algorithm therefore becomes,

$$|a\rangle_N |0\rangle_Q |b\rangle_D \rightarrow |r\rangle_N |q\rangle_Q |b\rangle_D. \quad (3.16)$$

As with the other algorithms for integer arithmetic, the most significant qubit will be taken as the qubit with the highest index, i.e. $|a\rangle_N = |a_{n-1}\rangle_{N_{n-1}} |a_{n-2}\rangle_{N_{n-2}} \cdots |a_1\rangle_{N_1} |a_0\rangle_{N_0}$, and $|b\rangle_D = |b_{n-1}\rangle_{D_{n-1}} |b_{n-2}\rangle_{D_{n-2}} \cdots |b_1\rangle_{D_1} |b_0\rangle_{D_0}$. The input and output states of the circuit are therefore as shown in Figure 3.18.

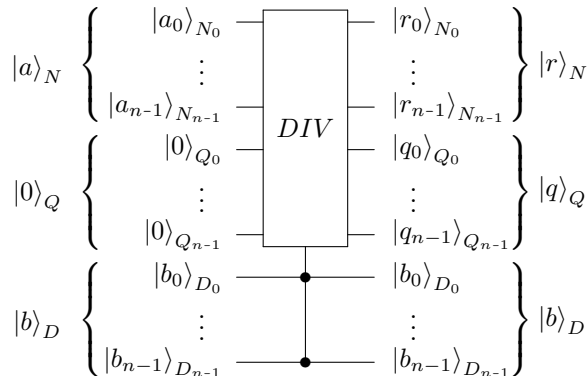


Figure 3.18: In- and outputs of the Thapliyal Integer Division Algorithm.

The algorithm applies a variation on the classical Long Division Algorithm [25]. Long Division is comparable to the manual method for division (“staartdeling”), and works in n iterations. First it looks at whether $2^{n-1} \cdot b$ can be subtracted from a . If that is possible, then the amount is removed from a , and 2^{n-1} is added to q . Otherwise, nothing is done. Secondly, the same steps are repeated, but now with 2^{n-1} replaced by 2^{n-2} , i.e. it is examined whether $2^{n-2} \cdot b$ can be subtracted from (what remains of) a . If that is possible, then the amount is removed from (what remains of) a , and 2^{n-2} is added to q . The process is repeated for smaller powers of two, until the n -th step where 2^0 is used instead of 2^{n-1} . After the n -th iteration, the only thing left of a will be the remainder r , while q has been transformed to its desired value $\lfloor a/b \rfloor$.

3.4.1.2 The Adapted Thapliyal Integer Division Algorithm

The Adapted Thapliyal Integer Division Algorithm uses the same n iterations, indicated by the index $i \in \{0, \dots, n-1\}$. In the i -th iteration, it examined whether $2^{n-i-1} \cdot b$ can be subtracted from register N (i.e. from what is left of a after the first $i-1$ iterations). If this is the case, then 2^{n-i-1} is added to register Q (i.e. to what will become q after all n iterations). In the actual algorithm, an iteration will more closely resemble the following steps,

1. subtract $2^{n-1} \cdot d$ from N ,
2. if the result is negative, add back $2^{n-1} \cdot b$ to N ,
3. otherwise if the result is non-negative, add 2^{n-1} to Q .

To facilitate the concrete definition of the iterations, two sub-registers $Y(i)$ and $Z(i)$ are defined for $i \in \{0, \dots, n-1\}$. Sub-register $Y(i)$, of length n , is the collection of the qubits with the same size as register N , but moved $n-i-1$ qubits downward. Sub-register $Z(i)$, of length 1, is the first qubit below the sub-register $Y(i)$. Concretely, these registers are defined as

$$\begin{aligned} |Y(i)\rangle &= |Q_{n-i-2} \cdots Q_0 N_{n-1} \cdots N_{n-i-1}\rangle = |\psi_{2n-i-2} \cdots \psi_{n-i-1}\rangle, \\ |Z(i)\rangle &= |Q_{n-i-1}\rangle = |\psi_{2n-i-1}\rangle. \end{aligned} \quad (3.17)$$

With the help of these sub-registers, the iterations are defined as,

1. subtract $|b\rangle_D$ from $|Z(i)\rangle \otimes |Y(i)\rangle$, with $|Z(i)\rangle$ overflow
2. add back $|b\rangle_D$ to $|Y(i)\rangle$ without overflow, controlled by $|Z(i)\rangle$,
3. invert $|Z(i)\rangle$.

Notice that these additions and subtractions can be performed without ancilla qubits when using the Muñoz-Coreas Adders shown in Figure 3.11 and Figure 3.10. One full iteration is shown in Figure 3.19.

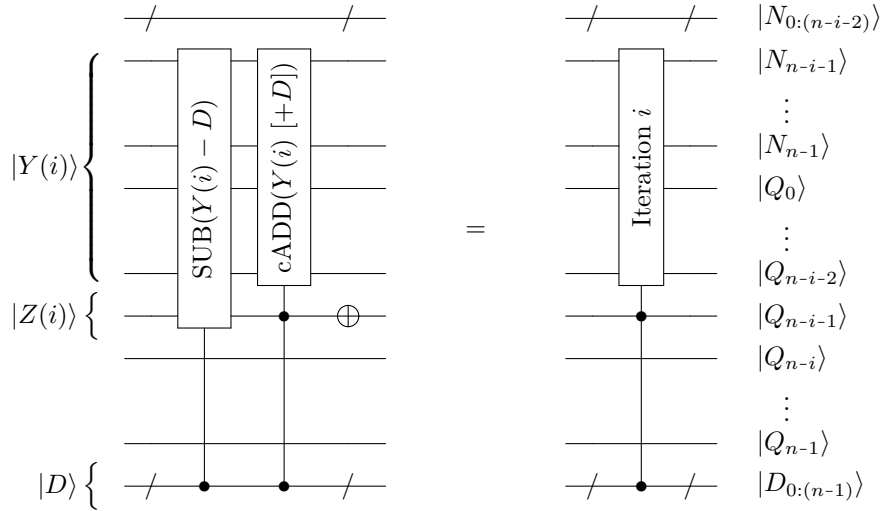


Figure 3.19: Single iteration in the Thapliyal Integer Division Algorithm for an arbitrary number of qubits n . The iteration number i should lie in the range $0 \leq i \leq n - 1$.

A complete circuit for $n = 4$ is shown in Figure 3.20.

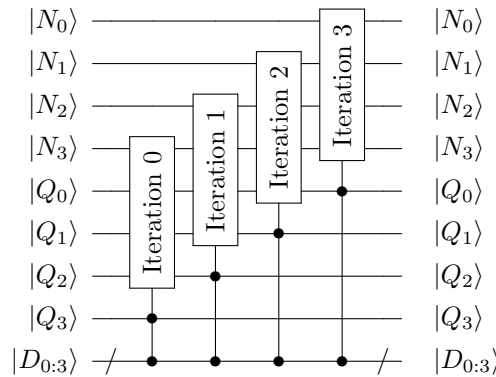


Figure 3.20: Complete Adapted Thapliyal Integer Division Algorithm for $n = 4$.

3.4.1.3 Step-by-step explanation of the algorithm

In this section the interpretation of the steps in the iterations is discussed. The three concrete steps taken in an iteration are examined to see that they closely align with the three-step breakdown shown earlier.

Recall that $2^{n-1} \cdot b$ written in binary is the same as b ($= |D\rangle$), but with $n - 1$ trailing zeroes. Also note that the combination $|RQ\rangle$ of registers $|R\rangle$ and $|Q\rangle$ can be viewed as a binary number

$R_{n-1} \cdots R_0 Q_{n-1} \cdots Q_0$. Since $|R\rangle$ is initialised at zero, and since $|Q\rangle$ start out as $|a\rangle$, the combination register $|RQ\rangle$ can also be viewed as being initialised as $|a\rangle$, but with extra zeroes before the number. Note that $|Y(1)\rangle$ lies within $|RQ\rangle$, in such a way that it is moved $n - 1$ digits from $|Q\rangle$ (to the more significant side). This means that if $|D\rangle$ ($= |b\rangle$) is subtracted from $|Y\rangle$ (as in step 1 of iteration 1), that this is the same as subtracting $2^{n-1} \cdot b$ from a , which is exactly as desired.

The two possible cases of $2^{n-1} \cdot b \leq a$ and $2^{n-1} \cdot b > a$ are now examined separately, in order to understand the consequences of Step 1. If $2^{n-1} \cdot b$ was smaller than or equal to a (which in the first Iteration is only the case when $b \leq 1$), then nothing happens to the overflow qubit $|Z(1)\rangle = |Q_{n-1}\rangle$. In other words, it remains zero since the result after the subtraction stays above or at zero. However, when $2^{n-1} \cdot b$ is larger than a , then the result is that the overflow qubit $|Z(1)\rangle$ rolls over, since the result after the subtraction becomes negative. The overflow qubit $|Z(1)\rangle = |R_{n-1}\rangle$ is therefore effectively behaves as a sign indicator.

If this sign is negative (i.e. $|1\rangle$) after Step 1, it means that b should be added back to $|Y(1)\rangle$, while this should not happen if the sign is positive (i.e. $|0\rangle$). This is exactly what happens in Step 2, where the sign qubit controls the re-addition of b to $|Y(1)\rangle$.

Now the only step left in the first iteration is Step 3. That is, to save 2^{n-1} to $|Q\rangle$ when the sign is positive (i.e. $|0\rangle$), and not to save it when the sign is negative. Notice that qubit $|Q_{n-1}\rangle$ is the qubit of $|Q\rangle$ with significance 2^{n-1} . But remember that precisely $|Q_{n-1}\rangle = |Z(1)\rangle$. However, $|Z(1)\rangle$ is exactly $|1\rangle$ if it is desired to be $|0\rangle$ and vice versa. Hence, only this sign qubit is to be flipped. This is precisely what is performed in Step 3.

Next, the second iteration is performed. Note that it is critical that $|Z(2)\rangle$ starts out at zero, since otherwise the controlled re-addition (Step 2) and qubit flip (Step 3) will happen exactly the wrong way around. However, this is inherently the case after the first iteration: if there was no sign-flip, the register- Q -qubits of $|Y(1)\rangle$ (and especially $|Z(2)\rangle = |Y(1)_{n-1}\rangle$) are left untouched and remain at zero, whereas if there was a sign flip (possibly changing $|Y(2)\rangle$), the subtraction is undone in Step 2, so again it is found that $|Y(2)\rangle$ ends up at zero!

The iterations can be repeated until the last iteration, after which a binary representation of q has formed in register $|Q\rangle$, and a binary representation of r is left in register $|R\rangle$.

An interesting realisation about the second to last paragraph, is that a single error in $|Y(i)\rangle$ for a low i can have a dramatic effect on the outcome of the algorithm. This may make the algorithm relatively prone to the noise experienced in real Quantum Computers. Thorough analysis of this possible phenomenon however is beyond the scope of this thesis.

3.4.1.4 Extensions to the algorithm and remarks

In this final section an extension to the algorithm is discussed, which allows for a larger register size for a than for b . The results of division by zero are also briefly touched upon, as well as possible future extensions for negative numbers.

First, the extension for a larger register size for a is discussed. Consider a value a that is saved in a register m qubits larger than that of b , which is still saved in an n qubit register. The value a is therefore saved in a $n + m$ qubit register. In the extension of the Integer Division algorithm, the quotient q and remainder r will not both have qubit size n . It turns out, however, that only the register for q will change in size. Namely, it was defined that r should be smaller than b , meaning that r keeps its maximum size of n qubits. The quotient q on the other hand can now become larger; suppose $b = 1$, then q will become the largest at $q = a$, implying that q now requires an $n + m$ qubit register. It is therefore observed that an equal amount of qubits is necessary to store q and r compared to a and b , at $2n + m$ qubits. Hence, no extra qubits are required at this stage. However, limitation will be necessary for the possible values for b . This will be discussed in depth later on. The only major change to the algorithm is that the Iterations subroutine will be

performed $n + m$ times instead of just n times. Additionally, the registers are renamed and new ones are introduced. The total number of required qubits will now be $3n + m$, which will again be split into three registers, only this time in twofold. Firstly into the three registers N , O and D (of sizes $n + m$, n and n , respectively), and afterwards into R , Q and D (of sizes n , $n + m$ and n , respectively), which are defined as follows,

$$\begin{aligned}
|N\rangle &= |N_{n+m-1} \cdots N_0\rangle = |\psi_{n+m-1} \cdots \psi_0\rangle, \\
|O\rangle &= |O_{n-1} \cdots O_0\rangle = |\psi_{2n+m-1} \cdots \psi_{n+m}\rangle, \\
|D\rangle &= |D_{n-1} \cdots D_0\rangle = |\psi_{3n+m-1} \cdots \psi_{2n+m}\rangle, \\
|R\rangle &= |R_{n-1} \cdots R_0\rangle = |\psi_{n-1} \cdots \psi_0\rangle, \\
|Q\rangle &= |Q_{n+m-1} \cdots Q_0\rangle = |\psi_{2n+m-1} \cdots \psi_n\rangle.
\end{aligned} \tag{3.18}$$

This means that all the qubits can be written in two different ways:

$$\begin{aligned}
&|\psi_{3n+m-1} \cdots \psi_{2n+m} \psi_{2n+m-1} \cdots \psi_{n+m} \psi_{n+m-1} \cdots \psi_n \psi_{n-1} \cdots \psi_0\rangle \\
&= |D_{n-1} \cdots D_0 O_{n-1} \cdots O_0 N_{n+m-1} \cdots N_n N_{n-1} \cdots N_0\rangle \\
&= |D_{n-1} \cdots D_0 Q_{n+m-1} \cdots Q_m Q_{m-1} \cdots Q_0 R_{n-1} \cdots R_0\rangle.
\end{aligned} \tag{3.19}$$

The inputs and outputs of the algorithms can now be defined as

$$|a\rangle_N |0\rangle_O |b\rangle_D \rightarrow |r\rangle_R |q\rangle_Q |b\rangle_D. \tag{3.20}$$

The total desired operation of the algorithm is shown in Figure 3.21.

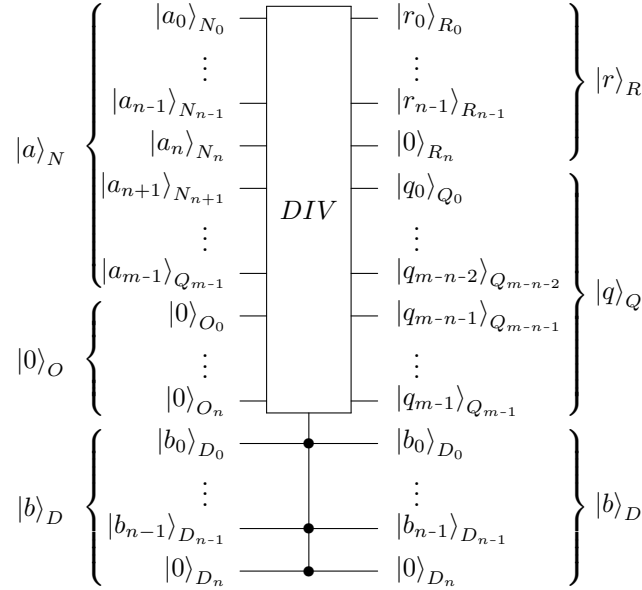


Figure 3.21: In- and outputs for the complete Adapted Thapliyal Integer Division Algorithm for unequal register sizes.

As mentioned, the process remains structurally unchanged, as a number of Iterations on sub-registers. Only the number of iterations changes. To accommodate the higher iterations, the sub-registers $|Y(i)\rangle$ and $|Z(i)\rangle$ are redefined. They remain the same size, and are identical for the early iteration (i.e. they start out at the same positions relative to register D , at 1 qubit above D),

$$\begin{aligned}
|Y(i)\rangle &= |\psi_{2n+m-i-2} \cdots \psi_{n+m-i-1}\rangle, \\
|Z(i)\rangle &= |\psi_{2n+m-i-1}\rangle,
\end{aligned} \tag{3.21}$$

for $i \in 0, 1, \dots, n + m - 1$. The Iterations themselves are kept the same as before, apart from the fact that the definitions of $|Y(i)\rangle$ and $|Z(i)\rangle$ have changed, and the fact that i now has a maximum

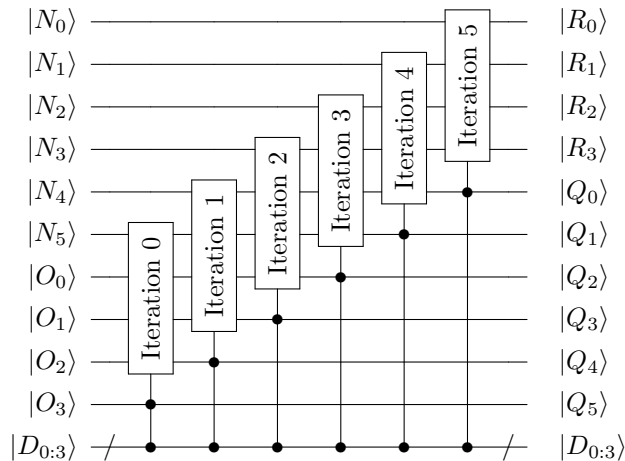


Figure 3.22: Complete Adapted Thapliyal Integer Division circuit for unequal register sizes for $n = 4$ and $m = 2$. Note that the most significant qubit of register D must be equal to $|0\rangle$ in order for the algorithm to work.

of $n+m-1$. The explanation steps in the previous section can be repeated to comprehend the main workings of this new adaptation. An example circuit for $n = 4$ and $m = 2$ is shown in Figure 3.22.

An attentive reader might have spotted a problem in the current implementation: what if b is larger than 2^{n-1} , i.e. it has its most significant qubit in the $|1\rangle$ state? In that case the algorithm will fail, since the $|Z(i)\rangle$ qubit might not be initially zero at some point. To illustrate, an example is examined. Consider $b = 1111$ and $a = 00100000$: in that case nothing happens in the first few iterations, since $2^7 \cdot b$ up until $2^2 \cdot b$ are clearly larger than a . The fact that $2^2 \cdot b = 111100$ is larger than $a = 100000$ will cause a problem however, since this means that in the next step $|Z(i)\rangle$ will start out at $|1\rangle$ instead of the required $|0\rangle$. From that point and on, all controlled operations will happen exactly the wrong way around, resulting in an incorrect output. The only way to fix this, is by forcing the most significant qubit of b to be zero. In practice this means that an extra (ancilla) qubit needs to be added to registers D and O if register N is larger than register D . This effectively means increasing n by one, and decreasing m by one. It also means that for a register size difference of $m = 1$, the ancilla qubits force the registers to be the same size.

Theoretical support for negative numbers is discussed next. It depends on the definition of integer division for negative numbers whether any changes need to be implemented. From the used definition of integer division, r cannot be negative. What this means, is that the answers for $(-a)/b$ and $a/(-b)$ are different: suppose $a = q \cdot b + r$, then $-a = (-q) \cdot (-b) + r$ and $-a = (-q-1) \cdot b + (b-r)$, which are different answers. It is easiest to allow r to be negative since q will only possibly change by one, and in such a way that the average error remains the same (rounding up instead of down). In this changed definition, and when using an independent sign qubit definition (e.g. the sign qubit performs the operation $(-1)^x$), no changes have to be made to the algorithm, apart from a $CNOT$ on the resulting signs of the outputs. If, on the other hand, there is a need for a non-negative r , the Thapliyal algorithm can easily be adapted to suit this requirement. First the algorithm is performed as normal. The sign qubit can then be used to control a -1 subtraction on q , and to perform the operation $b - r$, which was seen to be able to be performed without any ancillae using the Muñoz-Coreas Adder.

Finally, a brief look is taken at dividing by zero. Tests show that afterwards the Q register ends up with the state $|q\rangle = |11 \dots 11\rangle$, while the R register remains unchanged at $|r\rangle = |a\rangle$. This satisfies the equation $a = q \cdot b + r = r$. However, it may be desirable to have the Q register end up in the $|0\rangle$ state. This can be done using two ancilla qubits: the Q register, initialised at zero, can be used as ancillae to perform a $C^n(X)$ as in Figure 1.5 on register D to check if it is entirely zero. Note that register Q is immediately reset to zero again. The output qubit of the $C^n(X)$ can now be saved in our first extra ancilla qubit, and it can be used to control all the gates in the algorithm from that point on, making sure nothing happens if register D is empty. The second extra ancilla

qubit is to make sure this is possible: many operations now need to be controlled by three qubits. This is only possible using an ancilla qubit, and hence the second extra ancilla.

Since negative numbers and division by zero are beyond the scope of this thesis, these implementations are purely theoretical, and will therefore not be implemented or tested. These implementations are left for future research.

3.5 QX implementation

The only real difficulties in the implementation of the discussed quantum arithmetic algorithms lie in the implementation of the Draper adder. Specifically, in the columns of $R_{\pm k}$ gates used in the Draper Adder circuit in Figure 3.3 and Figure 3.13. Figure 3.23 and Figure 3.23 show a proposed implementation of these gate columns.

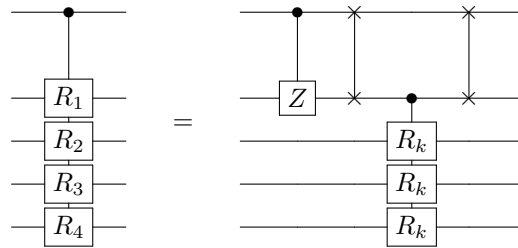


Figure 3.23: Method for applying a R_k column from the circuit in Figure 3.3 up to $k = 4$

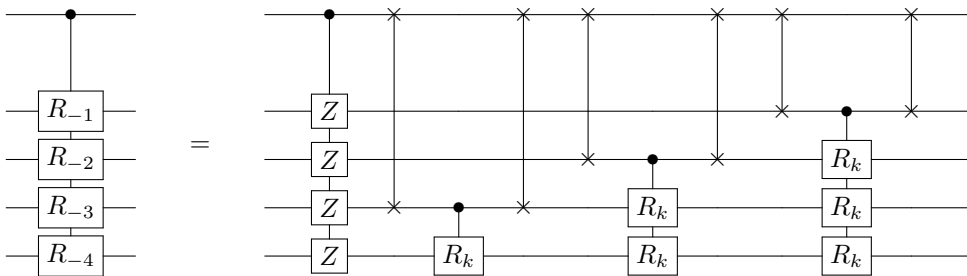


Figure 3.24: QX implementation of the R_{-k} column used in the circuit in Figure 3.13

Nearly all arithmetic algorithms discussed in this chapter were implemented in the QX simulator. Specifically,

- The three adders (all both controlled and uncontrolled, the Cuccaro and Muñoz-Coreas adders also with a choice of whether to include overflow, the Draper adder with full support for different register sizes).
- Subtracters of all three adder algorithms (Draper adder using its specific subtracter extension only in the $a - b$ form, the Cuccaro and Muñoz-Coreas adders with both the $b - a$ and $a - b$ extensions, all possibly controlled and with a choice of overflow or different register sizes as for the adders).
- Straightforward Multiplication (only for large number addition/subtraction i.e. only the second MULADD and MULSUB variation).
- Thapliyal Division for unequal register sizes.

Altogether over 30 variations of the algorithms have been implemented. When an algorithm is run, the compiler returns the inputs and outputs of said algorithm to gain insight into correctness of the output. All implementations have been thoroughly tested and verified for all possible four-qubit

values, as well as for a multitude of values with higher and lower numbers of qubits. It would be unwise to show all raw results. Instead, an example for each algorithm will be shown, in the form of the output of the Python compiler. The tested circuits are the controlled adders, the controlled $a - b$ subtracters, the multiplier and divider of same-size inputs. The example values taken are $a = 1010_2 = 10$ and $b = 0011_2 = 3$, meaning that the outputs are expected to be $a + b = 13$, $a - b = 7$, $a \cdot b = 30$ and the integer division $q = 3$ and $r = 1$ such that $a = q \cdot b + r$. The outputs are shown in Figure 3.25 for the adders and Figure 3.26 for the multiplier and divider. It is seen that the outputs indeed match the expectations.

<p>Adder QFT Controlled:</p> <p>input a = 1010 = 10 input b = 0011 = 3 input ctrl = 1</p> <p>output a+b = 1101 = 13 output a-b = 0111 = 7</p> <p>(a) Draper.</p>	<p>Adder Cuccaro Controlled:</p> <p>input a = 1010 = 10 input b = 0011 = 3 input ctrl = 1</p> <p>output a+b = 01101 = 13 output a-b = 00111 = 7</p> <p>(b) Cuccaro.</p>	<p>Adder Munoz-Coreas Controlled:</p> <p>input a = 1010 = 10 input b = 0011 = 3 input ctrl = 1</p> <p>output a+b = 01101 = 13 output a-b = 00111 = 7</p> <p>(c) Muñoz-Coreas.</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 3.25: Outputs of the different adders and subtracters.

<p>Multiplier:</p> <p>input a = 1010 = 10 input b = 0011 = 3 input c = 00000000 = 0</p> <p>output a*b = 00011110 = 30</p> <p>(a) Multiplier.</p>	<p>Division Thapliyal:</p> <p>input a = 1010 = 10 input b = 0011 = 3</p> <p>output q = a/b = 0011 = 3 output r = a%b = 0001 = 1</p> <p>(b) Thapliyal Division.</p>
----------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 3.26: Outputs of the Straightforward Multiplier and Thapliyal Division.

4. Quantum Linear Solver Algorithm

4.1 General problem statement

Consider the following linear algebra problem: given an invertible $N \times N$ matrix A and an $N \times 1$ vector \vec{b} , find the $N \times 1$ vector \vec{x} , such that

$$A\vec{x} = \vec{b}. \quad (4.1)$$

It is assumed that A is a Hermitian matrix, that is, A is equal to its complex conjugate, $A \equiv A^\dagger$ [26]. Two crucial properties for the QLSA will be that the eigenvectors \vec{v}_j of a Hermitian matrix are linearly independent, and the eigenvalues λ_j are real [26]. The vectors \vec{x} and \vec{b} are assumed to be such that they can be implemented on a quantum computer, as $|x\rangle$ and $|b\rangle$, respectively. This is the case when $\|\vec{x}\| = \|\vec{b}\| = 1$, with $\|\cdot\|$ defined as the ℓ^2 -norm, i.e., $\|\vec{x}\| = \|\vec{x}\|_2 \equiv (\sum_{i=1}^N |x_i|^2)^{1/2}$ [27]. The vector \vec{b} can simply be chosen such that $\|\vec{b}\| = 1$, but vector \vec{x} need not satisfy $\|\vec{x}\| = 1$ when A has any eigenvalue λ with $|\lambda| \neq 1$. Therefore the problem reduces to finding the state $|x\rangle$ that satisfies,

$$A|x\rangle = |b\rangle. \quad (4.2)$$

In other words, $|x\rangle$ is the solution

$$|x\rangle = \frac{A^{-1}|b\rangle}{\|A^{-1}|b\rangle\|}. \quad (4.3)$$

Two important quantities that characterise a matrix, are its condition number κ and sparsity s . The condition number κ of the matrix A is defined as $\kappa(A) \equiv \|A\| \cdot \|A^{-1}\|$, where the value $\|A\|$ is defined as the spectral norm $\|A\| \equiv \limsup_{\vec{x}} \frac{\|A\vec{x}\|}{\|\vec{x}\|}$ [27]. The value for $\|A\|$ is equal to the maximum absolute eigenvalue of A , which allows the condition number to be written as the maximum ratio between the eigenvalues of A , that is, $\kappa(A) = \frac{\max_j |\lambda_j|}{\min_j |\lambda_j|}$ [27]. The sparsity s of a matrix is related to the number of non-zero elements. The matrix A is called s -sparse, when, in each row, only s elements or fewer are non-zero [28]. Together with the size of the matrix N , the quantities κ and s play an important role in the required circuit width and depth to solve a linear system, both classical and quantum.

Consider the situation where the given N -by- N matrix A is invertible, but not Hermitian. Then, a method exists to convert the matrix A and vector \vec{b} (with $\|\vec{b}\| = 1$) to a form that can be solved using the HHL QLSA. Specifically, A and \vec{b} can be transformed into a $2N$ -by- $2N$ Hermitian matrix A' and $2N$ -by-1 vector \vec{b}' (with $\|\vec{b}'\| = 1$), such that the $2N$ -by-1 output vector of the HHL QLSA, \vec{x}' , closely resembles the desired output \vec{x} . Define A' and \vec{b}' as follows,

$$A' \equiv \begin{bmatrix} 0 & A \\ A^\dagger & 0 \end{bmatrix}, \quad \vec{b}' \equiv \begin{bmatrix} \vec{b} \\ \vec{0} \end{bmatrix}, \quad (4.4)$$

then it is evident that A' is Hermitian, and that $\|\vec{b}'\| = \|\vec{b}\| = 1$, implying that the linear system $A'\vec{x}' = \vec{b}'$ can indeed be solved using the HHL QLSA. The new system is easily seen to have the solution

$$\vec{x}' = \begin{bmatrix} \vec{x} \\ \vec{0} \end{bmatrix}, \quad (4.5)$$

with \vec{x} the solution of the original system $A\vec{x} = \vec{b}$. Hence, the linear system $A\vec{x} = \vec{b}$ can be solved using the HHL QLSA for any invertible matrix A at only linear increase in calculation complexity compared to a Hermitian matrix A .

4.2 Classical alternative

The most efficient classical algorithm to solve a system of linear equations is Gauss's method. This is the method most often taught in linear algebra courses, and it has a complexity of $\mathcal{O}(N^3)$ [29]. To solve sparse systems, there are faster methods such as the Conjugate Gradient Method, which can solve an s -sparse semi-definite system with condition number κ in $\mathcal{O}(Ns\kappa)$ time [30]. When only a scalar quantity is needed that can be computed as a matrix-induced inner product of the solution vector \vec{x} , i.e. $\vec{x}^\dagger M \vec{x}$, then the answer may be found in $\mathcal{O}(N\sqrt{\kappa})$ time [30].

4.3 Literature review

In 2009, a paper on solving linear systems of equations using quantum algorithms was published by Harrow, Hassidim and Lloyd [30]. This was the first paper detailing how a linear solver could be implemented on a quantum computer with exponentially improved complexity as compared to a classical computer, with a circuit depth of $\mathcal{O}(\kappa^2 \log N)$, although this only holds for sparse matrices. The algorithm is often referred to as the HHL QLSA. The authors did not specify how exactly the subroutines in the algorithm should be implemented, and significant improvements could still be made. In the following years multiple papers have contributed improvements to the algorithm, or proof-of-concept implementations of the algorithm. The papers will now be discussed, split into the two categories of theoretical improvements and practical implementations.

Firstly, the theoretical improvements will be examined. These are improvements which prove what improvements should be possible, without necessarily showing a method to implement these changes concretely. The focus has mainly been on the Hamiltonian simulation in the algorithm, which is the conversion of a matrix H into $\exp(iHt)$. However, the first improvement was not of this kind, and already came one year after the original paper. In [31], Ambainis describes an amplitude amplification method, which as a result gives a near quadratic improvement in runtime for κ , to a circuit depth of $\mathcal{O}(\kappa \log^3 \kappa \log N)$. In subsequent years, Berry et al. focused on Hamiltonian simulation in [32, 28]. Especially in the latter paper, they prove that this process can be realised in a near optimal circuit depth of $\mathcal{O}(\tau \log(\tau/\epsilon) / \log \log(\tau/\epsilon))$, where $\tau = s \|H\|_{\max} t$ and ϵ denotes the accuracy. This method was then used by Childs et al. in [33] to make an exponential improvement in accuracy to the HHL algorithm, with a circuit depth of $\mathcal{O}(\text{poly}(\log N, \log 1/\epsilon))$ for sparse matrices, whereas the original HHL algorithm has a circuit depth of $\mathcal{O}(1/\epsilon)$. More recently, in 2017, Wossnig et al. described the first adaptation to the HHL algorithm that can efficiently solve dense matrices. Their approach, however, only results in a quadratic speedup compared to classical computers, with a circuit depth of $\mathcal{O}(\kappa^2 \sqrt{n} \text{polylog}(n)/\epsilon)$.

Concerning publications on the practical aspects of the QLSA, the focus has been on proof-of-concept circuits, and especially on their implementation on real quantum computers. The first proof-of-concept theoretical implementation of the HHL QLSA was by Cao et al. [34] in 2012. Here, they show an explicit circuit for solving a specific 2-by-2 and 4-by-4 matrix, respectively. However, the circuits were not implemented on an actual quantum computer. This would change in the next two years, as three independent research groups managed to implement a proof-of-concept circuit on a real quantum computer in 2013 and 2014. Pan et al. managed to solve a specific 2-by-2 matrix on a magnetic resonance quantum computer for different input vectors in [35]. Cai et al. and Barz et al. followed with their respective papers [36] and [37], both also solving a specific 2-by-2 matrix, but this time on a photonic quantum computer. All of their setups were highly specific to their respective matrices, and their methods were not easily expandable. In 2017, Zheng et al. were the first to use a superconducting quantum processor to solve a 2-by-2 matrix, which method is more suitable for expansion [38].

In this thesis, the knowledge from the listed papers is used to implement a general Quantum Linear Solver on a Quantum Computer Simulator. The implementation will be based on the original method described in the HHL paper [30], and on the theoretical implementation by Cao et al. in [34]. The Cao et al. paper splits one of the steps from Harrow et al. into two, leading to an

algorithm that is easier to implement. Only the Cao et al. variation is discussed in this thesis.

4.4 The HHL algorithm

4.4.1 Introduction

Before going into the technical details of the algorithm, it is important to understand the critical thinking steps behind the algorithm. This section will attempt to explain these steps.

In quantum computing there are only a couple of algorithms known that can perform calculations exponentially faster than this is possible on classical computers. The challenge is to rewrite a problem to have one of these algorithms as core. This is exactly what Harrow, Hassidim and Lloyd did in their approach for solving linear systems of equations. Specifically, the problem will be rewritten to include the Quantum Phase Estimation algorithm [13]. The algorithm is explained in detail in Chapter 2.2; it finds the phase $\varphi \in [0, 1)$ of eigenvalues $e^{i\varphi}$ of a unitary operator by cleverly applying the corresponding eigenvector.

Instead of directly inverting the Hermitian matrix A to find \vec{x} , Harrow et al. make clever use of the properties of eigenvectors and eigenvalues. Let \vec{u}_j be the eigenvectors of A , with $\lambda_j \in \mathbb{R}$ denoting the corresponding eigenvalues for $j \in \{0, 1, \dots, N-1\}$. These eigenvectors are defined by the property $A\vec{u}_j = \lambda_j\vec{u}_j$ [26]. Furthermore, since A is Hermitian, the eigenvectors form a linearly independent set spanning the entire vector space \mathbb{R}^n [39]. Hence, \vec{b} and \vec{x} can be written as a linear combination of these eigenvectors, as $\vec{b} = \sum_j \beta_j \vec{u}_j$ and $\vec{x} = \sum_j \beta_j \lambda_j^{-1} \vec{u}_j$, respectively.

The critical new idea on how to perform these steps efficiently, is to use the Hamiltonian of the matrix A , $\exp(iAt)$, for a certain time t . This is again a matrix with the same eigenvectors \vec{u}_j , but with the original eigenvalues λ_j transformed to $\exp(i\lambda_j t)$ [26]. From the fact that A is Hermitian, it follows that $\lambda_j \in \mathbb{R}$. Hence, when applied to $\exp(iAt)$ for different times t , the eigenvectors \vec{u}_j rotate in the complex domain with the angular velocity of their eigenvalues λ_j .

This angular velocity is precisely what the Quantum Phase Estimation algorithm is designed to estimate. Hence, it can be applied to $\exp(iAt)$ to obtain the eigenvalues of A . The eigenvalues should then be inverted and multiplied with their respective eigenvectors to obtain the solution of \vec{x} .

The way the HHL algorithm applies these ideas, will be detailed in the next section.

4.4.2 Realisation of the HHL algorithm

The HHL algorithm uses four sets of qubits:

- an $\mathcal{O}(\log(n))$ qubit memory register m to initially store \vec{b} , and which is transformed to \vec{x} ,
- an $\mathcal{O}(n)$ qubit register r to store the eigenvalues of A ,
- another $\mathcal{O}(n)$ qubit register q to store the inverted eigenvalues of A ,
- a single ancilla qubit register a .

Suppose \vec{b} , \vec{u}_j , λ_j and λ_j^{-1} can be implemented as $|b\rangle_m$, $|u_j\rangle_m$, $|\lambda_j\rangle_r$ and $|\lambda_j^{-1}\rangle_q$ for all $j \in \{1, \dots, N\}$, respectively, such that $|b\rangle_m$ is written as a superposition of $|u_j\rangle_m$,

$$|b\rangle_m = \sum_{j=1}^N \beta_j |u_j\rangle_m. \quad (4.6)$$

How this is performed will be discussed later. This also means that the requested vector \vec{x} can be written as $|x\rangle_m = \sum_{j=1}^N \beta_j \lambda_j^{-1} |u_j\rangle_m$.

The Cao et al. adaptation to the HHL algorithm [34] performs the following steps to acquire the state $|x\rangle_m$:

$$\begin{aligned} 0. \text{ Initialise: } & |0\rangle_a |0\rangle_q |0\rangle_r |b\rangle_m \\ &= \sum_{j=1}^N \beta_j |0\rangle_a |0\rangle_q |0\rangle_r |u_j\rangle_m \end{aligned}$$

1. Perform Quantum Phase Estimation using e^{iAt} , and map the eigenvalues λ_j into the register r :

$$\rightarrow \sum_{j=1}^N \beta_j |0\rangle_a |0\rangle_q |\lambda_j\rangle_r |u_j\rangle_m$$

2. Invert the eigenvalues λ_j to λ_j^{-1} , and map them to the second register q :

$$\rightarrow \sum_{j=1}^N \beta_j |0\rangle_a |\lambda_j^{-1}\rangle_q |\lambda_j\rangle_r |u_j\rangle_m$$

3. Rotate the ancilla qubit to the state $\sqrt{1 - \frac{c^2}{\lambda_j^2}} |0\rangle_a + \frac{c}{\lambda_j} |1\rangle_a$ for each j using a controlled rotation on the $|0\rangle_a$ qubit, for a predefined $c < \lambda_{\max}$:

$$\rightarrow \sum_{j=1}^N \beta_j \left(\sqrt{1 - \frac{c^2}{\lambda_j^2}} |0\rangle_a + \frac{c}{\lambda_j} |1\rangle_a \right) |\lambda_j^{-1}\rangle_q |\lambda_j\rangle_r |u_j\rangle_m$$

4. Perform the opposite operation of Steps 1 and 2 to reset registers q and r :

$$\rightarrow \sum_{j=1}^N \beta_j \left(\sqrt{1 - \frac{c^2}{\lambda_j^2}} |0\rangle_a + \frac{c}{\lambda_j} |1\rangle_a \right) |0\rangle_q |0\rangle_r |u_j\rangle_m$$

5. Measure the ancilla qubit. If the $|1\rangle_a$ state is obtained, the resulting vector in the m register is $|x\rangle_m$:

$$\begin{aligned} &\rightarrow \sqrt{\frac{1}{\sum_{j=1}^N c^2 |\beta_j|^2 / |\lambda_j|^2}} \sum_{j=1}^N \beta_j \left(\frac{c}{\lambda_j} \right) |1\rangle_a |0\rangle_q |0\rangle_r |u_j\rangle_m \\ &\propto |1\rangle_a |0\rangle_q |0\rangle_r \sum_{j=1}^N \frac{\beta_j}{\lambda_j} |u_j\rangle_m \\ &= |1\rangle_a |0\rangle_q |0\rangle_r |x\rangle_m \end{aligned}$$

However, if the $|0\rangle_a$ state is obtained, then the outcome is garbage, and the algorithm needs to be retried.

The probability to measure $|1\rangle_a$ is equal to $\mathbb{P}(|1\rangle_a) = c \sqrt{\sum_{j=1}^N |\beta_j / \lambda_j|^2}$. Since this probability increases linearly with c , it is desirable to choose this number as large as possible without violating the condition that $c < \lambda_{\max}$.

Step 2 and register q are not present in the original HHL paper, and were introduced by Cao et al. in [34]. This is due to the fact that the transformation immediately from λ_j to a rotation proportional to $1/\lambda_j$ is an unreasonably complex step, and can be implemented more easily using this split-up.

4.4.3 In depth discussion

In order to better understand the steps taken in the HHL algorithm, it is insightful to first assume that \vec{b} is an eigenvector of A , say $\vec{b} = \vec{u}_1$, with eigenvalue $\lambda_1 \in \mathbb{R}$. In that case, when in Step 1 the $\exp(iAt)$ transformation is applied for different t , the only thing that happens, is that the complex amplitude of the state starts to rotate in the complex plane, with the angular velocity λ_1 . Note that the amplitude of the state does not change. This rotation is exactly what the Quantum Phase Estimation algorithm is designed to detect, and it will save this angular velocity (equal to the eigenvalue) to the register. Now that the eigenvalue λ_1 has been obtained, Step 2 is used to find the inverse of this eigenvalue, λ_1^{-1} . This inverted eigenvalue in turn is used in Step 3 to rotate the ancilla qubit to a state, such that the complex probability to measure the $|1\rangle_a$ state becomes proportional to λ_1^{-1} . If, after clearing the registers in Step 4, the ancilla qubit is measured in Step 5 and the $|1\rangle_a$ state is found, then the final state should now be a state proportional to the quantum representation of $\vec{u}_1/\lambda_1 = \vec{b}^{-1}$, precisely as desired.

However, it should be noted that in this case where \vec{b} is exactly an eigenvector of A , the normalisation of the final state causes the destruction of all information stored in the complex amplitude. This is expected, since the result for $|x\rangle_m$ will be proportional to $|u_1\rangle_m$. The latter state was already normalised, and hence after normalisation $|x\rangle_m$ will be precisely equal to $|u_1\rangle_m$ again. The algorithm only has a net effect when \vec{b} is not an eigenvector of A , but instead a linear combination of its eigenvectors.

In that case, $|b\rangle_m$ can be viewed as a superposition of these eigenvectors, as was required in the algorithm. Since the operations are performed on a quantum computer, all the following steps in the algorithm also happen as a superposition of these eigenvectors: with a complex amplitude of β_1 for the case that $|u_1\rangle_m$ (as described above); with complex amplitude β_2 for the case that $|u_2\rangle_m$; etcetera. The ancilla rotation in Step 3 will now be different for each eigenvector, so when the $|1\rangle_a$ state is measured in Step 4, the relative probabilities for the different eigenvectors change. This is not changed by the re-normalisation, since that only occurs over all states as a whole. The final state is a superposition of eigenvectors $|u_j\rangle_m$ with relative amplitudes β_j/λ_j . Due to the way the implementation of the vectors was defined, this is exactly equivalent to the desired state $|x\rangle_m$.

4.4.4 Implementation

A high-level circuit of the complete Cao adaptation to the HHL QLSA is shown in Figure 4.1.

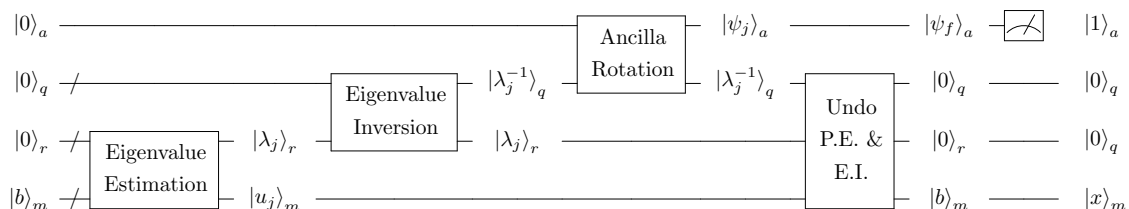


Figure 4.1: High-level overview of the Cao et al. implementation [34] of the HHL QLSA [30]. After the Quantum Phase Estimation subroutine, $|b\rangle_m$ is written as $|u_j\rangle_m$ to emphasise that all steps should be viewed in the basis of the eigenvectors of A , and that $|b\rangle_m$ is a superposition of these eigenvectors. The state $|\psi_j\rangle_a$ is defined as $|\psi_j\rangle_a \equiv \sqrt{1 - C^2/\lambda_j^2} |0\rangle_a + C/\lambda_j |1\rangle_a$, and $|\psi_f\rangle_a$ is the entangled superposition of these states with the input vector state $|b\rangle_m$.

In this overview, after the Quantum Phase Estimation subroutines, $|b\rangle_m$ is written as $|u_j\rangle_m$ to emphasise that $|b\rangle_m$ is a superposition of these eigenvectors, and that all subsequent steps should be viewed in the basis of those eigenvectors. The ancilla state $|\psi_j\rangle_a$ is defined as the desired rotated ancilla state for eigenvalue λ_j , $|\psi_j\rangle_a \equiv \sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a$. The final ancilla state $|\psi_f\rangle_a$ is defined as the superposition of the $|\psi_j\rangle_a$ states entangled with the $|u_j\rangle_m$ states as in Step 4 of the Cao et al. HHL algorithm implementation.

The subroutines described in Figure 4.1 will be the focus of the remainder of this thesis. Each of the three subroutines will be discussed in their own chapter, and solved where reasonably possible. At the end, the solutions and implementations to subroutines will be combined to form a working prototype Quantum Linear Solver that is as complete as possible.

Before the subroutines can be discussed, it is essential to know how the vectors \vec{b}_m and \vec{x}_m can be implemented on a quantum computer as $|b\rangle_m$ and $|x\rangle_m$, and how the values λ_j and λ_j^{-1} can be constructed as $|\lambda_j\rangle$ and $|\lambda_j^{-1}\rangle$, such that the implementations have the required properties. The latter two can be constructed as was discussed in the previous chapter. In [34], an implementation for the vectors is shown which will thereafter be discussed in greater detail.

In a quantum system of n qubits, there are 2^n possible pure states: $|00\dots 0\rangle$ through $|11\dots 1\rangle$. If only positive integers are considered, then any value smaller than 2^n can simply be saved as its binary representation, i.e.: $|0\rangle \equiv |00\dots 00\rangle$, $|1\rangle \equiv |00\dots 01\rangle$ etc. until $|2^n - 1\rangle \equiv |11\dots 11\rangle$.

Now consider the vector \vec{b} of length $N \leq 2^n$,

$$\vec{b} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{bmatrix}, \quad (4.7)$$

with $\|\vec{b}\| = 1$, that is, $\sum_{j=1}^N |\beta_j|^2 = 1$. This vector can then be implemented in a quantum register as,

$$|b\rangle_m = \sum_{j=1}^N \beta_j |j-1\rangle_m, \quad (4.8)$$

where $|j\rangle_m$ is the positive integer j state of the memory as defined above. This implementation is well-defined, since all these states are independent of each other. It also entails that only $\lceil \log_2(N) \rceil$ qubits in the memory are required to be able to store \vec{b} .

For example, if $N = 4$, then only $\log_2(N) = 2$ qubits are required in the memory to store the vector. Let $|j=0\rangle_m \equiv |00\rangle_m$, $|j=1\rangle_m \equiv |01\rangle_m$, $|j=2\rangle_m \equiv |10\rangle_m$ and $|j=3\rangle_m \equiv |11\rangle_m$, then the quantum representation of the vector $\vec{b} = [\beta_1 \ \beta_2 \ \beta_3 \ \beta_4]^T$ is written as $|b\rangle_m = \beta_1 |00\rangle_m + \beta_2 |01\rangle_m + \beta_3 |10\rangle_m + \beta_4 |11\rangle_m$.

An inherent downside of this way to store vectors, is that all the information of $|b\rangle$ (and eventually $|x\rangle$) is stored in the complex amplitudes of the memory qubits. The complex amplitudes are impossible to measure directly, since the state collapses into one of the $|j\rangle$ states when doing so [3]. Therefore, the HHL algorithm can only be used as either an intermediate step in a larger quantum algorithm, or when only a single property of the output vector is desired, like $\vec{x}^\dagger M \vec{x}$ [30].

With the knowledge of what the inputs and outputs of the different subroutines should be, and how they should be stored, the subroutines themselves can now be studied in detail. The next three chapters will be dedicated to this purpose. Firstly, in Chapter 5, the Eigenvalue Estimation subroutine is examined, which is an adaptation on the Quantum Phase Estimation Algorithm. The main challenge of this subroutine will be implementation of the $\exp(iAt)$ operations and vector $|b\rangle$. Those implementation, however, will be left for future research. Next, in Chapter 6, the Eigenvalue Inversion subroutine is examined, and three different solutions are implemented. After that, in Chapter 7, the Ancilla Rotation subroutine is investigated and solved using multiple implementations. Finally, in Chapter 8, the subroutines are combined to form a complete prototype Quantum Linear Solver.

5. Eigenvalue Estimation

In the previous chapter it was claimed that using the Hamiltonian $\exp(iAt)$ for $t \in \mathbb{R}_+$ and the Quantum Phase Estimation algorithm [13], the quantum representation $|b\rangle$ of a vector \vec{b} can be decomposed into eigenvectors of A , and the corresponding eigenvalues of A can be found [30]. In this chapter it is investigated why this holds, and the challenges of constructing the Hamiltonian $\exp(iAt)$ and vector $|b\rangle$ are examined.

5.1 The Quantum Phase Estimation Algorithm

First a brief recap of the Quantum Phase Estimation Algorithm, on which the Eigenvalue Estimation Algorithm is based. The Quantum Phase Estimation Algorithm has been discussed in depth in Chapter 2.2.

Consider a unitary operator U with an eigenvector $|u\rangle$ and corresponding eigenvalue $e^{2\pi i\varphi}$, where $\varphi \in \mathbb{R}$ is unknown. Without loss of generality, it is assumed that φ lies on the interval $[0, 1)$, since any value outside this domain will be mapped onto it. The Quantum Phase Estimation Algorithm approximates this value. The requirements are (1) a memory register with the $|u\rangle$ state, (2) a work register of n qubits that starts out in the $|00 \dots 0\rangle$ state, and (3) an operator capable of performing a controlled- U^{2^k} operation for $k \in \mathbb{N}$. Note that since the value φ lies in $[0, 1)$, it can be written as and approximated to,

$$\begin{aligned} \varphi &= 0.\varphi_1\varphi_2 \dots \varphi_n\varphi_{n+1} \dots \\ &\approx 0.\varphi_1\varphi_2 \dots \varphi_n \\ &\equiv \tilde{\varphi}, \end{aligned} \quad (5.1)$$

with φ_j either 0 or 1. The net process of the Quantum Phase Estimation Algorithm can be described as

$$|0\rangle |u\rangle \xrightarrow{\text{Phase Est.}} |\tilde{\varphi}\rangle |u\rangle, \quad (5.2)$$

using the circuit given in Figure 5.1, where the subroutines in the dashed box are as shown in Figure 5.2.

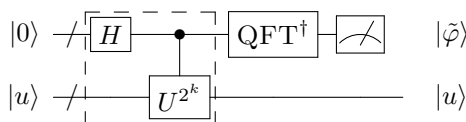


Figure 5.1: Complete Quantum Phase Estimation Algorithm. The encircled part of the Hadamard gate and controlled- U^{2^k} gates is as described in Figure 5.2, and the Inverse Quantum Fourier Transform is defined as the reverse of the circuit in Figure 2.1.

It was shown Chapter 2.2 that the application of a $C(U^{2^k})$ gate, controlled by a quantum state $\alpha|0\rangle + \beta|1\rangle$, has the following effect,

$$\left(\alpha|0\rangle + \beta|1\rangle\right) |u\rangle \xrightarrow{C(U^{2^k})} \left(\alpha|0\rangle + e^{2\pi i\varphi 2^k} \beta|1\rangle\right) |u\rangle \approx \left(\alpha|0\rangle + e^{2\pi i \cdot (0.\varphi_{k+1}\varphi_{k+2} \dots \varphi_n)} \beta|1\rangle\right) |u\rangle, \quad (5.3)$$

which shows that the state of the top register after the U^{2^k} subroutine can be written as,

$$\frac{1}{2^{n/2}} \left(|0\rangle + e^{2\pi i \cdot (0.\varphi_n)} |1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i \cdot (0.\varphi_{n-1}\varphi_n)} |1\rangle\right) \otimes \dots \otimes \left(|0\rangle + e^{2\pi i \cdot (0.\varphi_1\varphi_2 \dots \varphi_n)} |1\rangle\right). \quad (5.4)$$

This is precisely the Quantum Fourier Transform of $\tilde{\varphi} \approx \varphi$, and hence the application of the Inverse Quantum Fourier Transform to the work register indeed results in the state $|\tilde{\varphi}\rangle$ in the work register.

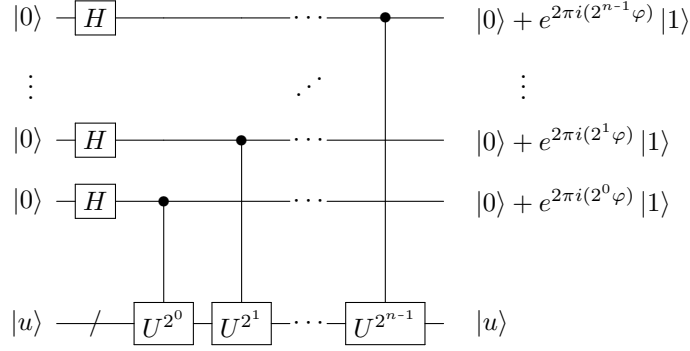


Figure 5.2: The U^{2^k} subroutine in the Quantum Phase Estimation Algorithm.

5.2 The Eigenvalue Estimation Algorithm

The Quantum Phase Estimation Algorithm will now be rewritten to calculate the eigenvalues of the matrix A . The adaptation will be referred to as the Eigenvalue Estimation algorithm. Since the matrices in the HHL QLSA are assumed to be Hermitian, the eigenvalues are real. In this explanation, it is additionally assumed that the eigenvalues of the matrix are positive integers. Extensions to allow for negative and non-integer eigenvalues will be discussed at the end of this section. The value for $n \in \mathbb{N}$ is chosen such that the largest eigenvalue λ_{\max} satisfies $\lambda_{\max} \leq 2^n$. This means that each eigenvalue can be written in binary form as $\lambda_j = \lambda_{j,1}\lambda_{j,2}\dots\lambda_{j,n}$, with $\lambda_{j,k} \in \{0, 1\}$ for all $j \in \{1, 2, \dots, N\}$ and $k \in \{1, 2, \dots, n\}$.

As was touched upon in the previous chapter, controlled- $\exp(iAt)$ operations are used to perform the Eigenvalue Estimation. It was stated that e^{iAt} is a matrix, with the same eigenvectors \vec{u}_j as A , but with eigenvalues transformed from λ_j to $\exp(i\lambda_j t)$. If the value for t is not taken as $t_0 = 2\pi/2^n$, then the eigenvalues become

$$e^{i\lambda_j t_0} = e^{2\pi i \cdot (0.\lambda_{j,1}\lambda_{j,2}\dots\lambda_{j,n})}, \quad (5.5)$$

for $j \in \{1, \dots, N\}$. This is the same as in the Quantum Phase Estimation algorithm for U^{2^0} when φ is replaced by λ_j . If the value for t is now changed again, this time to $t_k = t_0 2^k = 2\pi/2^{n-k}$ for some $k \in \mathbb{N}_\mu$, then the following eigenvalues are obtained,

$$e^{i\lambda_j t_k} = e^{2\pi i \cdot (0.\lambda_{j,k+1}\lambda_{j,k+2}\dots\lambda_{j,n})}. \quad (5.6)$$

Comparing this result to Equation (5.3) shows that these eigenvalues are exactly equal to those of U^{2^k} , when the binary values φ_k are replaced by $\lambda_{j,k}$. Therefore, if in the Quantum Phase Estimation Algorithm U^{2^k} is replaced by e^{iAt_k} and $|u\rangle$ is replaced by an eigenvector of A , i.e. $|u\rangle = |u_j\rangle$ for some $j \in \{1, \dots, N\}$, then the net operation becomes,

$$|0\rangle |u_j\rangle \xrightarrow{\text{Phase Est.}} |\lambda_j\rangle |u_j\rangle, \quad (5.7)$$

with $|\lambda_j\rangle = |\lambda_{j,1}\rangle |\lambda_{j,2}\rangle \dots |\lambda_{j,n}\rangle$. Its circuit implementation is shown in Figure 5.3, where the subroutines in the dashed box are as shown in Figure 5.4.

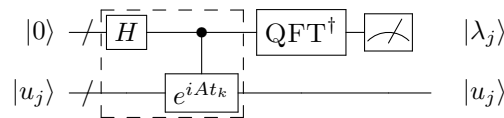


Figure 5.3: The Quantum Phase Estimation Algorithm rewritten to approximate the Quantum Fourier Transform of the eigenvalue λ_j of the matrix A . The two subroutines in the dashed box are shown in Figure 5.4.

However, this still requires the knowledge and implementation of the eigenvector \vec{u}_j . Ideally,

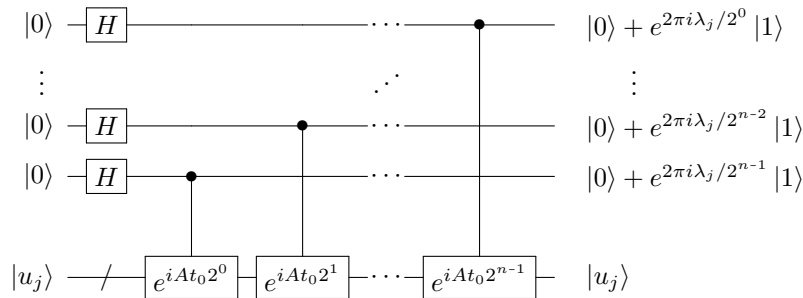


Figure 5.4: The U^{2^k} subroutine in the Quantum Phase Estimation Algorithm rewritten to approximate the Quantum Fourier Transform of the eigenvalue λ_j of the matrix A .

$|b\rangle$ is automatically split up into its eigenvector decomposition. Due to the clever implementation of $|b\rangle$ and the fact that A is Hermitian, it turns out that this process is already performed automatically. In the previous chapter, it was stipulated that the vector $|b\rangle$ can be written as $|b\rangle = \sum_{j=1}^N \beta_j |u_j\rangle$. The Quantum Phase Estimation Algorithm simply solves the problem as a superposition of each $|u_j\rangle$, and creates a superposition of entangled $|\lambda_j\rangle \otimes |u_j\rangle$ states between the register and the memory,

$$|0\rangle |b\rangle \equiv |0\rangle \sum_{j=1}^N \beta_j |u_j\rangle = \sum_{j=1}^N \beta_j |0\rangle |u_j\rangle \xrightarrow{\text{Phase Est.}} \sum_{j=1}^N \beta_j |\lambda_j\rangle |u_j\rangle. \quad (5.8)$$

A complete circuit to find the eigenvalues therefore is as shown in Figure 5.5.

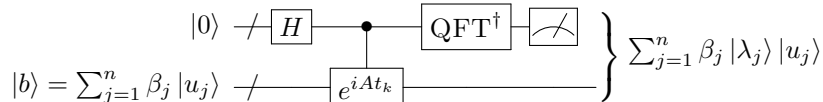


Figure 5.5: Complete algorithm to find the eigenvector decomposition with eigenvalues of a vector \vec{b} (implemented as $|b\rangle$) and matrix A for integer eigenvalues. The final state is the entangled superposition $\sum_{j=1}^N \beta_j |\lambda_j\rangle |u_j\rangle$. The part of the Hadamard gate and controlled- e^{iAt_k} gates is as described in Figure 5.2, with $|u\rangle = |b\rangle$ and $U^{2^k} = e^{iAt_k}$, where $t_k = 2\pi/2^{n-k}$. The Inverse Quantum Fourier Transform is defined as the reverse of the circuit in Figure 2.1.

At the start of this explanation it was assumed that the eigenvalues λ_j were positive integers. In these final paragraphs, extensions to the Eigenvalues Inversion Algorithm are proposed which make the algorithm function for non-positive and non-integer eigenvalues λ_j . These proposed extensions are untested theoretical concepts which will require further research.

In the Quantum Phase Estimation Algorithm, negative numbers $-\varphi$ are mapped to $1 - \varphi$. In the Eigenvalue Estimation Subroutine this is translated to that a negative eigenvalue $-\lambda$ is mapped to $2^n - \lambda$. This will create problems in the coming subroutines if left unchanged. A solution is to increase n by one, such that $|\lambda_{\max}| < 2^{n-1}$. In that case, if $\lambda > 0$, then the most significant qubit of the work register will always be left $|0\rangle$, while if $\lambda < 0$, the most significant qubit will always roll over to $|1\rangle$. This makes the most significant qubit effectively a sign symbol. Since the negative eigenvalue is still mapped as $2^n - \lambda - 1$, a CNOT can be performed on all other qubits of the register, with the sign qubit being the control. The value then becomes $2^{n-1} + \lambda$, with the 2^{n-1} the sign-qubit. The sign of the estimated eigenvalue has therefore been separated from the absolute value, and can be applied when necessary in later subroutines. Specifically, it will be needed in the Ancilla Rotation subroutine, where it can be used to control the sign of the rotations.

Implementing non-integer numbers also requires only minor changes to the Eigenvalue Estimation Algorithm. Consider non-negative eigenvalues again, such that $2^{n-1} < \lambda_{\max} \leq 2^n$. Consider the situation where an absolute accuracy of 2^{-m} is desired on all eigenvalues. That is, all λ in binary form are to be approximated as $\lambda_1 \lambda_2 \cdots \lambda_n \cdot \lambda_{n+1} \cdots \lambda_{n+m}$. This can be achieved by increasing the size of the register to $n + m$ qubits, and by changing the range of the e^{iAt_k} gates from

$k = 0, 1, \dots, n-1$ to $k = -m, -m+1, \dots, -1, 0, 1, \dots, n-1$. After the Inverse Quantum Fourier Transform in the Eigenvalue Estimation Algorithm, the resulting values in the work register will precisely be the requested approximations. The extension to facilitate negative numbers can be used in combination with this method.

5.3 Hamiltonian Simulation

Up to this point, it was assumed that a memory in the $|b\rangle$ state is available, and that black boxes capable of performing $\exp(iAt_k)$ for $k \in \mathbb{N}$ are available as well. Building the vector, and especially the matrix exponential is not at all trivial however. Construction of the vector $|b\rangle$ falls under the category of building a general quantum state [3], and is left completely for future research. The process of constructing the Hamiltonian $\exp(iAt_k)$ from the matrix A is well known in quantum mechanics, as the problem of Hamiltonian Simulation [40]. The problem will hence be referred to as the Hamiltonian Simulation Problem, and it will briefly be discussed in the next section. Its implementation, however, is again left for future research.

5.3.1 Classical Implementation

The classical definition of the matrix exponent e^X from a square matrix X is defined as [26],

$$e^X \equiv \sum_{k=0}^{\infty} \frac{1}{k!} X^k, \quad (5.9)$$

where X^k represents the k -times multiplication of the matrix X with itself. The Hamiltonian $\exp(iAt)$ can therefore be constructed from A by replacing X with iAt in Equation (5.9), yielding the equation,

$$e^{iAt} = \sum_{k=0}^{\infty} \frac{1}{k!} (iAt)^k = I + iAt - \frac{1}{2}A^2t^2 - \frac{1}{6}iA^3t^3 + \dots \quad (5.10)$$

Since A was assumed to be Hermitian, it has linearly independent eigenvectors and can consequently be diagonalised as $A = PDP^{-1}$, with D a diagonal matrix of all eigenvalues of A , and P the matrix of all corresponding eigenvectors [26]. Using this notation, the k -times multiplication of A can be written as $A^k = PD^kP^{-1}$, and hence the Hamiltonian $\exp(iAt)$ can be diagonalised as well, as

$$e^{iAt} = Pe^{iDt}P^{-1}. \quad (5.11)$$

This rewrite shows that $\exp(iAt)$ indeed has the same eigenvectors as A , only with the eigenvalues λ_j transformed to $\exp(i\lambda_j t)$. Therefore, if the eigenvectors and eigenvalues of A are known, it is near trivial to calculate its matrix exponential. In the more common case where the eigenvectors and eigenvalues are unknown however, the problem becomes much harder, since now only Equation (5.10) can be used.

5.3.2 Quantum Implementations

The problem of implementing a Hamiltonian as in Equation (5.10) is discussed in this section. The problem is well known in the field of Quantum Computing, as it is required in the implementation of Quantum Simulation [40]. A major issue in the implementation of the Hamiltonian lies in the fact that in general, a matrix A cannot directly be performed on a quantum computer. As was mentioned in Chapter 1 on the Introduction to Quantum Computing, only unitary operations (meaning $|\lambda| = 1$ for all eigenvalues λ) can be implemented on a Quantum Computer, as it would otherwise be possible for the complex amplitudes of a state to fail the normalisation requirement. However, this means that it is impossible to directly implement any matrix A with any eigenvalue $|\lambda| \neq 1$. This is a significant hurdle in the implementation of Hamiltonians. A number of different algorithms have been developed to solve the problem of Hamiltonian Simulation. Some of these algorithms are listed below. In this thesis the choice was made to focus on the other two subroutines of the HHL QLSA, and therefore the implementation of Hamiltonian Simulation algorithms is left for future research.

5.3.2.1 Group Leadership Algorithm

In [41], Daskin et al. show a general method for the implementation of any Hamiltonian, called the Group Leadership Algorithm. This however is not a quantum algorithm, but instead a classical genetic algorithm that finds an approximate circuit for the Hamiltonian Simulation. Since the aim of this thesis is to find direct Quantum Algorithms and not efficient ways of classically approximating them, this algorithm was not investigated further. The implementation of the Hamiltonian that will be used in Chapter 8 was conceived using the Group Leadership algorithm by Cao et al. in [34].

5.3.2.2 Quantum Walks Algorithm

Childs, Berry, et al. have published several papers on the implementation of Hamiltonians using the Quantum Walks Algorithm [28, 32, 33, 42, 43, 44, 45]. This algorithm decomposes any sparse matrix A implemented as a quantum oracle into a sum of matrices with low sparsity, so that they can efficiently be implemented. In [28] it is shown that the Hamiltonian can be simulated in a circuit depth of $\mathcal{O}(\tau \log(\tau/\epsilon)/\log \log(\tau/\epsilon))$ queries to the oracle. The papers, however, do not elaborate on the implementation of the oracle and many other critical subroutines, which makes an implementation outside the reach of this thesis.

5.3.2.3 Quantum Singular Value Estimation Algorithm

In [46], Wossnig et al. present a method to implement the Hamiltonian of any dense matrix using the Quantum Singular Value Estimation Algorithm, which is an algorithm that decomposes a matrix into a sum of vector cross products. This algorithm, however, reduces the theoretical exponential speedup of the original HHL algorithm for sparse matrices to only a quadratic speedup over any classical algorithm. The paper also does not present implementation details of most of its subroutines, again leaving a practical implementation outside the reach of this thesis.

6. Eigenvalue Inversion

In this chapter, implementations of the Eigenvalue Inversion subroutine are examined and implemented. The algorithms are responsible for finding the inverses λ_j^{-1} of the eigenvalues λ_j that were found in the Eigenvalue Estimation subroutine. A general Eigenvalue Inversion subroutine makes use of at least two registers. One memory register containing $|\lambda_j\rangle$, and an empty register that will store the inverse $|\lambda_j^{-1}\rangle$. The algorithm should leave the memory untouched and only overwrite the second register. Hence, the algorithm should perform the following operation,

$$|\lambda_j\rangle |0\rangle \xrightarrow{\text{Eig.Inv.}} |\lambda_j\rangle |\lambda_j^{-1}\rangle. \quad (6.1)$$

One of the algorithms will only make use of a single register, in which $|\lambda_j\rangle$ is directly transformed into $|\lambda_j^{-1}\rangle$. It is, however, only a very specific proof-of-concept Eigenvalue Inversion implementation. Since inversion is closely linked to division, a division algorithm may be used instead of a specific inversion algorithm. A division algorithm calculating a/b for some n qubit numbers $a, b \in \mathbb{N}$ can be used for calculating the inverse or reciprocal of a number λ_j by taking the values $a = 1$ and $b = \lambda_j$.

6.1 Classical Approaches

On a classical computer, inversion is mainly performed through division a/b using $a = 1$, as described above. Hence, solely division algorithms are examined. The easiest method for computing a/b , is to repeatedly subtract b from a until the result is smaller than b . The amount of times that b has been subtracted from a is then an approximation for a/b , specifically, $\lfloor a/b \rfloor$. This method, however, is relatively inefficient, with a circuit depth $\mathcal{O}(n2^n)$; it takes $\mathcal{O}(n)$ operations to subtract a single b from a , and a maximum of $\mathcal{O}(2^n)$ subtractions are possible (e.g. when $b = 1$ and $a = 2^n - 1$). Two main types of more efficient division algorithms exist: *fast algorithms* and *slow algorithms* [25]. The first category contains the Long Division Algorithm (“staartdeling”) commonly taught at schools, which has a circuit depth of $\mathcal{O}(n^2)$. All slow division methods use a variation of the principles used in the Long Division method, where one significant bit is resolved at a time. Fast division algorithms on the other hand are methods that can be applied when a relatively close estimate is already available. These algorithms allow for faster approximation than the slow algorithms from that first estimate. Examples are the Newton-Raphson and the Goldschmidt algorithms. These approximate the answer quadratically faster than slow algorithms [25].

6.2 Methods for Computing the Reciprocal of a number

In this chapter, three quantum algorithms for Eigenvalue Inversion will be discussed. Each of them is based on another method. The first algorithm is an extremely simple algorithm only designed for inverting powers of two, useful for, e.g., proof-of-concept circuits. Secondly, an inversion algorithm based on the Newton-Raphson Algorithm is discussed, which was suggested by Cao et al. in [47]. Finally, an algorithm based on long division, proposed by Thapliyal et al. [18, 19] is discussed. The algorithms are all implemented in the QX simulator and compared to each other at the end of this chapter. A fourth algorithm, proposed by Cao et al. in [34], was investigated. Initial attempts at implementation, however, have proved unsuccessful. A description of the algorithm and the attempts at implementation are found in Appendix B.

Before proceeding to the descriptions of the algorithms, a quick recap about binary numbers will be made, including notes on inverses of binary numbers. Since only finitely many qubits are available, say n qubits for a positive integer x , there will always be a largest possible value, which is $2^n - 1$ in this case. Any positive integer x that is smaller than 2^n can be written as $x_{n-1}x_{n-2} \cdots x_0$,

and its inverse $1/x = y$ as $y = y_0.y_{-1}y_{-2}\cdots y_{-n+1}y_{-n}\cdots$. Note that the bit representation of the exact inverse may have infinitely many non-zero bits. When the solution is saved as a binary number, then it must be truncated at a finite amount of bits, e.g., it is approximated to m qubits as $\tilde{y} \equiv y_0.y_{-1}y_{-2}\cdots y_{-m-1} \approx y_0.y_{-1}y_{-2}\cdots y_{-m-1}\cdots y_{-n}\cdots = y$.

If y is multiplied by some power of two 2^ℓ for $\ell \in \mathbb{N}$, then only the position of the decimal point changes: $2^\ell \cdot y = y_{-1}y_{-2}\cdots y_{-\ell}.y_{-\ell-1}\cdots$. The decimal point itself will not explicitly be saved in the implementations in this thesis; instead it will follow from the implementation of the circuit. This implies that the inverse $1/x$ of x is actually equivalent to performing $2^\ell/x$ for any $\ell \in \mathbb{N}$. Due to this property, $2^\ell/x$ will also interchangeably be called an inverse x^{-1} of x . In some cases, the inverse is approximated by integer values, which means that a higher ℓ may lead to a higher accuracy, depending on the implementation.

6.2.1 Powers-of-Two Algorithm

Computing the reciprocal of a number a is easiest when only positive integer powers of two are considered as inputs. For example, let $a = 2^k$ for some $k \in \mathbb{N}$. In that case the binary representation may be written as $a = 00\cdots 010\cdots 00_2$, where only the k -th bit from the right is non-zero. Its inverse $1/a$ is then equal to $0.00\cdots 010\cdots_2$, where only the $(k-1)$ -th bit to the right of the decimal point is non-zero (for $k=1$, i.e. $a = 00\cdots 01_2$, the solution is $1/a = a = 1_2$). Consider the situation where n qubits are available and only positive integers are considered. Then, the largest possible power of two is 2^{n-1} , and the smallest one is $2^0 = 1$. The inverses therefore lie between 2^{-n+1} and $2^0 = 1$. The critical observation is that if these inverses are multiplied by 2^{n-1} , then the output range is again between 2^0 and 2^{n-1} ; the same as for the inputs. Now define a^{-1} as $2^{n-1}/a$. Then, $(2^0)^{-1} = 2^{n-1}$ and $(2^{n-1})^{-1} = 2^0$. The same goes for 2^1 and 2^{n-2} , etcetera. This shows that the inversion of a number that is a power of two can be computed by reversing the qubit order, using a reverse subroutine as is shown in Figure 6.1.

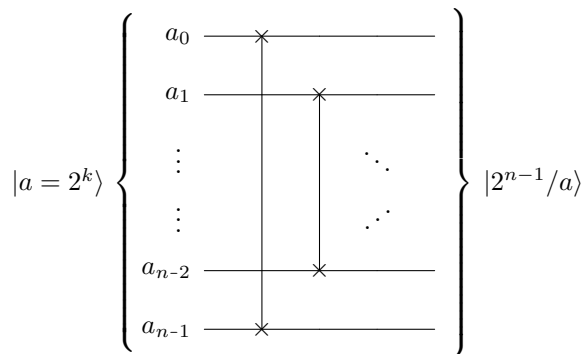


Figure 6.1: Eigenvalue inversion for inputs that are solely powers of two.

This inversion method is mainly useful for showing a proof-of-concept, as will be done in Chapter 8 on the implementation of the full HHL QLSA.

6.2.2 Newton-Raphson Algorithm

One method for inverting the eigenvalues, is to approximate the inverted values using a variation on the Newton-Raphson Root Finding Algorithm, as was proposed in [47]. It is based on the function

$$f(x) = 2x - \lambda x^2, \quad (6.2)$$

which is a parabola with its top at $(x, y) = (\lambda^{-1}, \lambda^{-1})$. Due to this top, if a first estimate x is chosen sufficiently close to the top, and the output of the function $f(x)$ is continually added back into the function, the output value will approach the value λ^{-1} arbitrarily closely. The first estimate is sufficiently close when it lies in the so-called region of convergence around the top, which

is where $|df/dx| < 1$. This constraint precisely holds for the open region $x \in (0, 2\lambda^{-1})$.

The first estimate that is used in [47], is

$$x_0 = 2^{-p}, \quad 2^{p-1} < \lambda \leq 2^p. \quad (6.3)$$

This value indeed lies in the convergence region: the estimate is indeed greater than zero, and can never be more than a factor of two away from λ^{-1} . The only problematic case could be for $\lambda = 2^k$ for some $k \in \mathbb{Z}$, but in that case the first estimate is already exactly equal to λ^{-1} . After this first estimate, all following approximations are defined as

$$\begin{aligned} x_{i+1} &= f(x_i) \\ &= 2x_i - \lambda x_i^2 \\ &= x_i (2 - \lambda x_i). \end{aligned} \quad (6.4)$$

In [47], a circuit is proposed for finding $x_0 = 2^{-p}$. This circuit for finding 2^{-p} is shown in Figure 6.2.

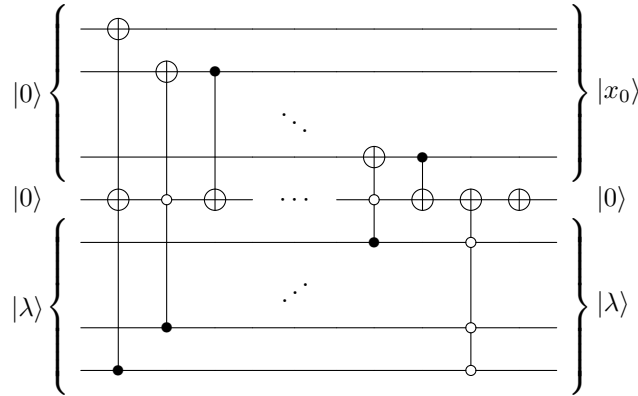


Figure 6.2: Circuit for creating the first estimate of λ^{-1} , $x_0 = 2^{-p}$ with $2^{p-1} < \lambda \leq 2^p$. The least significant qubit for both registers is the highest qubit.

However, for subsequent iterations of x_i , no explicit circuits are presented in [47]. In this section a circuit will be proposed to perform these steps, with an extra circuit specifically constructed to perform the step from x_0 to x_1 .

First, the general step from x_i to x_{i+1} for any $i \in \mathbb{N}_0$ is discussed. To this end, the last equality of Equation (6.4) is used: $x_{i+1} = x_i (2 - \lambda x_i)$. An intermediate register is used to first calculate $2 - \lambda x_i$, which is then multiplied with x_i to obtain the value for x_{i+1} . The circuit for this process is shown in Figure 6.3.

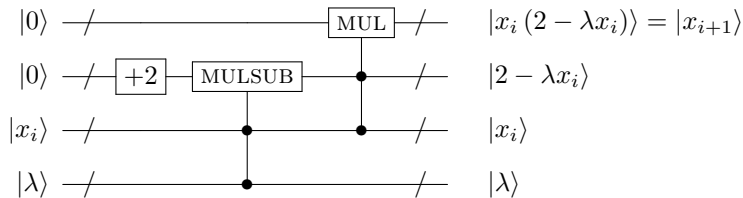


Figure 6.3: Circuit for performing the Newton-Raphson iteration step from x_i to x_{i+1} for any $i \in \mathbb{Z}_{\geq 0}$.

In the circuit, a MUL and MULSUB subroutine are used. These subroutine were defined in Chapter 3.3.

A possible bottleneck with this implementation is the number of qubits required. Assume that

λ is a positive integer of n qubits, meaning that the inverse satisfies $0 < \lambda^{-1} \leq 1$, and therefore $0 < x_i < 2\lambda^{-1} \leq 2$ for any i (so also for $i+1$). Consequently, the operation from 2 to $2 - \lambda x_i$ yields an output value of $0 < 2 - \lambda x_i < 2$. Hence, the value in the register never exceeds 2 , implying that this register containing $|2 - \lambda x_i\rangle$ is allowed have as most significant a qubit with value 2 . Then, the $+2$ operation is performed by flipping the most significant qubit. Now suppose that x_i is stored in an m_i qubit register, then $n + m_i$ qubits are required to store $|2 - \lambda x_i\rangle$, which in turn shows that $m_{i+1} = n + 2m_i$ are required to store x_{i+1} . Therefore, on top of the n qubits to store $|\lambda\rangle$, $2n + 4m_i$ qubits are required to calculate a single Newton-Raphson iteration. In the case of $i = 0$ (where $m_0 = n$), it means that to reach x_1 at least $7n$ qubits are needed, with an $m_1 = 3n$ qubit long output state. A second iteration therefore requires another $2n + 4 \cdot 3n$ qubits and results in an $m_2 = 7n$ qubit output state. More generally, x_k is an $m_k = (2^{k+1} - 1)n$ qubit value, and it takes $(2^{k+3} - 2 * k - 7)n$ qubits to find it. This is an exponential increase of qubits for higher order approximations, which is not acceptable with the current amount of qubits available on quantum computers and simulators.

The number of qubits can be reduced significantly when rounding in between steps. It is assumed that the least significant qubits in the x_i for smaller i are effectively noise, as the total error is still so large that the less significant qubits do not contribute significantly to the accuracy. The results in the QX implementation indicate that the assumption holds for x_1 . For higher values, it will not be checked due to the amount of qubits required. Thorough analysis is left for future research. Suppose it is allowed to only use the first m qubits of x_i in the next iteration. Note that this rounds down the result but keeps it above zero, which means that the rounded result inherently stays within the stable region. The rounded value can be combined with the circuit shown in Figure 1.6b to clear garbage states, to only require m extra qubits for each added step. The number of required qubits does increase to implement the garbage removal algorithm, at m qubits to copy the first x_i . The precise implementation is left for future research.

Next, the process of specifically building x_1 from x_0 is examined, to construct a circuit that performs this step more efficiently than the general method. Since $|x_0\rangle$ is a power of two, it only has a single qubit unequal to $|0\rangle$, which can be used to create a more efficient circuit. Note that x_1 will be equal to:

$$x_1 = 2^{-p+1} - 2^{-2p}\lambda. \quad (6.5)$$

This operation can be performed using only n controlled subtractions. First, $x_0 = 2^{-p}$ is multiplied by two. This does not need any circuitry; only the interpretation of the qubits building x_0 is to be changed, such that each qubit representing 2^k for some $k \in \mathbb{N}$ now represents 2^{k+1} . Just one of these qubits, namely the qubit with $k = p$, will have value $|1\rangle$, whereas all others have value $|0\rangle$. The qubits of x_0 can therefore be used as the control qubits for the subtraction of $2^{-2k}\lambda$. Only for the qubit with $k = p$ this subtraction will be performed, so only $2^{-p}\lambda$ will be subtracted, yielding $2^{-p+1} - 2^{-2p}\lambda = x_1$.

To be able to perform the $2^{-k}\lambda$ subtractions, additional less-significant qubits are necessary in the x_0 register, since $2^{-2(n-1)}\lambda$ may contain bits with significance less than 2^{-n+2} (which is the least significant qubit in $2x_0$). Specifically, the least significant possible value in $2^{-2(n-1)}\lambda$ can be $2^{-2(n-1)}$, so the x_0 register will need to be enlarged with n extra qubits.

A major gain in efficiency can be accomplished through the range of the subtractions. When qubit 2^{-p+k+1} is in the $|1\rangle$ state, it means that the k most significant qubits of λ are zero, i.e. that only the last $n - k$ qubits of $|\lambda\rangle$ need to be taken into account when subtracting. These are exactly the qubits of λ such that all non-zero qubits of $2^{-2(p+k)}\lambda$ are less significant than 2^{-p+k+1} .

A problem with the controlled subtractions is that controlling a subtraction by its most significant qubit is not possible with the subtractors discussed in this thesis. A solution is to copy the value into an ancilla qubit using a CNOT. This copied value can then be used to control the subtraction. This method, however, yields a new problem. The problem lies in the fact that subtracting any value from $2x_0$ will cause the non-zero qubit to overflow, meaning that its value is not constant and can therefore not be used to reset the ancilla qubit. Therefore, n of these ancillae

are required. If however in the future a method becomes available to control a subtracter by its most significant qubit, these ancillae would not be necessary anymore. The circuit for the first Newton-Raphson iteration is shown in Figure 6.4.

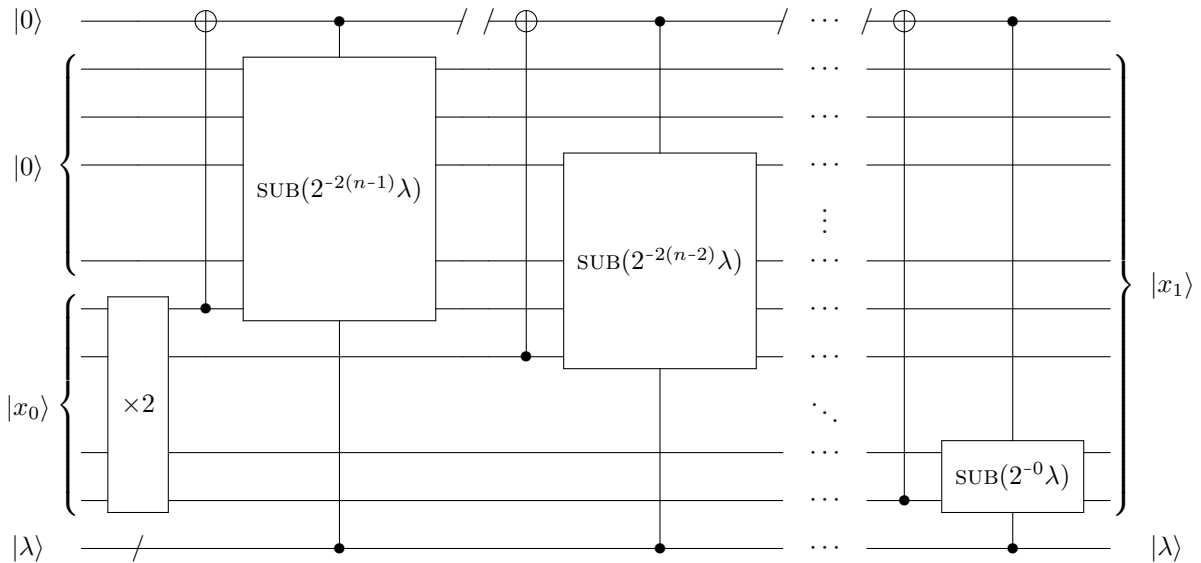


Figure 6.4: Circuit for performing the first Newton-Raphson iteration x_0 to x_1 .

The approximations created using the Newton-Raphson method have different errors for different values. The first four approximations are shown in Figure 6.5.

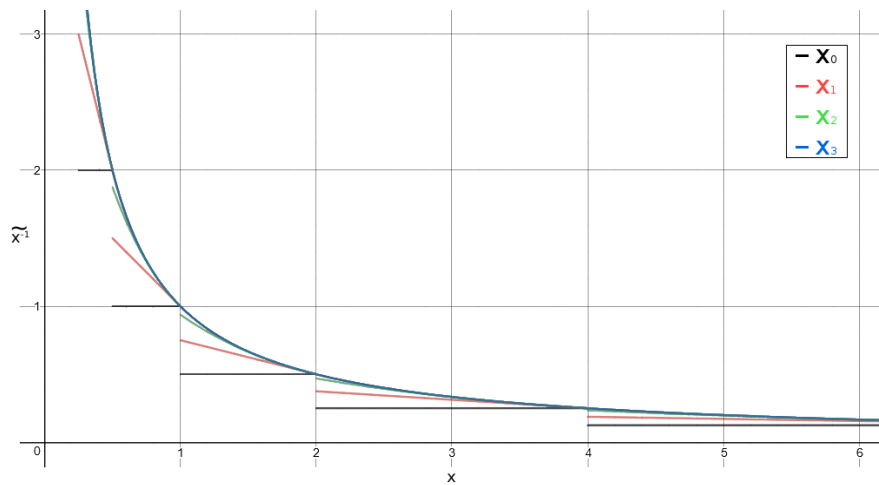


Figure 6.5: Approximations of x^{-1} using Newton-Raphson iterations.

It can be seen that for $\lambda = 2^n$ the approximations are perfect, while for values downward towards a lower power of two the approximation becomes less accurate. For the first approximation x_0 , the maximum relative error is 50% as the approximation for $\lambda = 2^k + \epsilon$ is equal to $x_0 = 2^{k-1}$. Each iteration yields a quadratically better approximation than the previous, meaning that the maximum relative error for any iteration x_k is 0.5^{2^k} . For example, the relative errors for the first

four approximations are,

$$\begin{aligned}
 x_0 : \quad \epsilon_{\text{rel, max}} &= 0.5, \\
 x_1 : \quad \epsilon_{\text{rel, max}} &= 0.25, \\
 x_2 : \quad \epsilon_{\text{rel, max}} &= 0.0625, \\
 x_3 : \quad \epsilon_{\text{rel, max}} &= 0.00375.
 \end{aligned} \tag{6.6}$$

As was discussed before, these approximations come at the cost of requiring a significant amount of ancillae. For example, when a relative error of less than 0.25 is desired, a bare minimum of $n + 2n + 6n = 9n$ qubits is necessary to store the final answers alone. With a maximum of around 40 qubits in simulation, the algorithm is therefore impractical for higher order approximations at the current state of technology.

6.2.3 Thapliyal Inversion Algorithm

The final algorithm applies the Adapted Thapliyal Integer Division Algorithm defined in Chapter 3.4.1. The extension of the algorithm to accommodate larger a will be used. A brief recap of the Extended Thapliyal Integer Division Algorithm is given, and it will be shown how the algorithm can be converted into an inversion algorithm.

For any m and $n \leq m$ qubit integers a and b the Extended Thapliyal Integer Division Algorithm finds the m and n qubit integers q and r , respectively, such that $a = q \cdot b + r$. To find q and r , the algorithm requires $2n + m + 2$ qubits. These qubits are always split up into three registers, although the registers before and after the algorithm are not equal. Before the algorithm, the qubits are split up into the registers N , O and D (of m , $n + 1$ and $n + 1$ qubits, respectively), and afterwards in the registers R , Q and D (of $n + 1$, m and $n + 1$ qubits, respectively). Using these registers, the algorithm performs the operation,

$$|a\rangle_N |0\rangle_O |b\rangle_D \xrightarrow{\text{DIV}} |r\rangle_R |q\rangle_Q |b\rangle_D. \tag{6.7}$$

The registers D and R are one qubit larger than their respective contents of b and r . The most significant qubit in these two registers must remain zero. Otherwise, the algorithm will give an erroneous result as was explained in Chapter 3.4.1. The complete in- and outputs of the algorithm are shown in Figure 6.6.

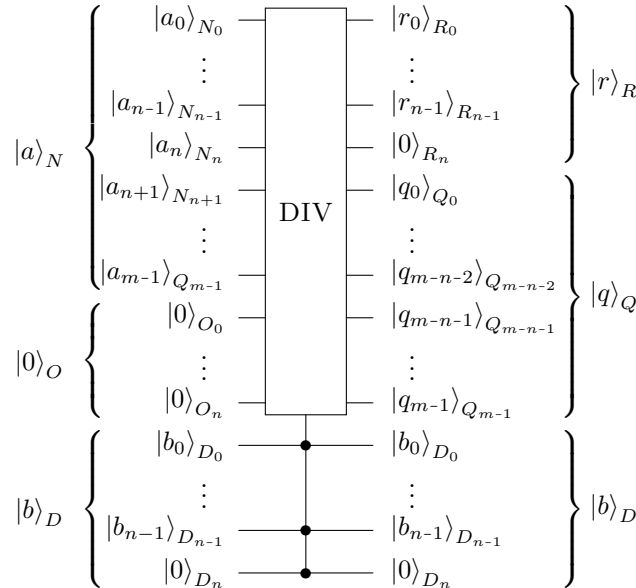


Figure 6.6: In- and outputs for the complete Adapted Thapliyal Integer Division Algorithm for unequal register sizes.

The Thapliyal Division Algorithm will now be used to find an approximation of the inverse of a positive integer eigenvalue λ . Earlier in this chapter it was observed that $2^k/\lambda$ is effectively equivalent to the actual inverse $1/\lambda$ for any integer k . Also note that q is equal to the floor of the real value a/b , i.e. $q = \lfloor a/b \rfloor$. Combining this shows that when $a = 2^{m-1}$ and $b = \lambda$ are taken, the result $q = \lfloor 2^{m-1}/\lambda \rfloor \equiv \widetilde{\lambda^{-1}}$ is obtained. With these modifications, the total process of the Thapliyal Inversion algorithm becomes

$$|2^{m-1}\rangle_N |0\rangle_O |\lambda\rangle_D \xrightarrow{\text{Thap. Inv.}} |r\rangle_R |\widetilde{\lambda^{-1}}\rangle_Q |\lambda\rangle_D. \quad (6.8)$$

The value of the power in $a = 2^{m-1}$ is not chosen arbitrarily: since N is an m qubit register, 2^{m-1} is the largest power of two which can be saved in it, as $|100 \cdots 00\rangle_N$. It should be noted that dividing q by a yield the approximation of the inverse $1/\lambda$ in decimal, since $q/a = \lfloor 2^{m-1}/\lambda \rfloor / 2^{m-1} \approx 1/\lambda$. This fact is used in the section on the QX implementation, to intuitively show the output of the algorithm.

A remaining question is the accuracy of $\widetilde{\lambda^{-1}}$. Register Q per definition contains an integer, with its least significant qubit having a value of $2^0 = 1$. Now since Q contains m qubits, its most significant qubit has a value of 2^{m-1} . If the register was instead interpreted as $1/\lambda$ instead of $2^{m-1}/\lambda$, then it follows that the most significant qubit now carries a value of 2^0 , and the least significant qubit a value of $2^{-(m-1)}$. Therefore, the maximum error in the approximation is $\epsilon_{\max} = 2^{-(m-1)}$. Since λ can have a value of up to $\lambda = 2^n - 1 < 2^n$, the minimum inverse becomes $(1/\lambda)_{\min} = 1/(2^n - 1) > 2^{-n}$. Therefore, the maximum relative error is

$$\epsilon_{\text{rel, max}} = \epsilon_{\max} / (1/\lambda)_{\min} < 2^{-(m-1)} / 2^{-n} = 2^{n-m+1} \quad (6.9)$$

For any $m > n + 1$ the relative error is always smaller than the inverse itself, with exponentially higher accuracy for higher m at the cost of a constant increase in circuit width and depth.

6.3 Comparison of the algorithms

In this chapter, three algorithms for Eigenvalue Inversion have been discussed. To build a complete Quantum Linear Solver however, only one algorithm is required. When choosing, the Thapliyal Inversion algorithm is the preferred algorithm. This is mainly due to the disadvantageous properties of the other algorithms, which will now be discussed. The Power-of-Two Inverter only works for powers of two as input. For a proof of concept implementation this can be ideal, but for a general linear solver it is not acceptable. The main alternative to the Thapliyal Inversion algorithm is therefore the Newton-Raphson Inverter. Its fast convergence makes it ideal in theory, but it is unusable at the current state of technology due to its large number of required ancillae. The Thapliyal Inversion algorithm, which only requires $n + 2$ ancillae for an n qubit input, is therefore the preferred algorithm going onward.

6.4 QX Implementation

None of the algorithms discussed in this chapter pose significant challenges in their QX implementations, although the higher-order Newton-Raphson Iterations are not implemented due to their high number of required qubits. The analysis of the Powers-of-Two inverter is also excluded from this section, as its functioning is inherent and does not require testing. The implementations of the Newton-Raphson and Thapliyal Inversion algorithms will therefore be discussed in the remainder of this chapter.

Of the Newton-Raphson Inversion, only the circuits for the first two iterations x_0 and x_1 have been implemented, since higher orders would take too many qubits to implement at over $7n$ qubits. The

n ancilla qubits were reduced to one single ancilla, by resetting the qubit after each subtraction. This makes the circuit irreversible and impossible to implement on a real quantum computer, but aids calculation time in the proof of concept.

The algorithm was tested for all possible inputs of four qubits and for several numbers of more qubits in length. The algorithm behaved as expected, i.e. its approximations were equal to the expectations. An example output for the $n = 5$ qubit input $\lambda = 01011_2 = 11$ is shown in Figure 6.7. The found approximations indeed align with the theoretical approximations of $x_0 = 2^{-\lceil \log(\lambda) \rceil} = 0.125$ and $x_1 = 2x_0 - \lambda x_0^2 = 0.078125$.

```

Inversion Newton-Raphson:

input a      =    01011 = 11

output x0    = 000100000 = 0.125
output x1    = 000010100 = 0.078125

expected x0  = 000100000 = 0.125
expected x1  = 000010100 = 0.078125

output ideal = 000010111 = 0.0909091

```

Figure 6.7: Output of the first two orders of Newton-Raphson Inversion for the input 01011.

The Thapliyal Inversion algorithm is performed using the Thapliyal Division implemented in Chapter 3.4.1, by taking the inputs $a = 2^k$ and $b = \lambda$. In Section 6.2.3, it was explained that dividing the output quotient q of the Thapliyal division algorithm by the input value a yields the decimal representation of the approximation of the inverse, that is, $q/a \approx 1/\lambda$. In validating the Thapliyal Division algorithm, the Thapliyal Inversion algorithm has effectively been validated as well. To give an idea of the output of the algorithm, the example from Newton-Raphson Inversion is repeated, i.e. $\lambda = 01011_2 = 11$. For k the value $k = 15$ is chosen, yielding $a = 16384$. For these inputs, 25 qubits are required in the calculations. Higher values of k result in significant calculation times and do not yield much extra insight in the working of the algorithm. The results of the calculation are shown in Figure 6.8. In this figure, the values a and b are written as n and d , which is another common notation. Additionally, the answers are shown as $q = 1489$ and $r = 5$, which need to be rewritten to a recognisable form using $q/a \approx 1/\lambda$. The found approximation for the inverse is therefore $q/a = 1489/16384 \approx 0.09088$, compared to the actual value of $1/11 \approx 0.090909$. When comparing the bit representation of the approximation $q/a = 0.00010111010001_2$ to that of the actual approximation $1/11 \approx 0.00010111010001011101_2$, it is clear that the approximation is indeed the first $k = 15$ bits of the real value, as expected. This leads to the conclusion that the implementation of the Thapliyal Inversion Algorithm functions properly.

```

Division Thapliyal:

input n      = 100000000000000 = 16384
input d      = 01011 = 11

output q = n/d = 000010111010001 = 1489
output r = n%d = 00101 = 5

test q*d + r = 1489*11 + 5 = 16384
correctness: True

```

Figure 6.8: Output of the Thapliyal Division algorithm used as Thapliyal Inversion for the input $\lambda = 01011_2 = 11$ and $k = 15$ bit precision. The approximated value for $1/11 \approx 0.00010111010001011101_2 \approx 0.090909$ is $q/a = 1489/16384 \approx 0.09088$.

7. Ancilla Rotation

In this chapter the third and final subroutine of the HHL algorithm is discussed, which is the Ancilla Rotation subroutine. The inputs for this subroutine are an ancilla qubit A initialised in the $|0\rangle$ state, and an m qubit register Q , containing an inverted eigenvalue $\lambda^{-1} \leq 1$. In the subroutine, the ancilla qubit A should be rotated to a state, where the complex amplitude of $|1\rangle_A$ is proportional to λ^{-1} . The desired operation of the Ancilla Rotation subroutine is therefore [34],

$$|0\rangle_A |\lambda^{-1}\rangle_Q \xrightarrow{\text{Anc. Rot.}} \left(\sqrt{1 - \frac{c^2}{\lambda^2}} |0\rangle_A + \frac{c}{\lambda} |1\rangle_A \right) |\lambda^{-1}\rangle_Q, \quad (7.1)$$

where $c \in \mathbb{R}$ is a predetermined value satisfying the condition $|c| < 1$. First, an implementation by Cao et al. [34] is discussed, after which two times two expansions are proposed which increase accuracy for constant c . Two extensions are implemented in the QX simulator and verified.

7.1 Cao implementation

In [34], Cao et al. propose a method to approximate the output state for the ancilla qubit. The algorithm solely uses m controlled- $R_y(\theta)$ gates. In Chapter 1, the $R_y(\theta)$ gate was defined as

$$R_y(\theta) = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}. \quad (7.2)$$

From this definition it is observed that by applying $R_y(\theta)$ to the $|0\rangle_A$ the state

$$R_y(\theta) |0\rangle_A = \cos \left(\frac{\theta}{2} \right) |0\rangle_A + \sin \left(\frac{\theta}{2} \right) |1\rangle_A \quad (7.3)$$

is obtained. Therefore, if an angle θ is found such that $\sin(\theta/2) = c/\lambda$, the desired rotation is performed. From the equality $\sin(\arcsin(\varphi)) = \varphi$ for $\varphi \in (-\pi, \pi)$, it is observed that this holds for $\theta = 2 \arcsin(c/\lambda)$. That is,

$$R_y \left(2 \arcsin \left(\frac{C}{\lambda} \right) \right) |0\rangle_A = \sqrt{1 - \frac{C^2}{\lambda^2}} |0\rangle_A + \frac{C}{\lambda} |1\rangle_A. \quad (7.4)$$

The challenge of the subroutine, hence, lies in the transformation $x \rightarrow \arcsin(x)$, or more specifically, $x \rightarrow 2 \arcsin(cx)$. Once this transformation is performed, only controlled y -rotations are required to perform the ancilla rotation. The transformation is where Cao et al. use an approximation. Through the Taylor expansion of $\arcsin(x)$ around $x = 0$, $\arcsin(x)$ can be written as [48],

$$\arcsin(x) = x + \sum_{k=1}^{\infty} \frac{1 \cdot 3 \cdot 5 \cdot \dots \cdot (2k-1)}{2 \cdot 4 \cdot 6 \cdot \dots \cdot (2k)} \frac{x^{2k+1}}{2k+1} = x + \frac{1}{6}x^3 + \frac{3}{40}x^5 + \frac{5}{112}x^7 + \frac{35}{1152}x^9 + \mathcal{O}(x^{11}). \quad (7.5)$$

In Equation (7.5), it is shown that $\arcsin(x)$ can be approximated to $\arcsin(x) \approx x$ for small x . Hence, with the y -rotations applied directly to λ^{-1} , the ancilla rotation can be performed arbitrarily precisely, as long as c is kept small. The implementation that Cao et al. use in [34] is shown in Figure 7.1.

The implementation exploits the fact that multiplication of R_y gates sums the angles (i.e. $R_y(\theta)R_y(\varphi) = R_y(\theta + \varphi)$), in combination with the fact that a binary number can be written as a sum, e.g. $x = x_0.x_1x_2 \dots x_{m-1} = x_0 + x_12^{-1} + x_22^{-2} + \dots + x_{m-1}2^{-m+1}$ with $x_j \in \{0, 1\}$. Hence,

$$R_y(cx) = R_y(cx_0) R_y(cx_12^{-1}) R_y(cx_22^{-2}) \dots R_y(cx_{m-1}2^{-m+1}). \quad (7.6)$$

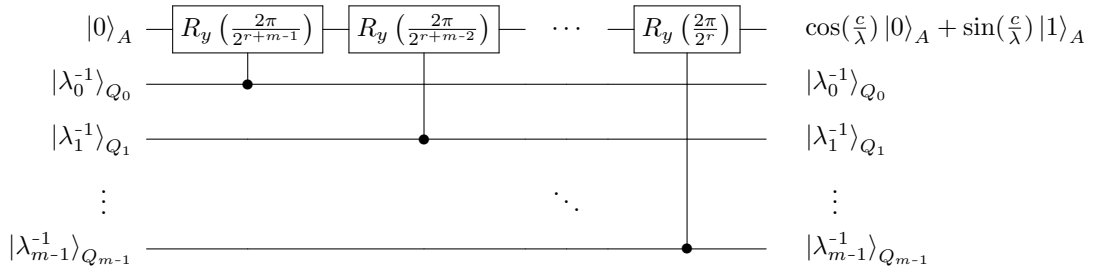


Figure 7.1: Circuit for performing the ancilla rotation $|0\rangle_A \rightarrow \cos(c/\lambda)|0\rangle_A + \sin(c/\lambda)|1\rangle_A \approx \sqrt{1-c^2/\lambda^2}|0\rangle_A + c/\lambda|1\rangle_A$, for $c = \pi/2^r$, with a freely selectable integer $r \in \mathbb{N}$.

Since the x_j are either zero or one, they effectively act as a control. For example, $R_y(cx_0)$ is equivalent to $C^{x_0}[R_y(c)]$. Equation (7.6) can therefore be reformulated as

$$R_y(cx) = C^{x_0}[R_y(c)] C^{x_1}[R_y(c2^{-1})] C^{x_2}[R_y(c2^{-2})] \cdots C^{x_{m-1}}[R_y(c2^{-m+1})]. \quad (7.7)$$

This is equivalent to the circuit shown in Figure 7.1 for $c = 2\pi/2^r$, which yields the output ancilla state,

$$\cos\left(\frac{c}{\lambda}\right)|0\rangle_A + \sin\left(\frac{c}{\lambda}\right)|1\rangle_A. \quad (7.8)$$

The value of the positive integer $r \in \mathbb{N}$ is to be chosen beforehand. Note that $\sin(x)$ is only monotonously increasing for $x \leq \pi/2$, which means that with a maximum value of $\lambda^{-1} = 1$ at least $r = 1$ is required in order to have a unique answer for each λ^{-1} . Increasing r will decrease c , and hence create a better approximation; the error increases as $\mathcal{O}(c^3)$. A downside of increasing r , however, is that this decreases the complex amplitude of $|1\rangle_A$, and hence also the probability of finding a correct answer in the HHL QLSA. This probability decrease is linear in c . The choice for r is therefore a consideration between precision and efficiency. The maximum error and probability of measuring $|1\rangle_A$ for different values of r are shown in Table 7.1.

r	1	2	3	4	5	6
max error	$5.71 \cdot 10^{-1}$	$7.83 \cdot 10^{-2}$	$1.00 \cdot 10^{-2}$	$1.26 \cdot 10^{-3}$	$1.58 \cdot 10^{-4}$	$1.97 \cdot 10^{-5}$
max probability	$1.00 \cdot 10^0$	$5.00 \cdot 10^{-1}$	$1.46 \cdot 10^{-1}$	$3.81 \cdot 10^{-2}$	$9.61 \cdot 10^{-3}$	$2.41 \cdot 10^{-3}$

Table 7.1: Table of the maximum error and probability for $|1\rangle_A$ for different values of r in the approximation used in the paper by Cao et al. [34]. The maximum error in the approximation is reached for the highest value of $\lambda^{-1} = 1$. In other words, the maximum error is equal to $\pi/2^r - \sin(\pi/2^r)$. The probability for measuring $|1\rangle_A$ is equal to the square of its complex amplitude. Again the maximum is for $\lambda^{-1} = 1$, for which the probability is equal to $\langle 1|1\rangle_A = \sin^2(\pi/2^r)$.

7.2 Proposed extensions

7.2.1 Third order approximations

An option for improved accuracy is to use a higher-order Taylor approximation of the $\arcsin(x)$ function as defined in Equation (7.5). Here, two possible extensions to the Cao Algorithm are proposed. Consider the first higher-order approximation, $\arcsin(x) \approx x + x^3/6$. This approximation can be built in two ways: either $(\lambda^{-1})^3 = \lambda^{-3}$ can explicitly be calculated, or the rotation $R_y(c^3/\lambda^3)$ can be applied directly from λ^{-1} . The two respective methods are shown in Figure 7.2.

7.2.1.1 Explicit Third Power Calculation Ancilla Rotation Algorithm

In the first case, where λ^{-3} is explicitly calculated, the rotation method from the Cao Implementation can be repeated on λ^{-3} , except with $c \rightarrow c^3/6$. The method has a runtime of $\mathcal{O}(n^2)$, and its circuit is shown in Figure 7.2a. The difficulty in implementing this algorithm is the calculation of $x \rightarrow x^3$. With the algorithms discussed in this thesis, this can only be accomplished by (1) copying

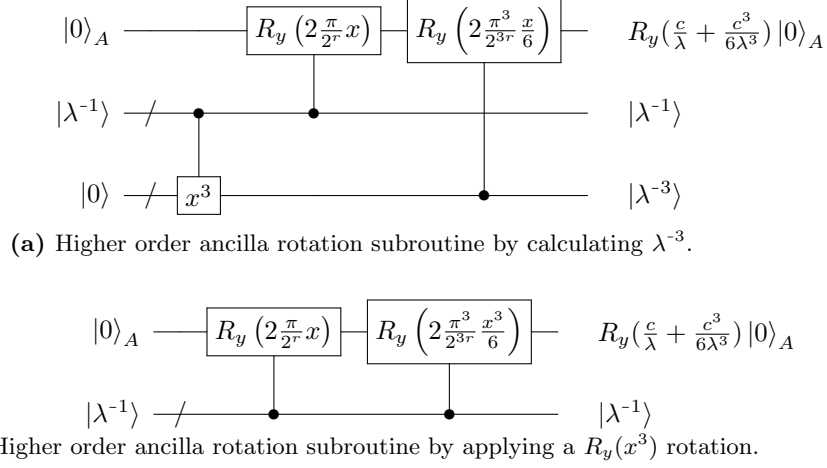


Figure 7.2: Two methods for applying a higher order ancilla rotation: one by calculating λ^{-3} explicitly, the other by directly applying the rotation $R_y(cx^3)$ with $c = \pi/2^r$.

x into a second register, (2) calculating x^2 in a third register by multiplying these two copies, and (3) calculating x^3 in a fourth register by multiplying x^2 and x . A circuit to perform this process is shown in Figure 7.3.

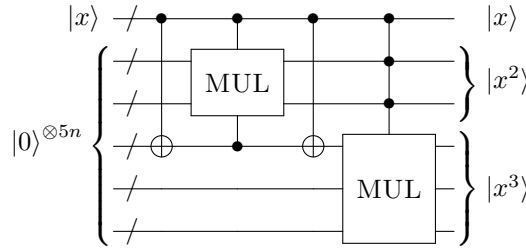


Figure 7.3: Circuit for calculating x^3 from an n -bit number x .

The copying subroutine is shown in Figure 1.7. The least amount of qubits required to calculate x^3 using this method is equal to $n + 2n + 3n = 6n$ in order to store x , x^2 and x^3 respectively. Note that no additional n qubits are required to save the copy of x , since after calculating x^2 the register storing the second x can be cleared again by repeating the copying procedure. It can then be reused to store x^3 , as is shown in the figure.

The copying subroutine requires n CNOT gates, leading to the circuit depth $\mathcal{O}(n)$. The multipliers implemented in this thesis have circuit depths of $\mathcal{O}(n^2)$. Therefore, the complete Explicit Third Power Calculation Ancilla Rotation Algorithm has a circuit depth of order $\mathcal{O}(n^2)$, and a circuit width of $6n = \mathcal{O}(n)$.

7.2.1.2 Direct Third Power Rotation Ancilla Rotation Algorithm

The second method, where the operation $R_y(x^3)$ is performed directly from x will be examined in this section. In contrast to the Explicit Third Order Calculation methods, it requires only a single extra qubit in practice. It is, however, very specific in its implementation for each number of qubits, and has a circuit depth of order $\mathcal{O}(n^3)$. The general circuit for the algorithm is shown in Figure 7.2b. The challenge is to find a subroutine such that $R_y(cx^3)$ can be applied directly from x . To this end, it is used that x^3 can be written as a sum of multiplications of the bits of x . Due to the writing complexity, the method will be explained for four qubits, but is similar for different numbers of qubits. Since it was assumed that $0 \leq x \leq 1$ (and that x is only four qubits long), x

can be written in binary as

$$\begin{aligned}
x &= x_0.x_{-1}x_{-2}x_{-3} \\
&= x_0 + 2^{-1}x_{-1} + 2^{-2}x_{-2} + 2^{-3}x_{-3} \\
&= a + b + c + d,
\end{aligned} \tag{7.9}$$

where a through d are defined as $a \equiv x_0$, $b \equiv 2^{-1}x_{-1}$, $c \equiv 2^{-2}x_{-2}$, and $d \equiv 2^{-3}x_{-3}$. This allows for the rewrite of x^3 as a sum of partial products, which in turn can be rewritten to conditional sums,

$$\begin{aligned}
x^3 &= (a + b + c + d)^3 \\
&= a^3 + b^3 + c^3 + d^3 \\
&\quad + 3[a^2b + ab^2 + a^2c + ac^2 \\
&\quad + a^2d + ad^2 + b^2c + bc^2 \\
&\quad + b^2d + bd^2 + c^2d + cd^2] \\
&\quad + 6[abc + abd + acd + bcd] \\
&= x_0^3 + x_{-1}^3 2^{-3} + x_{-2}^3 2^{-6} + x_{-3}^3 2^{-9} \\
&\quad + 3[x_0^2 x_{-1} 2^{-1} + x_0 x_{-1}^2 2^{-2} + x_0^2 x_{-2} 2^{-2} + x_0 x_{-2}^2 2^{-4} \\
&\quad + x_0^2 x_{-3} 2^{-3} + x_0 x_{-3}^2 2^{-6} + x_{-1}^2 x_{-2} 2^{-4} + x_{-1} x_{-2}^2 2^{-5} \\
&\quad + x_{-1}^2 x_{-3} 2^{-5} + x_{-1} x_{-3}^2 2^{-7} + x_{-2}^2 x_{-3} 2^{-7} + x_{-2} x_{-3}^2 2^{-8}] \\
&\quad + 6[x_0 x_{-1} x_{-2} 2^{-3} + x_0 x_{-1} x_{-3} 2^{-4} + x_0 x_{-2} x_{-3} 2^{-5} + x_{-1} x_{-2} x_{-3} 2^{-6}] \\
&= x_0 + x_{-1} 2^{-3} + x_{-2} 2^{-6} + x_{-3} 2^{-9} \\
&\quad + 3[x_0 x_{-1} 2^{-1} + x_0 x_{-1} 2^{-2} + x_0 x_{-2} 2^{-2} + x_0 x_{-2} 2^{-4} \\
&\quad + x_0 x_{-3} 2^{-3} + x_0 x_{-3} 2^{-6} + x_{-1} x_{-2} 2^{-4} + x_{-1} x_{-2} 2^{-5} \\
&\quad + x_{-1} x_{-3} 2^{-5} + x_{-1} x_{-3} 2^{-7} + x_{-2} x_{-3} 2^{-7} + x_{-2} x_{-3} 2^{-8}] \\
&\quad + 6[x_0 x_{-1} x_{-2} 2^{-3} + x_0 x_{-1} x_{-3} 2^{-4} + x_0 x_{-2} x_{-3} 2^{-5} + x_{-1} x_{-2} x_{-3} 2^{-6}] \\
&= x_0 + x_{-1} 2^{-3} + x_{-2} 2^{-6} + x_{-3} 2^{-9} \\
&\quad + 3[x_0 x_{-1} (2^{-1} + 2^{-2}) + x_0 x_{-2} (2^{-2} + 2^{-4}) + x_0 x_{-3} (2^{-3} + 2^{-6}) \\
&\quad + x_{-1} x_{-2} (2^{-4} + 2^{-5}) + x_{-1} x_{-3} (2^{-5} + 2^{-7}) + x_{-2} x_{-3} (2^{-7} + 2^{-8})] \\
&\quad + 6[x_0 x_{-1} x_{-2} 2^{-3} + x_0 x_{-1} x_{-3} 2^{-4} + x_0 x_{-2} x_{-3} 2^{-5} + x_{-1} x_{-2} x_{-3} 2^{-6}]
\end{aligned} \tag{7.10}$$

A couple of remarks on this rewrite, in order: (1) From step 2 to step 3, the x_i are reintroduced. However, now when two x_i are next to each other, it is not meant that they are forming a bitwise number, but instead that they are multiplied. (2) From step 3 to step 4, the powers of all the x_i are removed. This is allowed, since all the x_i either have value 0 or 1, which are both number with the property that $x_i^k = x_i$ for all $k \in \mathbb{N}$. (3) In the last step, all the same multiplications of x_i 's are grouped.

The value x^3 is now split into a sum of products, which means that $R_y(cx^3)$ can be written as a number of separate rotations, comparable to Equation (7.6),

$$\begin{aligned}
R_y(cx^3) &= \\
&R_y(cx_0) \ R_y(cx_{-1} 2^{-3}) \ R_y(cx_{-2} 2^{-6}) \ R_y(cx_{-3} 2^{-9}) \\
&R_y(3cx_0 x_{-1} (2^{-1} + 2^{-2})) \ R_y(3cx_0 x_{-2} (2^{-2} + 2^{-4})) \ R_y(3cx_0 x_{-3} (2^{-3} + 2^{-6})) \\
&R_y(3cx_{-1} x_{-2} (2^{-4} + 2^{-5})) \ R_y(3cx_{-1} x_{-3} (2^{-5} + 2^{-7})) \ R_y(3cx_{-2} x_{-3} (2^{-7} + 2^{-8})) \\
&R_y(6cx_0 x_{-1} x_{-2} 2^{-3}) \ R_y(6cx_0 x_{-1} x_{-3} 2^{-4}) \ R_y(6cx_0 x_{-2} x_{-3} 2^{-5}) \ R_y(6cx_{-1} x_{-2} x_{-3} 2^{-6})
\end{aligned} \tag{7.11}$$

The value of a product of x_i 's is binary, with only an output value of 1 when all the x_i in the product are equal to 1. In the section on the Cao implementation it was shown that a binary value in an R_y gate effectively act as a control, so a product of k x_i 's in an R_y gate behaves as a k -fold

controlled y -rotation. Hence, the third power rotation $R_y(cx^3)$ can be rewritten to a number of controlled- R_y gates,

$$\begin{aligned}
R_y(cx^3) = & \\
& C^{x_0}[R_y(c)] \quad C^{x_1}[R_y(c2^{-3})] \quad C^{x_2}[R_y(c2^{-6})] \quad C^{x_3}[R_y(c2^{-9})] \\
& C^{x_0x_1}[R_y(3c(2^{-1} + 2^{-2}))] \quad C^{x_0x_2}[R_y(3c(2^{-2} + 2^{-4}))] \quad C^{x_0x_3}[R_y(3c(2^{-3} + 2^{-6}))] \\
& C^{x_1x_2}[R_y(3c(2^{-4} + 2^{-5}))] \quad C^{x_1x_3}[R_y(3c(2^{-5} + 2^{-7}))] \quad C^{x_2x_3}[R_y(3c(2^{-7} + 2^{-8}))] \\
& C^{x_0x_1x_2}[R_y(6c2^{-3})] \quad C^{x_0x_1x_3}[R_y(6c2^{-4})] \quad C^{x_0x_2x_3}[R_y(6c2^{-5})] \quad C^{x_1x_2x_3}[R_y(6c2^{-6})]
\end{aligned} \tag{7.12}$$

This final rewrite is precisely equivalent to the circuit performing a third-order rotation, except not yet written in a recognisable form. The actual circuit is shown in Figure 7.7 on page 66. In this circuit, three-fold controlled gates are used, which are not gates that are available in QX by default. In the section on the QX Implementation it will be shown that the three-fold-controlled- R_y gate can be implemented using a single ancilla qubit. This is precisely the reason for requiring an ancilla qubit in the algorithm.

For different numbers of qubits, the circuit will be similar in the sense that it will consist of rotations of every possible combination of controlled- R_y gates, controlled by either one, two or three qubits. Specifically, $\binom{n}{1} = n$, $\binom{n}{2} = \frac{1}{2}n(n-1)$ and $\binom{n}{3} = \frac{1}{6}n(n-1)(n-2)$ of these controlled rotations are required, respectively. This means that the operation scales as $\mathcal{O}(n^3)$. The angles of the gates can be found in the same manner as in Equation (7.10). A more direct method will be discussed in the next section.

7.2.1.3 Comparison of the methods

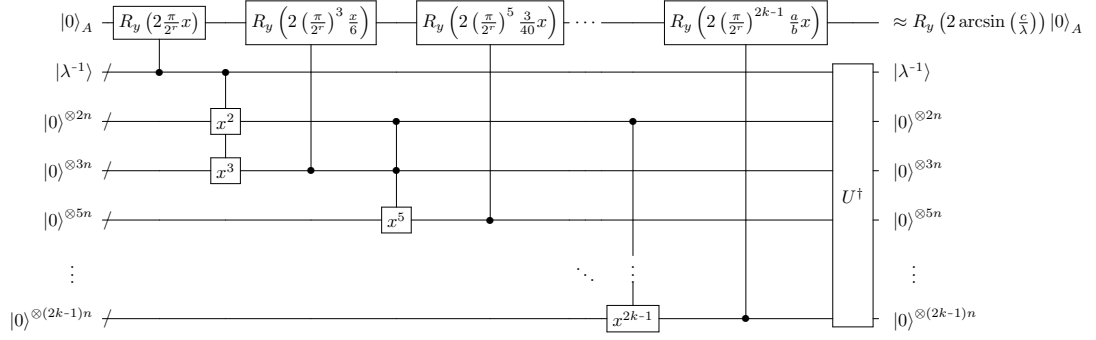
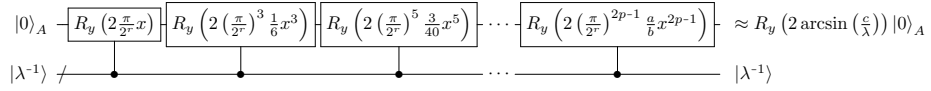
When explicitly calculating the third power λ^{-3} , the circuit has a circuit depth of order $\mathcal{O}(n^2)$, and the number of required number of qubits scales as $\mathcal{O}(n)$. Compare this to the second method of directly rotating by the third power, which has circuit depth $\mathcal{O}(n^3)$ and circuit width $\mathcal{O}(n)$. Written this way, it seems that the explicit method is more convenient. The notation however does not show the large constant factors in that method: roughly $12n^2 + \mathcal{O}(n)$ and $6n + \mathcal{O}(1)$ gates and qubits, respectively, compared to the $\frac{1}{6}n^3 + \frac{5}{6}n$ gates and $n + \mathcal{O}(1)$ qubits for the second method. It takes values saved in registers of over $n = 70$ qubits long (i.e. values greater than roughly 10^{21}) to need a larger circuit depth using the direct rotation method compared to the explicit method. Currently, there are no quantum computers or simulators that can even get close to that number of qubits, so the direct rotation method is preferred at this stage. However, given that one of the goals for quantum computers is to compute especially large problems at some point, the explicit method will possibly become preferred in the future.

7.2.2 Higher order approximations

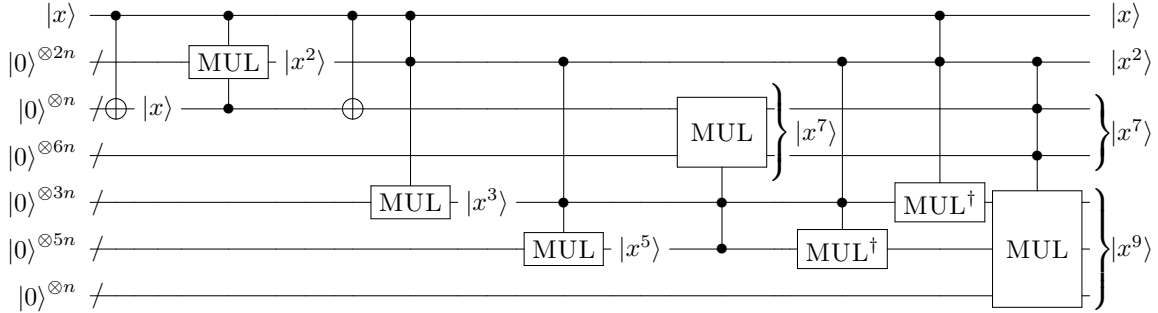
The two methods from the previous section can be adapted to implement higher orders than only the third order. With higher-order methods it is meant that $\arcsin(x)$ is approximated as a higher-order polynomial, according to Equation (7.5). Consider the approximation up to x^{2k+1} . Then the powers x , x^3 , x^5 , \dots , x^{2k+1} are required. The two adaptations to the algorithms from the previous section are shown in Figure 7.4. Again both methods will be discussed.

7.2.2.1 Explicit Higher-Power Calculation Ancilla Rotation Algorithm

In this method, comparable to the third power approximation, all x^k are explicitly calculated. The first two powers x^2 and x^3 are calculated the same way as before. Using these values, x^5 can be calculated by multiplying x^2 and x^3 , x^7 can be calculated by multiplying x^2 and x^5 etcetera, up until x^{2n-1} from x^2 and x^{2k-1} . Using the direct approach, this requires $n + 2n + 3n + 5n + 7n + \dots + (2k+1)n = (k^2 + 2k + 3)n$ qubits, as in Figure 7.4a. Comparable to the uncomputation of the second x before, it is possible to first build the lower-power numbers in the registers of the higher-power numbers. The rotations of these lower power numbers can be performed directly

(a) Higher order ancilla rotation subroutine by explicitly calculating λ^{-p} from λ^{-1} .(b) Higher order ancilla rotation subroutine by directly applying $R_y(x^k)$ rotations to λ^{-1} .**Figure 7.4:** Two methods for applying a higher order ancilla rotation: one by finding the λ^{-k} explicitly, the other by applying the rotations $R_y(cx^k)$.

after the number is constructed, and can then directly be uncomputed to make room for the higher-power numbers. An example for $k = 4$ (i.e. up to x^9) is shown in Figure 7.5. Now instead of at least $27n$, only a minimum of $n + 2n + (2k - 1)n + (2k + 1)n = (4k + 3)n = 19n$ qubits are required, in order to save x , x^2 , x^7 and x^9 (since x cannot be cleared, and x^2 and x^7 are required to build x^9). This process cannot easily be generalised for any k however, since the total size of the lower-power-number registers scales quadratically, compared to the linear increase in individual register sizes. This implies that for higher k not all lower-power numbers will fit into the larger-power-number registers as with $k \leq 4$, which in turn will mean that extra qubits will be required. With current technologies, however, where a maximum simulation can only handle around 40 qubits, these higher powers cannot be reached anyway for any reasonably sized input value. The large benefit of this explicit method though is that it will always scale with $\mathcal{O}(n^2)$, so it may become preferred once large-scale quantum computers become available.

**Figure 7.5:** Circuit for building x^3 , x^5 , x^7 and x^9 from x using multiplication.

7.2.2.2 Direct Higher-Power Rotation Ancilla Rotation Algorithm

The second method, that is, direct rotation proportional to x^p from x , is currently much more attractive due to its smaller memory footprint, i.e. the number of required qubits. In this method, implementing the highest power $R_y(cx^{2k+1})$ requires R_y gates controlled to up to $2k + 1$ qubits, with $\binom{n}{p}$ rotations controlled by p qubits. In the section on QX implementation, it will be shown that a p -fold-controlled- R_y gate can be implemented using $p - 2$ ancillae, implying that $2k - 1$ ancillae are required for an approximation up to the $(2k + 1)$ -th order. The circuit depth of the complete algorithm scales as $\mathcal{O}(n^{2k+1})$.

In this section it will be explained how a x^p rotation can be implemented for n qubits, with an x^5 rotation controlled by 5 qubits as example. For a complete ancilla rotation, multiple x^p rotations are required as detailed earlier in this section. The process for a single x^p rotation consists of three nested steps and a calculation step; (1) the gates are separated by the amount of unique controlling qubits ℓ , (2) per amount of controlling qubits ℓ , each combination \mathcal{C} of said amount of controlling qubits is determined, (3) for these combinations \mathcal{C} , all possible permutations \mathcal{F} of the amount each qubit controls the combination are found, including relative frequency of occurrence \mathcal{F} , (4) these combinations \mathcal{C} and frequencies \mathcal{F} are used to determine the rotation value of each combination of controlling qubits:

1. Cycle over the amount of unique qubits p that control a gate, with $\ell = 1, 2, \dots, p$.

- For example $\ell \in \{1, 2, 3, 4, 5\}$ for $p = 5$.

2. For any given ℓ , find all $\binom{n}{\ell}$ combinations of ℓ out of n qubits. Call these combinations $\mathcal{C}_{\ell,i}(n, p) = \{\mathcal{C}_{\ell,i}^{(0)}(n, p), \mathcal{C}_{\ell,i}^{(1)}(n, p), \dots, \mathcal{C}_{\ell,i}^{(\ell-1)}(n, p)\}$ for $i \in \{1, 2, \dots, \binom{n}{\ell}\}$.

- For the example with $n = 5$ and $p = 5$, consider $\ell = 3$. For this ℓ , there are $\binom{5}{3} = 10$ combinations, namely,

$$\begin{aligned} \mathcal{C}_{3,1}(5, 5) &= \{q_0, q_1, q_2\}, & \mathcal{C}_{3,6}(5, 5) &= \{q_0, q_3, q_4\}, \\ \mathcal{C}_{3,2}(5, 5) &= \{q_0, q_1, q_3\}, & \mathcal{C}_{3,7}(5, 5) &= \{q_1, q_2, q_3\}, \\ \mathcal{C}_{3,3}(5, 5) &= \{q_0, q_1, q_4\}, & \mathcal{C}_{3,8}(5, 5) &= \{q_1, q_2, q_4\}, \\ \mathcal{C}_{3,4}(5, 5) &= \{q_0, q_2, q_3\}, & \mathcal{C}_{3,9}(5, 5) &= \{q_1, q_3, q_4\}, \\ \mathcal{C}_{3,5}(5, 5) &= \{q_0, q_2, q_4\}, & \mathcal{C}_{3,10}(5, 5) &= \{q_2, q_3, q_4\}. \end{aligned}$$

3. Find all $\binom{p-1}{\ell-1}$ permutations of the amount of times each qubit is controlled in each of these combinations, including their frequencies. Name these permutations $\mathcal{P}_{\ell,i,j}(n, p) = \{\mathcal{P}_{\ell,i,j}^{(0)}(n, p), \mathcal{P}_{\ell,i,j}^{(1)}(n, p), \dots, \mathcal{P}_{\ell,i,j}^{(\ell-1)}(n, p)\}$ and frequencies $\mathcal{F}_{\ell,i,j}(n, p)$ for combination $\mathcal{C}_{\ell,i}(n, p)$ and $j \in \{1, 2, \dots, \binom{p-1}{\ell-1}\}$. The permutations need to satisfy two conditions: they must have $\mathcal{P}_{\ell,i,j}^{(m)}(n, p) \in \mathbb{Z}_{\geq 1}$ for all m , and $\sum_{m=0}^{\ell-1} \mathcal{P}_{\ell,i,j}^{(m)}(n, p) = p$. Elementary probability theory shows that the frequency $\mathcal{F}_{\ell,i,j}(n, p)$ is equal to [49],

$$\mathcal{F}_{\ell,i,j}(n, p) = \frac{p!}{\prod_{m=0}^{\ell-1} [\mathcal{P}_{\ell,i,j}^{(m)}(n, p)]!}. \quad (7.13)$$

- For the example of $n = 5$, $p = 5$ and $\ell = 3$ there are $\binom{4}{2} = 6$ permutations. Take $i = 1$, i.e. $\mathcal{C}_{3,1}(5, 5) = \{q_0, q_1, q_2\}$, then the possible permutations with frequencies are

$$\begin{aligned} \mathcal{P}_{3,1,1}(5, 5) &= \{3, 1, 1\}, & \mathcal{F}_{3,1,1}(5, 5) &= 20, & & \sim 20q_0^3q_1q_2 \\ \mathcal{P}_{3,1,2}(5, 5) &= \{1, 3, 1\}, & \mathcal{F}_{3,1,2}(5, 5) &= 20, & & \sim 20q_0q_1^3q_2 \\ \mathcal{P}_{3,1,3}(5, 5) &= \{1, 1, 3\}, & \mathcal{F}_{3,1,3}(5, 5) &= 20, & & \sim 20q_0q_1q_2^3 \\ \mathcal{P}_{3,1,4}(5, 5) &= \{2, 2, 1\}, & \mathcal{F}_{3,1,4}(5, 5) &= 30, & & \sim 30q_0^2q_1^2q_2 \\ \mathcal{P}_{3,1,5}(5, 5) &= \{2, 1, 2\}, & \mathcal{F}_{3,1,5}(5, 5) &= 30, & & \sim 30q_0^2q_1q_2^2 \\ \mathcal{P}_{3,1,6}(5, 5) &= \{1, 2, 2\}, & \mathcal{F}_{3,1,6}(5, 5) &= 30, & & \sim 30q_0q_1^2q_2^2 \end{aligned}$$

4. Sum the values found in the third step to get the angle $\theta_{\ell,i}(n, p)$ of the rotation gate controlled by the combination of qubits $\mathcal{C}_{\ell,i}(n, p)$, $C^{\mathcal{C}_{\ell,i}(n, p)}[R_y(c\theta_{\ell,i}(n, p))]$. Here, it is assumed that the number is written as $q = q_0.q_1q_2 \dots q_{n-1} = q_0 + q_110^{-1} + q_210^{-2} + \dots + q_{n-1}10^{-n+1}$. the value for $\theta_{\ell,i}(n, p)$ in the $C^{\mathcal{C}_{\ell,i}(n, p)}[R_y(c\theta_{\ell,i}(n, p))]$ gate then becomes

$$\theta_{\ell,i}(n, p) = \sum_{j=1}^{\binom{p-1}{\ell-1}} \mathcal{F}_{\ell,i,j}(n, p) \prod_{m=0}^{n-1} 10^{-m \cdot \mathcal{P}_{\ell,i,j}^{(m)}(n, p)}. \quad (7.14)$$

- For the example with $n = 5$, $p = 5$, $\ell = 3$ and $i = 1$ (which corresponds to combination $\{0, 1, 2\}$, i.e. the gate controlled by q_0 , q_1 and q_2), the angle becomes,

$$\begin{aligned}\theta_{3,1}(5, 5) &= 20(10^{-3} + 10^{-5} + 10^{-7}) + 30(10^{-4} + 10^{-5} + 10^{-6}) \\ &= 2 \cdot 10^{-2} + 3 \cdot 10^{-3} + 5 \cdot 10^{-4} + 3 \cdot 10^{-5} + 2 \cdot 10^{-6} \\ &= 0.023532.\end{aligned}$$

Hence, the rotation gate controlled by qubits q_0 , q_1 and q_2 is $C^{q_0q_1q_2} [R_y(0.023532c)]$.

In Step 3, the fact that there are $\binom{p-1}{\ell-1}$ permutations is not completely trivial. The process is equivalent to dividing p balls over ℓ buckets (where “the number of balls in bucket x ” represents “the number of times qubit x occurs in the specific permutation”), with the additional condition that no bucket may remain empty (because an empty bucket would mean that one or more qubits do not occur in the permutation, which would mean that permutations are counted doubly). The additional condition can be met when one ball is added preemptively to each of the ℓ buckets. Hence, there are effectively only $p - \ell$ free balls. The number of ways to put x balls into y buckets is equal to $\binom{x+y-1}{x-1}$ [49]. Taking $x = p - \ell$ and $y = \ell$ shows that there are indeed $\binom{p-1}{\ell-1}$ permutations.

Some remark is in order for small qubit numbers. Consider the implementation of a p -th power rotation on an $n < p$ qubit number. From the explanation above, one would expect to have rotation gates controlled up to p unique qubits. This, however, is impossible with fewer than p qubits; the maximum is n unique qubits. Generally, ℓ only goes up to $\min(n, p)$ instead of up to p . The number of ancilla qubits is therefore only $\max(0, \min(n, p) - 2)$ instead of $\max(0, p - 2)$. For example for $n = 4$, the maximum required number of qubits is only 2 for approximation up to an arbitrarily high order, making it relatively inexpensive to do so in terms of circuit depth.

7.2.2.3 Comparison of the methods

For the higher-order approximations, the same considerations hold as for the third power approximation. The method that explicitly calculates the higher powers retains a circuit depth of $\mathcal{O}(n^2)$, but with a high constant factor. Additionally, although it requires $\mathcal{O}(n)$ qubits, the constant factor is again large; at least $(4k+3)n$ qubits are required. The second method only requires $k - 2$ ancillae for the k -th approximation (up to power x^{2k+1}), but it does scale as $\mathcal{O}(x^{2k+1})$, albeit with small constant factors. This thesis has not gone in depth on what these constant factors exactly are, and the thorough analysis is left open for future research. Due to the high circuit width for the explicit method, the direct rotation method is preferred going forward.

7.2.3 Error analysis

Finally the errors of the higher-order approximations are examined. Instead of the r factor as before, it will now be examined up to what input number for cx the results of the ancilla rotation $\sim \sqrt{1 - c^2x^2} |0\rangle + cx |1\rangle$ stay under a certain error. The errors looked at will be 5%, 1% and 0.1% for k from 0 to 5 (i.e. for the highest power ranging between x and x^{11}). The results are shown in Table 7.2. It might seem strange that some maximum input values exceed $cx = 1$, while the maximum output is equal to one. This is due to the fact that input values cx can be larger than one, and values slightly larger than one are mapped close to it. Hence, for example, for input values up to roughly 1.05 the output error can stay under 5%, yielding the results in the table. The error is monotonous up to $\pi/2 \approx 1.57$, which is greater than the maximum input, so any values smaller than those given in the table will result in a smaller error. For higher-order approximations, however, inputs significantly larger than one will result in rapid oscillations of the output, hence c and x should be chosen such that always $cx \leq 1$.

7.3 QX Implementation

The main challenge in the implementation of the algorithms described in this chapter lies in the construction of the (multiply-)controlled- R_y gates. For the Cao implementation and the Explicit

k	0	1	2	3	4	5
error 0.05	0.5519	0.8900	0.9889	1.0257	1.0414	1.0485
error 0.01	0.2453	0.5996	0.7652	0.8493	0.8974	0.9274
error 0.001	0.0775	0.3389	0.5264	0.6440	0.7208	0.7738

Table 7.2: Values for cx such that the error in $\sim \sqrt{1 - c^2x^2} |0\rangle + cx |1\rangle$ stays under a certain level, approximated up to the $a_{2k+1}x^{2k+1}$ term.

Higher Powers implementation, only singly-controlled- R_y gates are required. These have successfully been implemented by application of the circuit in Figure 1.3 in combination with the realisation that the square root of the $R_y(\theta)$ gate is the $R_y(\theta/2)$ gate. The circuit for the controlled- R_y gate is found in Appendix A on page 82. The decomposition for the doubly-controlled- R_y gate can also be found in that section, which was constructed using the singly-controlled- R_y gate and the circuit shown in Figure 1.4, which shows how to construct a doubly-controlled gate using its singly-controlled-square-root gates. For a general k -fold-controlled- R_y gate the circuit shown in Figure 1.5b can be applied to the doubly-controlled- R_y gate. This final method does require $k-2$ ancillae.

Due to the amount of required qubits, the Explicit Higher Powers method has not been implemented in QX. The Direct Higher Order Rotations method on the other hand has been implemented for any number of qubits and up to any power. The higher power rotations have been implemented as stand alone functions. They have been tested for all four qubit values x , all powers p up to five, a number of different values for c and a multitude of larger x and p . For any input $cx^p \leq \pi/2$ the algorithm performs a correct rotation, with absolute errors never exceeding $5 \cdot 10^{-7}$, which corresponds to the accuracy of the results in the QX simulator. An example of the output of the implementation for $c = 10$, $x = 0.1011_2 = 0.6875$ (implemented as $|01011\rangle$) and $p = 7$ is shown in Figure 7.6a. This implementation requires 8 qubits.

Ry($c*x^p$) rotation:	Ancilla rotation:
input c = 10.0	input c = 1.1
input x = 0.6875	input x = 0.6875
input p = 7	input k = 4
output sin(r) = 0.663849	output sin(r) = 0.755004
calc r = 0.7259537202194326	expectation sin(r) = 0.7550042148744679
test $c*(x^p)$ = 0.7259536907076836	test $c*x$ = 0.7562500000000001
rel error = 4.065238523587311e-08	rel expectation error = 2.8460043640215567e-07
	rel output error = 0.0016476033057852301
(a) Single rotation.	(b) Complete arcsin rotation.

Figure 7.6: Outputs of the Direct Higher Order Rotation Ancilla Rotation Algorithm. (a) Single rotation for $c = 10$, $x = 0.1011_2 = 0.6875$ and $p = 7$. (b) Complete algorithm for $c = 1.1$, $x = 0.1011_2 = 0.6875$ and $k = 4$.

The independent rotations were then used to build the complete ancilla rotation as described in the chapter. As for the single rotations, the algorithm was validated for all four qubit values, all $k \leq 3$ and various c . The outputs never had a relative error larger than 10^{-6} compared to the expectation, so the error analysis from the previous section still holds. As an example consider again $x = 0.1011_2 = 0.6875$, but this time with $c = 1.1$ and $k = 4$, meaning that the approximation is up to x^9 . The results are shown in Figure 7.6b. It is observed that the output value is within the range of the rounding error of the expected value, and the relative error follows the analysis from the previous section.

The Direct Higher Order Rotation Ancilla Rotation Algorithm is therefore a successful method for increasing the accuracy up to arbitrary precision while retaining the highest possible measurement probability.

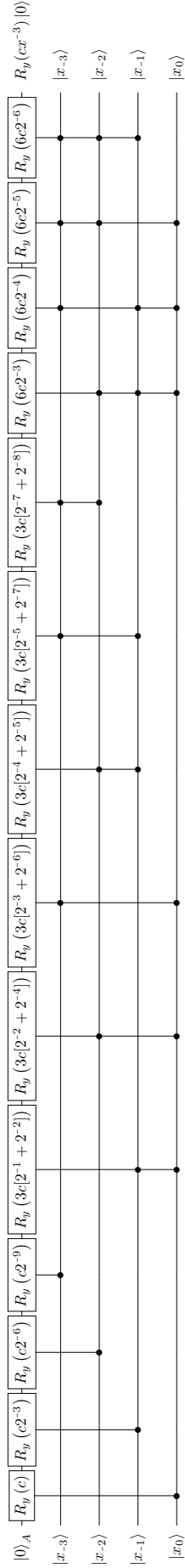


Figure 7.7: Circuit for performing the rotation $R_y(cx^3)$ using x and a predetermined c for an $n = 4$ qubit x , i.e. $x = x_0.x_{-1}x_{-2}x_{-3}$.

8. Quantum Linear Solver Algorithm

In this chapter, practical implementations of the HHL Quantum Linear Solver Algorithm are discussed. First, the proof of concept Quantum Linear Solver presented by Cao et al. in [34] is examined, implemented and tested on the QX simulator. After this, an alternative realisation of the HHL QLSA is developed and implemented from the subroutines discussed in this thesis and using the Hamiltonian simulation algorithm implemented in the first section.

8.1 Baseline circuit by Cao et al.

8.1.1 Overview

In [34], Cao et al. present a proof-of-concept realisation of the HHL QLSA, which can solve a specific system with a 4-by-1 vector \vec{b} and a 4-by-4 matrix A . Specifically, the matrix and vector are:

$$A = \frac{1}{4} \begin{bmatrix} 15 & 9 & 5 & -3 \\ 9 & 15 & 3 & -5 \\ 5 & 3 & 15 & -9 \\ -3 & -5 & -9 & 15 \end{bmatrix}, \quad \vec{b} = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (8.1)$$

The matrix and vector were not chosen randomly, but instead have a number of properties that make it suitable for a proof of concept implementation. The matrix's eigenvalues are all powers of two, namely,

$$\lambda_1 = 1, \quad \lambda_2 = 2, \quad \lambda_3 = 4, \quad \lambda_4 = 8, \quad (8.2)$$

with respective eigenvectors,

$$\vec{v}_1 = \frac{1}{2} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \vec{v}_2 = \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \quad \vec{v}_3 = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \quad \vec{v}_4 = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}. \quad (8.3)$$

Since the eigenvalues are all powers of two, the proof-of-concept eigenvalue inversion circuit of Chapter 6.2.1 can be used. The sizes of the eigenvalues imply that they can be saved in a four qubit register, while the size of A and \vec{b} with $N = 4 = 2^2$ imply that the vectors may be encoded using two qubits, as was shown earlier in this thesis. Specifically, the vectors \vec{b} and eigenvectors \vec{v}_i for $i \in \{1, 2, 3, 4\}$ are encoded as

$$|b\rangle = \frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle), \quad (8.4)$$

$$|v_i\rangle = \frac{1}{2} ((-1)^{\delta_{i1}} |00\rangle + (-1)^{\delta_{i2}} |01\rangle + (-1)^{\delta_{i3}} |10\rangle + (-1)^{\delta_{i4}} |11\rangle), \quad (8.5)$$

where δ_{ij} is the Kronecker delta. The vector $|b\rangle$ is chosen such that it is the equal superposition of the eigenvectors A , i.e.,

$$|b\rangle = \frac{1}{2} (|v_1\rangle + |v_2\rangle + |v_3\rangle + |v_4\rangle). \quad (8.6)$$

In the original explanation of the HHL algorithm, it was shown that \vec{x} may be written as the same linear combination of eigenvectors as \vec{b} , except with all complex amplitudes of the eigenvectors divided by the respective eigenvalues. In the example this becomes,

$$\begin{aligned} |x\rangle &\propto \frac{1}{2} \left(\frac{1}{1} |v_1\rangle + \frac{1}{2} |v_2\rangle + \frac{1}{4} |v_3\rangle + \frac{1}{8} |v_4\rangle \right) \\ &\propto -1 |00\rangle + 7 |01\rangle + 11 |10\rangle + 13 |11\rangle. \end{aligned} \quad (8.7)$$

This final state is proportional to what is expected as output of the implementation of the algorithm.

For the ancilla rotation, a first order approximation of $\arcsin(x) \approx x$ is used. The circuit presented in the original paper [34] is shown in Figure 8.1. In the main circuit (Figure 8.1a), the r in the rotation gates can be chosen freely, as was discussed in the previous chapter. Since $|b\rangle$ is the equal superposition of two qubits, it can be constructed by applying a Hadamard gate on two qubits in the $|0\rangle$ state. In the main circuit, the gates up until the QFT^\dagger subroutine form the eigenvalue estimation. The SWAP gate thereafter is the eigenvalue inversion, and the four R_y gates after that are the ancilla rotation. The uncompute subroutine is meant to reverse the eigenvalue inversion and eigenvalue estimation. The angles in the rotation gates yield a prefactor of $\pi/2^{r+2}$ in Equation (8.7).

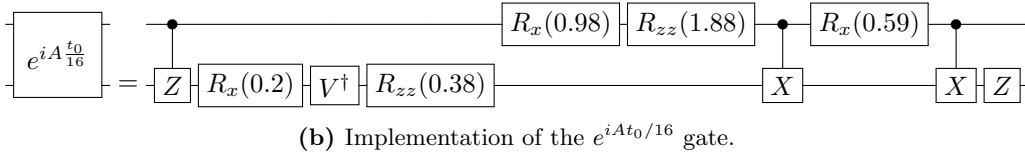
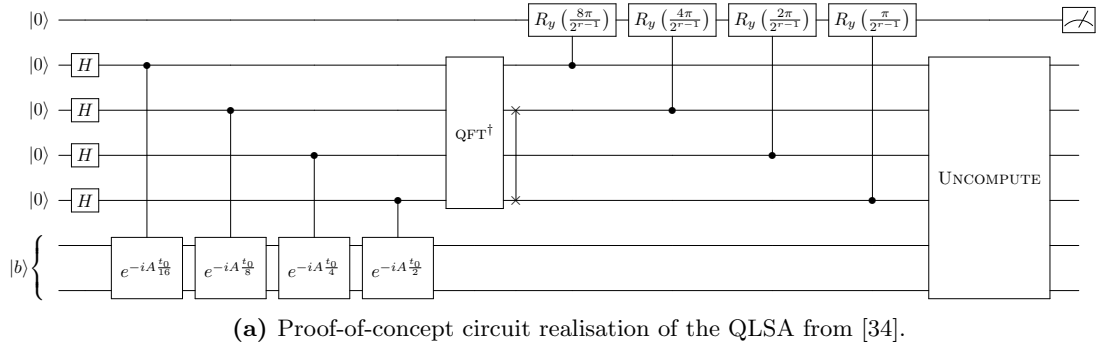


Figure 8.1: Complete proof-of-concept QLS as presented in [34], where $t_0 \equiv 2\pi$.

Figure 8.1b shows the circuit for the $e^{iAt_0/16}$ gate as given in [34]. In the paper, it is explained that for larger angles (i.e. times 2^k), the angles in rotation gates need to be multiplied by the same amount (i.e. also times 2^k). The circuit for $e^{iAt_k/16}$ are constructed in this fashion.

8.1.2 Implementation

The practical implementation of the circuit revealed a couple of subtleties that will be discussed in what follows. They are related to the eigenvalue inversion, with the output of the e^{iAt} gates, with the higher angles in the e^{iAt} gates, and with the rotation of the Fourier transform.

- In the output of the $e^{-iAt_0/16}$ gate, the complex amplitude of the $|10\rangle$ state is negative compared to what it should be, i.e.

$$|v_i\rangle \xrightarrow{e^{-iAt_0/16}} e^{i\varphi_i} \frac{1}{2} \left((-1)^{\delta_{i1}} |00\rangle + (-1)^{\delta_{i2}} |01\rangle - (-1)^{\delta_{i3}} |10\rangle + (-1)^{\delta_{i4}} |11\rangle \right). \quad (8.8)$$

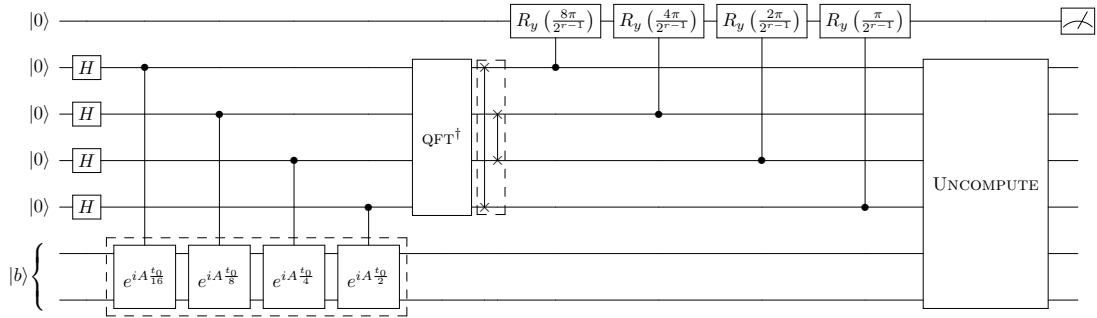
This problem is easily fixed by replacing the last (Z) gate in the circuit depicted in Figure 8.1b by a Z gate controlled by the top qubit.

- For higher times t in the matrix exponential gate, the original paper states that the angles in the rotation gates are multiplied by the same amount (i.e. 2^k). What is not mentioned, however, is the fact that the V^\dagger gate also effectively works as a rotation gate. In order for the higher orders to function properly, the V^\dagger gate should be taken to the power 2^k . This

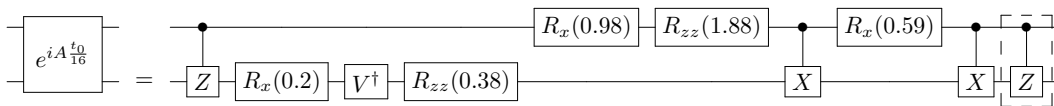
means that the V^\dagger gate should be replaced by $(V^\dagger)^2 = X$ for $k = 1$, and $(V^\dagger)^{2^k} = I$ for $k > 1$.

- The eigenvalues that are obtained after the eigenvalue estimation subroutine are $|1110\rangle$, $|1101\rangle$, $|1011\rangle$ and $|0111\rangle$, instead of the desired $|0001\rangle$, $|0010\rangle$, $|0100\rangle$ and $|1000\rangle$. The found values are exactly the complements of the desired values. This is equivalent to finding the negative versions of the eigenvalues, since $-x$ is mapped to $2^n - 1 - x = 1111_2 - x = \bar{x}$. The negated eigenvalues are not a surprise when comparing the given eigenvalue estimation subroutine to the Eigenvalue Estimation algorithm defined in Figure 5.5: in the latter the exponentials do not have the minus in the exponent. In order to fix this, it is solely needed to remove the minusses in the matrix exponential gates in Figure 8.1a. The minusses are likely to be a typo, given the fact that the Hamiltonian shown in Figure 8.1b does not have it.
- Comparing the eigenvalue inversion subroutine in Figure 8.1a to the eigenvalue inversion circuit in Figure 6.1, shows that these circuits do not match. The subroutine shown in the former figure performs the possible operations: $|1\rangle \rightarrow |4\rangle$ and $|4\rangle \rightarrow |1\rangle$, while $|2\rangle$ and $|8\rangle$ remain the same. This is not an inversion, so the single SWAP is replaced by two SWAP gates: one swapping qubits 2 and 5, and the other swapping qubits 3 and 4, which is an Eigenvalue Inversion subroutine as defined in Figure 8.1a.

Combining these remarks and fixes, the new experimental circuit becomes as is shown in Figure 8.2, where the changed parts are highlighted by dashed lines.



(a) Improved proof-of-concept circuit realisation of the QLSA from [34].



(b) Improved implementation of the $e^{iA t_0/16}$ gate.

Figure 8.2: Improved version of the complete proof-of-concept QLS as shown in Figure 8.1, where $t_0 \equiv 2\pi$. The changes compared to Figure 8.1 are encircled with dashed lines.

The $R_{zz}(\theta)$ and $R_z(\theta)$ gates are defined as follows in [34],

$$R_z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, \quad R_{zz}(\theta) = \begin{bmatrix} e^{i\theta} & 0 \\ 0 & e^{i\theta} \end{bmatrix}. \quad (8.9)$$

It was already mentioned that this definition of $R_z(\theta)$ is equivalent to the more common definition introduced in Chapter 1, except for a negligible global phase shift of $e^{i\theta/2}$. For the $R_{zz}(\theta)$ gate there is no equivalent in the QX simulator. That is not a problem though, since it will be used as $C(R_{zz}(\theta))$, which is equivalent to performing $R_z(\theta)$ on the controlling qubit. The $C(e^{iA 2^k t_0/16})$ gates are therefore implemented as shown in Figure 8.3.

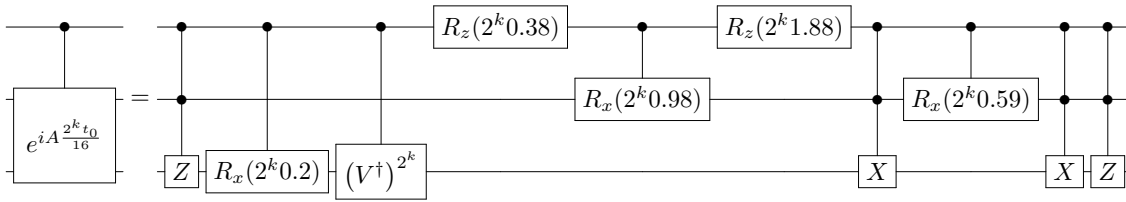


Figure 8.3: Implementation for the controlled $e^{iA2^k t_0/16}$ gates, for $k \in \mathbb{Z}_{\geq 0}$.

Multiple of the controlled gates used in Figure 8.3 are not available in QX. These can be emulated, however, using gates available in QX. Appendix A on page 82 shows the respective circuits.

For the uncomputation subroutine, the inverses of the $e^{iA2^k t_0/16}$ gates are necessary, which are $e^{-iA2^k t_0/16}$ gates. Their implementation is trivial; the order of all gates is reversed, and all angles are replaced by their negatives, and the V^\dagger gate is replaced by a V gate. Note that still $V^2 = X$ and $V^{2^k} = I$ for $k \in \mathbb{Z}_{\geq 2}$. The circuit is shown in Figure 8.4.

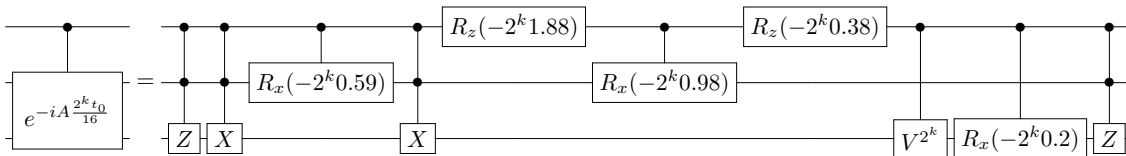


Figure 8.4: Implementation for the controlled $e^{-iA2^k t_0/16}$ gates, for $k \in \mathbb{Z}_{\geq 0}$.

Finally, through a simple brute force fivefold for-loop with stepsize 0.001, an improvement to the rotation angles was found. From the first rotation gate to the last rotation gate, the improved rotation angles are as follows,

$$0.20 \rightarrow 0.196; \quad 0.38 \rightarrow 0.375; \quad 0.98 \rightarrow 0.982; \quad 1.88 \rightarrow 1.883; \quad 0.59 \rightarrow 0.589. \quad (8.10)$$

These improved angles were sought for to exclude any major errors introduced by the Hamiltonian simulation. The angles will be compared to the original angles in the next section.

8.1.3 Results

Firstly the adapted Hamiltonian simulation as defined in Figure 8.3 is examined, for both the original angles and improved angles as defined in Equation (8.10). The Hamiltonians e^{iAt_k} have been tested on the eigenvectors $|v_j\rangle$ of A , which should yield the output state $e^{i2\pi 2^k \lambda_j} = e^{i2\pi 2^{j+k}}$ for $j, k = 0, 1, 2, 3$. This is a complex rotation of $22.5 \cdot 2^{j+k}$ degrees. The results are shown in degrees, as this means that the ideal outputs are round numbers and therefore easy to compare. Tests of the Hamiltonian simulations for the different eigenvectors have shown that for every possible input and for both the original and new angles, the rotations of all four qubit states are identical. Hence, only the total angle of each input has to be examined. The ideal results, and the results using both the original and improved angles are shown in Table 8.1a, Table 8.1b and Table 8.1c, respectively. The absolute errors for both the original and improved angles are shown in Table 8.2a and Table 8.2b, respectively.

Table 8.2 shows that the errors for larger rotations k scale as 2^k , which is equal to the increased factor in total rotation, as expected. The maximum error for the original angles is a rotation of 1.47° , or 0.41% of the total unit circle. This compared to the maximum error of 0.14° , or 0.039% of the total unit circle, for the improved angles. It is concluded that the improvements to the angles have been successful, with a more than tenfold increase of accuracy over the original angles. It is however the relative error in the rotations that determines the possible error after the Inverse Quantum Fourier Transform in the outcome of the Eigenvalue Estimation subroutine. These relative errors are the same for different values of k , so only the relative errors for $k = 0$ are observed,

$k \setminus j$	0	1	2	3	$k \setminus j$	0	1	2	3	$k \setminus j$	0	1	2	3
0	22.5	45	90	180	0	22.5289	44.8743	90.1836	-179.8620	0	22.5003	45.0175	89.9830	179.9947
1	45	90	180	0	1	45.0579	89.7486	-179.6329	0.2758	1	45.0006	90.0351	179.9661	-0.0107
2	90	180	0	0	2	90.1156	179.4971	0.7342	0.5518	2	90.0010	-179.9100	-0.0680	-0.0212
3	180	0	0	0	3	-179.7688	-1.0059	1.4684	1.1035	3	-179.9979	0.1400	-0.13580	-0.0425

(a) Ideal rotations

(b) Original rotations

(c) Improved rotations

Table 8.1: Rotations in degrees of the Hamiltonian simulation using the Cao matrix for time $t_k = t_0 2^k = 2\pi \cdot 2^k$ and eigenvector v_j as defined in Equation (8.5). (a) Ideal rotations, (b) Rotations using the circuit shown in Figure 8.3, with original angles, (c) Rotations using the circuit shown in Figure 8.3, with the improved angles shown in Equation (8.10).

$k \setminus j$	0	1	2	3	$k \setminus j$	0	1	2	3
0	$2.89 \cdot 10^{-2}$	$1.26 \cdot 10^{-1}$	$1.85 \cdot 10^{-1}$	$1.38 \cdot 10^{-1}$	0	$2.75 \cdot 10^{-4}$	$1.75 \cdot 10^{-2}$	$1.70 \cdot 10^{-2}$	$5.27 \cdot 10^{-3}$
1	$5.78 \cdot 10^{-2}$	$2.51 \cdot 10^{-1}$	$3.67 \cdot 10^{-1}$	$2.76 \cdot 10^{-1}$	1	$5.67 \cdot 10^{-4}$	$3.51 \cdot 10^{-2}$	$3.39 \cdot 10^{-2}$	$1.07 \cdot 10^{-2}$
2	$1.16 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$7.34 \cdot 10^{-1}$	$5.52 \cdot 10^{-1}$	2	$1.03 \cdot 10^{-3}$	$7.00 \cdot 10^{-2}$	$6.79 \cdot 10^{-2}$	$2.12 \cdot 10^{-2}$
3	$2.31 \cdot 10^{-1}$	$1.01 \cdot 10^0$	$1.47 \cdot 10^0$	$1.10 \cdot 10^0$	3	$2.06 \cdot 10^{-3}$	$1.40 \cdot 10^{-1}$	$1.36 \cdot 10^{-1}$	$4.25 \cdot 10^{-2}$

(a) Absolute errors

(b) Relative errors

Table 8.2: Absolute errors in degrees of the rotations of the Hamiltonian simulation using the Cao matrix shown in Table 8.1. (a) Errors using the original angles, (b) errors using the improved angles.

which are shown in Table 8.3.

j	0	1	2	3
$\epsilon_{\text{rel, original}}^{(j)}$	$1.28 \cdot 10^{-3}$	$2.80 \cdot 10^{-3}$	$2.04 \cdot 10^{-3}$	$7.67 \cdot 10^{-4}$
$\epsilon_{\text{rel, improved}}^{(j)}$	$1.22 \cdot 10^{-5}$	$3.89 \cdot 10^{-4}$	$1.88 \cdot 10^{-4}$	$2.93 \cdot 10^{-5}$

Table 8.3: Relative errors of the rotations of the Hamiltonian simulation using the Cao matrix shown in Table 8.1 of both the original and improved angles, for $k = 0$.

It is observed that an improvement in errors of more than a factor of seven has been achieved, with a decrease in maximum relative error from $2.80 \cdot 10^{-3}$ to $3.89 \cdot 10^{-4}$. All further tests and implementations have therefore solely been performed using the improved angles. Although it was not thoroughly researched what the precise impact of the relative errors is on the output of the Eigenvalue Estimation subroutine, it is known that these errors result in non-zero amplitudes for incorrect estimates of eigenvalues. The probabilities of these undesirable states, however, scale as the square of the sine of the error, which is a function that can be approximated by x^2 for small values. Hence, the resulting error is unlikely to be significant. Initial testing showed that the errors are indeed insignificant. This will later implicitly be confirmed in the results of the full QLSA, since the errors keep decreasing for higher accuracy of the Ancilla Rotation.

The complete QLS circuit shown in Figure 8.2 was implemented and ran in the QX simulator for different values of r . An example of the output for $r = 5$ is shown in Figure 8.5, giving the final real and complex amplitude of each state. The given numbers are the decimal values that the state would have when viewing the ancilla qubit as the most significant qubit. The outputs are near purely real, and only states $|0_{10}\rangle \equiv |000000\rangle$ through $|3_{10}\rangle \equiv |0000011\rangle$ and $|6_{4_{10}}\rangle \equiv |1000000\rangle$ through $|6_{7_{10}}\rangle \equiv |1000011\rangle$ have significant probabilities, with no other state exceeding a probability of 10^{-8} . This is precisely as desired. The second four non-zero states are the interesting ones, as these are the desired outcome of the HHL algorithm $|x\rangle$ as defined in Equation (8.7).

The accuracy of the output of the $|x\rangle$ state for different values of r is now examined. The values of the four amplitudes in the $|x\rangle$ state for different r are shown in Table 8.4a. Since the amplitudes decrease for higher r , it is instructive to look at their relative sizes, as that is what remains after normalisation. The relative values, which are obtained by defining $x_{\text{rel},00} \equiv -1$, are therefore shown in Table 8.4b. For larger values of r , especially for $r \geq 7$, it is observed that the answer

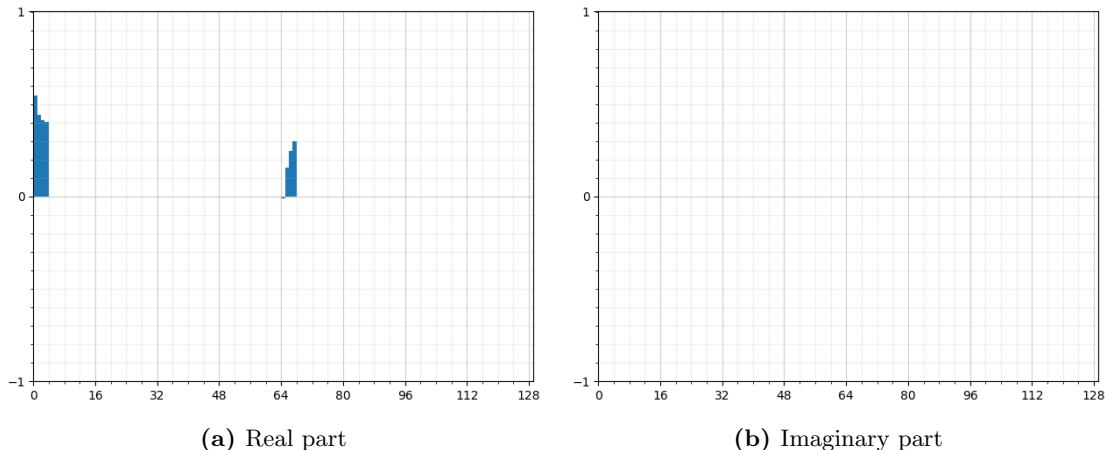


Figure 8.5: Output state of the circuit adapted from [34], using $r = 5$. The horizontal axis is the state, transformed from a binary state to a decimal number, with the ancilla state as most significant qubit. The vertical axis is the real or imaginary part of the complex amplitude of each state. The non-zero states correspond to $|0000000\rangle \equiv |0\rangle$ through $|0000011\rangle \equiv |3\rangle$ and $|1000000\rangle \equiv |64\rangle$ through $|1000011\rangle \equiv |67\rangle$. The real amplitudes of the first four non-zero states are $[0.548184, 0.439798, 0.411345, 0.404145]$, and the real amplitudes of the second four states are $[-0.007829, 0.154382, 0.248179, 0.296716]$.

indeed approaches the correct answer of $|x\rangle \propto -1|00\rangle + 7|01\rangle + 11|10\rangle + 13|11\rangle$. For lower values of r , especially for $r \leq 4$, the answer does not resemble the desired output at all. This is not entirely unexpected, as for $r \leq 4$ the rotations in the Ancilla Rotation subroutine can exceed $\pi/2$, which results in a non-uniform increase of the amplitude of the $|1\rangle_A$ state, causing a completely unrecognisable output.

r	x_{00}	x_{01}	x_{10}	x_{11}	r	$x_{\text{rel},00}$	$x_{\text{rel},01}$	$x_{\text{rel},10}$	$x_{\text{rel},11}$
3	0.52245	0.02245	0.16889	0.33110	3	-1.00000	-0.04297	-0.32328	-0.63376
4	0.07122	0.21767	0.37988	0.47367	4	-1.00000	-3.05625	-5.33385	-6.65086
5	-0.00783	0.15438	0.24818	0.29672	5	-1.00000	19.71925	31.69996	37.89960
6	-0.01013	0.08367	0.13221	0.15668	6	-1.00000	8.26197	13.05480	15.47161
7	-0.00614	0.04295	0.06750	0.07977	7	-1.00000	7.27412	11.44647	13.53699
8	-0.00303	0.02144	0.03370	0.03984	8	-1.00000	7.06691	11.10877	13.13085
9	-0.00153	0.01073	0.01687	0.01994	9	-1.00000	7.01503	11.02484	13.03007

Table 8.4: Absolute and relative outputs of the real parts of the $|x\rangle$ state for different r .

Next, the probability to measure $|1\rangle_A$ is examined. This probability is equal to the sum of the absolute square of all complex amplitudes containing $|1\rangle_A$. Due to the many zero states, the probability can be rewritten to $\mathbb{P}(|1\rangle_A) = |x_{00}|^2 + |x_{01}|^2 + |x_{10}|^2 + |x_{11}|^2$. Since all x_{ij} are dependent on the rotation proportional to 2^{-r} , it is seen that the probability scales as $\mathbb{P}(|1\rangle_A) \propto 2^{-2r} = 4^{-r}$ for larger r . The results for the different values of r are shown in Table 8.5, which support the calculation. For values of r higher than 9, the probability to measure $|1\rangle_A$ becomes so small that the values are not included. Already for $r = 9$ on average over 1000 runs are required for a single positive outcome.

r	3	4	5	6	7	8	9
$\mathbb{P}(1\rangle_A)$	$4.12 \cdot 10^{-1}$	$4.21 \cdot 10^{-1}$	$1.74 \cdot 10^{-1}$	$4.91 \cdot 10^{-2}$	$1.27 \cdot 10^{-2}$	$3.19 \cdot 10^{-3}$	$8.00 \cdot 10^{-4}$

Table 8.5: Probability to measure the desired state $|1\rangle_A$ for different r .

Finally, the absolute and relative errors in the final answer are reviewed. The answer for some value of r should ideally be $|x\rangle = \frac{\pi}{4 \cdot 2^r} (-1|00\rangle + 7|01\rangle + 11|10\rangle + 13|11\rangle)$ for a perfect Ancilla Rotation subroutine. This ideal output is what the earlier results found are compared to in Table 8.4. The absolute errors are displayed in Table 8.6a, and the relative errors in Table 8.6b. Both the absolute and relative errors decrease for larger values of r with a factor of roughly $8 = 2^3$ for each increment of r . This is in line with the expectation, as the error in the approximation of $\arcsin(x)$ has an order of $\mathcal{O}(x^3)$, and for each increment of r the rotation is halved. Although the absolute errors in the first term are the smallest, they are the largest relatively by a margin. This is due to the fact that x_{00} is roughly ten times smaller than the other terms.

r	ϵ_{00}	ϵ_{01}	ϵ_{10}	ϵ_{11}	r	$\epsilon_{\text{rel},00}$	$\epsilon_{\text{rel},01}$	$\epsilon_{\text{rel},10}$	$\epsilon_{\text{rel},11}$
3	0.62062	0.66478	0.91103	0.94517	3	6.32158	0.96734	0.84361	0.74057
4	0.12031	0.12595	0.16008	0.16446	4	2.45088	0.36653	0.29647	0.25772
5	0.01671	0.01742	0.02180	0.02235	5	0.68102	0.10142	0.08075	0.07005
6	0.00214	0.00223	0.00278	0.00285	6	0.17478	0.02601	0.02063	0.01788
7	0.00027	0.00028	0.00035	0.00036	7	0.04399	0.00655	0.00519	0.00450
8	0.00003	0.00003	0.00004	0.00004	8	0.01107	0.00162	0.00129	0.00112
9	0.000004	0.000005	0.000006	0.000006	9	0.00260	0.00045	0.00034	0.00029

(a) Absolute errors

(b) Relative errors

Table 8.6: Absolute and relative errors in the outputs of the real parts of the $|x\rangle$ state for different r .

Since the results asymptotically approach the actual answer with the expected orders, it is concluded that all parts in the circuit work properly.

8.2 General circuit using Cao matrix

In this thesis, general methods for Eigenvalue Inversion and higher-order methods for Ancilla Rotation have been discussed. Only an implementation for the process of transforming a matrix into the circuit for its Hamiltonian, and for implementing a general vector are lacking. In order to build a proof-of-concept complete Quantum Linear Solver, the Hamiltonian and vector from the previous section are used, including the adapted circuits for implementing the controlled- $\exp(iAt)$ gates. For Eigenvalue Inversion, the Thapliyal Inversion introduced in Chapter 6.2.3 is used, and for the Higher-Order Ancilla Rotations the Direct Higher-Order Rotation method introduced in Chapter 7.2.2.2 is used.

Define the registers V , E , Q , R , B and A , for respectively: the vector $|b\rangle$; the eigenvalues; the inverted eigenvalues; the remainder in the Thapliyal inversion; the ancillae for the higher order ancilla rotation; and the rotation ancilla. Due to the sizes of the known vector and eigenvalues, registers V and E require at least 2 and 4 qubits, respectively. The Thapliyal inversion algorithm requires the eigenvalues to have a $|0\rangle$ as most significant qubit, meaning that in practice register E requires 5 qubits. The registers Q and R for the Thapliyal inversion require at least as many qubits as the register being inverted. Taking this minimum results in register sizes of 5 qubits for both Q and R . The number of ancillae in the higher order ancilla rotation algorithm is bound by size of the register containing the rotation angle (i.e. Q). The maximum size of register B therefore becomes $5 - 2 = 3$ qubits. Finally, register A contains only one qubit. The total number of qubits required is therefore equal to $2 + 5 + 5 + 5 + 3 + 1 = 21$, and the complete circuit for an $k = 2$ Ancilla Rotation is shown in Figure 8.6.

The algorithm was performed for different values of k ranging from 0 to 9, i.e. for different orders of approximation for the ancilla rotation up to x^{19} . The value for r was kept constant throughout the runs, at $r = 5$. This value of r was specifically chosen, as it is the lowest value of r for which the answer can get arbitrarily close to the correct answer, while the effects of the higher-order approximations are especially visible. The results for the different k are shown in Table 8.7. Note

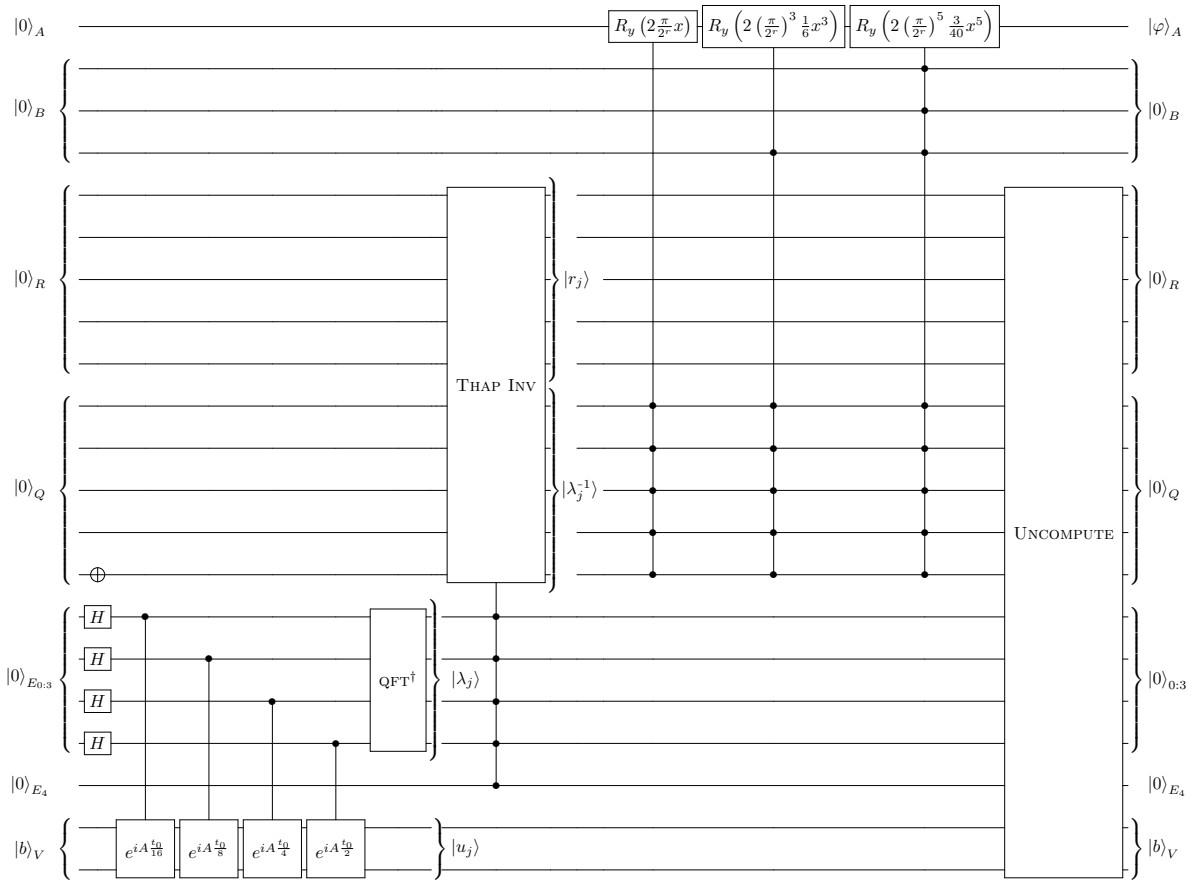


Figure 8.6: Quantum Linear Solver circuit using Cao matrix and vector, for $k = 2$.

that for $k = 0$, the circuit is effectively identical to that of the previous section. As expected, the result is also identical. For higher approximation orders, the results start to resemble the desired result more closely, again as desired.

k	x_{00}	x_{01}	x_{10}	x_{11}	k	$x_{\text{rel},00}$	$x_{\text{rel},01}$	$x_{\text{rel},10}$	$x_{\text{rel},11}$
0	-0.00783	0.15438	0.24818	0.29672	0	-1.00000	19.71925	31.69996	37.89948
1	-0.01884	0.16609	0.26392	0.31299	1	-1.00000	8.81706	14.01056	16.61592
2	-0.02225	0.16951	0.26765	0.31674	2	-1.00000	7.61822	12.02890	14.23496
3	-0.02353	0.17079	0.26896	0.31805	3	-1.00000	7.25928	11.43197	13.51838
4	-0.02406	0.17133	0.26950	0.31859	4	-1.00000	7.11956	11.19926	13.23915
5	-0.02431	0.17157	0.26974	0.31883	5	-1.00000	7.05842	11.09734	13.11680
6	-0.02442	0.17169	0.26986	0.31895	6	-1.00000	7.02964	11.04942	13.05929
7	-0.02448	0.17174	0.26992	0.31900	7	-1.00000	7.01536	11.02557	13.03072
8	-0.02451	0.17177	0.26995	0.31903	8	-1.00000	7.00796	11.01330	13.01595
9	-0.02453	0.17179	0.26996	0.31905	9	-1.00000	7.00432	11.00718	13.00860

(a) Absolute outputs
(b) Relative outputs

Table 8.7: Absolute and relative outputs of the real parts of the $|x\rangle$ state for different k and $r = 5$.

The probability to measure $|1\rangle_A$ will converge to a specific value. This probability is $\mathbb{P}(|1\rangle_A) = \frac{|x_{00}|^2 + |x_{01}|^2 + |x_{10}|^2 + |x_{11}|^2}{4 \cdot 2^5}$ for $|x\rangle = \frac{\pi}{4 \cdot 2^5} (-1|00\rangle + 7|01\rangle + 11|10\rangle + 13|11\rangle)$, so

$$\lim_{k \rightarrow \infty} \mathbb{P}(|1\rangle_A) = \left(\frac{\pi}{2^7}\right)^2 (|-1|^2 + |7|^2 + |11|^2 + |13|^2) \approx 0.2048. \quad (8.11)$$

Due to the monotonically increasing approximation of $\arcsin(x)$, the probability will be increasing to this limit. This is confirmed by the actual probabilities shown in Table 8.8.

k	0	1	2	3	4	5	6	7	8	9
$\mathbb{P}(1\rangle_A)$	0.1735	0.1956	0.2012	0.2032	0.2041	0.2044	0.2046	0.2047	0.2048	0.2048

Table 8.8: Probability to measure the desired state $|1\rangle_A$ for different k and $r = 5$.

Finally, the errors will again be investigated. The errors are depicted in Table 8.9. Note that they are decreasing for higher k as expected. Unlike with the first-order approximation, there is no such simple connection between the errors. It is, however, clear that the errors are rapidly decreasing for higher r . For example for $k = 7$ (and $r = 5$), the relative error is smaller than the relative error for $r = 9$ (and $k = 0$) in Table 8.6b, while the probability to measure the $|1\rangle_A$ state is several orders of magnitude larger. This difference in probability is roughly $2^{2(r_2 - r_1)} \approx 2.5 \cdot 10^2$.

The implementation is therefore successful in its goals of obtaining up to arbitrary precision in results while retaining the highest possible measurement results. Hence, it is concluded that in this thesis, the implementation of a general Quantum Linear Solver is implemented, on the condition that the vector and Hamiltonian implementations are provided.

k	ϵ_{00}	ϵ_{01}	ϵ_{10}	ϵ_{11}	k	$\epsilon_{\text{rel},00}$	$\epsilon_{\text{rel},01}$	$\epsilon_{\text{rel},10}$	$\epsilon_{\text{rel},11}$
0	0.01671	0.01742	0.02180	0.02235	0	0.68102	0.10142	0.08075	0.07006
1	0.00571	0.00572	0.00606	0.00607	1	0.23251	0.03329	0.02246	0.01904
2	0.00229	0.00229	0.00233	0.00233	2	0.09341	0.01335	0.00861	0.00729
3	0.00102	0.00102	0.00102	0.00102	3	0.04142	0.00592	0.00378	0.00320
4	0.00048	0.00048	0.00048	0.00048	4	0.01954	0.00280	0.00178	0.00151
5	0.00024	0.00024	0.00024	0.00024	5	0.00964	0.00138	0.00088	0.00075
6	0.00012	0.00012	0.00012	0.00012	6	0.00492	0.00070	0.00045	0.00038
7	0.00006	0.00006	0.00006	0.00006	7	0.00255	0.00037	0.00024	0.00020
8	0.00003	0.00003	0.00003	0.00003	8	0.00133	0.00020	0.00012	0.00011
9	0.00002	0.00002	0.00002	0.00002	9	0.00072	0.00010	0.00007	0.00006

(a) Absolute errors

(b) Relative errors

Table 8.9: Absolute and relative errors in the outputs of the real parts of the $|x\rangle$ state for different k and $r = 5$.

9. Conclusions and Future Work

In this thesis, the *HHL Quantum Linear Solver Algorithm* has been analysed, and a possible realisation of the QLSA has been implemented and tested on the QX Quantum Computer Simulator. The subroutines that have been implemented are the Eigenvalue Inversion subroutine and the Ancilla Rotation subroutine. Those subroutines have been combined to form a Quantum Linear Solver that can solve any matrix and vector, with the condition that the implementations for the vector and Hamiltonian of the matrix are provided. A specific Hamiltonian and vector implementation have been used to validate the functionality of the solver.

For the *Eigenvalue Inversion subroutine* four solutions have been analysed, which are in order the Powers-of-Two Inverter, the Cao Inverter, the Newton-Raphson Inverter and the Thapliyal Inverter. The Powers-of-Two Inverter works as advertised, which is only for powers of two. The Cao Inverter was implemented in the QX simulator, but did not yield the expected results. The origin of the erroneous results has not conclusively been determined. The Newton-Raphson Inverter has been implemented for the first two orders, and functions as expected. At a circuit width of at least $9n$ qubits for higher orders, the algorithm requires too many qubits to implement higher orders, and hence, yields inaccurate results. The Thapliyal Inverter has been implemented successfully and has been expanded to yield arbitrary precision when desired. This and its low number of required ancillae at $n+1$ ancillae result in the conclusion that it is the preferred Eigenvalue Inversion subroutine out of the four considered in this thesis.

For the *Ancilla Rotation subroutine*, the externally proposed first-order circuit by Cao et al. was successfully implemented in the QX simulator. Two extensions yielding up to arbitrary precision while retaining constant probability of success have been proposed and analysed, specifically the Explicit Higher-Order Value Ancilla Rotation Algorithm and the Direct Higher-Order Rotation Ancilla Rotation Algorithm. The first algorithm has low asymptotic scaling in circuit depth at $\mathcal{O}(n^2)$, but has high constant factors, and at a circuit width of over $(4k+3)n$ qubits it requires too many qubits to be implemented at the current state of technology. The second algorithm has higher order asymptotic scaling of circuit depth at $\mathcal{O}(n^{2k+1})$, but low constant factors and requires only $2k-1$ ancillae. This makes it more suitable for implementation on current and mid-future quantum computers. At this stage, the Direct Higher-Order Rotation algorithm is therefore the preferred higher order approximation algorithm.

An externally proposed proof-of-concept Quantum Linear Solver by Cao et al. has been implemented and thereby adapted to successfully function in the QX simulator. Its specific Hamiltonian and vector implementation in combination with the Thapliyal Inversion and Direct Higher-Order Ancilla Rotation subroutines have been used to build a complete prototype Quantum Linear Solver using general circuitry to invert and rotate over the eigenvalues. The correct functioning of the algorithm has been demonstrated for a particular choice of input data, leading to the conclusion that in this thesis a complete Quantum Linear Solver has successfully been implemented, on the condition that the implementations for the vector and Hamiltonian of the matrix have to be provided by the user.

In the process of implementing the different subroutines, a number of useful by-products have been developed and implemented. A number of gates that are not natively available in the QX simulator have been implemented as circuits, especially a number of (doubly-)controlled versions of basic gates. A full framework for basic Quantum Arithmetic has been implemented, with three Adders, two Subtractors, a Multiplier and an Integer Divider successfully implemented in the QX simulator. A number of improvements and expansions were made over algorithms and implementations published in the literature, allowing them to function in the QX simulator.

The work of this thesis suggests different directions for future research. Firstly, the main open

question that remains in the implementation of a general purpose Quantum Linear Solver, lies in the implementation of the vector and the Hamiltonian of the matrix. A couple of papers on this subject have been published in literature, and creating a systematic overview of the available algorithms is advised. Secondly, a number of subroutines in this thesis have not been thoroughly analysed with regard to circuit width and depth, specifically the Thaliyal Inversion and Higher-Order Ancilla Rotation subroutines. In-depth research of those aspects will yield a better overview of the system requirements for a general-purpose Quantum Linear Solver. Thirdly and finally, the degree of error proneness of any of the subroutines has not at all been considered in this thesis. For an eventual physical implementation of the algorithm this would be essential in order to predict its functioning. With these three areas of further research combined, the HHL Quantum Linear Solver Algorithm may be ready for implementation on physical Quantum Computers and help pave the road for a faster Quantum future.

List of references

- [1] J. Preskill, “Quantum Computing in the NISQ era and beyond,” [arXiv:1801.00862](https://arxiv.org/abs/1801.00862).
- [2] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, “QX: A high-performance quantum computer simulation platform,” in *Proceedings of the 2017 Design, Automation and Test in Europe*. 2017.
- [3] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
- [4] T. Kadowaki and H. Nishimori, “Quantum annealing in the transverse Ising model,” *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics* **58** no. 5, (1998) 5355–5363.
- [5] P. A. M. Dirac, “A new notation for quantum mechanics,” *Mathematical Proceedings of the Cambridge Philosophical Society* **35** no. 3, (Jul, 1939) 416–418.
- [6] C. Dickel, “Programming for the Quantum Computer,” 2016. <https://blog.qutech.nl/index.php/2016/11/03/programming-for-the-quantum-computer/>.
- [7] D. J. Griffiths, *Introduction to Quantum Mechanics*. Cambridge University Press, 2nd ed., 2017.
- [8] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels, “cQASM v1.0: Towards a Common Quantum Assembly Language,” [arXiv:1805.09607](https://arxiv.org/abs/1805.09607).
- [9] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, “A new quantum ripple-carry addition circuit,” [arXiv:0410184](https://arxiv.org/abs/0410184) [quant-ph].
- [10] G. Van Rossum and F. L. Drake Jr, *Python tutorial*. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.
- [11] L. K. Grover, “A fast quantum mechanical algorithm for database search,” [arXiv:9605043](https://arxiv.org/abs/9605043) [quant-ph].
- [12] P. W. Shor, “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer,” [arXiv:9508027](https://arxiv.org/abs/9508027) [quant-ph].
- [13] M. Mosca, *Quantum Computer Algorithms*. PhD thesis, University of Oxford, 1999. <http://www.karlin.mff.cuni.cz/~holub/soubory/moscathesis.pdfpapers2://publication/uuid/8D7E986E-95EE-4A2C-8B47-AEFE2698D295>.
- [14] E. O. Brigham, *The Fast Fourier Transform and its Applications*. Prentice Hall, Englewood Cliffs, N.J., 1988.
- [15] M. van der Lans, *Quantum Algorithms and their Implementation on Quantum Computer Simulators*. BSc Thesis, Delft University of Technology, Jun, 2017.
- [16] T. G. Draper, “Addition on a Quantum Computer,” [arXiv:0008033](https://arxiv.org/abs/0008033) [quant-ph].
- [17] E. Muñoz-Coreas and H. Thapliyal, “T-count Optimized Design of Quantum Integer Multiplication,” [arXiv:1706.05113](https://arxiv.org/abs/1706.05113).
- [18] H. Thapliyal, T. S. S. Varun, and E. Munoz-Coreas, “Quantum Circuit Design of Integer Division Optimizing Ancillary Qubits and T-Count,” [arXiv:1609.01241](https://arxiv.org/abs/1609.01241).
- [19] H. Thapliyal, E. Muñoz-Coreas, T. S. S. Varun, and T. S. Humble, “Quantum Circuit Designs of Integer Division Optimizing T-count and T-depth,” [arXiv:1809.09732](https://arxiv.org/abs/1809.09732).
- [20] V. Vedral, A. Barenco, and A. Ekert, “Quantum Networks for Elementary Arithmetic Operations,” [arXiv:9511018](https://arxiv.org/abs/9511018) [quant-ph].

- [21] A. Q. Nguyen, “TR-2004007: Optimal Reversible Quantum Circuit for Multiplication,” tech. rep., City University of New York (CUNY), 2004. https://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1245&context=gc{}_cs{}_tr.
- [22] L. A. B. Kowada, R. Portugal, and C. M. H. De Figueiredo, “Reversible Karatsuba’s Algorithm,” *Journal Of Universal Computer Science* **12** no. 5, (2006) 499–511.
- [23] A. Parent, M. Roetteler, and M. Mosca, “Improved reversible and quantum circuits for Karatsuba-based integer multiplication,” [arXiv:1706.03419](https://arxiv.org/abs/1706.03419).
- [24] H. V. Jayashree, H. Thapliyal, H. R. Arabnia, and V. K. Agrawal, “Ancilla-Input and Garbage-Output Optimized Design of a Reversible Quantum Integer Multiplier,” [arXiv:1608.01228](https://arxiv.org/abs/1608.01228).
- [25] M. Flynn and S. Oberman, “Division,” in *Advanced Computer Arithmetic Design*, ch. 5, pp. 129–143. J Wiley, Stanford, 2001. <https://web.stanford.edu/class/ee486/doc/chap5.pdf>.
- [26] D. C. Lay, S. R. Lay, and J. J. McDonald, *Linear Algebra and Its Applications*. Pearson Education, College Park, 5 ed., 2016.
- [27] C. Vuik, F. Vermolen, M. van Gijzen, and M. Vuik, *Numerical Methods for Ordinary Differential Equations*. Delft Academic Press, Delft, The Netherlands, 2nd ed., 2016.
- [28] D. W. Berry, A. M. Childs, and R. Kothari, “Hamiltonian Simulation with Nearly Optimal Dependence on all Parameters,” *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS* (Jan, 2015) 792–809, [arXiv:1501.01715](https://arxiv.org/abs/1501.01715).
- [29] X. G. Fang and G. Havas, “On the worst-case complexity of integer Gaussian elimination,”.
- [30] A. W. Harrow, A. Hassidim, and S. Lloyd, “Quantum algorithm for linear systems of equations,” *Physical Review Letters* **103** no. 15, (Nov, 2009) 1–15, [arXiv:0811.3171](https://arxiv.org/abs/0811.3171).
- [31] A. Ambainis, “Variable time amplitude amplification and a faster quantum algorithm for solving systems of linear equations,” [arXiv:1010.4458](https://arxiv.org/abs/1010.4458).
- [32] D. W. Berry, A. M. Childs, R. Cleve, R. Kothari, and R. D. Somma, “Exponential improvement in precision for simulating sparse Hamiltonians,” [arXiv:1312.1414](https://arxiv.org/abs/1312.1414).
- [33] A. M. Childs, R. Kothari, and R. D. Somma, “Quantum algorithm for systems of linear equations with exponentially improved dependence on precision,” [arXiv:1511.02306](https://arxiv.org/abs/1511.02306).
- [34] Y. Cao, A. Daskin, S. Frankel, and S. Kais, “Quantum circuit design for solving linear systems of equations,” *Molecular Physics* **110** no. 15-16, (Oct, 2012) 1675–1680, [arXiv:1110.2232](https://arxiv.org/abs/1110.2232).
- [35] J. Pan, Y. Cao, X. Yao, Z. Li, C. Ju, X. Peng, S. Kais, and J. Du, “Experimental realization of quantum algorithm for solving linear systems of equations,” [arXiv:1302.1946](https://arxiv.org/abs/1302.1946).
- [36] X. D. D. Cai, C. Weedbrook, Z. E. E. Su, M. C. C. C. Chen, M. Gu, M. J. J. Zhu, L. Li, N. L. Liu, C. Y. Lu, and J. W. Pan, “Experimental quantum computing to solve systems of linear equations,” *Physical Review Letters* **110** no. 23, (Feb, 2013) 1–5, [arXiv:1302.4310](https://arxiv.org/abs/1302.4310).
- [37] S. Barz, I. Kassal, M. Ringbauer, Y. O. Lipp, B. Dakić, A. Aspuru-Guzik, and P. Walther, “A two-qubit photonic quantum processor and its application to solving systems of linear equations,” *Scientific Reports* **4** no. 1, (May, 2014) 1–6, [arXiv:1302.1210](https://arxiv.org/abs/1302.1210).
- [38] Y. Zheng, C. Song, M. C. Chen, B. Xia, W. Liu, Q. Guo, L. Zhang, D. Xu, H. Deng, K. Huang, Y. Wu, Z. Yan, D. Zheng, L. Lu, J. W. Pan, H. Wang, C. Y. Lu, and X. Zhu, “Solving Systems of Linear Equations with a Superconducting Quantum Processor,” *Physical Review Letters* **118** no. 21, (Mar, 2017) 2–6, [arXiv:1703.06613](https://arxiv.org/abs/1703.06613).

- [39] L. A. Sadun, *Applied Linear Algebra: The Decoupling Principle*. American Mathematical Society, 2nd ed., 2007.
- [40] D. W. Berry, G. Ahokas, R. Cleve, and B. C. Sanders, “Quantum Algorithms for Hamiltonian Simulation,” in *Mathematics of Quantum Computation and Quantum Technology*, G. Chen, L. Kauffman, and S. J. Lomonaco, eds., ch. 4, pp. 89–110. Taylor & Francis, Oxford, 2007. <https://www.iqst.ca/people/home/bsanders/BACSCChapter4.pdf>.
- [41] A. Daskin and S. Kais, “Group Leaders Optimization Algorithm,” [arXiv:1004.2242](https://arxiv.org/abs/1004.2242).
- [42] D. W. Berry, G. Ahokas, R. Cleve, and B. C. Sanders, “Efficient quantum algorithms for simulating sparse hamiltonians,” *Communications in Mathematical Physics* **270** no. 2, (Aug, 2007) 359–371, [arXiv:0508139](https://arxiv.org/abs/0508139) [quant-ph].
- [43] D. W. Berry and A. M. Childs, “Black-box Hamiltonian simulation and unitary implementation,” [arXiv:0910.4157](https://arxiv.org/abs/0910.4157).
- [44] A. M. Childs and R. Kothari, “Simulating sparse Hamiltonians with star decompositions,” [arXiv:1003.3683](https://arxiv.org/abs/1003.3683).
- [45] A. M. Childs and N. Wiebe, “Hamiltonian Simulation Using Linear Combinations of Unitary Operations,” [arXiv:1202.5822](https://arxiv.org/abs/1202.5822).
- [46] L. Wossnig, Z. Zhao, and A. Prakash, “A quantum linear system algorithm for dense matrices,” [arXiv:1704.06174](https://arxiv.org/abs/1704.06174).
- [47] Y. Cao, A. Papageorgiou, I. Petras, J. Traub, and S. Kais, “Quantum algorithm and circuit design solving the Poisson equation,” *New Journal of Physics* **15** (Jul, 2013) 1–30, [arXiv:1207.2485](https://arxiv.org/abs/1207.2485).
- [48] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Washington D.C., 9th print ed., 1964.
- [49] G. Grimmett and D. Welsh, *Probability: An Introduction*. Oxford University Press, Oxford, 2nd ed., 2014.
- [50] A. Daskin and S. Kais, “Decomposition of unitary matrices for finding quantum circuits: Application to molecular Hamiltonians,” *Journal of Chemical Physics* **134** no. 14, (Sep, 2011) 1–15, [arXiv:1009.5625](https://arxiv.org/abs/1009.5625).

A. Controlled gates

The QX simulator only offers a limited number of gates, with especially few controlled versions of gates. Some of these gates are desired in QX implementation in this thesis, and hence they need to be emulated using other gates. Most circuits are based on the circuits in Figure 1.3 and Figure 1.4.

The gates that are implemented in this thesis are,

- $V^{(\dagger)}$
- controlled- $S^{(\dagger)}$
- controlled- H
- controlled- $V^{(\dagger)}$ (requires V , controlled- H and controlled- $S^{(\dagger)}$)
- controlled- $R_x(\theta)$, $-R_y(\theta)$ and $-R_z(\theta)$
- doubly-controlled- $R_x(\theta)$, $-R_y(\theta)$ and $-R_z(\theta)$
- doubly-controlled- Z (requires controlled- $S^{(\dagger)}$)

The following figures will provide the circuits for these gates. All gates were tested for every possible basis state (e.g. $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$) for a two qubits gate). Due to linearity, this proves the functionality of the gates for all other states as well.

$$\begin{array}{cc} \text{---} [V] \text{---} = \text{---} [H] [S] [H] \text{---} & \text{---} [V^\dagger] \text{---} = \text{---} [H] [S^\dagger] [H] \text{---} \\ \text{(a) QX implementation of the } V \text{ gate.} & \text{(b) QX implementation of the } V^\dagger \text{ gate.} \end{array}$$

Figure A.1: QX implementation of the V and V^\dagger gate.

$$\begin{array}{cc} \begin{array}{c} \bullet \\ | \\ \text{---} [S] \text{---} \\ \bullet \\ | \\ \text{---} [T] \oplus [T^\dagger] \oplus \text{---} \\ \bullet \\ | \\ \text{---} [T] \text{---} \end{array} = \begin{array}{c} \bullet \\ | \\ \text{---} [T] \oplus [T^\dagger] \oplus \text{---} \\ \bullet \\ | \\ \text{---} [T] \text{---} \\ \bullet \\ | \\ \text{---} [T] \text{---} \end{array} & \begin{array}{c} \bullet \\ | \\ \text{---} [S^\dagger] \text{---} \\ \bullet \\ | \\ \text{---} [T^\dagger] \oplus [T] \oplus \text{---} \\ \bullet \\ | \\ \text{---} [T^\dagger] \text{---} \end{array} \\ \text{(a) QX implementation of the } S \text{ gate.} & \text{(b) QX implementation of the } V^\dagger \text{ gate.} \end{array}$$

Figure A.2: QX implementation of the controlled- S and controlled- S^\dagger gate.

$$\begin{array}{c} \bullet \\ | \\ \text{---} [T] [X] [T^\dagger] [X] \text{---} \\ \bullet \\ | \\ \text{---} [H] \text{---} \\ \oplus \\ \text{---} [H] [T] \oplus [T] [H] [S] [X] \text{---} \end{array}$$

Figure A.3: QX implementation of the controlled- H gate.

$$\begin{array}{cc} \text{---} [V] \text{---} = \text{---} [H] [S] [H] \text{---} & \text{---} [V^\dagger] \text{---} = \text{---} [H] [S^\dagger] [H] \text{---} \\ \text{(a) QX implementation of the controlled-} V \text{ gate.} & \text{(b) QX implementation of the controlled-} V^\dagger \text{ gate.} \end{array}$$

Figure A.4: QX implementation of the controlled- V and controlled- V^\dagger gate. The implementations of the controlled- H , controlled- S and controlled- S^\dagger gates are shown in Figures A.3, A.2a and A.2b respectively.

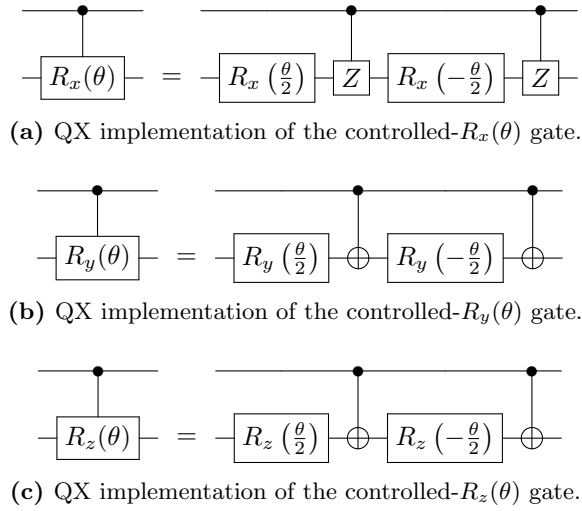


Figure A.5: QX implementation of the controlled- $R_x(\theta)$, controlled- $R_y(\theta)$ and controlled- $R_z(\theta)$ gate.

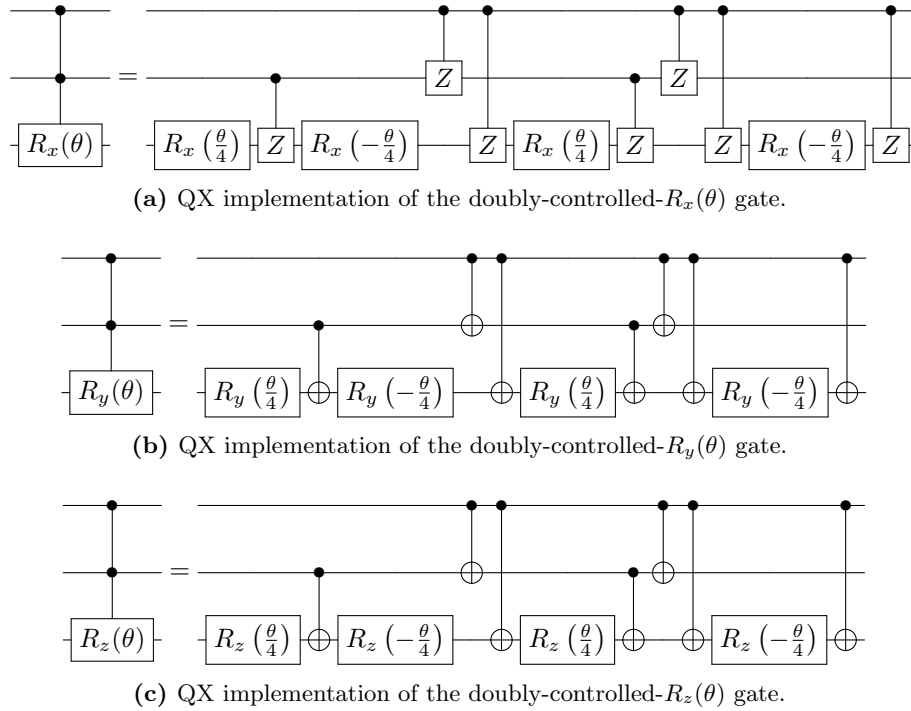
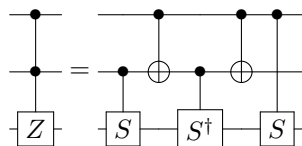
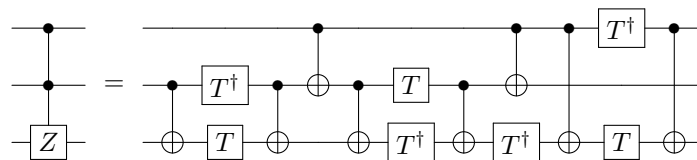


Figure A.6: QX implementation of the doubly-controlled- $R_x(\theta)$, doubly-controlled- $R_y(\theta)$ and doubly-controlled- $R_z(\theta)$ gate.

(a) QX implementation of the doubly-controlled- Z gate.(b) QX implementation of the doubly-controlled- Z gate, compact.**Figure A.7:** QX implementation of the doubly-controlled- Z gate.

B. Cao Eigenvalue Inversion Algorithm

The Eigenvalue Inversion algorithm proposed by Cao et al. in [34] takes advantage of quantum effects to invert the eigenvalues and is the only algorithm not based on a classical one. However, attempts to implement the algorithm have proved unsuccessful, which means that this section can be skipped if only algorithms with proven functionality are of interest. More in depth analysis on the attempts at implementation will follow in the section on QX implementation.

The algorithm makes use of three registers L , M and C , with ℓ , m and n qubits respectively. The C register is used to store the value to be inverted, say λ . The L register stores the final inverted values, and the M register is a work register that will end up with garbage states. Cao et al. propose the following algorithm.

0. Initially:

$$|0\rangle_L \otimes |0\rangle_M \otimes |\lambda\rangle_C$$

1. Apply Walsh-Hadamard transform to registers L and M (i.e. a Hadamard gate to each qubit in registers L and M):

$$\rightarrow \sum_{s=0}^{2^\ell-1} |s\rangle_L \otimes \sum_{p=0}^{2^m-1} |p\rangle_M \otimes |\lambda\rangle_C$$

2. Apply the R_{zz} subroutine:

$$\rightarrow \sum_{s=0}^{2^\ell-1} \sum_{p=0}^{2^m-1} \exp\left[2\pi i \frac{p}{2^m}\right] |s\rangle_L |p\rangle_M \otimes |\lambda\rangle_C$$

3. Apply the $e^{iH_0 t_0}$ subroutine:

$$\begin{aligned} &\rightarrow \sum_{s=0}^{2^\ell-1} \sum_{p=0}^{2^m-1} \exp\left[2\pi i \frac{p}{2^{m+1}} (2^\ell - \lambda s)\right] |s\rangle_L |p\rangle_M |\lambda\rangle_C \\ &\approx \sum_{p=0}^{2^m-1} |2^\ell/\lambda\rangle_L |p\rangle_M |\lambda\rangle_C \\ &= |2^\ell/\lambda\rangle_L |\lambda\rangle_C \otimes \sum_{p=0}^{2^m-1} |p\rangle_M \end{aligned}$$

A circuit of the proposed algorithm is shown in Figure B.1.

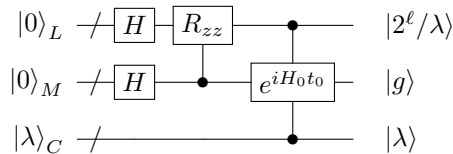


Figure B.1: Circuit for the Eigenvalue Inversion Algorithm proposed by Cao et al. in [34].

The R_{zz} and $e^{iH_0 t_0}$ subroutines, which are hereafter interchangeably referred to as the Rotation Subroutine and Hamiltonian Subroutine, will be examined later. First, however, the rewrite in step 3 from the first to the second step is discussed, as it is not a trivial rewrite. The step makes use of the equality [34],

$$\sum_{k=0}^{N-1} \exp\left(2\pi i k \frac{r}{N}\right) = \delta_{(r,0 \bmod N)}. \quad (\text{B.1})$$

The state in the first line of Step 3 can be rewritten into the left hand side of this equality, when $N = 2^m$, $k = p$ and $r = \frac{1}{2}(2^l - \lambda_j s)$ are taken. This implies that according to the equality, only the states with $r = \frac{1}{2}(2^l - \lambda_j s) = 0$ are non-zero. Solving the equality for s shows that only the state $s = 2^l/\lambda_j$ remains. This is precisely an inverse of s , as was requested.

In what follows, the two subroutines R_{zz} and $e^{iH_0 t_0}$ will be explained in more detail.

B.1 The Rotation subroutine

First is the R_{zz} Rotation Subroutine. The R_{zz} subroutine is defined as the transformation,

$$|s\rangle_L |p\rangle_M \xrightarrow{R_{zz}} \exp\left[2\pi i \frac{p}{2^m}\right] |s\rangle_L |p\rangle_M. \quad (\text{B.2})$$

To implement this subroutine, Cao et al. [34] make use of so-called $R_{zz}(\theta)$ gates, which are defined as global rotation gates,

$$R_{zz}(\theta) = \begin{bmatrix} e^{i\theta} & 0 \\ 0 & e^{i\theta} \end{bmatrix}. \quad (\text{B.3})$$

Note that the effect of an R_{zz} gate is independent of the gate to which it is applied. With the R_{zz} gates, the implementation of the R_{zz} subroutine as defined in [34] is shown in Figure B.2, alongside the Hadamard transforms. In the subroutine, a value t_0 is used. This value is defined as 2π .

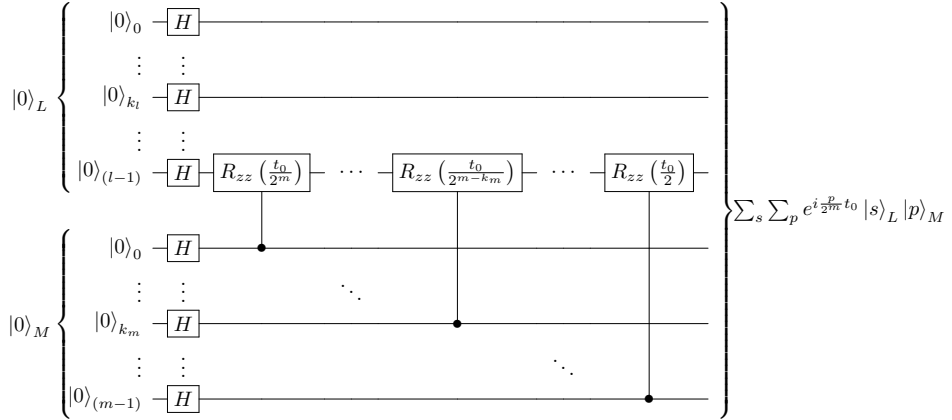


Figure B.2: Walsh-Hadamard transform and R_{zz} subroutine as defined in [34], where $t_0 = 2\pi$.

B.2 The Hamiltonian Subroutine

The matrix H_0 is defined as a diagonal matrix, with the values $[1, 2, 4, \dots, 2^{m-1}]$ on its diagonal. The subroutine performs a Hamiltonian simulation of H_0 over time t_0 on register M , controlled by $|\lambda\rangle_C$ and register L , in such a way that it performs the operation

$$|s\rangle_L |p\rangle_M |\lambda\rangle_C \xrightarrow{\exp(iH_0 t_0)} \exp\left[-2\pi i \frac{p}{2^{m+\ell}} \lambda s\right] |s\rangle_L |p\rangle_M |\lambda\rangle_C. \quad (\text{B.4})$$

Since H_0 is a diagonal matrix, its Hamiltonian simulation $e^{iH_0 t}$ is easily found as was shown in [50]. The circuit for $e^{iH_0 t/2^m}$ as defined in [34] is shown in Figure B.3.

It is important to note that Cao et al. use a different definition for the $R_z(\theta)$ gate. Their definition differs only in global phase, and is defined as

$$R_z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}. \quad (\text{B.5})$$

$$\not\vdash \boxed{e^{iH_0 \frac{t}{2^m}}} \not\vdash \equiv \left\{ \begin{array}{l} \boxed{R_z \left(\frac{t}{2} \right)} \\ \boxed{R_z \left(\frac{t}{2^2} \right)} \\ \boxed{R_z \left(\frac{t}{2^3} \right)} \\ \vdots \\ \boxed{R_z \left(\frac{t}{2^m} \right)} \end{array} \right.$$

Figure B.3: Decomposition of the Hamiltonian Simulation $e^{iH_0 t}$ of matrix H_0 .

Since only the unimportant global phase is different, the difference is ignored. With this definition of the $R_z(\theta)$ gate, Cao et al. stipulate that the circuit in Figure B.3 is a decomposition for the Hamiltonian Simulation of H_0 .

Using the $e^{iH_0 t/2^m}$ decomposition, Cao et al. implement the $e^{iH_0 t_0}$ subroutine as a number of $G(\ell - k)$ operations as defined in Figure B.4.

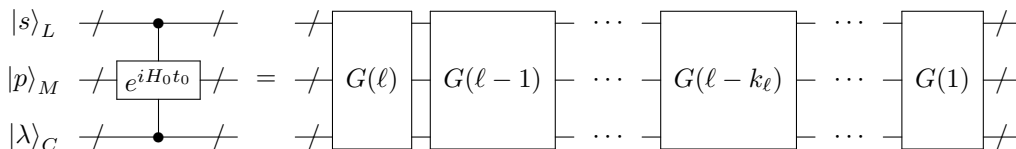


Figure B.4: Implementation of the $e^{iH_0 t_0}$ subroutine as defined in [34].

The $G(\ell - k_\ell)$ operations in turn are defined as a number of Hamiltonian simulations, as is shown in Figure B.5. The gates labelled (u, v) (where $u = c, \dots, c - k_c, \dots, 1$ and $v = \ell - k_\ell$) in the circuit represent an $\exp\left(-\frac{iH_0 t_0}{2^{u+v}}\right)$ gate. This means that the (u, v) -gates can be realised using an $e^{iH_0 \frac{t_0}{2^{u+v-m}}}$ simulation as shown in Figure B.3 with $t = \frac{t_0}{2^{u+v-m}}$.

A remaining question is what sizes registers L and M need to be in order to be able to perform the inversion of λ 's. Since M and L need to store all values λ_j and $\frac{1}{\lambda_j}$ of matrix A , respectively, the minimum required size are $m \geq \lceil \log \lambda_{\max} \rceil$ and $\ell \geq \lceil \log 1/\lambda_{\min} \rceil$. These constraints can be rewritten to the more general constraint of: $m - \ell \approx \log \kappa$. Increasing the sizes of both registers together will improve accuracy.

B.3 Results

The Cao Inversion Algorithm makes use of $R_{zz}(\theta)$ gates. These $R_{zz}(\theta)$ gates are not available in the QX Simulator and are impossible to recreate. However, only controlled- $R_{zz}(\theta)$ gates are used in the algorithm. These controlled versions can be implemented by directly applying an $R_z(\theta)$ gate (with the same angle) to the control qubit. The target qubit drops out of the implementation, as the output of the R_{zz} is independent of the qubit state it is applied to. The QX implementation only results in an unimportant global phase difference compared to using a $C(R_{zz}(\theta))$ gate.

The Cao Inversion Algorithm was implemented for any number of input qubits. However, due to the large number of required ancillae, it has only been run for $n \leq 4$, in which case the algorithm requires up to $4n = 16$ qubits for $n = 4$. Despite the thorough explanations by Cao et al, the implementation of the Cao Inversion Algorithm does not show any of the expected results. Especially none of the cancelling of states in Step 3 is observed. Instead, every state from Step 2 is still present, but with different rotations depending on the value of the M register (higher value means larger rotation). Due to the many hundreds of states in superposition (for example $2^{10} = 1024$ states for $n = 3$), the states will not be plotted. The superposition of rotations is not

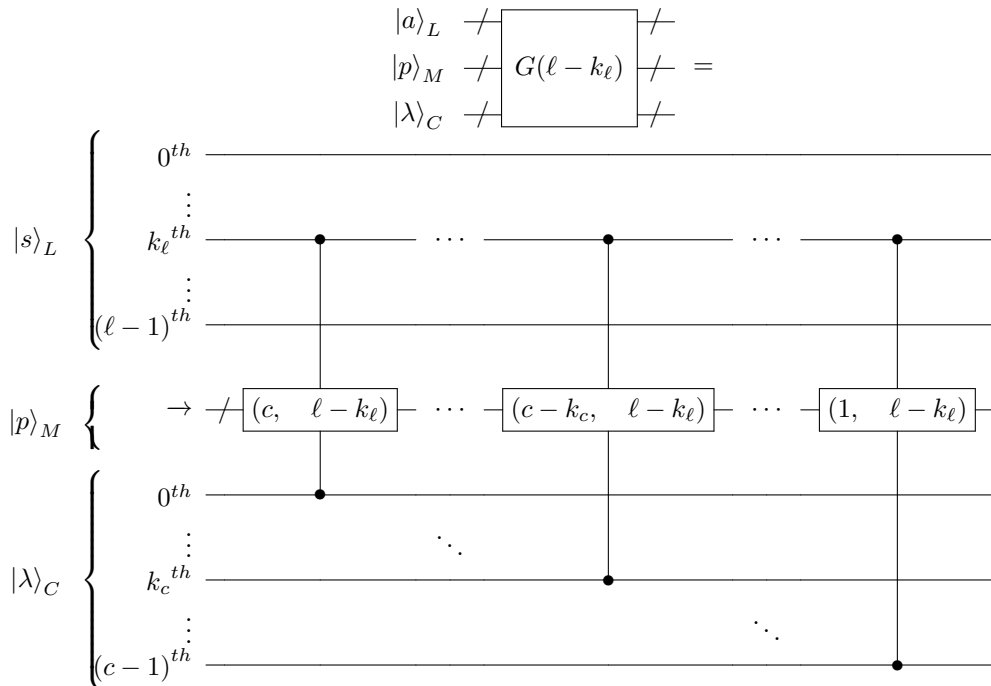


Figure B.5: Decomposition of the $G(\ell - k_\ell)$ block used in the $e^{iH_0 t_0}$ subroutine. The gates labelled (u, v) in the diagram, for $u = c, \dots, c - k_c, \dots, 1$ and $v = \ell - k_\ell$, represent the quantum gate $\exp\left(-\frac{iH_0 t_0}{2^{u+v}}\right)$, which can be realised using the circuit in Figure B.3.

unsurprising, since z -rotations by themselves cannot have a cancelling effect.

The main suspicion with regard to the unexpected results, was that the final states were obfuscated by the garbage states in register M . To get rid of these states, the method from Figure 1.6b was applied: first, the algorithm was performed, then the result in register L was copied to a fourth register K , and finally the algorithm was reversed. This, however, did not have any of the desired effects, as the resulting state in the K register was simply a pure Walsh-Hadamard transform of the initial $|0\rangle_K$ state. Snippets of this final state for input $|010\rangle$ are shown in figure B.6.

The main suspicion to why the algorithm does not work lies in the $e^{iH_0 t_0}$ subroutine. The Cao paper does not explain in detail why exactly this subroutine should work, and it is not easily verifiable whether it does. Future research could further look into this method, but due to the other successful number inversion algorithms described in this thesis it is doubtful whether this is worthwhile.

```

-----[quantum state]-----
(+0.125000,+0.000000) |0000100000000000> +
(+0.125000,+0.000000) |0000100000000001> +
(+0.125000,+0.000000) |0000100000000010> +
(+0.125000,+0.000000) |0000100000000011> +
(+0.125000,+0.000000) |0000100000000100> +
(+0.125000,+0.000000) |0000100000000101> +
(+0.125000,+0.000000) |0000100000000110> +
(+0.125000,+0.000000) |0000100000000111> +
(+0.000000,+0.000000) |0000100000001000> +
(+0.000000,+0.000000) |0000100000001001> +
(+0.000000,+0.000000) |0000100000001010> +
(+0.000000,+0.000000) |0000100000001011> +
(+0.000000,+0.000000) |0000100000001100> +
(+0.000000,+0.000000) |0000100000001101> +
(+0.000000,+0.000000) |0000100000001110> +
(+0.000000,+0.000000) |0000100000001111> +
.
.
.
(-0.000000,+0.000000) |0000101111111100> +
(-0.000000,+0.000000) |0000101111111101> +
(-0.000000,+0.000000) |0000101111111110> +
(-0.000000,+0.000000) |0000101111111111> +
(+0.125000,+0.000000) |0010100000000000> +
(-0.125000,-0.000000) |0010100000000001> +
(+0.125000,+0.000000) |0010100000000010> +
(-0.125000,-0.000000) |0010100000000011> +
(+0.125000,+0.000000) |0010100000000100> +
(-0.125000,-0.000000) |0010100000000101> +
(+0.125000,+0.000000) |0010100000000110> +
(-0.125000,-0.000000) |0010100000000111> +
(+0.000000,+0.000000) |0010100000001000> +
(-0.000000,-0.000000) |0010100000001001> +
(+0.000000,+0.000000) |0010100000001010> +
(-0.000000,-0.000000) |0010100000001011> +
.
.
.
(+0.000000,-0.000000) |1110101111111000> +
(-0.000000,+0.000000) |1110101111111001> +
(-0.000000,+0.000000) |1110101111111010> +
(+0.000000,-0.000000) |1110101111111011> +
(-0.000000,+0.000000) |1110101111111100> +
(+0.000000,-0.000000) |1110101111111101> +
(+0.000000,-0.000000) |1110101111111110> +
(-0.000000,+0.000000) |1110101111111111> +
-----

```

Figure B.6: Snippets of the output of the Cao Inversion Algorithm for $n = 3$ and input $|010\rangle$. The registers in the output are respectively the K , C , M and L registers of lengths 3, 3, 6 and 3.