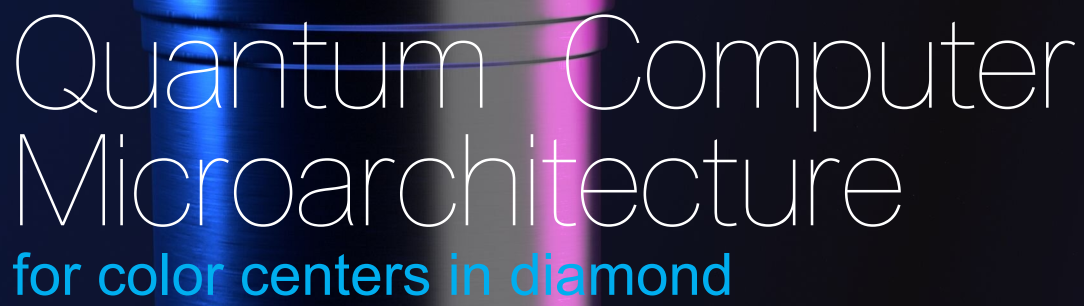
Quantum Computer Microarchitecture



for color centers in diamond

Q. van Wingerden

TU Delft

Quantum Computer Microarchitecture

for color centers in diamond

by

Q. van Wingerden

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday, August 26 2021 at 14:00.

Student number:	4476913	
Project duration:	Nov 1, 2020 – July 1, 2021	
Thesis committee:	Prof. dr. ir. J.S.S.M Wong,	TU Delft, supervisor
	Dr. F. Sebastiano,	TU Delft
	Dr. S. Feld,	TU Delft

This thesis is confidential and cannot be made public until August 26, 2021.

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Nowadays, the need for better and faster computing searches for solutions in the field of quantum computing. It is believed that quantum computing can and will surpass conventional, classical computing. When quantum computing surpasses classical computing, it is called quantum supremacy. There are many different projects that use different quantum technologies aim to achieve quantum supremacy, such as projects from Google and Intel. In this thesis, the reader is introduced to a new project: the Fujitsu Project. The project is a collaboration between TU Delft and Fujitsu and has the goal of fabricating a new distributed quantum quantum computer based on color centers in diamond. The focus of the thesis is the definition of the microarchitecture.

The quantum computer stack is explained in detail. Precise and robust definitions for each of the seven layers of the quantum computer stack are given, where the definition of the microarchitecure is most important. A microarchitecture is defined as the list of instructions of the lowest level before entering control electronics. To define the diamond microarchitecture, a set of goals have to be completed. Other architectures such as QuTech's QuMA and QuTech's Central Controller were identified and analyzed. Combined with the requirements of the overall diamond system architecture as defined by the project, a list of requirements for the microarchitecture is created. Using these requirements, a Quantum Instruction Set Architecture (QISA) and a microarchitecture are defined.

Based on the defined QISA and microarchitecture, a compiler is designed. The compiler is made in OpenQL. OpenQL is a framework for high-level quantum programming that targets different quantum computing platforms. The main task of the compiler is to translate high-level quantum algorithms, expressed in a high-level quantum programming language (C++/Python APIs), to quantum microcode. The quantum microcode consists of instructions that are defined in the microarchitecture.

The QISA, microarchitecture and compiler are verified against the requirements that are set at the beginning of their respective definition phases. The microarchitecture supports for all gates that are part of cQASM 1.0. In addition, it supports all diamond color center specific protocols and rules. The compiler supports all instructions from cQASM as well. Moreover, the compiler supports all standard (gate) instructions and diamond specific instructions through OpenQL's Python API.

The work is intended to be a solid baseline, where the future of the project can rely and improve upon. Possible improvements could be the adaption of the microarchitecture to the growth of the number of controlled qubits per controller, the parallelization of the microarchitecture instructions to improve instruction throughput, and alignment with the quantum network group. The compiler can also be improved with for example enhanced scheduling, differentiation between qubit types and an entanglement library. The next step in the design of the microarchitecture will be the design of a microarchitecture simulator, which takes the microcode as input and simulates the hardware architecture.

Preface

Starting my studies at TU Delft I always had affinity with computer. I always wanted to know how electronics worked and how computers worked from the inside. Because of this, I started my bachelor Electrical Engineering at the TU Delft. After completing the BSc, I pursued a master's degree in Computer Engineering to learn more about how computers work. Nearing the end of the first year of this enjoyable master it was time to search for a thesis project. I had gathered interest for quantum computing and, *Stephan Wong* answered my email with an interesting project about quantum computer microarchitectures. After an online meeting we decided that I would take the project.

Now, a year later, the thesis project has come to an end. I have learned many things, not only about quantum computing and microarchitectures, but also how to write, present, and have meaningful discussions with other people in (online) meetings. I am proud to have completed the thesis, but not without the help of the following people:

First, I would like to thank my supervisor *Stephan Wong* for always being critical and getting the most out of me. He challenged me in multiple ways - by having me give presentations to large crowds in online meetings or by reviewing my thesis time after time until it met the requirements.

Second, I would like to thank *my parents* for supporting me and always being there to talk to. They taught me valuable skills when I was growing up and that helped me complete the thesis.

Third, I would like to thank *Jeroen van Straten* for assiting me with all the work I have done with the OpenQL compiler. Jeroen was one of the maintainers of OpenQL and helped me understand the program and guided me through adding my own code to the framework. I could not have built the compiler as is without Jeroen. I would also like to thank *Luc Enthoven* for the valuable discussions about the workflow of the quantum computer and about the requirements of the microarchitecture. I also would like to thank *Erwin van Zwet* and *Jaco Morits* from TNO that helped form the blueprint for the overall system architecture and thus a blueprint for the microarchitecture.

Fourth, I would like to thank my friends for always being there when I had something to complain about or when I had good results that I could share with them. I want to thank two people in special that helped me do so, namely *Thijs Timmer* and *Suzanne Brand*. I could not have done this project without you.

I hope you enjoy reading this thesis as much as I had fun reading, learning from, thinking, programming, reviewing and submitting it.

Q. van Wingerden
Ridderkerk
August 26, 2021

Contents

1 Introduction **1**
1.1 The Fujitsu Project . . . 1
1.2 Problem Statement . . . 2
1.3 Methodology . . . 2
1.4 Thesis Overview . . . 2

2 Background **4**
2.1 Quantum Information Theory . . . 4
2.1.1 Qubits . . . 4
2.1.2 Gates . . . 5
2.1.3 Quantum Circuits and Algorithms . . . 6
2.2 The QuTech Quantum Computer Stack . . . 6
2.2.1 Quantum Algorithm . . . 6
2.2.2 Programming Paradigm & Languages . . . 7
2.2.3 Quantum Arithmetic, Runtime and Compiler . . . 8
2.2.4 Quantum Instruction Set Architecture . . . 8
2.2.5 Microarchitecture . . . 8
2.2.6 Quantum to Classical . . . 8
2.2.7 Quantum Chip . . . 8
2.3 NV-centers in Diamond . . . 8
2.3.1 Other types of Qubit . . . 9
2.4 Quantum Computer Microarchitectures . . . 10
2.4.1 QuMA . . . 10
2.4.2 QuTech Central Controller . . . 14
2.4.3 Central Controller . . . 15
2.4.4 CC-Spin . . . 18
2.5 Quantum Networking . . . 19
2.5.1 Quantum Communication . . . 19
2.5.2 Entanglement and Teleportation . . . 20
2.5.3 Quantum Tools for Networking . . . 20
2.6 OpenQL . . . 21
2.7 Conclusion . . . 22

3 QISA and Microarchitecture **23**
3.1 Diamond Microarchitecture Requirements . . . 23
3.2 System Architecture . . . 24
3.3 Quantum Instruction Set Architecture (QISA) . . . 25
3.3.1 Refining of the QISA . . . 25
3.4 Finalized QISA and microarchitecture . . . 27
3.4.1 Supported High-Level Instructions . . . 27
3.4.2 Example of decomposition . . . 28
3.5 Conclusion . . . 31

4 Tool **33**
4.1 OpenQL . . . 33
4.2 Compiler Design . . . 33
4.2.1 Compiler Requirements . . . 33
4.2.2 Diamond backend in OpenQL . . . 34
4.2.3 Microcode Translator Pass . . . 34

4.3 Using the Compiler . . . 38
4.4 Conclusion . . . 39

5 **Verification** **40**
5.1 QISA and microarchitecture . . . 40
5.1.1 Deterministic and precise control of the control electronics . . . 40
5.1.2 Classical instructions for flow control . . . 40
5.1.3 Basic quantum gate instructions (X, Y, Z, S and T) . . . 41
5.1.4 cQASM Gateset . . . 41
5.1.5 Diamond specific protocols and instructions . . . 41
5.2 Compiler Tool . . . 43
5.3 Conclusion . . . 45

6 **Conclusion** **47**
6.1 Summary . . . 47
6.2 Main Contributions . . . 48
6.3 Future Work . . . 49
6.3.1 Development of the Fujitsu Project . . . 49
6.3.2 Alignment with the Fujitsu Global Controller . . . 49
6.3.3 Design of the microarchitecture . . . 50
6.3.4 Improvements of the OpenQL Compiler . . . 50

A **Microarchitecture Documentation** **52**
A.1 ISA Instructions . . . 55
A.1.1 Qubit Gate . . . 55
A.1.2 Qubit Rotation . . . 55
A.1.3 Qubit Readout . . . 55
A.1.4 Qubit Initialize . . . 55
A.1.5 nop . . . 56
A.1.6 Entanglement . . . 56
A.1.7 Nuclear Spin Operations . . . 56
A.1.8 Biasing and Checks . . . 56
A.1.9 Calibration . . . 57
A.1.10 Timing . . . 58
A.1.11 Standard (Classical) Instructions . . . 58
A.2 microarchitecture Instructions . . . 59
A.2.1 Quantum Operations . . . 59
A.2.2 Timing . . . 59
A.2.3 Additional Instructions . . . 60
A.2.4 Standard (Classical) Instructions . . . 60

B **Decomposition Microcode** **62**
B.1 Measurement . . . 62
B.2 Initialization . . . 62
B.3 qentangle . . . 62
B.4 NVentangle . . . 62
B.5 memswap . . . 63
B.6 sweep_bias . . . 63
B.7 decouple . . . 63
B.8 calculate_bias . . . 64
B.9 calculate_voltage . . . 64
B.10 cal_meas . . . 64
B.11 cal_pi . . . 65
B.12 cal_halfpi . . . 65

C List of Supported Functions and Gates of the Compiler **67**
C.1 Initialization . . . 67
C.1.1 prep_z . . . 67
C.1.2 prep_x . . . 67
C.1.3 prep_y . . . 67
C.1.4 initialize . . . 67
C.2 Measurement . . . 68
C.2.1 measure . . . 68
C.2.2 measure_z . . . 68
C.2.3 measure_x . . . 68
C.2.4 measure_y . . . 68
C.3 Single Qubit Gates . . . 68
C.4 Two Qubit Gates . . . 68
C.5 Three Qubit Gate . . . 68
C.6 Diamond Calibration . . . 69
C.6.1 cal_measure . . . 69
C.6.2 cal_pi . . . 69
C.6.3 cal_halfpi . . . 69
C.6.4 decouple . . . 69
C.6.5 Custom Rotations . . . 69
C.6.6 rz . . . 69
C.6.7 rx . . . 69
C.6.8 ry . . . 69
C.6.9 cr . . . 69
C.6.10 crk . . . 70
C.7 Diamond Protocols and Sequences . . . 70
C.7.1 crc . . . 70
C.7.2 rabi_check . . . 70
C.7.3 excite_mw . . . 70
C.7.4 qentangle . . . 70
C.7.5 nventangle . . . 70
C.7.6 memswap . . . 71
C.7.7 sweep_bias . . . 71
C.8 Timing . . . 71
C.8.1 wait . . . 71
C.8.2 qnop . . . 71
C.9 Classical Support Functions . . . 71
C.9.1 calculate_current . . . 71
C.9.2 calculate_voltage . . . 71

Bibliography **72**

1 Introduction

Nowadays, in the search for better and faster computing, there is a very large interest in quantum computing. It is believed that quantum computing can solve some problems exponentially quicker using its quantum technology than with its classical counterpart [1]. When quantum computing surpasses classical computing, it is called quantum supremacy. To achieve quantum supremacy, quantum algorithms are needed. An example of such an algorithm is Shor's algorithm [2], which is designed to factor a number N into its prime factors p and q using the power of the Quantum Fourier Transform (QFT). Shor's algorithm can be used to break RSA-encryption because RSA assumes that it is a hard problem to factor a large number N into its prime integers p and q. Another well-known example is Grover's Search Algorithm [3], which provides a quadratic speedup in the unstructured search. Unstructured search is that out of N options the correct option ω has to be picked. Classically, at best one try is needed and at worst N tries. On average, $\frac{N}{2}$ tries are needed. With Grover's algorithm, $\sqrt{N}$ tries are needed, thus providing quadratic speedup.

The field of quantum computing is rapidly evolving. Several quantum computers have been made already, such as the systems by D-Wave [4], which uses quantum annealing, or Sycamore, the 54-qubit quantum processor from Google that claims it showed quantum supremacy [5]. Within QuTech and through the Quantum Inspire platform [6], two quantum accelerators (Spin-2 and Starmon-5) are made publicly available. More examples are the Quantum Processing Units (QPUs) from Intel, Microsoft and IBM, but there are many more QPUs in existence.

Each of these QPUs work with different types of quantum bits (qubits), which are the physical entities that are used to perform calculations on. The qubits are fundamental particles where quantum computers are based on. There are many different technologies available for the creation of different types of qubits. Examples of these types of qubits are transmon, majorana fermion, quantum dots and color centers in diamond (such as NV-centers) [7].

This thesis documents the (beginning of) the development of a microarchitecture for quantum computing using color centers in diamond. This Chapter will introduce the project. The thesis is part of the Fujitsu Project, that is introduced in Section 1.1. After that, the problem statement is introduced in Section 1.2 and the project methodology is discussed in Section 1.3. The Chapter ends with a overview of the thesis in Section 1.4.

1.1. The Fujitsu Project

The Fujitsu Project is a collaboration between TU Delft and Fujitsu Limited. The project has a duration of five years, and the goal of is to develop a distributed quantum computer based on color centers in diamond. Color centers contain a particle, typically an electron, that has a spin and thus can be used as a qubit. The goal is to develop and fabricate a *scalable* quantum computer using these qubits. Because of the modular nature of the architecture, retaining scalability should be expected. In addition to that, the color center qubits have special characteristics compared to other types of qubit[1]. In contrast to other qubits, color center qubits can function at room temperature. This means less cooling and thus

[1]This also depends on the type of color center qubit. At Fujitsu, NV-centers as well as SnV-centers are used.

less energy consumption. In addition, the qubits have long coherence times. This means that the qubit can hold its state longer, which is beneficial for quantum computing. Color center qubits are *ODMR*, which stands for Optically Detected Magnetic Resonance. This means that the qubits can be read out with lasers and photondetectors, and can be controlled by magnetic fields. Using these qubits, there are two important goals that the project aims to achieve:

1. Demonstration of scalable fabrication of the integrated circuits that contain the qubit, the control electronics and other parts needed to function.
2. Demonstration of a unit with a 20 qubits operating with a high fidelity.

1.2. Problem Statement

For the Fujitsu Project, a microarchitecure needs to be developed that will work with the diamond spin qubits. It should interface with the control electronics that are directly connected to the qubits. The following research question arises:

How is a microarchitecture for a quantum computer based on color centers in diamond defined?

To answer this research question, three goals have been set:

1. Define the overarching Quantum Instruction Set Architecture (QISA) that interfaces with the higher layers of the quantum computer stack.
2. Define the microarchitecture.
3. Build a compiler that is able to translate a quantum algorithm to quantum microcode, defined by the microarchitecture, using the higher layer and interfaced with an existing compiler framework.

1.3. Methodology

To complete these goals and answer the research question, a number of tasks are performed:

1. Identify how other quantum microarchitectures are defined.
2. Determine the requirements of the diamond microarchitecture.
3. Define the microarchitecture.
4. Fabricate a simple compiler that compiles an algorithm to diamond-specific assembly code.
5. Verify the results by comparing them to their requirements.

1.4. Thesis Overview

In Chapter 2, the background material needed to understand the rest of the thesis is provided. First, it covers basics, such as quantum information theory. Second, it covers the quantum computer stack that is defined within QuTech. Third, NV-centers, a type of color center in diamond, are discussed. Fourth, the reader is introduced to quantum computer microarchitectures. An overview of developments in quantum networking is given in the following section. Finally, an overview about OpenQL, the framework for high-level quantum programming in C++/Python, is given.

In Chapter 3, the design of the microarchitecture is discussed. It will cover what the microarchitecture needs to be able to do. After that, an overview of the system where the microarchitecture is being designed for is given. The design process of the QISA as well as the microarchitecture is discussed. The finalized version of both the QISA and the microarchitecture is presented. At the end of the chapter, decompositions from the QISA to the microarchitecture are presented.

In Chapter 4, we discuss the compiler tool that translates high-level quantum algorithms to low-level assembly code (microcode). OpenQL is briefly refreshed upon. After that, the requirements of the design and the design itself are presented. At the end of the chapter it is explained how to use the compiler tool.

In Chapter 5, we present the verification of the designed microarchitecture. It is split in two parts. The first part discusses the completeness of the microarchitecture. It compares its functions with the

requirements that are stated in Chapter 3. In the second part of the Chapter, the compiler is verified against the requirements introduced in Chapter 4.

In Chapter 6, we conclude the thesis by taking a look at at the work that is done. A summary is given and the research question is answered using the goals and tasks defined in this introduction. Furthermore, the main contributions are listed as well as ideas and suggestions for future work.

Appendix A documents the documentation for reading and using the defined microarchitecture.

Appendix B presents details about decompositions that are explained in Chapter 3.

Appendix C contains the documentation for the compiler tool.

2
Background

This chapter will present all the information that is needed to understand the thesis work. In Section 2.1, the theory behind quantum computing, also called quantum information theory, is briefly touched upon. Here, qubits are explained, just as basic quantum operations, gates and how to build circuits and algorithms. Then in Section 2.2, the QuTech quantum computer stack is explained. The stack is an important reference for the design of quantum computers. In this section, the definition of a microarchitecture is given. After the stack has been presented, NV-centers in diamond will be explained in Section 2.3. Section 2.4 will discuss quantum computer microarchitectures. After that, a short introduction to quantum networking is presented in Section 2.5. Finally, in Section 2.6, OpenQL, a framework for high-level quantum programming, is introduced.

2.1. Quantum Information Theory

This section will give a short summary of quantum information theory. It will cover quantum bits (qubits) by explaining what they are and how they differ from classical bits. The section will also cover quantum gates (operations on the qubits) and quantum algorithms (quantum programs).

2.1.1. Qubits

A qubit is different from a classical bit. Where a classical bit can be either in state '0' or '1', a qubit can be in a linear superposition of these states. The notation of qubits is called the *Dirac* notation [8]. This can be illustrated by the following formula:

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

Here, it can be seen that the qubit consists of state $|0\rangle$ by part α and state $|1\rangle$ by part β. However, since a qubit must be 'complete', the chance of the qubit existing must be 1. Therefore, it must hold that $|\alpha|^2 + |\beta|^2 = 1$.

There are a other rules that the qubits must follow. One of the most important ones is that when you measure a qubit, the outcome is not deterministic. Measuring a qubit is a probabilistic process. For example, if α and β have the same value (which is valid for a value of $\frac{1}{\sqrt{2}}$), the chance of measuring $|0\rangle$ and $|1\rangle$ is the same, as $|\frac{1}{\sqrt{2}}|^2 = 0.5$. If measuring that specific qubit and the outcome is $|0\rangle$, then the full state becomes $|\Psi\rangle = |0\rangle$. This means that the qubit state collapses to either $|0\rangle$ or $|1\rangle$, depending on the measurement result. In the case of the example, the state has been collapsed to the zero-state.

A good way to visualize the state of a qubit is by using the Bloch-sphere, depicted in Figure 2.1. Note that the Bloch-sphere visualization only works for a single qubit. The qubit will be anywhere on the edge of the sphere. It can help to understand what the state of the qubit is when applying the quantum gates in algorithms. It also means that the formula from earlier can be rewritten as:

$$|\Psi\rangle = cos(\frac{\theta}{2})\,|0\rangle + e^{i\phi}sin(\frac{\theta}{2})\,|1\rangle$$

In addition to the previous notations, there is also a notation that is particularly helpful when doing (hand) calculations on qubits when they are in the so-called Clifford states, which are found on the Bloch-sphere as +z, -z, +x, -x, +y and -y axis.

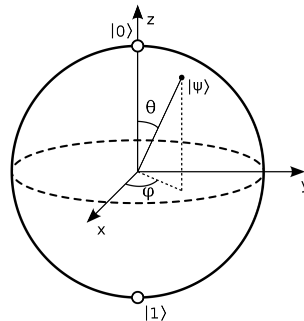


Figure 2.1: The Bloch-sphere [9].

The Clifford states are:

1. $|0\rangle \equiv \begin{bmatrix}1\\0\end{bmatrix}$
2. $|1\rangle \equiv \begin{bmatrix}0\\1\end{bmatrix}$
3. $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \equiv |+\rangle \equiv \begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{bmatrix}$
4. $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle \equiv |-\rangle) \equiv \begin{bmatrix}\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\end{bmatrix}$
5. $\frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle \equiv |+i\rangle) \equiv \begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{i}{\sqrt{2}}\end{bmatrix}$
6. $\frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle \equiv |-i\rangle) \equiv \begin{bmatrix}\frac{1}{\sqrt{2}}\\-\frac{i}{\sqrt{2}}\end{bmatrix}$

By representing the states in their vector notations, they can be transformed into new states by using linear algebra.

Until now, only single qubit states are presented. However, multiple qubits can share a state together. For example, when two qubits share a state together, it can be represented like the following:

$$|\Psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

The rules that apply to a single qubit state still hold. Multiple qubit states are very useful because they allow for the use of quantum entanglement. Entanglement can be seen as the quantum-mechanical connection between two qubits. It can be seen as a *resource* [8] that plays a key role in quantum algorithms.

2.1.2. Gates

Gates can also be represented by their vector notation. This makes that if the X-gate (quantum equivalent of the NOT-gate) is applied, the calculation becomes:

$$X\,|0\rangle = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} = |1\rangle$$

The X-gate is given by the first (2x2) matrix. The qubit itself is represented by the second (2x1) matrix. An overview of the most frequently used gates are given in Figure 2.2, together with their circuit representation. The Hadamard gate, H, maps a qubit into equal chances of measurement of $|0\rangle$ or $|1\rangle$, creating superposition. The X, Y and Z gates rotate the qubit around the specified axis (x, y or z) with π radians. The S and T gate both shift the phase of the qubit over the Z-axis, with $\frac{\pi}{2}$ and $\frac{\pi}{4}$ respectively. Gates are the building blocks of quantum algorithms, as illustrated in the next section. It is worth noting that some gates, such as the Pauli-X gate, can be written as a HZH gate. Other examples are Pauli-Y, which equals iXZ and Pauli-Z that equals HXH.

Hadamard	H	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
Pauli-X	X	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
Pauli-Y	Y	$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
Pauli-Z	Z	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
Phase	S	$\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$
$\pi/8$	T	$\begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$

Figure 2.2: Overview of qubit gates [8, Fig. 4.2]

2.1.3. Quantum Circuits and Algorithms

Quantum algorithms are described using discrete time-based schematics. An example can be found in Figure 2.3 where the state of the top qubit is teleported to the bottom qubit. As can be seen, multiple gates from Figure 2.2 are present in the circuit together with two measurements. The majority of quantum circuits and algorithms are visualized like this. In the Figure, there is an X-gate on qubit 0, followed by CNOT with qubit 0 as control and qubit 1 as target.Then, a Hadamard gate is performed on qubit 0 followed by a measurement on qubits 0 and 1. Depending on the outcome of the measurements, an X gate and a Z gate are performed on qubit 2. The example is explained in more detail in [8, Section 1.3.7].

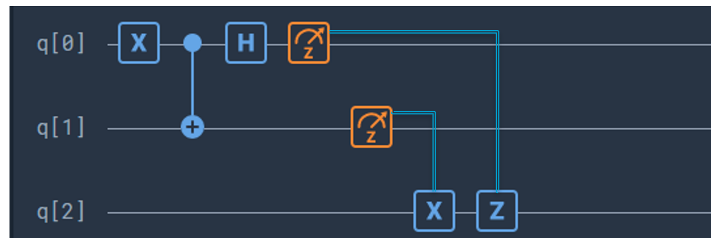


Figure 2.3: Teleportation Circuit. Made with Quantum Inspire [6].

2.2. The QuTech Quantum Computer Stack

As this thesis focuses on the definition of a microarchitecture for a quantum computer based on diamonds, it is important to get the definitions of said microarchitecture correct. A clear definition of what the microarchitecture is already exists for classical computing systems and so the same must be done for quantum computing systems. A clear definition will add clarity for all who are involved in the development of quantum computers, as different research facilities and different quantum computing companies might have different definitions right now. Based on work that has been done before, an abstract quantum computer stack can be created, as depicted in Figure 2.4. The quantum computer stack presents an overview of the different layers of a quantum computer. It can be observed that there are a total of seven layers. In this section, the definition of each layer is given, starting with the top layer and working to the bottom layer.

2.2.1. Quantum Algorithm

This layer contains the definition of the quantum algorithm. Contrary to classical algorithms, where the data input and data output set are defined, as well as the operations that transform the input data to the output data, a quantum algorithm consists of a set of discrete-time gates operating on qubits as depicted in Figure 2.3. Examples for use cases are data encryption, finding a new molecule or synthesizing a

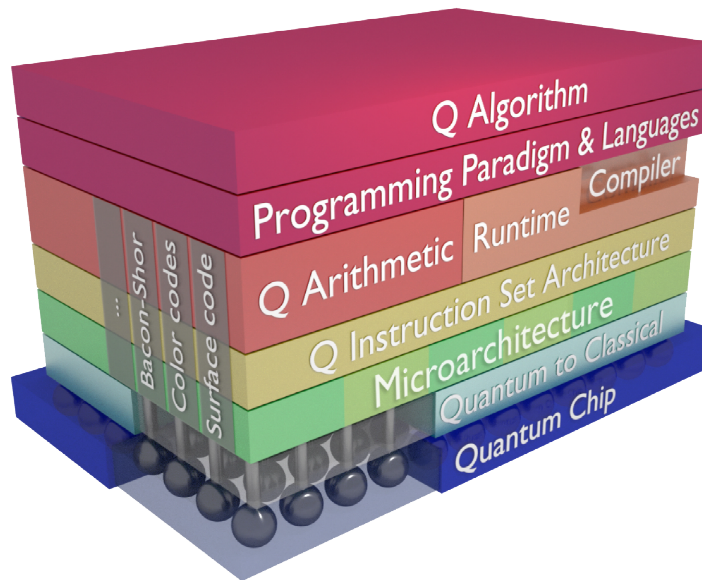


Figure 2.4: The QuTech system stack. Q stands for Quantum.

new vaccine for COVID-19 (caused by the SARS-CoV-2 virus). The quantum algorithms have the goal of speeding up a use case using quantum computation characteristics, such as superposition or entanglement. Examples of well-known quantum algorithms are Shor's Algorithm [2] or Grover's Search Algorithm [3], as explained in the introduction of the thesis.

2.2.2. Programming Paradigm & Languages

In this layer, the quantum algorithm defined in the Q Algorithm layer (for example: Shor's Algorithm) gets transformed into a specification using a high-level quantum programming language. There are different high-level quantum programming languages available today, such as Scaffold, Q, Q#, Quipper, LIQUI|⟩ and Project Q, see Figure 2.5. For some of these languages, compilers have been written. For example, to compile Scaffold, the ScaffCC compiler and analysis framework is used. Here at QuTech, we use the OpenQL framework, which uses a Python API or a C++ API as algorithm input. OpenQL is currently used for the QuTech Central Controller and for the hardware backends that can be programmed through Quantum Inspire as well [6]. More information about OpenQL can be found in Section 2.6 or at `openql.readthedocs.org`.

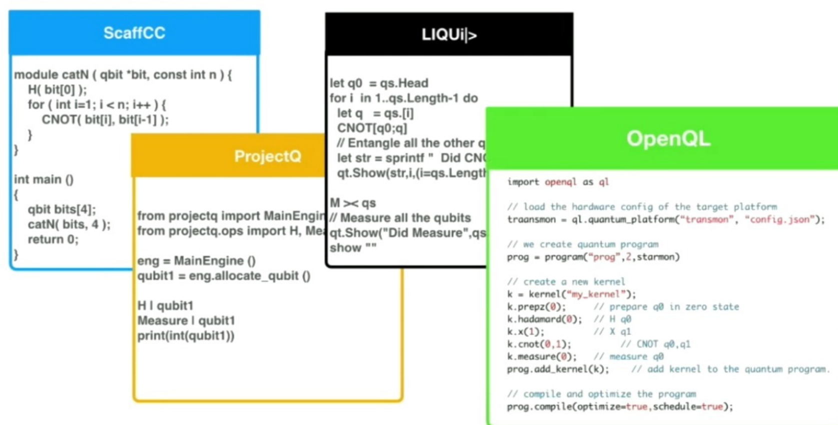


Figure 2.5: Different programming languages for quantum computing.

2.2.3. Quantum Arithmetic, Runtime and Compiler

At this layer, the initial translation from a quantum algorithm towards the signals used to control the electronics controlling the qubits is performed. For this purpose, a set of quantum arithmetic operations is assumed and the target outcome of the compiler is a program that is composed of instructions defined in the QISA. Similar to traditional compilers, the runtime refers to a high-level scheduler that is needed for a larger multi-qubit or multi-node (with multiple qubits per node) system. The runtime is envisioned to be programmed using instructions from the QISA.

2.2.4. Quantum Instruction Set Architecture

The QISA can be seen as the bridge between (high-level) quantum algorithms, expressed using quantum programming languages or quantum arithmetic operations, and the microarchitecure level below. The layer defines the functionality of the quantum computer expressed as a list of quantum instructions similar to the ISA of a traditional computer. This includes instructions for the functionality of a qubit as well as instructions for the necessary control logic, including classical instructions. These classical instructions are also used by the runtime to perform scheduling between qubits. The definition of the QISA is important, as a well defined QISA allows for adaptability and flexibility of the functions of the quantum computer.

2.2.5. Microarchitecture

The goal of the microarchitecture is to execute QISA instructions generated by the higher layers in a deterministic manner with ample run-time support. It should, just like the QISA, support both quantum operations as classical operations. The microarchitecure is defined as a collection of simple(r) micro-instructions of the lowest level that implements the functionalities of the QISA by controlling the complex electronics that control qubits.

2.2.6. Quantum to Classical

The Quantum to Classical layer, also called the control electronics layer, is responsible for driving the physical qubits. It is located very close to the qubits. The layer is controlled by the microarchitecure layer. Some examples of control electronics are Digital-Analog Converters (DACs), Analog-Digital Converters (ADCs), Arbitrary Waveform Generators (AWGs) and Ultra-High Frequency Quantum Analyzers (UHF-QAs) as well as relays, optical switches and current sources. These *all* need to be properly controlled to ensure correct and full functionality of the quantum computer.

2.2.7. Quantum Chip

The bottom layer of the stack would be the quantum chip layer, or the qubits themselves. There are different types of qubit available (superconducting, spin, NV-centers and more) and each type demands different requirements from the higher layers.

2.3. NV-centers in Diamond

There are many ways to implement a qubit available today. Each type of qubit has its own advantages and disadvantages. The most known types of qubit are superconducting qubits, spin qubits, majorana-fermion qubits and diamond color center qubits [7]. In this thesis, the focus lies on the fourth qubit type: color center qubits. Within these color center qubits, the focus lies on NV-centers.

A NV-center is an open space in diamond caused by the replacement of a carbon atom with a nitrogen atom in the structure of the diamond. The open space, or vacancy, is occupied by a particle that can act as a qubit. These vacancies can form naturally or be formed by humans in artificial diamonds [10]. The needed structure can be made using different approaches. One of these approaches could be to try to “grow chemical vapor-deposited diamond onto etchable substrates” [11].

As said previously, an NV-center is a type of color center, A color center is an impurity or defect that gives emeralds and rubies their color [11]. Diamond is the optimal environment for an artificial atom because of the thermal and mechanical properties of the material. Also, there are very few nuclear-spins present in diamond. The nuclear spins that are present can even be used as an advantage.

As the particle inside the vacancy can be used as a qubit, is must be able to have superposition of quantum states. It is also important that the particle has a long coherence time and this is usually achieved by isolating the particle from uncontrolled perturbations in the form of noise [10].

In [10], it is stated that all of the previously mentioned properties are in fact satisfied by the NV-center. Today, it is widely known that NV-centers can be used to generate, measure and control qubits, but at the time of [10] this was not trivial.

These particles in the vacancy, from now on qubits, can be controlled with both magnetic as optical interactions [10]. Readout of the qubit is can be done optically. This is supported by the fact that the ground state of the NV-center shines brighter than the excited state [11]. This is one of the properties of color centers: they react strongly with microwaves and optical fields. Because of these properties, the qubit can be classified as *Optically Detected Magnetic Resonance*, or ODMR.

The created diamond and corresponding NV-center are made purer and purer, enhancing the life-time of the qubit. In 2013, [10] shows that this can be extended to the millisecond range by removing spins, both nuclear and electronic. These spins normally act as sources of decoherence. Thus, by minimizing these sources the coherence time of the qubit increases. However, not all material spins are necessarily bad. It is possible to create Quantum Memory by using the nuclear spin of the carbon atoms to store data without losing the state because of entanglement generation [12]. In particular, natural material contains spins with weak hyperfine couplings, a magnetic coupling between atoms/-electrons, between 20-50KHz. If these coupling strengths are weak enough (<10KHz), the spins can survive up to hundreds of entanglement attempts [12]. The Fujitsu Project aims to use the nuclear spins surrounding the qubit as additional qubits.

These nearby nuclear carbon-atom spins can also be used to implement Quantum Error Correction (QEC) [13]. The authors use the electron found in the NV-center to initialize, control and read out the carbon-13 spins found close to the center. They have implemented 1- and 2-qubit gates with high fidelity, as well as a three-qubit QEC [13]. This supports the statement that nuclear spins around the qubit do not necessarily influence the NV-center qubit system badly.

It is important to stress that not every NV-center is the same. Thus, not every trapped particle is the same. There are differences in brightness of emitted light and in decay rate [14]. Therefore, a metric has to be introduced which will indicate the efficiency of a NV-center. This is done in [14]. Quantum Efficiency, or QE, is quantified experimentally. They measured between 30 and 40 NV-centers and mapped their brightness. A broad distribution was found in accordance with previous reports. They found that the decay rate is influenced by two parts: Radiative and Nonradiative decay rates. NV-centers have a QE of between 10 and 90 percent [14].

Comparing the NV qubit with other types of qubits, we see that the NV electron spin lifetimes are much longer, about a couple of milliseconds even at room temperature. Other types of qubit are usually cooled down to less than 4K to be useful.

However, the biggest challenge that comes with NV-center based qubits is its scalability. The scal-ability of the qubits is very hard to develop well. This is mostly because the quality of the devices and materials is nowhere near it should be right now. On top of that, optical coupling between qubits is also needed. Optical coupling between qubits would mean they can also be coupled (and subsequently entangled) over long distances [11], resulting in a quantum network.

There are also ideas to combine multiple types of qubits into one system. [10] names the combina-tion of superconducting qubits (which are fast and high-controllable) and NV-qubits (to act as quantum memory and transducers).

Other types of color centers are also present in diamond. Research has been done to other types as NV-centers have a “large static and dynamic inhomogeneous linewidth”, [15]. Another type of color center, the negatively charged SiV-center has good optical behavior and a narrow inhomogeneous linewidth, making it a good candidate for quantum communication. However, the SiV-center has a much smaller coherence time. This can be elongated to one minute at a temperature of 4K [15]. As mentioned before, the Fujitsu Project also does research about SnV-centers.

A simulator has been proposed by [16] to learn the Hamiltonian that describes the behavior of the NV-center qubit. It consists of three parts: 1) a Silicon Quantum Photonic Simulator, 2) a Diamond NV electron spin system and 3) a classical computer. The system proves that the verification and validation of quantum systems and quantum devices can be done with a hybrid-system, which is part classical and part quantum [16].

2.3.1. Other types of Qubit

NV-centers can be used as qubits, and can be controlled using lasers, magnetic fields en electric fields (ODMR). This is quite different from other types of qubit and also what makes the technology special.

For example, the decoherence time of NV-centers is longer than superconducting qubits.

Superconducting is perhaps the most popular and therefore most explored type of qubits. Sycamore, the quantum computer from Google, uses 53 of these type of qubits [5]. The qubits, as the name suggests, consist of superconducting electrodes that are interconnected with Josephson-Junction. Within the superconducting qubits, there are different types. Examples are: charge, flux, phase and fluxonium. At QuTech, the transmon qubit is used. There are two islands, interconnected with a junction. This junction is comparable with an LC-oscillator. Surface code is an example of an application of transmon qubits. It uses bus-resonators, readout-resonators and microwave-inputs to control the qubits [7].

Spin qubits are more comparable to NV-centers than superconducting qubits. Spin qubits also utilize the spin of an isolated electron to do calculations. Charge sensing can be used to detect if an electron spin is present. The space wherein a spin qubit is present is called a quantum dot. Two quantum dots can be cross coupled, but that has as a disadvantage that the voltage applied to dot #1 also influences the potential of dot #2. Instead of control with lasers, the control is now done with alternating magnetic fields. When this alternating field is introduced, the spin starts to rotate as a function of time. The oscillations are called Rabi-oscillations. These are important as they form the basis of single qubit rotations. The oscillations decay because of nuclear spins present in the environment, and so the frequency of the qubit changes over time. Silicon-28 has 0 nuclear spin, thus there is no decay of oscillations. Together with sequences and other materials an increase of the coherence-time of a qubit is possible [7].

2.4. Quantum Computer Microarchitectures

Because there are different types of qubits, there must also be different ways to control them, either because the underlying qubit requires that or because the design philosophy is different. In this section, a few quantum computer microarchitectures will be highlighted. Note that, because there is a discrepancy between the definitions of microarchitecture used in between these examples, the definition might not add up completely to the definition set in Section 2.2.

2.4.1. QuMA

QuMA, or **Qu**antum **M**icro **A**rchitecture [17], is a heterogeneous microarchitecture aimed at the control of superconducting qubits. There is a host classical CPU, with a quantum co-processor. There are, essentially, three versions of QuMA. The first version, called QuMA, is based on four concepts at the core: codeword-based event control, Queue-based event timing control, multilevel instruction decoding and QuMIS, the **Qu**antum **M**icro **I**nstruction **S**et. The other versions of QuMa, QuMa_v2 (featuring eQASM microarchitecture) and QuMa_FT are discussed later.

Codeword-based event control

In QuMa, every event is indexed with a codeword. The events are then triggered by the codewords, improving control flexibility. Additionally, there is less memory usage. For the *AllXY*-experiment that serves as a benchmark for QuMA, there is an 83% reduction in used memory.

Queue-based event timing control

Events are buffered in a group of queues. Then, the events can be triggered at the expected timing resulting in deterministic and precise execution under non-deterministic timing.

The concept is quite simple. Each operation has a time label. A timeline can be created, going from left to right, where different operations have their own time label. Once the system clock T_d has reached the correct time on which the operation with time label (x) has to be executed, it executes the operation and resets the system clock for the next event.

Multilevel instruction decoding

Because the preceding two concepts already handle a lot of things, other parts of QuMA can focus on flexibly decoding instructions. In addition, this part also focuses on filling the queue as fast as possible.

Quantum instructions are defined in the Quantum Instruction Set (QIS). It contains classical and quantum instructions, where the classical instructions are used for basic arithmetic and logic operations and program flow control. The quantum instructions determine when and what operation is applied on which qubit.

Multilevel instruction decoding supports technology-independent QIS definitions. At the core, quantum instructions are decoded into quantum micro-instructions, micro-operations and eventually codeword triggers.

Quantum Micro Instruction Set
QuMIS is the microinstruction set that is used within QuMA. it has four different instructions:

1. Wait: wait between consecutive time points.
2. Pulse: apply quantum gates on qubits.
3. MPG: generate the measurement pulse.
4. MD: trigger the measurement discrimination process.

The first version of QuMA is no longer used, because there are a flaws in the architecture that prohibit the architecture from functioning optimally. There is no feedback based on qubits measurement results, there is limited scalability because of a low instruction information density and it is limited in flexibility because of the tight bound to the hardware. Because of these flaws, QuMIS can **not** be defined as a QISA [17].

This is why Fu proposed a new version featuring eQASM, the executable quantum assembly [17, 18]. Under the definitions set up in Section 2.2, eQASM is an example of a microarchitecture. eQASM has support for the following:

- Runtime Feedback: Two kinds of feedback are supported within eQASM: *fast conditional execution* and *comprehensive feedback control (CFC)*.
- Operational Implementation: the definition of eQASM lies on the assembly level and basic rules of mapping assembly to binary.
- Increased **Q**uantum **O**peration **I**ssue **R**ate (QOIR): By adapting a VLIW architecture in addition to Single-Operation-Muliple-Qubit (SOMQ) execution, alleviating the Quantum Operation Issue Rate (QOIR).
- Configurable QISA at compile time: eQASM allows the user to configure quantum operations at compile time instead of at the ISA design stage.

Overview
Quantum computing can be compared with computing in for example OpenCL. OpenCL is a open industry standard for programming for GPUs. Following this heterogeneous computing mode, we can also do do the same for a quantum computer, or quantum accelerator. A host program, typically written in Python or C++ is used with kernels written in OpenQL, a quantum compiler, for the quantum part of the program. The kernels are compiled to eQASM instructions, which are then loaded into the QPC by the host CPU and executed. eQASM is defined by five principles, [17, p. 77]:

1. "eQASM should include classical instructions to support quantum program flow control including runtime feedback;
2. eQASM should contain well-defined methods to specify the timing of quantum operations;
3. Low-level hardware information should be abstracted away from the eQASM assembly as much as possible to avoid eQASM being stuck to a particular hardware implementation;
4. The quantum operation issue rate is a potential bottleneck of the quantum microarchitecture, and should be addressed, e.g., by densely encoding the instructions such as done with SIMD and VLIW for classical architectures;
5. Different experiments and radical compiler-based optimization techniques such as quantum optimal control [144, 145] may use a different set of quantum operations, which can be uncalibrated or uncommon. eQASM should be flexible to allow different quantum operations via configuration."

The architectural state of the QPU (Quantum Processing Unit) is shown in Figure 2.6. The **Data Memory** buffers intermediate compute results and acts as the communication channels between CPU and QPU. Instructions are obviously stored in the **Instruction Memory** while the **Program Counter (PC)** keeps track of the instruction addresses. **General Purpose Registers (GPRs)** are 32-bit registers, starting with `r1`. The comparison result from comparison and branch instructions are stored in the **Comparison Flags**. There are two types of **Quantum Operation Target Registers**: single qubit registers for single-qubit operations and two-qubit registers for two-qubit operations. The physical address

of a set of qubits is stored in a target register. Same as with QuMA, eQASM features a queue-based timing control scheme, splitting the process into two domains: deterministic and non-deterministic timing. This is done in the **Timing and Event Queues**. For storing the measurement results of the qubit, a set of **Qubit Measurement Result Registers** are available. There is an **Execution Flag Register** for quick conditional execution. The **Quantum Registers** which serve as a collection of all physical qubits. A unique index is assigned to each qubit called the physical address.

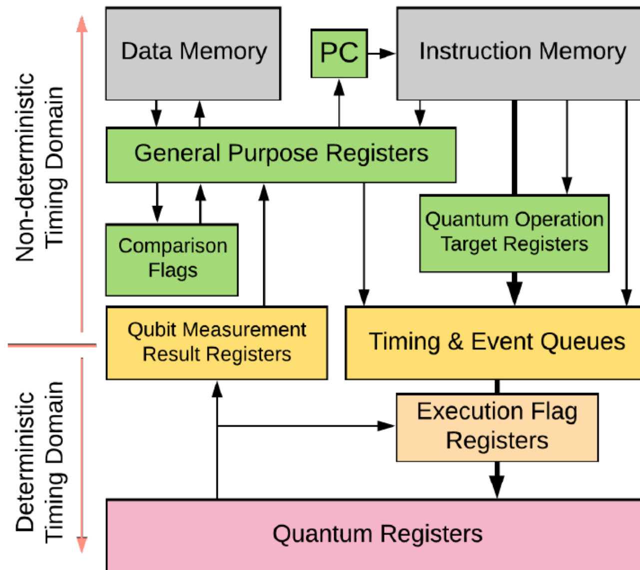


Figure 4.2: Architectural state of eQASM. Arrows indicates the possible information flow. The thick arrows represent quantum operations, which reads information from the modules passed through.

Figure 2.6: Architectural state of eQASM [17, Figure 4.2]

An important part of eQASM is that one of the goals is that it is technology independent. This means that eQASM is directed at the assembly level with rules of mapping the assembly code to binary. eQASM enables freedom for the design of the microarchitecture to enhance performance or usability.

A program for eQASM consists of quantum and classical operations. An overview of all instructions can be found in Figure 2.7.

Based on these instructions, eQASM has four features, as also mentioned previously: 1) eQASM can specify the timing for each quantum operation explicitly, 2) The quantum operations are defined by the programmer at compile time, meaning there is more flexibility for compiler-based optimization and, for example, different technologies, 3) Single-Operation-Multiple-Qubit with VLIW architecture and 4) Two kind of feedback - fast conditional for go and no-go decision and CFC for redirecting program flow.

Queue-based Timing Control eQASM also supports an increased QOIR by combining a wait statement and a quantum operation onto one instruction: `[PI,] <Quantum Operation>`, where the content between [...] is optional.

Quantum Operation Definition and Decoding The quantum operations are defined at compile time. This means that the programmer can choose what gate set to use and more. The only thing that needs to be done, is that the assembler translates the quantum operation to the expected opcode and then the microcode unit translates the opcode to micro-instructions using the decoding scheme [17]. These are then translated into pulses by the pulse generator - which applies the operation on qubits with precise and deterministic timing. All units are configured at compile time.

Table 4.1: Overview of eQASM Instructions.

Type	Syntax	Description
Control	`CMP Rs, Rt`	Compare GPR `Rs` and `Rt` and store the result into the comparison flags.
	`BR <Comp. Flag>, Offset`	Jump to `PC + Offset` if the specified comparison flag is '1'.
Data Transfer	`FBR <Comp. Flag>, Rd`	Fetch the specified comparison flag into GPR Rd.
	`LDI Rd, Imm`	Rd = sign_ext(Imm[19..0], 32).
	`LDUI Rd, Imm, Rs`	Rd = Imm[14..0]::Rs[16..0].
	`LD Rd, Rt(Imm)`	Load data from memory address `Rt + Imm` into GPR `Rd`.
	`ST Rs, Rt(Imm)`	Store the value of GPR `Rs` in memory address `Rt + Imm`.
	`FMR Rd, Qi`	Fetch the result of the last measurement instruction on qubit `i` into GPR `Rd`.
Logical	`AND/OR/XOR Rd, Rs, Rt` `NOT Rd, Rt`	Logical and, or, exclusive or, not.
Arithmetic	`ADD/SUB Rd, Rs, Rt`	Addition and subtraction.
Waiting	`QWAIT Imm` `QWAITR Rs`	Specify a timing point by waiting for the number of cycles indicated by the immediate value `Imm` or the value of GPR `Rs`.
Target Specify	`SMIS Sd, <Qubit List>` `SMIT Td, <Qubit Pair List>`	Update the single- (two-)qubit operation target register `Sd` (`Td`).
Q. Bundle	`[PI,] Q_Op [| Q_Op]*`	Applying operations on qubits after waiting for a small number of cycles indicated by `PI`.

Figure 2.7: Overview of eQASM microarchitecure instructions [17, Table 4.1]

Address Mechanism Because of SOMQ, we can apply one operation on multiple qubits at once. This is useful if we want to initialize the qubits into superposition, which is done by applying the Hadamard gate on multiple qubits. SOMQ is comparable with SIMD in classical computing.

Very Long Instruction Word In addition to SOMQ, eQASM also allows to apply different operations on different qubits, in parallel. This is written as a *quantum bundle* in VLIW:

```
[PI,] <Quantum Operation> [| <Quantum Operation>]*
```

Where | is the separator between operations, and * indicates how many times the operation between [...] can be repeated. Also see the example in [17, p. 86] on SOMQ and VLIW instructions.

Fast Conditional Execution When doing conditional operations, for example branch operations, classical computing uses methods to speed up branch operations. Branch operations are the most time-intensive operations in for example MIPS. To combat the length of these operations, Fu *predicts* the outcome of the branch based on previous results - called branch prediction.

In eQASM, something similar is done. When the result of the check is stored in the execution flag, the next operation is either executed or canceled. The execution flags are updated automatically through the combinatorial logic circuit when the result is read from the qubit. An example can be found in [17, p. 87].

Comprehensive Feedback Control (CFC) CFC can adjust program flow. It does this based on the measurements of qubits. The advantage is that there is flexible user-defined feedback but the latency on the feedback is larger. CFC has three steps: 1) Measurement instruction on qubit i, 2) Fetch the value of the measurement to a GPR, 3) The GPR is used in a branch instruction in the CPU. Example in [17, p. 88].

Hardware architecture

eQASM has been adopted to work with the first version of QuMA, featuring codeword-based event control, multilevel instruction decoding and queue-based timing control. QuMA_v2, as the new architecture is named, supports these three features including all of eQASM. The architecture can be found in Figure 2.8. On details on what each part does, refer to [17, sec. 4.4.3].

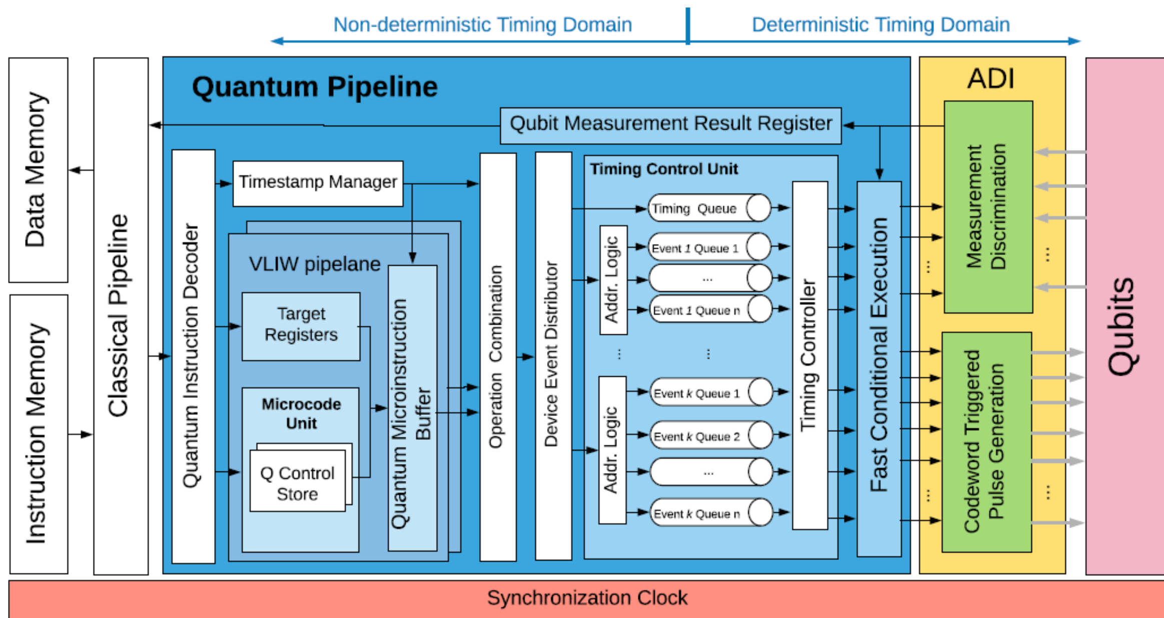


Figure 4.9: Quantum microarchitecture implementing the instantiated eQASM for the seven-qubit superconducting quantum processor.

Figure 2.8: Overview of QuMA_v2 [17].

2.4.2. QuTech Central Controller

The QuTech Central Controller, or QCC, is a system controller based on QuMA [17], controlling a 17-qubit surface code quantum processor based on superconducting qubits [19]. Moreira analysed QuMA to find weaknesses. By eliminating the weaknesses, he could improve upon QuMA and adapt it to the 17-qubit architecture.

The disadvantages that are found in QuMA are:

- Centralized Design makes it so that there is limited scalability.
- Control and scalability challenge because there is a lack of full integration support due to the number of available inputs and outputs.
- QuMa is not flexible because of the rigid control scheme. Changes in chip size, layout or instrument configuration are not supported.

In the adaption, for unknown reason, the platform was first ported to a new hardware platform featuring an Xilinx Zynq-7000 SoC. The second adaption needed to make QCC ready for 17 qubits. QuMA did not support such a high number of qubits, so naturally some things needed to be changed. First and foremost, the instructions SMIS and SMIT in eQASM needed to be enlarged to fit the addresses for 17 different qubits. Luckily, SMIS was big enough to accompany the increase of qubit. SMIT on the other hand, required an expansion of 32 two-qubit target registers because of the increased amount of two-qubit pairs.

Additionally, the part that is responsible for the distribution of events among the devices needed to be expanded. In total, there are now five AWGs and three UHFQCs. Because of the change of topology, several other systems also had to undergo changes.

QuMA was quick enough to not throttle under the *Quantum Operation Rate Issue*-requirements of the 17-qubit architecture. However, the increased number of connected devices means that 288 pins are now needed to connect everything. The FPGA that is used, however, only supports 48 I/O pins. Moreira solved this by connecting the main (CORE) FPGA with other, similar, FPGA boards in a star-configuration. They are connected using a custom backplane, which allows high-speed serial communication between the I/O boards and the CORE FPGA-board.

As to date, the system architect of the Central Controller, Wouter Vlothuizen, wrote that both QuMA and QCC are no longer in use at QuTech. The main architecture is the Central Controller, which is highlighted in the next section.

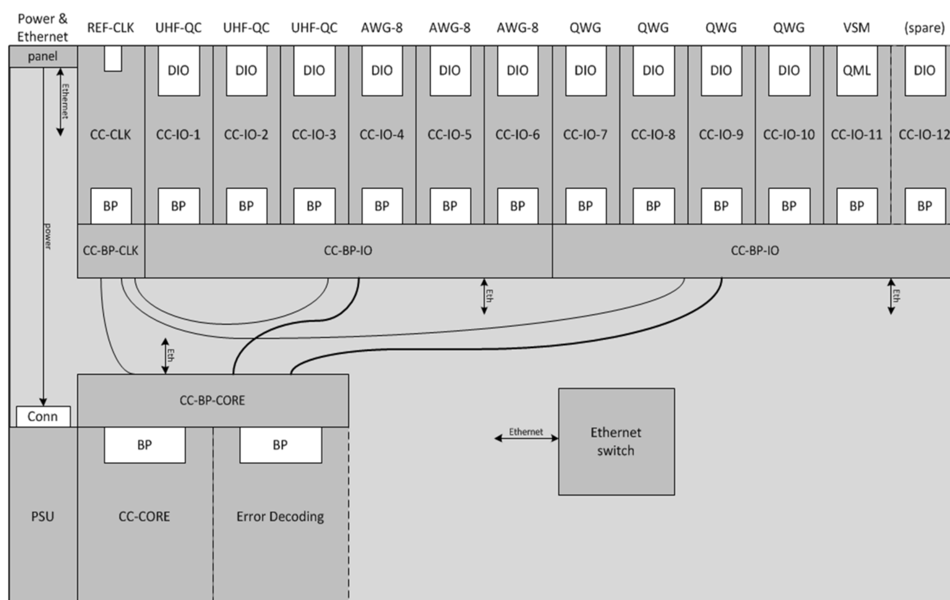


Figure 2.9: The distributed hardware architecture of the Central Controller (CC). Provided by Wouter Vlothuizen, TNO.

2.4.3. Central Controller

The Central Controller (CC), not to be confused with he QuTech Central Controller, is a *distributed* control architecture used at QuTech, developed by TNO, TU Delft, Zurich Instruments and ETH-Zurich. It aims to control transmon qubits, which are a form of superconducting qubits. The goal is to build a surface-17 quantum error correction logical qubit. Part of the development of the Central Controller was the CBox - a master controller with 3 built in AWGs using QuMA_v1. The second generation, CC-light, featured a master controller running QuMA_v2 with multiple external instruments (AWGs, UHFQC). The third and most recent generation is the Central Controller (CC) as presented in this section. All information in this subsection is gathered through emails and presentations with and from Wouter Vlothuizen from TNO.

Hardware Architecture

The CC is designed for 17 qubits, but can be scaled to 49 qubits and even beyond that. It features a distributed hardware architecture with 12 CC-IOs and a CC-CORE. An overview can be found in Figure 2.9. A CC-IO consists of an FPGA + ARM processor. Every CC-IO has one connected instrument - this can be either an AWG, a UHF-QC or a Vector Switch Matrix (VSM). The CC-CORE also features an FPGA and an ARM processor, but acts as a communication hub for low-latency communication.

Contrary to the architecture of eQASM/QuMA, the CC knows absolutely nothing about quantum operations. It only knows about the waveform generators and the signals that it needs to send out. In CC, it is determined in the software and thus not the hardware what signals go where, and then they are executed by the CC-IO FPGAs. The system achieves its scalability by just adding more CC-IO modules - no changes in firm- or software are needed. For 17 qubits, approximately 10 instruments are needed whilst for 49 qubits, approximately 20 instruments are needed. Theoretically, no changes in the firmware are needed if the qubit chips are updated, a new qubit technology is hooked up to the controller, new instruments are added etc. This is because the knowledge about the system is located in the compiler. However, new technologies might require signals that are not supported by the CC.

Each CC-IO board features a Sequence Processor (SP). Each instrument, whether is is an AWG, an UHG-QC or a VSM, needs a SP to function properly. SPs operate from independent, but coordinated programs with a synchronous start. An SP has an instruction size of 64-bit, with 65 32-bit wide registers. It supports arithmetic, flow control, co-processing and sequencing operations. The core runs at 200 MHz whilst the sequencer clock runs at 50 MHz. The SPs have cycle accurate synchronization of the distributed processors. A patent has been filed for this technique. There is fast feedback through the shared memory. This memory is shared between all CC-IO boards in a star-configuration. Writes

are distributed to all SPs, while reads are performed locally. In the star-configuration, the CC-CORE acts as the network switch.

The Q1 Instruction Set

Here, the instruction set that is used withing the CC-IO and CC-CORE are described. Below, in Table 2.1, the operand of the instruction set is shown. This information was directly gained from the instruction set file of the CC, `Q1instructionSetCC.rst`, that was received from Wouter Vlothuizen. Note that this file is not complete as there are several “FIXME”s written in the document. However, it should give a good overview of the instructions of the CC.

In Table 2.2, the actual instructions can be seen. They are divided over multiple categories, of which the most notable two are at the bottom of the table: q1sec-0 and q1qec-0. There is a difference between these two categories and they do not run on the same processor. There are two types of processors, with one type running the instruction set with the q1sec-0 instructions and the other with the q1qec-0 instructions. In addition, to clarify the instructions more, the instructions starting with ’seq’ are instructions that are executed in the sequencer part of the CPU. The ’swreq’ instructions need action from the ARM processor. The other instructions are for flow control, arithmetic etc as indicated in Table 2.2.

Comparing this instruction set with the one from QuMA (eQASM), it can be seen there are vast differences in how they operate. This is entirely due to the fact that eQASM has a centralized architecture, while the CC is decentralized.

Table 2.1: The operands that are used in the Q1 instruction set as gained from the instruction set file by Wouter Vlothuizen.

Name	Description
Rs, Rd, Rn	Source and destination registers. Varying from R0 through R63.
Sn	Shared memory address. These range from S0 through S127 and are byte addressable.
addr12	A 12-bit direct address or lower 12 bits of a register.
src32	Source operand with a 32-bit immediate value or register.
imm32	32-bit immediate value.
cnt32	
val32	
sizeX	

Table 2.2: Q1 Instruction Set - based on cc_firmware_implementation.docx - Wouter Vlothuizen

Category	Instruction	Description
General	illegal	causes an illegal instruction exception, e.g. for empty memory locations since opcode is 0
	stop	Stop instruction
	nop	No-operation instruction
flow control	jmp addr12	pc = addr12
	jge Rs,imm32m,addr12	if(Rs >= imm32) pc = addr12
	jge Rs,imm32,addr12	if(Rs <imm32) pc = addr12
	loop Rn,addr12	if(–Rn != 0) pc = addr12
arithmetic	move src32,Rd	Rd = src32
	add Rs,src32,Rd	Rd = Rs + src32
	sub Rs,src32,Rd	Rd = Rs - src32
	not src32,Rd	Rd = !src32
	and Rs,src32,Rd	Rd = Rs & src32
	or Rs,src32,Rd	Rd = Rs
	xor Rs,src32,Rd	Rd = Rs ^src32
	asl Rs,src32,Rd	Rd = Rs<<src32
	asr Rs,src32,Rd	Rd = Rs>>src32
coprocessing	sw_req src32	software request to ARM processor, src32 is passed
shared memory	move_sm Rd	Rd = 32-bit shared memory element as read by the "seq_cl_sm" instruction. Blocks until data is available.
real–time	seq_wait cnt32	wait cnt32 cycles
	seq_inv_sm Sn,size	locally invalidate a region of shared memory from address n (byte aligned) with size size. Reads from invalidated memory flag an error; use this mechanism to guarantee that late arrival of results is flagged
	seq_bar	synchronize all participating modules (CCIOs, CCCORE) using the MOD_SYNC party-line, and then wait for the number of cycles set by Q1 configuration register "SEQ_BAR_CNT" (which is set using CC SCPI command setSeqBarCnt)
ipc	seq_cl_sm Sn	output 32-bits SM element at address n (32-bit aligned) to the classical part of the Q1 via a FIFO, so that the Q1 can read it using "move_sm"
	seq_sw_sm Sn	reads a 128-bits shared memory element at address n (128-bit aligned), latches it into a FIFO buffer together with the Q1 program counter and generates the corresponding interrupt so that ARM processor software can read it.
real–time: q1seq–0	seq_out val32,cnt32	output the digital value val32 for cnt32 cycles
	seq_in_sm Sn,mux,s	read the digital input FIFO, pass the value through multiplexer mux, and distribute the result to shared memory address n (size aligned) with transfer size determined by s (0=8-bit, 1=16-bit, 2=32-bit). If the input FIFO is empty an error is flagged. Writes from different instruments to overlapping regions should be prevented.
	seq_out_sm Sn,pl,duration	read a 128-bit shared memory element at address n (128-bit aligned), pass it through programmable logic pl and output the result for duration cycles. Also updates PL state.
	seq_state_sm Sn,pl	read a 128-bit shared memory element at address n (128-bit aligned) and pass it through programmable logic pl, thus updating PL state. Does not perform output
	seq_state val32	set the state register in Programmable Logic to val32
real–time: q1qec–0	seq_init_a_qb src32	write src32 into the QEC as initial ancilla state
	seq_init_a_qb_sm Sn,lut	write 128-bit shared memory element at SM address n (128-bit aligned) through the multiplexer into the QEC as initial ancilla qubit state. FIXME: lut
	seq_a_qb Sn,lut,cnt32	write 128-bit shared memory element at SM address n (128-bit aligned) through the multiplexer into the QEC as ancilla qubits and wait cnt32 cycles
	seq_d_qb Sn,Sm,s,lut,cnt32	write 128-bit shared memory element at SM address n (128-bit aligned) through the multiplexer into the QEC as data qubits and wait cnt32 cycles; FIXME: SM address m (32-bit aligned), size

Software Stack

For the software stack, there are four steps, starting at the quantum algorithm and ending at the binary program that will be executed by the SP.

1. Add timing to the quantum gates. This step is done in OpenQL on the control PC.
2. Convert the quantum gates to codewords that can be used by the instruments. This is done in the OpenQL CC-backend on the control PC.
3. Then, the programs are compiled and distributed to the CC-IO boards by the CC-CORE compiler, that is running on the ARM processor of the CC-CORE board.
4. Ultimately, the programs are assembled on the CC-IO assemblers running on the CC-IO ARM processors.

2.4.4. CC-Spin

CC-Spin [20] is a microarchitecture that focuses on controlling spin qubits. Spin qubits are yet another type of qubits realized by quantum dots. Inside a quantum dot, there is an electron that spins just like it does in NV-centers. However, the control of the qubits in CC-Spin is very different from e.g. superconducting and NV-center qubits. First and foremost - the design philosophy of the CC-Spin microarchitecture looks a lot like how it's done with QuMA [17]. There is a part cQASM, that is technology independent, and a technology dependent part eQASM.

CC-Spin QISA

The QISA of CC-Spin has a set of requirements [20]. First and foremost, it must include both classical as quantum instructions with run-time feedback. Run-time feedback is necessary for implementation of decisions based on measurement results. Additionally, the timing constraints should not be present in the QISA and it should also be hardware independent. Ideally, the QISA should also be qubit-technology independent. Additionally, the instructions should be dense enough so that the *Quantum Operation Issue Rate* does not arise. Ultimately, there should be enough flexibility to accommodate different sets of operations - including complex waveform generation.

In CC-Spin, the *general* 32-bit QISA instruction format has specified four things: 1) the instruction type (1 bit), 2) the opcode (9-bits), 3) the quantum channel mask, which tells what qubit is at which address (8-bits) and the 4) payload, or the information that needs to be sent, such as amplitude, phase and frequency (14-bits). An example can be see in [20, p. 28-29]. Yadav introduces two QISA instruction format designs. In the first design, the number of payload bits have been reduced from 14 to 12 bits. The two free bits are now used to create a mask index, that is fed together with the channels mask into a simple decoder to enlarge the channel mask from 8-bits to 32-bits, enabling the control of 32 qubits simultaneously. In the second design, the channel masks are put into a lookup table (LUT). This method requires more memory but can be done in a single cycle. In addition, there are 256 qubit channels available. This format also supports the SMIS and SMIT instructions from [17].

Ultimately, however, the LUT approach was too complex for the master thesis project of Yadav, and the general QISA instruction format was chosen without explicit definition of the payload parameters.

Hardware of CC-Spin

The design of the architecture was divided into two areas: *Datapath Design* and *Control Unit Design*. The definition of functional blocks, such as GPRs, Bus sizes etc, are done in the Datapath design whilst the control unit design focuses on flow control - instruction registers, decode logic, program counters etc. The control unit is also responsible for translating each QISA instruction to microarchitecture-operations that can be executed and ultimately sending signals to the control electronics. In Figure 2.10, the overview of the hardware architecture of CC-Spin can be seen. The flow begins at the left. At the host PC, the QISA is generated (with OpenQL), then forwarded to the Master Controller. The Master Controller then translates the QISA operations to micro-operations (or micro-instructions), which get send through the SERDER (Serializer-Deserializer) to slave FPGAs. The slave FPGAs are connected to the AWGs - which are responsible for the envelope and waveform generation. The waveforms are then sent to the Analog-to-Digital interface.

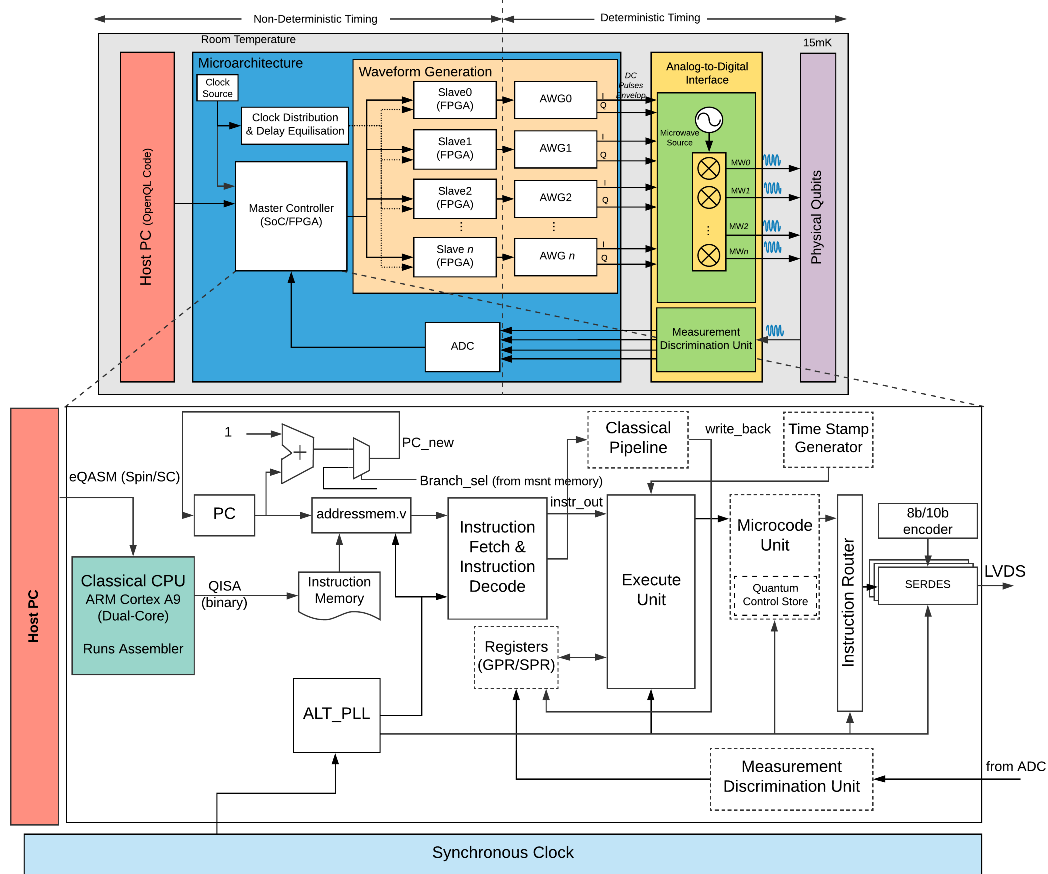


Figure 2.10: The system view of CC-spin, using AWGs and/or DDS-DAC [20, Figure 3.2].

2.5. Quantum Networking

If the goal is to build large quantum computing systems, communication between quantum computers is needed. This means that a network is needed between these so-called nodes. When nodes are connected together using quantum communication, we speak of a Quantum Network.

This network can be used to connect large compute systems together, but also to connect smaller nodes together in one system. As seen with QuMA [17], an idea is to make multiple nodes consisting of tens of qubit and connect them together to make one powerful quantum computer. This is sometimes called a modular quantum computer or scalable distributed quantum computer, comparable with the Fujitsu Project.

2.5.1. Quantum Communication

As stated previously, quantum communication can be used to link quantum computers or quantum nodes together. There are two big reasons for the need of a quantum communication network, of which the first has been mentioned already: using multiple nodes or multiple computers to solve a problem that needs more compute power. The second reason is that a quantum network supports tasks that a classical network can not [21]. Examples are secure communications, secure cloud computing and quantum-enhanced measurement networks [22]. Quantum Key Distribution (QKD) is the best known application for such networks.

To realize both types of networks, a few challenges have to be solved. However, the amount of challenges and the details of these challenges are unknown. There are at least three important challenges, being transmission losses, decoherence and no-cloning [22]. Construction requirements for large-scale quantum networks do not exist yet.

Quantum networks consist of both classical and quantum hardware. Both types of hardware need

to be integrated together very precisely so that the network works optimally. This calls the need for a new quantum network stack. Unfortunately, the definition of this stack is challenged by technological limitations. Moreover, protocols need to be defined as done in [22]. However, despite these limitations, QuTech's quantum networking group led by Stephanie Wehner has done research in the field of quantum network stacks, operating systems and architectures.

Further insights in quantum networks and communications have been found using ad-hoc studies [21]. The effort required to set up the conditions of the ad-hoc simulation is very high, and thus the method is tiresome. To gain more knowledge about quantum networks and quantum networking, researchers have investigated how quantum communication works. For example, QuTech made a quantum network simulator called NetSquid [21]. It can be used to predict and test the performance of both quantum networks as modular quantum computers. There are many other tools available with different goals in mind.

2.5.2. Entanglement and Teleportation

As shown by [23, Figure 1] and elaborately described in [8], teleportation and entanglement are closely related. Essentially, when entanglement between two nodes has been established, one can send over a data qubit. This will, however, consume the entanglement pair. If one wants to send over another data qubit, entanglement has to be created again.

According to [8], "Entanglement is a uniquely quantum mechanical *resource* that plays a key role in many of the most interesting applications". When two particles are entangled with each other, they are 'linked' by quantum channels. So, teleportation of quantum states is a utilization of quantum entanglement. With teleportation, the state can be measured at point A and then re-made at point B based on the measurement values of the entangled states. For more on teleportation, refer to [8, Sec. 1.3.7]. Entanglement and teleportation solve the challenges of losses and no-cloning [22].

2.5.3. Quantum Tools for Networking

[24] describes what the theoretical limits for quantum entanglement are from a mathematical perspective. This is useful when comparing different network simulators. It also describes how to estimate waiting times and fidelity for quantum networks. Then, in [24, sec. IV], the author describes simulation tools including NetSquid, a quantum network simulator made by QuTech. The goal of the paper is to review the research dedicated to analytical and simulation tools for quantum network communication, making a unified description.

Generally speaking, there are three choices of evaluation: experimental, through simulation and analytical. The choice of the method depends on factors such as cost, accuracy and flexibility [24] and each choice has its drawbacks and advantages. For example, experimental methods give the most accurate answers to concrete scenarios. Such methods are costly and if the hardware does not exist yet, evaluation is not possible. Both simulation and analytical methods do not have these drawbacks, but they are less accurate. They are, however, less time consuming and cheaper.

Simulation Tools

Analytical models are well suited for smaller and simpler models, but they evaluate them at different points, such as protocols, topology and network usage [24]. Compared to analytical tools, simulation tools are used in more complicated scenarios. Simulation of network usually undergoes four steps [24]:

1. Problem Formulation: Definition of the goal of the performance evaluation.
2. Theoretical Modeling: Definition of the boundary conditions, assumptions and the level of detail for protocols and models. A full specification, including behavior of elements and their interactions, is the output of this step.
3. Implementation: The model that was specified in the previous step is written in software.
4. Simulation: Run the simulation and verify the outcome.

The main limitation for simulation is the network size. When simulating quantum networks classically, only networks consisting of a few bits can be simulated. However, since many interesting scenarios only consist of Clifford gates with Pauli noise, classical simulation is still viable. Classical simulation is also viable for small-qubit sized networks.

At the time of writing, there are multiple platforms available that simulate quantum networks. SimulaQron [25] and QuNetSim [26] both focus on the facilitation of network development, where QuNetSim

is more aimed at ease of implementation. It uses the an OSI-inspired model and is used to explore routing schemes [24]. SimulaQron can simulate the network by having the different nodes at different classical computers. It is used to explore the verification of entanglement and other protocols. SQUANCH, the Simulator for Quantum Networks and CHannels [27], is tailored to simulate quantum information network protocols with realistic noise.

QUantum Internet Simulator Package (QuISP) [28] captures the complete complex quantum network behavior. It is a module of OMneT++, which is an open-source classical network simulator. The main mode of operation of QuISP is tracking down Pauli errors. Simulator of QUantum Network Communication (SeQUeNCE) [29] aims at accurate physical simulation. NetSquid [21] also does this. NetSquid has shown that its modular design and quantum engine makes it possible to simulate up to one thousand of nodes for *some* models [21, 24].

2.6. OpenQL

OpenQL is used when programming quantum computers. It converts an algorithm written in a higher-level language, such as cQASM or OpenQL (C++/Python), to an form of executable code, which can be cQASM or the diamond microcode. This executable code can then be run on the corresponding quantum accelerator. In other words: “The objective of an OpenQL compiler is to produce an output external representation of the input program that satisfies the needs of what comes next”, [30].

An overview of the basic operation of OpenQL can be found in Figure 2.11. The input on the right is fed into the framework together with the configuration file, which contains information about the quantum computer, such as: the number of available qubits, the primitive gate set, topology of the target. The configuration file is crucial to the OpenQL framework, as without it, there is no way of the framework knowing the compilation target. The input undergoes a series of compiler passes, which can be defined as an updater to the quantum internal representation, or QIR. The QIR is key to understanding how OpenQL works. There are different types of passes such as a mapper, a scheduler and a Clifford optimizer. For the full list, see [30].

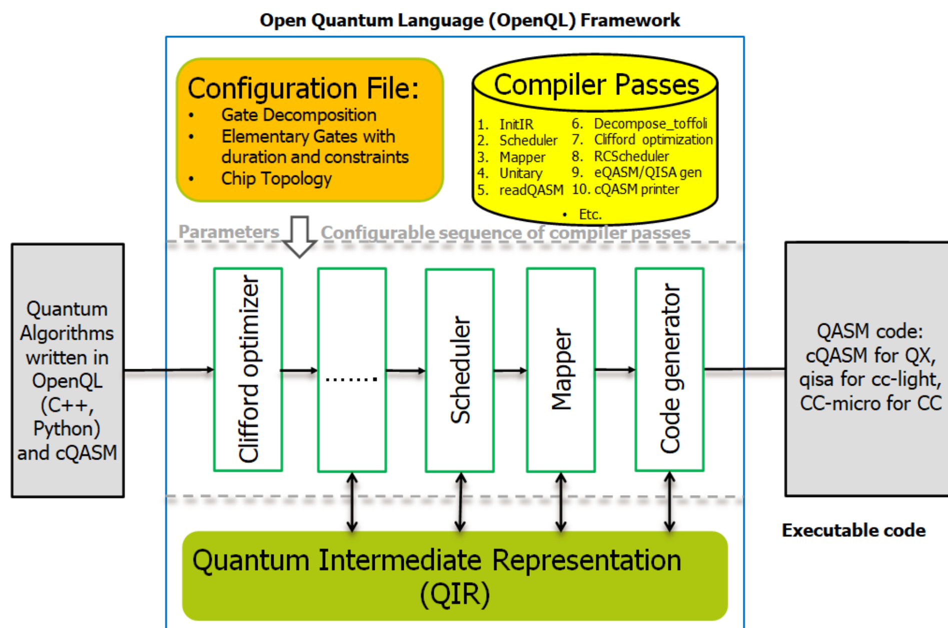


Figure 2.11: A schematic overview of the OpenQL framework. Source: Hans van Someren

Currently, the OpenQL framework has several uses at TU Delft. The starmon experiments as well as the Quantum Inspire back-end for the starmon accelerators use OpenQL.

As for current development, the developers are looking at the possibility to parallelize the compiling so that OpenQL can run more efficiently on multi-core processors. In addition, a second version of cQASM, called cQASM 2.0 is being developed and support for that is also in the OpenQL roadmap.

2.7. Conclusion

In this chapter, we introduced the reader to some specific topics needed to understand the rest of the thesis. We started with the introduction of quantum information theory in Section 2.1. We explained qubits, quantum gates and quantum algorithms. In Section 2.2, the quantum computer stack, that serves as a lead thread in the quantum computer design process, was discussed. Each level was presented briefly and definitions were made for each layer. Then, in Section 2.3, NV-centers were discussed. NV-centers are discussed because the Fujitsu Project focuses on color center qubits, which NV-centers are. It is made clear what they are, how they can be made and how they can be controlled. This is especially useful, as in Section 2.4 the different quantum computer microarchitectures were presented. We saw that there are a few of them, but none of them are aimed towards color center qubits in diamond. However, a lot of information can still be used to work towards the goal of a color center quantum computer. In the next section, Section 2.5, an overview of quantum networks was given. Quantum networks are important, as when we are scaling the Fujitsu Project computer, it feels and behaves a lot like a network. We learned about why quantum networks and quantum internet are important as they enable applications such as QKD and quantum device communication. Finally, in Section 2.6, we introduced OpenQL, the platform for high-level quantum programming developed by QuTech. OpenQL aims at translating a technology independent algorithm to a technology dependent assembly code.

3
QISA and Microarchitecture

In this chapter, the definition process of the microarchitecture is discussed. It will start with the requirements in Section 3.1. From there, the overall system architecture (as to date) is discussed briefly in Section 3.2. The signals needed by the layer below the microarchitecture layer (control electronics) are identified and operations are mapped in the operation flow chart. This is presented in Section 3.3. Later on, the microarchitecture is refined using the overall system architecture and the flow chart. Ultimately, the microarchitecure is decomposed from the instruction set architecture (ISA). The microarchitecture forms the lowest level of instructions before signals are sent to the control electronics. After the presentation of the QISA and microarchitecture in Section 3.4, an example is given on how to read them, in addition to the decompositions from QISA to microarchitecture.

3.1. Diamond Microarchitecture Requirements

As stated in the introduction of this chapter, the microarchitecture is responsible for controlling the control electronics. This means that it is responsible for sending the correct signals at the needed moment to the expected destination. Failure to do so will result in unwanted behavior, such as wrong application of gates, false measurements and more that will result in incorrect algorithm execution.

A detailed list of what the microarchitecture needs to support and why can be created:

1. **Deterministic and precise control of the control electronics.**

   This is needed to ensure that the qubit is excited with the correct signals at the expected time. If this does not happen, then wrong operations will be acted out on the qubit. In addition, the instructions of the microarchitecture should be of the lowest level possible. There should be no possibility for decompositions in the microarchitecture instructions.

2. **Classical instructions for flow control.**

   Flow control is needed because the diamond operations require checks with branches, loops and other arithmetic operations. This automatically requires flow control. Moreover, the microarchitecture needs to perform classical calculations. Therefore, classical instructions are also needed.

3. **Basic quantum gate instructions (X, Y, Z, H, S and T).**

   Without basic quantum gate instructions, the microarchitecture cannot execute basic quantum algorithms. The ability to support quantum gates is trivial.

4. **Additional gate instructions according to cQASM 1.0, as found on the knowledge base of Quantum Inspire [6].**

Aligning the available gate set with the entire gate set of cQASM 1.0 makes sure that the microarchitecture is future-proof: it has to support most algorithms that are known today and will be discovered in the future.

5. **Support for diamond specific protocols and instructions.**

   Diamond color centers require specific protocols in order to have a quantum computer that functions properly. Examples are the magnetic biasing of the color center or the charge resonance check. Without support for these functionalities, the completeness of the microarchitecture cannot be ensured.

3.2. System Architecture

Before thinking about the QISA and microarchitecture, the overall system architecture should be known. This is important as it can give information about what instructions and functionality is needed. Additionally, it can be seen what types of electronic devices need to be controlled and what signals they need.The overall system architecture is observed in Figure 3.1. A more closed up figure is depicted in Figure 3.2. Looking at the figures, a number of things can be concluded. First, there are two types of controllers. There are local real-time controllers and there is a global controller. Per color center, or per node, there is *one* local real-time controller. However, the system architecture may change in the future, so for now it is assumed that each color center has a dedicated local real-time controller. The microarchitecure that is defined in the rest of this chapter will be present on the local controllers. The global real-time controller oversees the local controllers and gives them instructions. Second, there is a number of electronic devices that need to be controlled by the local controller (and thus, microarchitecture):

- A voltage source,
- A current source,
- Three MEMS switches, which are optical switches,
- A variable optical attenuator per MEMS switch,
- A Photon Sideband (PSB) detector, which detects photons,
- and an Arbitrary Waveform Generator (AWG).

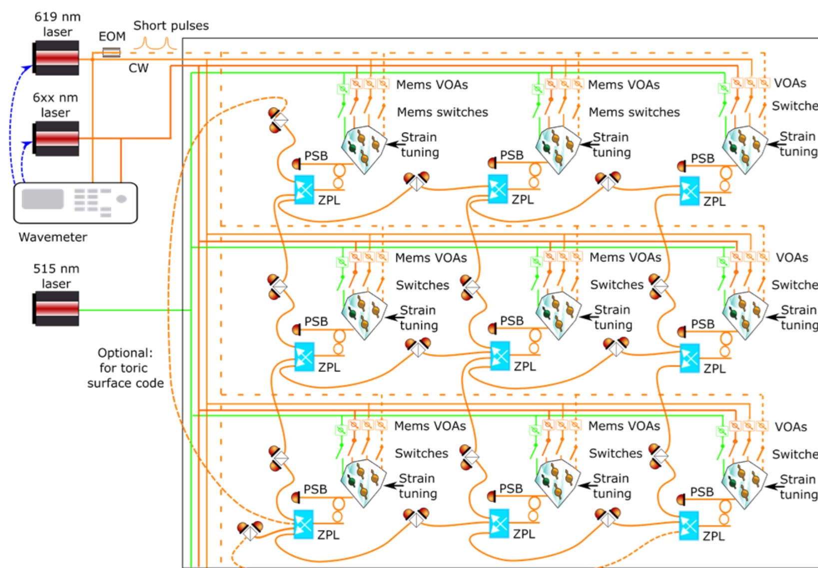


Figure 3.1: The system overview of the color center layout. Made by Erwin and Jaco, TNO.

The global controller interfaces with the local real time controller with two connections. One connection is for data and one is for timing synchronization. The other connections of the global controller

are for a set of photon-detectors that are needed to herald entanglement, and an external computer. The master controller that is being used in NV-center experiments uses an ADwin-based microcontroller [31, Supplementary Information]. The external computer is responsible for sending the quantum algorithm/program to the global microcontroller, which in turn sends the desired instructions to the local controllers. The local controllers will then report back the the global controller when the instructions are completed. Depending on the instruction, the local controllers will also send additional data to the global controller. As an example, the local controller sends back what the measurement result is or what the amount of detected photons for certain calibrations is.

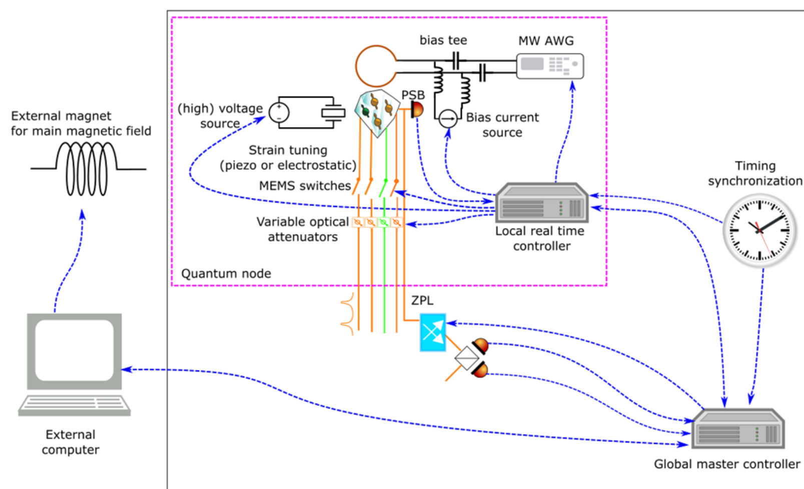


Figure 3.2: A closed up system overview of a color center layout. Made by Erwin and Jaco, TNO.

3.3. Quantum Instruction Set Architecture (QISA)

With the information from the overall system architecture in mind, the flow of the quantum program execution was determined. The flow graph can be seen in Figure 3.3. The flow graph is very useful to determine what needs to be done at what time, but can also be used to determine instructions needed in the QISA. For example, it can be seen that if a qubit is measured (readout), there are more steps involved then just measure, namely that the readout laser has to be enabled and photons have to be detected. Moreover, it is important to differentiate between diamond-specific instructions and the actual quantum algorithm. The actual quantum algorithm is represented in the `gates`-section, which, as the name suggests, consists only of quantum gates, as is expected with quantum algorithms as seen in Section 2.2. Most of the other blocks, such as `magnetic biasing` or `determine rabi frequency` are diamond specific, or not used or not common to other quantum technologies.

The large, colored blocks were used to determine the functionalities of the QISA. Using these functionalities, a first draft could be made. The instructions of the QISA were high-level. Although these instructions should not be low-level, the level of the instructions was a bit too high to be used to control the electronics directly. The instructions were based on the names and functions of the colored blocks of Figure 3.3, and thus could be decomposed into smaller instructions. In the first draft, the decomposition was done irregularly and therefore the QISA was inconsistent. This first version of the QISA, however, proved to be a good starting point for discussions about the overall functionality and the requirements of the QISA.

3.3.1. Refining of the QISA

The discussions with other work packages led to refinements within the QISA. Through these discussions, it was more clear what was expected from the QISA. It also become more clear what the general idea was. For example, it became clear that, in order to shine a laser pulse on a color center, there had to be an optical path that is controlled with MEMS optical switches (as also seen in the system overview, Figure 3.1. These advancements led to choices made that would have instructions be part of the QISA and of the microarchitecture. Additionally, it became clear that calibration sequences were

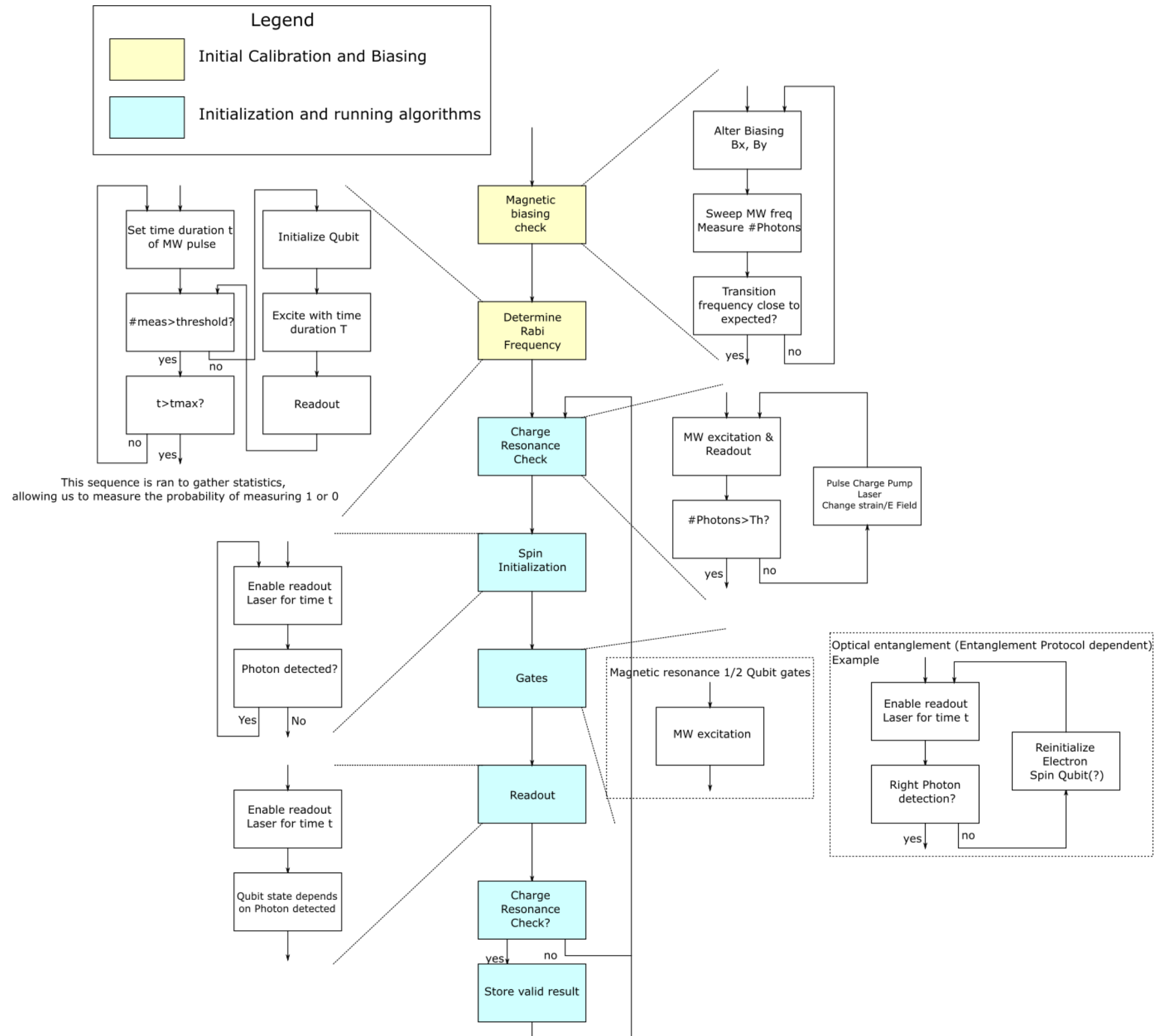


Figure 3.3: The operation flow graph for quantum computing in NV-centers.

needed. At first, the magnetic biasing, rabi check and charge resonance check were the only methods of calibration. Now, measurement, pi rotation and half-pi rotation should also be options for calibration. Each calibration method is aimed at increasing the fidelity of the operations of the quantum computer.

QISA and microarchitecture
Included in the refinement is that the QISA was split up into two tables: the QISA and the microarchitecture. The QISA that was developed so far still had instructions that could be decomposed. For example, the measurement instruction was still called `qread`, but when looking at the operation flow chart it consists of multiple steps. These steps could be part of a microarchitecture, defined as the list of instructions that are of the lowest level before the electronics. These instructions could not be decomposed and are aimed at controlling the control electronics directly or through a dedicated control chip [1]. Because in the quantum computer stack the QISA and the microarchitecture are direct neighbors, they should work together seamlessly. This is why the definition of a QISA and microarchitecture that work together is important, as it allows the *decomposition* of high-level quantum instructions into smaller technology specific low level assembly-like instructions, often called microcode.

3.4. Finalized QISA and microachitecture

Below in Table 3.1, the finalized QISA is presented. The QISA lists the type, syntax, description and decomposability for each instruction. The type shows what category the instruction belongs to as it is a quantum gate or a classical instruction. The actual instruction name and its syntax is given next. It shows how to use the instruction and what its arguments are. After that, the description of the instruction is presented. The description gives a short summary about the instruction. Finally, the last column presents whether the instruction is decomposable or not. If the instruction is decomposable, it can be split up into multiple smaller instructions. These microcode instructions will be part of the defined microarchitecture.

In Table 3.2, the microarchitecture can be seen. Just like with the QISA, the type, syntax and description of the instructions are presented. The instructions have all been verified and checked to make sure they are not decomposable, therefore the decomopsability is not listed in the table.

More details about the instructions of both the QISA and the microarchitecture can be found in Appendix A. Larger versions of the QISA and the microarchitecture can be found in Table A.1 and Table A.2 respectively.

3.4.1. Supported High-Level Instructions

The combination of the QISA and microarchitecture supports all instructions that are needed to enable quantum computing in diamond. However, there are some caveats. For example, it is assumed that most electronic devices, such as the MEMS switches, are controlled by the microarchitecture simply by writing a value to a register. Another example for this is the magnetic biasing. The microarchitecture itself is not able to create the current that is needed to change the magnetic field. So instead, it supplies the value that is needed to a control chip that connects to a current source that is able to create the currents that are needed. This will not always be the case, as some hardware could be directly controlled by the microarchitecture without a dedicated control chip in between. For most electronic devices however, the assumption that there is a control chip between the microarchitecture and the actual electronics is valid. Furthermore, it is easier to create the microarchitecture because there are less instructions needed, to write a value to a register in a control chip, the move-command suffices. As per the requirements, the QISA and microarchitecture should have support for all diamond protocols and functions. The following (diamond specific) instructions and protocols are supported:

- Magnetic Biasing
- Determine Rabi frequency
- Charge Resonance Check
- Initialization of qubits
- Calibration
    - measurement
    - pi rotation

[1] For example, it is assumed the MEMS switches have a control chip

Table 3.1: The QISA for quantum computing in diamond color centers.

Type	Syntax	Description	Decomposable
Qubit Gate	X, Y, Z, S, T qRd	Apply a single Quantum gate on qubit qRd,	No
	CNOT, CZ, CR qRs, nqRd, [<angle>]	Apply a two-qubit Quantum gate with qRs as control and qRd as destination. Angle is optional and only used for CR and CR_k gates.	No
Qubit Rotation	excite_MW <env>, <dur>, <freq>, <phase>, <amplitude>, qRd	Excite the qubit for time duration <duration>with frequency <freq>, phase <phase>and <amplitude> on qubit Rd. The value of these arguments determine the rotation / action of the excitation.	No
Qubit Readout	measure qRd, Rd	Measure qubit qRs and stores result in Rd. First, excite the laser on a particular NV-center. Then, the number of photons are counted by the control electronics and put into a register. The microarchitecture then needs to fetch the result from the control electronics register and store it in a main register.	Yes
Qubit init	initialize qRd	Initialize the qubit to state 0 using readout until no photons are measured.	Yes
nop	qnop	The quantum no-operation. Can be used as a filler operation.	No
Entanglement	qentangle qRs, nqRd	Entangle an NV-center qubit with a nuclear spin qubit. This is a sequence of gates.	Yes
	Nventangle qRs, qRd	Entangle an NV-center qubit with a NV-center qubit. There are multiple steps and thus multiple gates.	Yes
Nuclear Spin Ops	memswap qRs, qRd	Swap the state from the NV-center to the nuclear spin qubit, functioning as memory. Is a sequence of gates.	Yes
Biasing and Checks	sweep_bias <value>, dacReg, <start>, <df>, <fstop>, <memoryaddress>	Set the new current value for the biasing on control electronics register and sweeps microwave whilst storing a frequency/photoncount pair for each swept frequency.	Yes
	decouple qRd	Decouple the qubit according to the XY-8 protocol	Yes
	calculate_bias Rs, Rt, Rd	Calculate the new value for the current in amperes. This operation is purely classical and involves standard instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.	Yes
	calculate_volt Rs, Rt, Rd	Calculate the new value for the voltage in Volt. This operation is purely classical and involves standard instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.	Yes
Calibration	cal_meas qRd	Calibrate the qubit readouttime to ensure optimal readout.	Yes
	cal_pi qRd	Calibrate the qubit rotation amplitude for optimal pi-rotation.	Yes
	cal_halfpi qRd	Calibrate the qubit rotation amplitude for optimal half pi-rotation.	Yes
Timing	wait <value>	Wait an amount of cycles,	No
Standard Instructions	AND/OR/XOR/NOT Rs, Rd	Apply a classical AND/OR/XOR/NOT gate to registers and store result.	No
	ADD(i)/SUB(i) Rs, Rt, Rd	Add or subtract two registers and store the result.	No
	MUL/DIV Rs, Rt, Rd	Multiply or divide two registers and store the result.	No
	MOV Rs, Rd	Move the contents of a register to another register.	No
	LD(i)/ST(i) Rs, Rd	Load from memory to register or Store from register to memory.	No
	BR <comp>, <address>	Jump if the comparisons statement is true.	No
	jump <address>	Jump unconditionally.	No

  - half pi rotation
- Application of gates on qubit according to cQASM 1.0 specifications
  - Standard gates: X, Y, Z, S(†), T(†)
  - 90 degree rotations: +/- x90, +/- y90
  - Two-qubit gates: CNOT, CZ, SWAP
  - Toffoli-gate through decompositions
  - Custom rotations: CR and CR_k
  - Full user-specific gates using `excite_MW`
  - Binary-controlled gates
- Readout and storage of quantum states

3.4.2. Example of decomposition

In this section, an example the decompositions will be given. The first two instructions can be seen in Figure 3.5, where `measurement` is seen in part a and `initialize` in part b. Note that the two instructions look a lot like each other. This is because `initialize` is based on measurements. As

Table 3.2: The microarchitecture for quantum computing in diamond color centers.

Type	Syntax	Description
Quantum Operation	qgate <type>, qRd	Apply a single Quantum gate. <type>defines which gate (X, Y, Z, H, I, S (= pi/2), T (= pi/2)). qRd is the target qubit.
	qgate2 <type>, qRs, nqRd, [<angle>]	Apply a two-qubit Quantum gate. <type>defines which gate (CNOT, CZ). qRs is the source qubit and qRd is the target qubit. The angle argument is optional and only used for CR and CR_k gates.
	excite_MW <env>, <dur>, <freq>, <phase>, <amplitude>, qRd	Excite the qubit for time duration <duration>with frequency <freq>, phase <phase>and <amplitude> on qubit Rd. The value of these arguments determine the rotation / action of the excitation.
	qnop	Apply a quantum no-operation. Can be used as a filler operation.
Timing	wait <cycles>	Wait an amount of <cycles>.
Support	switchOn <address>, [<duration>]	Switch the optical switch with address <address>to the "on" position (optional: for time <duration>).
	switchOff <address>	Switch the optical switch with address <address>to the "off" position.
Standard Instructions	AND/OR/XOR/NOT Rs, Rd	Apply a classical AND/OR/XOR gate to registers Rs and Rd, storing the result in Rd.
	ADD(i)/SUB(i) Rs, Rt, Rd	Add or subtract two registers Rs and Rt (or values when using immediate), storing the result in Rd.
	MUL/DIV Rs, Rt, Rd	Multiply or divide two registers Rs*Rt or Rs/Rt and store the result in Rd.
	MOV Rs, Rd	Move the contents of register Rs to Rd.
	LD/ST Rs/Rd, Rt	Load from memory to register or Store from register to memory.
	BR <comparison>, <offset>/Rd	Jump to <offset>is the <comparison>is true. If the offset is a register, store the comparison result in that register.
	jump <address>	Jump to <address>.
	LDi <value>, Rd	Load immediate value <value>in register Rd.

one would expect with decompositions, the single QISA instruction is decomposed into a set of smaller instructions. In the case of measurement and initalization, a single instruction can be realized by decomposing it into 6 (or 7, for initialize) instructions. The smaller instructions are always part of the microarchitecture defined earlier in this chapter. The remaining instructions that are decomposable are discussed later in this section, and the microcode for each decomposition can be found in Appendix B.

measurement

The measurement instruction uses two input arguments. The first is the qubit to be measured and the second is the place where the result of the measurement needs to be stored. These arguments are taken into the decomposition. It can be seen that the optical path is switched on for qubit q1 as required by the QISA instruction. Following this instruction, the counter for the number of photons has to be reset to zero. This is to ensure that only the photons emitted during the duration are measured. Then, the laser is enabled with its parameters (envelope, duration, frequency and phase) on the requested qubit. The photondetectors count the number of photons automatically and stores it in the photonReg-register, which is read by the microarchitecture by using the mov-command. Mov moves the contents of a classical register to another classical register. Then, the optical path is closed. The number of counted photons is then compared to a pre-defined threshold, which is stored in R33. Depending if the number of counter photons is lower or higher than the threshold, the result register is set to either '0' or '1'.

initialize

The initialize instruction works largely the same as measurement, but instead of comparing with a set threshold, it compares with '0'. If the amount of detected photons is higher than 0, it means the quantum state is not in $|0\rangle$. The process needs to be repeated until 0 photons are detected.

qentangle

With qentangle, the electron qubit is entangled with a nuclear spin qubit. This happens according to a set protocol, found in [32, Fig. 4]. After the decomposition, the user needs to measure the electron qubit and depending on the result, a half-pi y rotation is acted out on another electron qubit. The protocol itself consists of three gates that are executed sequentially: 1) -x90 gate on electron, 2) +-x90 gate with electron as control and nuclear spin qubit as target and 3) a +90x gate on the electron.

NVentangle

NVentangle, as the name suggests, entangles two color centers. In this case, the color centers are NV-centers. This means that if other color centers are to be used, such as SnV-centers, then the protocol showed here might not work. Additionally, the Fujitsu project might use a different scheme nonetheless [2] For now, the decomposition is based on a protocol using the method from Barrett and Kok [33]. The protocol from [33] has four steps: 1) apply a pi-pulse to both NV-centers, 2) wait t_wait for photon detection, 3) wait t_relax to relax the qubit cavity system and 4) X rotation for both qubits and repeat step one, two and three. If '0' or '2' photons have been detected, entanglement has failed. Else, heralded entanglement has been achieved.

Translating this protocol to microcode means that an optical path has to be created for both qubits, then the laser has to be enabled and then the system has to wait for a moment to detect photons. After the wait time is over, we can switch off the optical path and copy values from the photondetectors to general purpose registers. After the first round, it has to be made sure that the x-gate is applied on both qubits and the steps are repeated. This is done using a combination of quantum gates and classical branch instructions.

memswap

This instruction swaps the quantum state of the electron qubit with the state of a nuclear spin qubit inside the same color center. This protocol has also been taken from [32, Fig. 4]. The protocol consists of a sequence of four quantum gates: 1) +-y90 with electron as control and nuclear qubit as target, 2) x90 on electron, 3) +-x90 with electron as control and nuclear qubit as target and 4) -y90 on the electron.

sweep_bias

When the user wants to sweep a laser with varying frequency over a qubit, sweep_bias is used. The instruction stores the frequency-photoncount pair and using that information, the optimal biasing current can be calculated. The function is to be used together with calculate_bias to perform magnetic biasing on the color center. First, a starting value for the biasing has to be set together with a start frequency, a step frequency and a stop frequency. The memory address, to where to store, also has to be set. Subsequently, there is a series of *load immediate* instructions taking care of this set-up first. Then a loop will be started, that excites the color center with a laser, counts the number of photons, stores the frequency-photoncount pair and increments the frequency by the step frequency. This is repeated until the stop frequency has been reached, which is checked by a branch instruction.

decouple

Periodically, the color center electron needs to be decoupled because it is influenced from natural occurrences such as magnetic and electric fields and nuclear spins. The decoupling 'resets' the electron spin, so it can retain its state longer. For the Fujitsu project, XY8-decoupling is used and that is what the decouple instruction implements. XY8-decoupling consists of a series of 8 rotations around x and y, as shown in Figure 3.4.

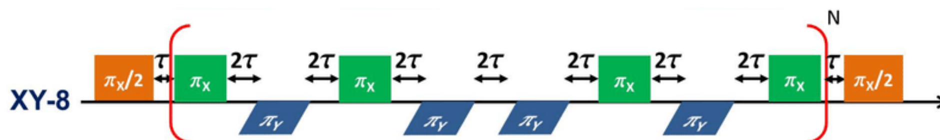


Figure 3.4: XY8-decoupling protocol.

calculate_bias

This instruction is used to calculate the new value for magnetic biasing. However, since there is no information yet about how to calculate the new value based on the old value and the stored frequency-photoncount pairs, the decomposition of this instruction into classical instructions is not done yet.

[2] At the time of writing, the desired entanglement protocol was not yet known.

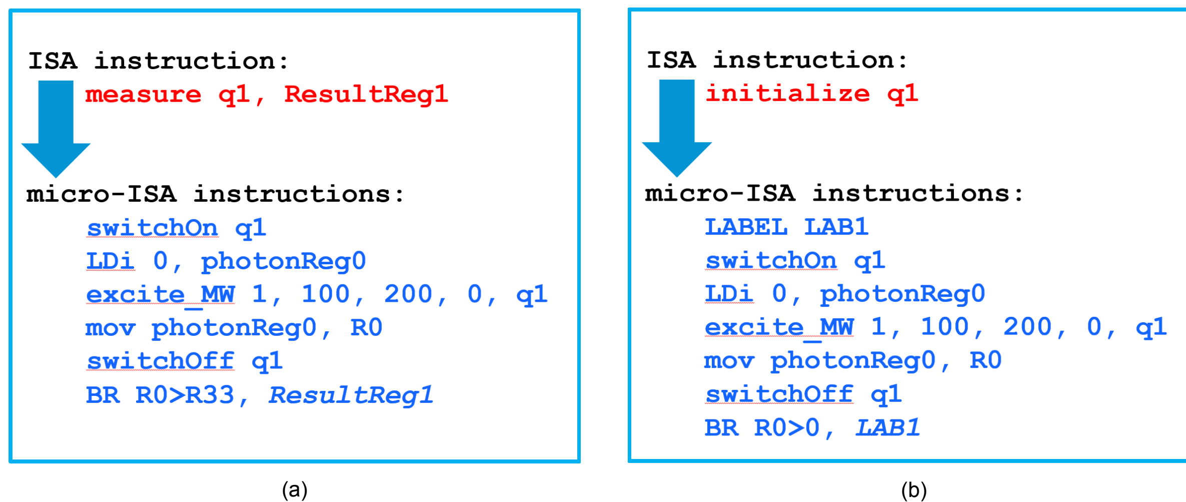


Figure 3.5: Examples of decomposition using the `measurement` (a) and `initialize` (b) instructions.

calculate_voltage
This instruction is the same as calculate_bias, but instead it calculates a new value for electrical biasing. Because the calculation of this value is different than for calculate_bias, a different function had to be introduced. Again, the calculation itself is unknown, so the decomposition is not done yet.

cal_meas
The system needs to be calibrated periodically, and one of the things that needs calibration is measurement. The time period that the laser excites the color center can be optimized, gaining maximum readout fidelity. The protocol consists of six steps: 1) prepare the qubit in $|0\rangle$, 2) laser on for 40us and count the emitted photons while timestamping them, creating a photon-time pair. Next, in step 3) the qubit is prepared in $|1\rangle$, 4) repeat step 2, 5) repeat the preferred amount of times and 6) calculate the new readout time, which is the point with the highers amount of emitted photons. The calculation process is a classical process which has not been elaborated on because that is out of the scope of this thesis.

cal_pi
Rotation with a angle of pi also has to be calibrated for maximum fidelity. Contrary to the measurement calibration, this protocol calibrates the amplitude of the laser signal. This protocol has four steps: 1) prepare the qubit in $|0\rangle$, 2) perform 11-pi pulses with the same amplitude, 3) measure the fidelity and 4) change the amplitude and repeat.

cal_halfpi
Rotation with a angle of pi/2 also has to be calibrated. This is done differently compared to a pi rotation, as there are now 5 rotations with pi/2 instead of 11, followed by a reset and 5 rotations of 3pi/2. This creates two curves, one going up and one going down, making the shape of an X. The crossing of the curves is the optimal point and the amplitude can be determined accordingly.

It can be a tedious process to manually translate the QISA instructions to microcode. Because of this, a tool is developed that auto-translates QISA-code to microcode. The tool is described in Chapter 4.

3.5. Conclusion

In this chapter, we defined the microarchitecture for diamond quantum computing. We listed the requirements the microarchitecture should have in Section 3.1, where it was determined that it should support all diamond-specific protocols and instructions. Moreover, it should support all cQASM 1.0 instructions as found on the knowledge base of [6]. In Section 3.2, we saw the overall system where the microarchitecture will be part of. The system architecture was introduced to give a better view of the job of the microarchitecture, as well as to give a better overview of the entire project. Next, we introduced

the Quantum Instruction Set Architecture (QISA) and the microarchitecture in Section 3.3. We saw that the first draft was created with the help of information obtained from the literature research and information about the required diamond-specific instructions and operations. We also saw that some instructions were decomposable. This resulted in a separation between the QISA and the microarchitecture, where the instructions of the microarchitecture are of the lowest level before entering control electronics. The result of the development was shown in Section 3.4, where the QISA and corresponding microarchitecture were presented. We went over the supported high-level instructions as well as an example of a decomposition.

4 Tool

This chapter will discuss the development of the compiler tool that translates a quantum algorithm to microcode (diamond quantum computer assembly). The compiler is made within the OpenQL platform, which was briefly discussed in Section 2.6. The chapter will begin with a recap of OpenQL in Section 4.1. Subsequently in Section 4.2, the design process will be laid out. Each step of the design process will be elaborated on. After that, a brief example on how to use OpenQL and the compiler pass is presented in Section 4.3. The chapter ends with a conclusion on the design of the tool. The tool can be found on GitHub [34].

4.1. OpenQL

OpenQL is QuTech's solution to quantum computer code compilation. It is a framework for high-level quantum programming in C++/Python, but also accepts cQASM files as input [30]. The focus lies on compiling down to assembly code for different quantum computing platforms, such as QuTech's Central Controller. OpenQL is publicly available under the Apache 2.0 license [34].

OpenQL takes two inputs. The first input is the algorithm, either written in the C++ API, the Python API or in a cQASM file. When using a cQASM file, the file has to be loaded in using the Python API. The second input is the platform configuration file that tells OpenQL what the target hardware looks like. It conveys the available gates, the available devices and more. Much more information about how OpenQL works and its concepts can be read at [30, /manual/concepts].

4.2. Compiler Design

In the list below, the workflow of the creation of the compiler is seen. Each step is elaborated into detail.

1. Identify requirements of the compiler.
2. Add the Diamond backend to OpenQL.
3. Add and design the microcode translator pass to OpenQL.
4. Test the compiler with commands.
5. Test the compiler with a quantum algorithm.

4.2.1. Compiler Requirements

The first step of the compiler design is to determine the requirements and goals of the compiler. The main goal of the compiler is to translate a high-level quantum algorithm to microcode. The microcode instructions are found in the microarchitecture table. Preferably, the compiler is more than a translator. It should also, to some extent, schedule operations, although this is typically done in the runtime part that is present in the same layer. Within OpenQL, several passes are already available that do mapping, scheduling and Clifford optimization [30]. From the goal, a number of requirements follow. The requirements of the compiler are found in the following list:

1. **Translate high-level quantum algorithms to diamond microcode**

This is the main function of the microcode compiler.

2. **Support for all instructions defined in the QISA/microarchitecture**

   Without support for all instructions defined in the QISA/microarchitecture, the compiler does not fully support the microarchitecture and therefore would not be complete.

3. **Support all diamond-specific protocols/sequences such as, but not limited to:**
   (a) Magnetic Biasing
   (b) Rabi Frequency Determination
   (c) Charge Resonance Check
   (d) Calibration of measurement, pi and pi/2 rotation

   The full list is already shown for the microachitecture in Subsection 3.4.1. These protocols/sequences are needed for the correct functionality of the color center qubit.

4. **To some extent, schedule necessary diamond-specific operations automatically**

   Instruction scheduling is important in the compiler. Without some form of scheduling, the compiler is nothing more than a simple translator. With the addition of scheduling, the compiler has more functionalities and will therefore be more complete.

The work on the compiler starts with installing OpenQL. Because contributions to OpenQL are going to be made, the framework has to be built from source. Every time a change is made to the source code, the program has to be build again. Instructions on how to build from source are documented in [30]. The system on which the compiler was developed is Ubuntu 20.04.2 LTS. The operating system was a virtual machine using VMWare Workstation 16 Player with Windows 10 as the host system. The compiler was programmed using CLion, a cross platform IDE.

4.2.2. Diamond backend in OpenQL

The first thing that was done, is creating a new backend and add it to OpenQL. A backend can be seen as a target system for which OpenQL compiles for. CC-light and CC are examples of backend. We create and add a new backend by creating a new folder in `/src/ql/` called Diamond. In here, a new file, called `info.cc` is made that serves as the information file about the new backend. It includes the standard platform configuration file, `hwconf_default.json`, that also has to be made inside `/src/ql/diamond/resources/`. Inside `info.cc`, some settings are defined, such as the name of the backend, the namespace which the backend falls under, and what passes are used in the backend. Note that the same diamond folder also has to be made in `/include/arch/` with inside the header file for `info.cc`, `info.h`. Besides these changes, the backend also needs to be registered to the system. This is done by adding the backend to the backend list in `/src/ql/arch/factory.cc`. Important is to not forget to add the new files to `CMakeLists.txt`.

Now, the newly made (but empty) backend should be present within OpenQL. However, because there is no pass added to the backend, there is no functionality.

4.2.3. Microcode Translator Pass

To ensure that the backend is functional, a pass has to be added. Usually, a pass is added in `/ql/-pass/`, but because the diamond microcode translator pass is backend-specific, the pass should be added in `/ql/arch/diamond/pass/`. Additionally, because the pass is a generator, it should be placed in a folder called `gen`.

After the pass has been configured, including the main functions that dumps documents, the generation of the friendly name (human readable name) and the constructor for the pass, the pass has to be added to the diamond backend. At the end of `info.cc`, found in the `src/.../diamond` folder, the user should add:

```
manager.append_pass(”arch.diamond.gen.Microcode”, ”diamond_codegen”);
```

in the `populate_backend_passes` function to add the pass to the diamond backend. Now, it is time to fill in the pass with the functionality that is desired.

```
for (const ir::KernelRef &kernel : program->kernels) {
    for (const ir::GateRef &gate : kernel->gates) {
        const auto &data = program->platform->find_instruction(gate->name);

        outfile << "# " << gate->qasm() << "\n";

        // Determine gate type.
        utils::Str type = "unknown";
        auto iterator = data.find("diamond_type");
        if (iterator != data.end() && iterator->is_string()) {
            type = iterator->get<utils::Str>();
        }

        if (type == "qgate") {
            // print code to outfile
        } else if (type == "qgate2" {
            // print code to outfile
        } else {
            if (gate->name == "measure") {
                // Code for measurement
            } else if (gate->name == "initialize") {
                // Code for initialization
            }
        }
    }
}
```

Listing 1: The idea of the microcode pass.

The main idea of the pass was to have it be an enormous switch-case statement, that checks the name of a gate, and based on the name of the gate writes something to the microcode file, or output file. While this idea is non-sophisticated, it can provide for all the functionality that is desired in the early stages of the compiler.

Under the `run`-function, the code for the pass behavior can be written. First and foremost, it should be clear to what file any output should be written. After that has been done, a nested for-loop has to be made that makes it possible to loop through every gate, as shown in lines 1-2 of Listing 1.

It can also be seen that the type of gate is determined. This is useful for grouping gates. For example, when using a standard quantum gate such as X, Y, Z, S and T, the output of the microcode is `qgate <gatename>, qubit`. This has the same output form for every standard gate, and thus can be standardized by giving a type to the instruction. This also holds for 2-qubit gates, as indicated in the listing. Gates can be brought under a type by defining that in the platform configuration file like this:

```
"x": {
    "duration": 40,
    "diamond_type": "qgate"
},
```

Using this method, custom rotations around an axis (rx, ry and rz gates) can also be categorized under a type to make the code more readable.

Next, all other gates defined in the QISA can be added to the if-else statement as shown in lines 19-23, Listing 1.

As an example, the measurement instruction can be looked into further, see Listing 2. It can be seen that the function is explained in comments, followed by a set of operations. Each operation is written to the outfile with the use of the «-operator. Note that operations begin with `detail::`. This is because the functions for a operation, for example switchOn, are defined in separate files, `/src/arch/diamond/pass/gen/detail/function.c` and its header file that is found in the same path.

The measure instruction takes two arguments, of which one is inherent to the first. In OpenQL's Python API, measurement is called by using `kernel.gate('measure', [0])`. The argument between [ ] is the qubit which needs to be measured. Inherently, this result is stored into a bit register called `breg[x]`, where x is the same number as the qubit number. The measurement operation has

the same behavior, where the qubit number is taken (`gate->operands[0]`) and used in the functions to always print the correct number when needed.

```
if (gate->name == "measure") {
    // Measures a qubit and stores the result in ResultRegQ,
    // where Q is the qubit number. Also stores the result in
    // breg[Q], as per OpenQL standard.
    Str qubit_number = to_string(gate->operands[0]);
    const Str threshold = "33";

    outfile << detail::switchOn(gate->operands[0]) << "\n";
    outfile << detail::loadimm("0", "photonReg", qubit_number) << "\n";
    outfile << detail::excite_mw("1", "100", "200", "0", gate->operands[0]) << "\n";
    outfile << detail::mov("photonReg", qubit_number, "R", qubit_number) << "\n";
    outfile << detail::switchOff(gate->operands[0]) << "\n";
    outfile << detail::branch("R", qubit_number, "<", "R", threshold,
        "ResultReg", qubit_number) << "\n";
 }
```

Listing 2: Example of the measurement instruction in the microcode pass.

Instructions with multiple parameters

Some instructions, such as `excite_mw`, take more arguments than the qubit number. OpenQL does not directly support this, but one of the maintainers, Jeroen van Straten, suggested annotations could be used. How these annotations work is “c++ magic”, according to Jeroen and subsequently will not be further discussed in this thesis. However, might the successor working on this compiler need to add more annotations, the process is described for the `excite_mw` instruction.

First and foremost, an annotations file should be included in the project. A new file, `annotations.h`, has been made, which is found in `/include/.../diamond/`. In this file, a struct is made, as seen in Listing 3. It can be seen that the additional parameters for the instruction are defined in this struct. Next, the instruction should be added to the kernel of OpenQL. In `api/kernel.h`, the instruction is defined as a function:

```
void diamond_excite_mw(size_t envelope, size_t duration, size_t frequency, size_t phase,
                       size_t qubit);
```

```
/** \file
 * Defines annotations about the diamond architecture.
 */

#pragma once

#include "ql/utils/num.h"

namespace ql {
namespace arch {
namespace diamond {
namespace annotations {

struct ExciteMicrowaveParameters {
    utils::UInt envelope;
    utils::UInt duration;
    utils::UInt frequency;
    utils::UInt phase;
};
```

Listing 3: An example of annotation setup.

This function is then filled in `kernel.cc`, as seen in Listing 4. Note that the name of the function has the prefix `diamond_`. This is to distinguish the instruction as diamond-specific.

The final step is to add the function to the documentation. This is done in `kernel.i`. Please keep in mind that unless the structure is the exact same as the other instructions (white rules etc), errors will come up when compiling. The code for the `excite_mw` instruction is seen in Listing 5.

```
/**
 * Appends the diamond excite_mw instruction.
 */
void Kernel::diamond_excite_mw(size_t envelope, size_t duration, size_t frequency,
    size_t phase, size_t qubit) {

    kernel->gate("excite_mw", qubit);
    kernel->gates.back()->set_annotation<ql::arch::diamond::annotations::
        ExciteMicrowaveParameters>({envelope, duration, frequency, phase});
}
```

Listing 4: Function definition for annotation setup.

```
%feature("docstring") ql::api::Kernel::diamond_excite_mw
"""
Appends the diamond \"excite_mw\" instruction.

Parameters
----------
envelope : int
    The envelope of the microwave.

duration : int
    The duration of the microwave in nanoseconds.

frequency : int
    The frequency of the microwave in kilohertz.

phase: int
    The phase of the microwave.

qubit: int
    The target qubit index.

Returns
-------
None
"""
```

Listing 5: An example of annotation documentation setup.

The final step is to 'get' the annotations. This is done inside the code for a particular gate after the name of the gate has been determined. This is done using the following line:

```
const auto &params = gate->get_annotation<annotations::ExciteMicrowaveParameters>();
```

The annotations, or set values or the envelope, duration and other defined parameters are stored in `params` and accessed using `params.envelope` (and similar for others). Now, the gate can be called from the Python API using `kernel.diamond_exite_mw(1, 100, 200, 0, 0)`. Unfortunately, due to the nature of OpenQL's internal representation, it is not possible to call annotated gates from cQASM. This limits the usability of cQASM as the input for algorithms.

Scheduling of Instructions

The compiler has to be smart. Now, all functionality is aimed at directly converting operations, gates, and diamond-specific sequences to microcode. However, some of this can be performed automatically. For example, from Figure 3.3, it can be seen that every time an algorithm is executed on the diamond quantum computer, there are magnetic biasing checks.

This can be automated by copying the OpenQL kernel to a temporary kernel, adding gates in front, and copying the temporary kernel back to the original kernel. This is precisely what is done, as can be seen in Listing 6.

From line 13 onwards, the functions that need to be put in front of the normal algorithm gates are added. For operations that need annotations (such as magnetic biasing's `sweep_bias` instruction), these annotations can be set using `kernel.gates.back()->set_annotation<struct>();`

```
// for each kernel
for (const auto &kernel : program->kernels) {

        // copy the kernel and put it into a temp kernel
        ir::Kernel temp_kernel(
            "dummy",
            program->platform,
            kernel->qubit_count,
            kernel->creg_count,
            kernel->breg_count
        );

        // for each qubit, do magnetic biasing
        // for each qubit...

    // Copy the gates to the original kernel and set the cycles_valid flag to false
    // because the cycle count is not accurate anymore.
    kernel->gates = std::move(temp_kernel.gates);
    kernel->cycles_valid = false;
}
```

Listing 6: The microcode scheduler.

Apart from the operations that are scheduled *before* the algorithm gates, there are some operations that go in between these gates. The Charge Resonance Check (CRC) is done for every qubit every 10 gates because this is needed in the diamond architecture to maintain correct functionality of the qubits. Additionally, the compiler also keeps track of labels. This is done by using a global variable that gets incremented each time a label is made.

Conditional Gates

Because cQASM 1.0 also supports conditional gates, the principle of conditional gates has also been added to the microcode pass. OpenQL also has support for conditional gates, so that functionality was used in the pass. When using a conditional gate, it gets translated to microcode using three instructions: 1) a branch instruction that jumps to a label if the condition is fulfilled, 2) the gate and 3) the label. If the condition is fulfilled, the gate gets skipped. Note that this branch condition has to be the opposite of the actual condition because of the structure.

If a conditional gate is used, then the branch is put automatically before the gate. At the end, if the branch is put before the gate, a label is automatically added. This way, all gates can be conditional like per cQASM specifications [6, Knowledge Base->binary controlled gates].

4.3. Using the Compiler

In this section, a brief introduction to the usage of OpenQL and the diamond backend including the microcode pass will be given. For all supported functions and gates, and how to use them, please refer to Appendix C. An example file with all possible gates can be found in `OpenQLexamplesdiamond`.

First and foremost, when OpenQL is installed, it can be called from any Python script as long as OpenQL is imported (`import openql as ql` for example). How this Python API works is written in the documentation, [30], but a brief example is being shown in Listing 7. It can be seen that in this example, the cQASM reader is used before the microcode pass. This ensures that the algorithm, written in `cqasm_test.cq`, gets loaded into OpenQL and gets translated to the IR of OpenQL, which serves as the input for the microcode pass. All gates that are added to the kernel (in this case, the X-gate on qubit 0 and the `diamond_excite_mw`-operation) are deleted if the cQASM pass is put before the microcode pass. This is intended behavior by the cQASM reader pass.

When not using the cQASM reader pass, the user has to put the algorithm in the Python API. Two examples of this are given in lines 13 and 14 of Listing 7. The two methods have advantages and disadvantages. The cQASM-method is easy to use and provides organized files. In addition, the cQASM files can be used in simulators such as QX [35]. However, it does not support all the instructions of the diamond QISA. Custom instructions, such as the `diamond_excite_mw`-instruction, that also has annotations, are not supported through cQASM because they are not supported (yet) by OpenQL. The Python API, whilst not that user friendly, does support these instructions. In the future, the OpenQL

```
import openql as ql

ql.set_option('prescheduler', 'no')
# Specify the platform
platform = ql.Platform("diamond_test", "diamond")

# Put a cQASM reader before the diamond pass
platform.get_compiler().prefix_pass('io.cqasm.Read', '', {'cqasm_file': 'cqasm_test.cq',
    'gateset_file': 'gateset.json'})

nqubits = 1^^I
p = ql.Program("testProgram", platform, nqubits)
k = ql.Kernel("testKernel", platform, nqubits)

k.gate('x', [0])
k.diamond_excite_mw(0, 100, 200, 0, 0)

p.add_kernel(k)
p.compile()
```

Listing 7: Example Code of using the diamond backend in OpenQL.

kernel and/or IR may be updated to support annotations and the custom instructions with cQASM. Once the algorithm is put into cQASM or into the Python API, OpenQL can be executed by running the Python script. For Ubuntu, that is done with `pyhton3 <filename>.py`. Note that in addition to the cQASM file, a gateset file also has to be included. This gateset file acts as the library that OpenQL uses to translate between cQASM and the OpenQL IR.

4.4. Conclusion

In this chapter, we discussed the design of the compiler. First, a brief recap about OpenQl was given in Section 4.1. We determined that OpenQL takes two inputs: the algorithm and the platform configuration file. Then, in Section 4.2, the design of the compiler was discussed. We learned the design methodology of the compiler and the requirements of the compiler. Using this information, we have seen how to create and add a backend to OpenQL. In addition, we have seen how to create and add a pass to OpenQL. The pass that was added to OpenQL is our microcode translator pass, which schedules the gates and operations of the algorithm and prints the output microcode to an outfile. We have seen the different design aspects of the pass, such as custom instructions with multiple operands and conditional gates. Finally, in Section 4.3, we made comments on how to use the compiler and how to use OpenQL.

5

Verification

In this chapter, the verification of the microarchitecture and the compiler tool is discussed. The requirements that were listed in Chapter 3 and Chapter 4 will be repeated and a brief discussion will be presented per requirement whether the design fails or passes the requirement and why.

5.1. QISA and microarchitecture

When designing the microarchitecture, several requirements were given, as seen in Section 3.1. In this section, the QISA and microarchitecture is tested on these requirements. For reference, the requirements for the microarchitecture are:

1. Deterministic and precise control of the control electronics.
2. Classical instructions for flow control.
3. Basic quantum gate instructions (X, Y, Z, S and T).
4. Additional gate instructions according to cQASM 1.0, as found on the knowledge base of Quantum Inspire [6].
5. Support for diamond specific protocols and instructions.

5.1.1. Deterministic and precise control of the control electronics

Deterministic and precise control of control electronics plays a key part in the microarchitecture. As stated previously the quantum computer cannot function properly and reliably without a microarchitecture. If an operation is too late, too long, too short or not up to specifications, then the intended rotation ends up being another unwanted rotation.

There is a distinction between the contents of a signal and when the signal is sent. Regarding when the signal is sent: the microarchitecture that was designed works sequentially. That is, every instruction gets executed one for one using only one thread. This means that timing information is not sent with the instruction but instead wait statements have to be added between the instructions to comply with correct scheduling. If instructions 1 to 20 have taken 200ns, and instruction 21 has to be executed at t = 250ns, then the system would have to wait the appropriate amount of cycles using the wait-instruction before being able to execute instruction 21 at the right time. This gets increasingly complex as, for example, the `excite_mw` instruction has a duration-parameter.

Concerning the content of the signals, the signal itself (X-gate, Y-gate, Rx-gate, switchOn etc) does not correlate to the moment the signal has to be send to the lower layer. This means that the signal should always be the correct signal, regardless of the timing of the signal. If an X-gate needs to be executed, it will always be an X-gate. Keep in mind that if the X-gate, for example, is sent to the qubit later than planned, the rotation of the qubit at the end of the algorithm may not be what is expected, because of the natural decoherence of the qubit.

5.1.2. Classical instructions for flow control

Besides flow control, classical instructions are also needed to perform other functions, such as calculations. With the possibility for branch instructions, jumps and label definitions, flow control has been

Table 5.1: The comparison between cQASM 1.0 instructions and the diamond microarchitecture. ✓: full support, ✗: no support, ✱: partial support [1].

	prep_z	prep_y	prep_x	X	Y	Z	H	I	Rx	Ry	Rz	90-deg rot	S
cQASM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Diamond MA	✓	✱	✱	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

	Sdag	T	Tdag	CNOT	CZ	SWAP	Toffoli	CR	CR_k	Measure_z	Measure_y	Measure_x	Measure_all
cQASM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Diamond MA	✓	✓	✓	✓	✓	✓	✱	✓	✓	✓	✱	✱	✗

adequately established. For calculations, such as (re)calculating the value for magnetic biasing, the microarchitecture supports various classical instructions that support the calculations. For example, arithmetic instructions such as adding, subtracting, multiplying and dividing are present. But also binary operations are present, such as AND and OR. In addition, loading and storing of data is included, as is moving/copying of data. As a result, the classical instructions supported by the microarchitecture should be sufficient to act out calculations and flow control.

5.1.3. Basic quantum gate instructions (X, Y, Z, S and T)

The microarchitecture supports the basic single qubit quantum gate instructions X, Y, Z, S, Sdag, T, Tdag, Hadamard (H), +/-X90, +/-Y90 and two-qubit gate instructions Besides that, the basic quantum instructions that cannot be executed by the standard gate operations (or for micro-ISA, `qgate` and `qgate2`) are able to be executed using the `excite_mw`-instruction when using the right parameter values.

5.1.4. cQASM Gateset

In order to verify whether the microarchitecture features the full cQASM gateset, a table has been made to verify this. The comparison is found in Table 5.1. It can be seen that most of the instructions are supported fully, but some instructions are not fully supported. For measurement in y and in x, the qubit first needs to be rotated into those axis before a normal measurement or preparation can be done. For preparation, the rotation happens after initialization. The toffoli gate is not natively supported on the diamond architecture, and therefore also not on the diamond microarchitecture. However, it can be decomposed into single and two qubit gates that are available, making the toffoli gate partially supported. Finally, if the user wants to measure all qubits, then all qubits should be measured sequentially.

5.1.5. Diamond specific protocols and instructions

To aid the verification process, a list of the diamond specific instructions and protocols can be seen below. For reference, Figure 3.3 is used to determine the needed protocols. Note that, whilst some instructions that are included by cQASM, such as measurement instructions, are handled differently by the diamond architecture than another qubit architecture, they are not labeled as *diamond specific*. Diamond specific instructions are solely the instructions that are not used in other architectures.

- Magnetic Biasing
- Determination of Rabi Frequency
- Charge Resonance Check
- Calibration
  - Readout
  - Pi rotation
  - Pi/2 rotation
- Enabling the optical path
- Pulse picking
- Read out photondetectors
  - PSB
  - ZPL
- Dynamical Decoupling

[1] Partial support means the instruction is supported through decomposition or a sequence of other instructions.

Magnetic Biasing
For magnetic biasing, the microarchitecture needs to be able to sweep a laser frequency over a qubit with a set value for the magnetic biasing. Then, depending on the measurements result - which is a combination of number of detected photons and current frequency, a new value can be calculated for the magnetic biasing. The microarchitecture features this functionality in `sweep_bias` and `calculate_bias`. Therefore, magnetic biasing is supported.

Determination of Rabi Frequency
As can be seen in the flow graph, determination of the rabi frequency is complex. It is an exhaustive protocol, but can be realized very well within the microarchitecture. In Subsection 5.1.4, it is seen that initialization and readout is supported, as well as custom excitation of a color center using `excite_mw`. The outer loop of the protocol can be realized with the classical instructions - loading, storing and flow control. Therefore, the determination of the rabi frequency is supported.

Charge Resonance Check
The CRC, is used to check if the color center is in the right charge state. If not, then the center needs to be charge pumped and the electric field has to be changed to verify if the center is still calibrated to the laser. Charging the color center is done by enabling the charge pump laser for a short amount of time. The electrical biasing is done similarly to magnetic biasing. The microarchitecture is able to do both, therefore is capable of performing a CRC.

Calibration
Calibration comes in threefold. The readout-operation has to be calibrated, as well as pi-rotation and pi/2 rotation pulses. These operations are not listed in the flow graph of Figure 3.3, but are still important to be able to perform.

Readout calibration is done by reading out the color center for 40us, and counting photons whilst time-stamping the photon measurement. This is done both for $|0\rangle$ and $|1\rangle$. It is then repeated and the readout time is set at the highest point of the mean curve (fidelity vs readout time). These operations are fully supported by the microarchitecture as it is possible to excite the color center, detect photons, store and load data, make loops and perform calculations.

Pi-pulse calibration is done similarly to readout calibration, but instead of measuring versus readout time, the fidelity of $|0\rangle$ is measured versus MW amplitude. First, 11 MW X-rotations are executed, resulting in an end-state of $|1\rangle$. Then, the state is measured and the fidelity/amplitude pair is stored. This is repeated a number of times, depending on what is required by the user, with a changing amplitude. After that, the lowest point of the curve (lowest fidelity $F(|0\rangle)$) is determined and the corresponding amplitude is set for pi-pulse. These are all basic operations that are supported by the microarchitecture. However, the microarchitecture does not support direct extraction of fidelity. This does not mean that it is not possible to execute the calibration. To get the fidelity, the rotation and measurement has to be repeated multiple times (under the same conditions) and then the fidelity can be calculated classically using the measurement data.

Pi/2 pulse calibration is, again, similar to pi-pulse calibration. However, now the experiment is done with 5 pi/2 pulses and then 5 pi/2 pulses followed by a pi-pulse. This creates a graph in the shape of a X, where one line is the first experiment and the other line the second. The point at which the lines intersect is found at the desired pi/2 amplitude. Again, this calibration protocol is supported by the microarchitecture.

Optical Path
Optical path switching is important as it allows the laser to reach the color center when desired. It is easier to control MEMS optical switches per color center than to control the laser itself. Therefore, as also seen in Figure 3.1, there are three MEMS switches per color center. These switches can be controlled using `switchOn` and `switchOff` respectively, where both take a argument with the address of the switch.

Pulse Picking
Pulse picking is extremely important for creating entanglement between color centers. At the time of writing, it is not clear yet what entanglement protocol is going to be used in the project, but there

is an idea, as can be seen in Figure 3.2. The idea is that pulses are 'picked' by two color centers' variable optical attenuator, ensuring that the two centers are excited by the same laser pulse at the exact same time, creating entanglement when reading out the ZPL photondetector. This means that the microarchitecture should at least be able to control the attenuators. This is abstracted to reading/writing a value from/to a register of the attenuator, meaning the microarchitecture can pick pulses.

Photondetectors

The photondetectors play a major part in the diamond architecture. Without it, a lot of operations are not possible. There is a distinction between two types of photondetectors - PSB and ZPL. PSB is mainly used for measurement operations (including initialization). ZPL is used for creating heralded entanglement between two color centers. Just like with the attenuator, it is expected that the photondetectors have their own controller and therefore the microarchitecture only has to read and write to and from a register on the controller, done typically with the `MOV`-instruction.

Dynamic Decoupling

The qubits need to be decoupled dynamically to preserve the state. This is supported through the `decouple`-instruction. This instruction works in both the Python API as in cQASM, because it was added to the `gateset.json` file.

5.2. Compiler Tool

The requirements for the compiler tool, listed in Subsection 4.2.1, are repeated below for convenience.

1. Translate high-level quantum algorithms to diamond microcode
2. Support for all instructions defined in the ISA/micro-ISA
3. Support all diamond-specific protocols/sequences
4. To some extent, schedule necessary diamond-specific operations automatically

To illustrate whether the compiler tool meets these requirements, a set of two examples is created. These examples can be found in Figure 5.1 and Figure 5.2 respectively. The first example is an example of automatic decomposition of a measurement of a qubit using a cQASM input. Because cQASM does not support all instructions, another example has been made that illustrates a Python API input. These examples are shown without the scheduling (magnetic biasing, rabi check, CRC check and initialization) because then the microcode would be too long. A example of scheduling is discussed later in the section.

```
version 1.0

qubits 1

measure q[0]
```

(a) Compiler cQASM input.

```
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0<R33, ResultReg0
```

(b) Compiler Output without scheduling.

Figure 5.1: An example of the in and output of the compiler using the measure-instruction. Note that the input is cQASM input only - the python API looks like in Listing 7.

```
k.diamond_excite_mw(1, 100, 200,
    0, 60, 0)
```

(a) Compiler cQASM input.

```
excite_MW 1, 100, 200, 0, 60, q0
```

(b) Compiler Output without scheduling.

Figure 5.2: An example of the in and output of the compiler using the excite_MW-instruction. This input is from the python API, as the function cannot be called from cQASM.

Translation of high-level algorithms to microcode
As can be seen by the examples, the compiler translates higher-level quantum algorithms, which is only one gate in the example, to diamond microcode.

Support for ISA instructions and diamond protocols
Ultimately, the quantum algorithm should be the main input for the compiler. Within quantum algorithms, calibration protocols are not called manually, just as classical instructions such as AND/OR etc. However, these instructions can be called when testing the quantum computer or under other circumstances, so it would still be wise to have support.

Quantum gate instructions, such as X, Y, Z etc are all supported. The input, either in cQASM as `'x q[0]'` or in the Python API as `k.gate('x', [0])` gets translated to the `qgate`-instruction appropriately. This is the same for the two-qubit operations, such as CNOT and CZ, but instead they get translated to `qgate2`-instructions. The toffoli-gate is also supported - it is properly decomposed to single and two qubit gates. Custom rotations, such as Rx, Ry and Rz gates are also supported. They get translated to custom `excite_MW`-instructions. Furthermore, measurement and initialization are also supported. Entanglement procedures, both color center to color center and color center to nuclear spin qubit are supported, but the final entanglement protocol for color center to color center has not yet been decided on by the people of the Fujitsu Project [2]. Furthermore, the instruction for swapping the state between an electron and a nuclear spin qubit is also supported.

For magnetic biasing, the frequency sweep instruction (`sweep_bias`) and the `calculate_bias` instruction are supported. However, the instruction for calculating the bias value has not been elaborated upon, because it is still needed to figure out how the new value must be determined. Therefore, the microcode instruction is just `calculate_current()`. The same goes for the calculation of the voltage for the CRC. Decoupling according to the XY-8 protocol is also supported.

For waiting, the amount of cycles that needs to be waited needs to provided. Because OpenQL supports waiting in its internal system, this is natively supported. In cQASM, we can just put the wait duration after the wait instruction (`wait 10`).

Finally, the classical standard instructions that are needed to perform flow control and calculations are not supported by the tool. The main reason for this is that they are not called manually, but they could be implemented if needed.

Table 5.2: Overview of the ISA instructions supported by the tool - divided into Python API and with cQASM.

Protocol/Instruction	Python API	cQASM
X, Y, Z, S, T etc	✓	✓
CNOT, CZ etc	✓	✓
excite_mw	✓	✗
measure	✓	✓
initialize	✓	✓
qnop	✓	✗
qentangle	✓	✗
NVentangle	✓	✗
memswap	✓	✗
Magnetic Biasing	✓	✗
Rabi Frequency	✓	✗
Charge Resonance Check	✓	✗
cal_measure	✓	✗
cal_halfpi	✓	✗

Protocol/Instruction	Python API	cQASM
XY-8 Decoupling	✓	✓
Toffoli-gate	✓	✓
calculate_bias	✓	✗
calculate_volt	✓	✗
wait	✓	✓
AND/OR/XOR/NOT	✗	✗
ADD(i)/SUB(i)	✗	✗
MUL/DIV	✗	✗
MOV	✗	✗
LD(i)/ST(i)	✗	✗
BR	✗	✗
jump	✗	✗
cal_pi	✓	✗

Scheduling of Instructions
The compiler does some form of automatic scheduling. The gates are still handled sequentially - that is if the algorithm wants gates in order 1, 2, 3 then the output microcode is also in order 1, 2, 3. In addition,

[2]at the time of writing

the compiler automatically puts the following in front of a algorithm: magnetic biasing for all qubits, rabi frequency determinations for all qubits, charge resonance check for all qubits and initialization for all qubits. It does this even when the qubits are specified but nothing is done with them, meaning the algorithm is empty. These operations can also be called manually from the Python API. Furthermore, every 10 gates, a CRC is also scheduled for each qubit. For one qubit and an empty algorithm, the microcode looks like found in Listing 8.

5.3. Conclusion

In this chapter, we verified the defined microarchitecture in Section 5.1. We saw that the QISA and the microarchitecture meet requirements for deterministic and precise control, including classical instruction support and most of the gates that are defined in the cQASM 1.0 gateset. Furthermore, the QISA and microarchitecture also support all diamond specific protocols such as magnetic biasing, CRC and more. It also fits well in the system-overview that was proposed earlier in the thesis, in Section 3.2. Subsequently, we have verified the tool in Section 5.2. An example was presented and we saw that the tool translated higher-level quantum code, either from the OpenQL Python API or using a cQASM reader gets translated correctly to diamond microcode. It has support for most instructions defined in the ISA, except for the standard classical instructions. This is not an issue, as the classical instructions are not part of quantum algorithms. The cQASM reader of OpenQL limits the functionality with cQASM algorithms as some functions are not supported because of OpenQL's IR. Some gates can be added to the gateset file of the cQASM reader, so the reader can understand the custom gate name and parse it correctly through the OpenQL IR. However, we still recommend to use the Python API as, while it is more messy, it does supports all diamond specific gates.

```
# sweep_bias q[0]
LDi 10, dacReg0
LDi 0, sweepStartReg0
LDi 10, sweepStepReg0
LDi 100, sweepStopReg0
LDi 0, memAddr0
LABEL LAB0
switchOn q0
excite_MW 1, 100, sweepStartReg0, 0, 60, q0
switchOff q0
mov photonReg0, R0
ST R0, memAddr0($0)
ST sweepStartReg0, memAddr0($0)
ADD sweepStartReg0, sweepStartReg0, sweepStepReg0
ADDi memAddr0, 4
BR sweepStartReg0<sweepStopReg0, LAB0

# calculate_current q[0]
calculate_current()

# rabi_check q[0]
LDi 100, R1
LDi 2, R2
LDi 3, R3
LDi 0, R32
LABEL LAB1
LABEL LAB2
LABEL LAB3
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB3
excite_MW 1, R2, 200, 0, 60, q0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0<R33, ResultReg0
ST ResultReg0, memAddress0($0)
ADDi memAddr0, 4
ADDi R32, 1
BR R32<R1, LAB2
ST R2, memAddr0($0)
ADDi memAddr0, 4
ADDi R2, 10
BR R2<R3, LAB1

# crc q[0]
LDi 5, treshReg0
LDi 0, dacReg0
LABEL LAB4
LDi 0, photon Reg0
switchOn q0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>treshReg0, LAB5
calculateVoltage()
JUMP LAB4
LABEL LAB5

# initialize q[0]
LABEL LAB6
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB6

// quantum algorithm here
```

Listing 8: Example of scheduling for a 1-qubit quantum algorithm.

6 Conclusion

This chapter will serve as a conclusion to the thesis project. It will summarize the work that has been conducted in Section 6.1. In the section Section 6.2, the main contributions will be discusses. The research question will also be answered in that section. Section 6.3 will lists any future work recommendations for the project.

6.1. Summary

In Chapter 2, we introduced the reader to some specific topics needed to understand the rest of the thesis. We started with the introduction of quantum information theory in Section 2.1. We explained qubits, quantum gates and quantum algorithms. In Section 2.2, the quantum computer stack, that serves as a lead thread in the quantum computer design process, was discussed. Each level was presented briefly and definitions were made for each layer. Then, in Section 2.3, NV-centers were discussed. NV-centers are discussed because the Fujitsu Project focuses on color center qubits, which NV-centers are. It is made clear what they are, how they can be made and how they can be controlled. This is especially useful, as in Section 2.4 the different quantum computer microarchitectures were presented. We saw that there are a few of them, but none of them are aimed towards color center qubits in diamond. However, a lot of information can still be used to work towards the goal of a color center quantum computer. In the next section, Section 2.5, an overview of quantum networks was given. Quantum networks are important, as when we are scaling the Fujitsu Project computer, it feels and behaves a lot like a network. We learned about why quantum networks and quantum internet are important as they enable applications such as QKD and quantum device communication. Finally, in Section 2.6, we introduced OpenQL, the platform for high-level quantum programming developed by QuTech. OpenQL aims at translating a technology independent algorithm to a technology dependent assembly code.

In Chapter 3, we defined the microarchitecture for diamond quantum computing. We listed the requirements the microarchitecture should have in Section 3.1, where it was determined that it should support all diamond-specific protocols and instructions. Moreover, it should support all cQASM 1.0 instructions as found on the knowledge base of [6]. In Section 3.2, we saw the overall system where the microarchitecture will be part of. The system architecture was introduced to give a better view of the job of the microarchitecture, as well as to give a better overview of the entire project. Next, we introduced the Quantum Instruction Set Architecture (QISA) and the microarchitecture in Section 3.3. We saw that the first draft was created with the help of information obtained from the literature research and information about the required diamond-specific instructions and operations. We also saw that some instructions were decomposable. This resulted in a separation between the QISA and the microarchitecture, where the instructions of the microarchitecture are of the lowest level before entering control electronics. The result of the development was shown in Section 3.4, where the QISA and corresponding microarchitecture were presented. We went over the supported high-level instructions as well as an example of a decomposition.

For the translation from high-level algorithms to diamond microcode instructions, a compiler was made using OpenQL. We discussed the design of the compiler in Chapter 4. First, a brief recap about

OpenQl was given in Section 4.1. We determined that OpenQL takes two inputs: the algorithm and the platform configuration file. Then, in Section 4.2, the design of the compiler was discussed. We learned the design methodology of the compiler and the requirements of the compiler. Using this information, we have seen how to create and add a backend to OpenQL. In addition, we have seen how to create and add a pass to OpenQL. The pass that was added to OpenQL is our microcode translator pass, which schedules the gates and operations of the algorithm and prints the output microcode to an outfile. We have seen the different design aspects of the pass, such as custom instructions with multiple operands and conditional gates. Finally, in Section 4.3, we made comments on how to use the compiler and how to use OpenQL.

In Chapter 5, we verified the defined microarchitecture in Section 5.1. We saw that the QISA and the microarchitecture meet requirements for deterministic and precise control, including classical instruction support and most of the gates that are defined in the cQASM 1.0 gateset. Furthermore, the QISA and microarchitecture also support all diamond specific protocols such as magnetic biasing, CRC and more. It also fits well in the system-overview that was proposed earlier in the thesis, in Section 3.2. Subsequently, we have verified the tool in Section 5.2. An example was presented and we saw that the tool translated higher-level quantum code, either from the OpenQL Python API or using a cQASM reader gets translated correctly to diamond microcode. It has support for most instructions defined in the ISA, except for the standard classical instructions. This is not an issue, as the classical instructions are not part of quantum algorithms. The cQASM reader of OpenQL limits the functionality with cQASM algorithms as some functions are not supported because of OpenQL's IR. Some gates can be added to the gateset file of the cQASM reader, so the reader can understand the custom gate name and parse it correctly through the OpenQL IR. However, we still recommend to use the Python API as, while it is more messy, it does supports all diamond specific gates.

6.2. Main Contributions

The problem statement of this thesis was:

How is a microarchitecture for a quantum computer based on color centers in diamond defined?

In this section, the problem statement is answered by checking whether each goal that was set at the beginning of the thesis is completed. Below, the goals are repeated:

1. Define the overarching QISA that interfaces with the higher layers of the quantum computer stack.
2. Define the microachitecture.
3. Build a compiler that is able to translate a quantum algorithm to quantum assembly, defined by the microarchitecture, using the higher layer and interfaced with a existing compiler framework.

The completion of the goals can be checked by following the methodology specified in Section 1.3. The first task was to identify other quantum microarchitectures. Using the QuTech quantum tack, the first thing that was done was to clear up the definitions of the stack and thus also the definition of microarchitecture. Different microarchitectures that are used within QuTech, such as QuMA, QuMA_v2, CC-Spin and CC were introduced and analyzed. It was explained how the microarchitectures work and can conclude that the first task of identifying other microarchitectures is completed.

Based on the knowledge gained from the other microarchitectures and from the meetings held within the Fujitsu project, a set of requirements on our microarchitecture are drawn. It became clear that the microarchitecture needs to support both classical and quantum instructions, as well as deliver them with precise timing to the classical-to-quantum (electronics) layer. In addition, the microarchitecture should have support for *all* diamond specific protocols and sequences.

Based on the requirements, the microarchitecture could be defined. However, before that, the overarching QISA had to be defined first. The requirements list was looked at, as well as the overall system architecture, which included the electronic devices that need to be controlled, and subsequently defined a set of instructions of which some could be decomposed into smaller instructions. After the QISA had been defined, the microarchitecture could be defined. The microarchitecture supports every instruction specified in the QISA. With the completion of this task the next goals that were set have been reached. Namely, the overarching QISA was defined and the microarchitecture was also defined.

The next task and goal was to build a simple compiler that translates a high-level algorithm (expressed in a high-level quantum programming language) to instructions from the microarchitecture.

The compiler was built and in addition to the translation process, it also decomposes certain instructions and has a simple form of scheduling as well, where it prepares each qubit for the algorithm. The compiler was built using OpenQL. The task and goal of building a compiler are therefore also met.

After the microarchitecture and the compiler have been defined and build respectively, the results are verified. In the verification process, it was checked if the microarchitecture met the requirements that were specified or not. This was also performed for the compiler.

6.3. Future Work

In this section, the future work for the project is discussed.

6.3.1. Development of the Fujitsu Project

The first thing that comes to mind is that the defined microarchitecture needs to keep in-line with the developments of the Fujitsu project. The microarchitecture has now been defined with the system architecture in mind as described in Chapter 3. However, this view can change, meaning some things in the microarchitecture can subsequently change. It is therefore important to always align with the overall system architecture.

For example, it could be that the local controller will grow in size, controlling more than one color center. Fortunately, the microarchitecture that was defined has taken this into account. If the number of electron qubits should grow, then the only changes that are needed is to change the qubit number index in the instructions. There are other changes in the development of the project that have impact on the microarchitecture, such as:

- The growth of the number of controlled color centers by the local controller.

  As described in the previous paragraph, this is supported by the microarchitecture.

- The system that is developed right now (and also the overall system architecture) is for a diamond color center system without the use of cavities.

  Should this happen, the microarchitecture needs changes to account for the use of cavities. At the moment, little is known about these cavities and thus no speculation can be done on the exact changes of the microarchitecture.

- The Quantum Operation Issue Rate (QOIR) of the microarchitecture might be too low - meaning that the system cannot perform the operations quickly enough.

  This may be solved with using a form of quantum SIMD - called single operation multiple qubit (SOMQ). Additionally, a VLIW architecture could be introduced like seen with eQASM/QuMA_v2 that also increases the QOIR. This means that additional hardware is needed as well, that is in charge of fetching instructions and translating them to a usable format. No changes are needed to the microarchitecture instructions.

- The direct driving of nuclear spin qubits is something that is wanted in the future of the Fujitsu project.

  This is currently not entirely supported by the microarchitecture. Using a DDRF-based scheme, one interleaves an RF pulse on the nuclear spin qubit with a pi-pulse on the electron qubit. The pi-pulse is supported, the RF pulse on the nuclear spin qubit is not. However, an instruction can be added easily that will send a RF pulse on a target spin qubit with a certain envelope, amplitude, frequency and duration that will make the direct driving possible.

6.3.2. Alignment with the Fujitsu Global Controller

The proposed microarchitecture is for the local real-time controllers. However, these controllers need to be controlled using the global controller that oversees multiple color centers. The global controller should send control signals along with timing synchronization signals to the local controllers based on the algorithm it receives from the classical control PC. It is of upmost importance to get this right, as

it is fundamental for the correct functioning of the quantum computer. Right now, the experiments are done using a ADwin-based global controller on NV-centers. It would be a good starting point to see how that is executed exactly.

6.3.3. Design of the microarchitecture

Since the microarchitecture has now been defined, the microarchitecture also has to be designed. This means the the components have to be sorted out and linked to each other correctly, as is done with eQASM for CC-Light by Fu, see Section 2.4. The QISA is what is fed into the microarchitecture, therefore the microarchitecture should have a decoder from QISA to microarchitecture before it executes the micro instructions, typically called a microcode unit. In addition, the microarchitecture should have a timing control unit that controls the timing of the instructions in another way than using wait statements to enable smart scheduling. There are a few other things that should be accounted for:

- The number and type of registers.
- The communication protocol with the electronics.
- Possibility of codeword based event control.
- The microarchitecture should be quick enough to not be throttled by the Quantum Operation Issue Rate (QOIR) - the microcode can be coded more densely by using SIMD-based instructions or a VLIW architecture.
- In line with the previous point, the microarchitecture should work more parallel, using more than one thread.
- The aforementioned microcode unit.
- The interface with the classical-to-quantum electronics.

Whilst designing the microarchitecture, it has to be kept in mind that the system should be easily expandable. For now, we assumed that each color-center would have one dedicated local controller, but there have also been ideas that a color centers can share a local controller. Therefore, the local controller microarchitecture has to be flexible and ready for the expansion of the system.

The next step would be the design of a simulator that takes the QISA code as input, converts the QISA code to microcode and has the needed (digital) signals ready at the output of the simulator. The simulator will be a very important phase of the project, as now the functionality, flexibility and adaptability can be tested far more easily than when the entire microarchitecture has to be fabricated for example.

Even thought the design of this microarchitecture is important, the people from the group of Stephanie Wehner have conducted research in the field of the Quantum Internet. Color center based quantum computers have a lot in common with quantum network. Because the Fujitsu project's goal is a distributed system, there are nodes that connect with each other and therefore can resemble a quantum network. New information about quantum networks and quantum internet, including their microarchitecture can be very helpful for the future of this project. Stephan Wong is already in (preliminary) contact with two people from Wehner's group, but further discussions have yet to take place. I highly recommend setting up and participating in these discussions to steer the project in the right direction.

In short:

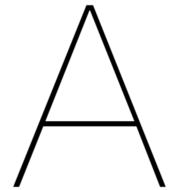
- Design the architecture components.
- Design the simulator and simulate a quantum algorithm.
- Align with Stephanie Wehner's group about the comparison of the diamond microarchitecture and the research towards quantum network microarchitectures.

6.3.4. Improvements of the OpenQL Compiler

The designed compiler is very simple - it supports the translation of algorithm code to microcode, and does very simple scheduling. There are few things that the compiler doesn't support, which it should in the future. Namely:

- Support for differentiation of electron qubits and nuclear spin qubits.
- A library that tells the compiler which qubits pairs can entangle with each other, which are entangled with each other (and therefore also which qubits cannot entangle with each other)
- Improved scheduling - this can also be something that is needed from the operating system (OS) of the quantum computer.

- The entanglement procedures have to be decided upon by the people working on the project and have to be added into the tool correctly.
- Now, values for example duration, frequency and various other things have been a educated guess, because those values are not known for the real color centers. However, as research progresses in these fields, it is important to keep aligning with other work packages.
- Support for all diamond specific instructions from the cQASM reader. This will have the advantage that cQASM is much easier to use than the C++ or Python API of OpenQL. In addition, it is more accessible.



Microarchitecture Documentation

This appendix will go into the details of the ISAs. It will not explain concepts such as decomposition and how the ISAs were designed, but instead it will go over the information in the tables. It will be explained, per instruction, what the instruction is, what it does and how it is used in more detail than is found in the table itself.

The following two pages show the ISA and microarchitecture respectively, but in a larger format than found in the thesis previously.

Table A.1: The QISA for quantum computing in diamond color centers.

Type	Syntax	Description	Decomposable
Qubit Gate	X, Y, Z, S, T qRd	Apply a single Quantum gate on qubit qRd,	No
	CNOT, CZ, CR qRs, nqRd, [<angle>]	Apply a two-qubit Quantum gate with qRs as control and qRd as destination. Angle is optional and only used for CR and CR_k gates.	No
Qubit Rotation	excite_MW <env>, <dur>, <freq>, <phase>, <amplitude>, qRd	Excite the qubit for time duration <duration>with frequency <freq>, phase <phase>and <amplitude> on qubit Rd. The value of these arguments determine the rotation / action of the excitation.	No
Qubit Readout	measure qRd, Rd	Measure qubit qRs and stores result in Rd. First, excite the laser on a particular NV-center. Then, the number of photons are counted by the control electronics and put into a register. The microarchitecture then needs to fetch the result from the control electronics register and store it in a main register.	Yes
Qubit init	initialize qRd	Initialize the qubit to state 0 using readout until no photons are measured.	Yes
nop	qnop	The quantum no-operation. Can be used as a filler operation.	No
Entanglement	qentangle qRs, nqRd	Entangle an NV-center qubit with a nuclear spin qubit. This is a sequence of gates.	Yes
	Nventangle qRs, qRd	Entangle an NV-center qubit with a NV-center qubit. There are multiple steps and thus multiple gates.	Yes
Nuclear Spin Ops	memswap qRs, qRd	Swap the state from the NV-center to the nuclear spin qubit, functioning as memory. Is a sequence of gates.	Yes
Biasing and Checks	sweep_bias <value>, dacReg, <start>, <df>, <fstop>, <memoryaddress>	Set the new current value for the biasing on control electronics register and sweeps microwave whilst storing a frequency/photoncount pair for each swept frequency.	Yes
	decouple qRd	Decouple the qubit according to the XY-8 protocol	Yes
	calculate_bias Rs, Rt, Rd	Calculate the new value for the current in amperes. This operation is purely classical and involves standard instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.	Yes
	calculate_volt Rs, Rt, Rd	Calculate the new value for the voltage in Volt. This operation is purely classical and involves standard instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.	Yes
Calibration	cal_meas qRd	Calibrate the qubit readouttime to ensure optimal readout.	Yes
	cal_pi qRd	Calibrate the qubit rotation amplitude for optimal pi-rotation.	Yes
	cal_halfpi qRd	Calibrate the qubit rotation amplitude for optimal half pi-rotation.	Yes
Timing	wait <value>	Wait an amount of cycles,	No
Standard Instructions	AND/OR/XOR/NOT Rs, Rd	Apply a classical AND/OR/XOR/NOT gate to registers and store result.	No
	ADD(i)/SUB(i) Rs, Rt, Rd	Add or subtract two registers and store the result.	No
	MUL/DIV Rs, Rt, Rd	Multiply or divide two registers and store the result.	No
	MOV Rs, Rd	Move the contents of a register to another register.	No
	LD(i)/ST(i) Rs, Rd	Load from memory to register or Store from register to memory.	No
	BR <comp>, <address>	Jump if the comparisons statement is true.	No
	jump <address>	Jump unconditionally.	No

Table A.2: The microarchitecture for quantum computing in diamond color centers.

Type	**Syntax**	**Description**
Quantum Operation	`qgate <type>, qRd`	Apply a single Quantum gate. <type>defines which gate (X, Y, Z, H, I, S (= pi/2), T (= pi/2)). qRd is the target qubit.
	`qgate2 <type>, qRs, nqRd, [<angle>]`	Apply a two-qubit Quantum gate. <type> defines which gate (CNOT, CZ). qRs is the source qubit and nqRd is the target qubit. The angle argument is optional and only used for CR and CR_k gates.
	`excite_MW <env>, <dur>, <freq>, <phase>, <amplitude>, qRd`	Excite the qubit for time duration <duration>with frequency <freq>, phase <phase>and <amplitude> on qubit Rd. The value of these arguments determine the rotation / action of the excitation.
	`qnop`	Apply a quantum no-operation. Can be used as a filler operation.
Timing	`wait <cycles>`	Wait an amount of <cycles>.
Support	`switchOn <address>, [<duration>]`	Switch the optical switch with address <address>to the ”on” position (optional: for time <duration>).
	`switchOff <address>`	Switch the optical switch with address <address>to the ”off” position.
Standard Instructions	`AND/OR/XOR/NOT Rs, Rd`	Apply a classical AND/OR/XOR gate to registers Rs and Rd, storing the result in Rd.
	`ADD(i)/SUB(i) Rs, Rt, Rd`	Add or subtract two registers Rs and Rt (or values when using immediate), storing the result in Rd.
	`MUL/DIV Rs, Rt, Rd`	Multiply or divide two registers Rs*Rt or Rs/Rt and store the result in Rd.
	`MOV Rs, Rd`	Move the contents of register Rs to Rd.
	`LD/ST Rs/Rd, Rt`	Load from memory to register or Store from register to memory.
	`BR <comparison>, <offset>/Rd`	Jump to <offset>is the <comparison>is true. If the offset is a register, store the comparison result in that register.
	`jump <address>`	Jump to <address>.
	`LDi <value>, Rd`	Load immediate value <value>in register Rd.

A.1. ISA Instructions

A.1.1. Qubit Gate

Qubit gates are divided into two options: 1-qubit gates and 2-qubit gates. 1-qubit gates are gates that act on a single qubit, such as X, Y, and Z.

X, Y, Z, S, T
Operation: X, Y, Z, S, T qRd
Description: Acts out a single qubit gate on qubit qRd. Supported gates are: X, Y, Z, S, T, S dagger, T dagger, Hadamard (H), +/- X90 and +/-Y90. Note that each of these gates can also be executed using the `excite_MW` instruction.
Decomposable: No.
cQASM Equivalent: x q[0]

Example: X q0

CNOT, CZ
Operation: CNOT, CZ qRs, nqRd, [<angle>]
Description: Acts out a two-qubit gate between qubit qRs (control qubit) and qRd (target qubit). Options are: CNOT, CZ, SWAP, PMX90, PMY90, PMX180, PMY180, CR and CR$_k$. When using CR or CR$_k$, the angle option has to be specified. Note that in diamond, two qubit gates can only happen between a electron and a nuclear spin qubit.
Decomposable: No.
cQASM Equivalent: cnot q[0], q[1]

Example: CNOT q0, q1

A.1.2. Qubit Rotation

excite_mw
Operation: excite_mw <envelope>, <duration>, <freq>, <phase>, <amplitude>, qRd
Description: This type of gate is a full custom rotation. The user can set the envelope, duration, frequency, phase, amplitude and target register when calling the instruction.
Decomposable: No.
cQASM Equivalent: -

Example: excite_mw 0, 100, 200, 0, 60, q0

A.1.3. Qubit Readout

measure
Operation: measure qRd, Rd
Description: This operation measures qubit qRd. It then stores the result in register Rd. The result is either ‘0’ or ‘1’.
Decomposable: Yes.
cQASM Equivalent: measure q[0]

Example: measure q0, ResultReg0

A.1.4. Qubit Initialize

initialize
Operation: initialize qRd
Description: Initializes a qubit to the $|0\rangle$-state. Should the qubit be initialized to the $|1\rangle$-state, the initialisation should be followed with an x-gate.
Decomposable: Yes.
cQASM Equivalent: prep_z q[0]

Example: initialize q0

A.1.5. nop

qnop
Operation: qnop
Description: The quantum no-operation. Useful when the system needs to do nothing for a cycle. Is equivalent to waiting one cycle. The instruction is hardly used.
Decomposable: No.
cQASM Equivalent: -

Example: qnop

A.1.6. Entanglement

qentangle
Operation: qentangle qRs, nqRd
Description: Creates quantum entanglement between two qubits, where one qubit (qRs) is the electron of the color center - the so called qubit, and the other qubit (nqRd) is one of the nuclear spin qubits that surround the electron. The nuclear spin qubits are carbon-13 atoms.
Decomposable: Yes.
cQASM Equivalent: -

Example: qentangle q0, nq5

NVentangle
Operation: NVentangle qRs, qRd
Description: This instruction attempts to achieve heralded entanglement between two qubits. Right now, this is implemented according to the protocol of Barrett and Kok [33], but the protocol that will be used in the Fujitsu Project will undoubtedly be something different.
Decomposable: Yes.
cQASM Equivalent: -

Example: NVentangle q0, q1

A.1.7. Nuclear Spin Operations

memswap
Operation: memswap qRs, qRd
Description: Swaps the state of a color center qubit (electron) with a nuclear spin qubit (carbon-13), similar to the normal SWAP operation.
Decomposable: Yes.
cQASM Equivalent: -

Example: memswap q0, nq6

A.1.8. Biasing and Checks

sweep_bias
Operation: sweep_bias qRd, <value>, dacReg, <start>, <df>, <fstop>, <memaddr>
Description: Sweeps the frequency of the laser over qubit qRd. The user also specifies the value of the magnetic biasing, the target DAC (dacReg), the start value of the sweep, the step of the sweep and the stop value of the sweep, all in kHz. The instruction stores the frequency/photoncount pair in memory with address memaddr.
Decomposable: Yes.
cQASM Equivalent: -

Example: sweep_bias q0, 0, dacReg0, 0, 200, 2000, 0

decouple
Operation: decouple qRd
Description: Decouple qubit qRd according to the XY-8 decoupling protocol [36].
Decomposable: Yes
cQASM Equivalent: -

Example: decouple q0

calculate_bias
Operation: calculate_bias Rs, Rt, Rd
Description: Calculates the new value for the current in milli-amperes. This operation is purely classical and involves standard (classical) instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.
Decomposable: Yes.
cQASM Equivalent: -

Example: calculate_bias R0, R1, R0

calculate_voltage
Operation: calculate_voltage Rs, Rt, Rd
Description: Calculates the new value for the voltage in milli-Volts. This operation is purely classical and involves standard (classical) instructions to calculate the new value and put it in a register. First, it fetches the data from memory location Rs and old value Rt, then calculates the new value and puts it in Rd.
Decomposable: Yes.
cQASM Equivalent: -

Example: calculate_voltage R0, R1, R0

A.1.9. Calibration

Additional calibration instructions

cal_measure
Operation: cal_measure qRd
Description: Calibrates the readouttime of the laser to ensure that the readout happens with maximum fidelity.
Decomposable: Yes.
cQASM Equivalent: -

Example: cal_measure q0

cal_pi
Operation: cal_pi qRd
Description: Calibrates the amplitude of the laser to ensure that a pi-rotation happens with maximum fidelity.
Decomposable: Yes.
cQASM Equivalent: -

Example: cal_pi q0

cal_halfpi
Operation: cal_halfpi qRd
Description: Calibrates the amplitude of the laser to ensure that a half pi-rotation happens with maximum fidelity.
Decomposable: Yes.

cQASM Equivalent: -

Example: cal_halfpi q0

A.1.10. Timing

wait
Operation: wait <value>
Description: Waits for <value> cycles. Can be used to stall the system or to wait between operations.
Decomposable: no.
cQASM Equivalent: wait 5, where the system waits 5 time units

Example: wait 5

A.1.11. Standard (Classical) Instructions

AND/OR/XOR/NOT
Operation: AND/OR/XOR/NOT Rs, Rd
Description: Applies the classical bit-wise operation on registers Rs and Rd. Stores the result in register Rd. For NOT, Rs is not used.
Decomposable: No.
cQASM Equivalent: -

Example: AND/OR/XOR r0, r1 ; NOT r2

ADD(i)/SUB(i)
Operation: ADD(i)/SUB(i) Rs, Rt, Rd
Description: Add Rs with Rt and store in Rd. For immediate, Rs can be a integer.
Decomposable: No.
cQASM Equivalent: -

Example: ADD/SUB r0, r1, r0 ; ADDi/SUBi 5, r0, r0

MUL/DIV
Operation: MUL/DIV Rs, Rt, Rd
Description: Multiply Rs with Rt and store in Rd or divide Rs by Rt and store in Rd.
Decomposable: No.
cQASM Equivalent: -

Example: MUL r0, r1, r0

MOV
Operation: MOV Rs, Rd
Description: Move (copy) the contents from register Rs to register Rd.
Decomposable: No.
cQASM Equivalent: -

Example: MOV r0, r1

LD(i)/ST(i)
Operation: LD(i)/ST(i) Rs, Rd($i)
Description: Load or store data from or to memory.
Decomposable: -
cQASM Equivalent: -

Example: LD/ST r0, r1($0) ; LDi/STi 0, r0

BR
Operation: BR <comp>, <register/address>
Description: Branch operation that, if the comparison flag is true, sets the output register to a value of ‘1’, or jumps to a defined label address.
Decomposable: No.
cQASM Equivalent: -

Example: BR r0>r1, LAB1

jump
Operation: jump <address>
Description: Unconditional jump to the specified address.
Decomposable: No.
cQASM Equivalent: -

Example: jump LAB1

A.2. microarchitecture Instructions

A.2.1. Quantum Operations

qgate
Operation: qgate <type>, qRd
Description: Executes a single qubit gate on qubit qRd. Options for <type> are: X, Y, Z, S, T, S dagger, T dagger, Hadamard (H), +/- X90 and +/-Y90.

Example: qgate X, q0

qgate2
Operation: qgate2 <type>, qRs, nqRd
Description: Executes a two qubit gate with qubit qRs as control and qRd as target. Options are: CNOT, CZ, SWAP, PMX90, PMY90, PMX180, PMY180.

Example: qgate2 CNOT, q0, q1

excite_mw
Operation: excite_mw <envelope>, <duration>, <freq>, <phase>, <amplitude>, qRd
Description: Excites a color center electron with a laser with a set envelope, duration, frequency, phase and amplitude. The combination of values determines the rotation on the qubit. <envelope>: for now, either ‘0’ or ‘1’. <duration>: time in us. <freq>: frequency in KHz, <phase>: phase in radians (0-2π), <amplitude>: amplitude in mV

Example: **Example:** excite_mw 0, 100, 200, 0, 60, q0

qnop
Operation: qnop
Description: The quantum no-operation. Useful when the system needs to do nothing for a cycle. Is equivalent to waiting one cycle. The instruction is hardly used.

Example: qnop

A.2.2. Timing

wait
Operation: wait <value>
Description: Waits for <value> cycles. Can be used to stall the system or to wait between operations.

Example: wait 5

A.2.3. Additional Instructions

switchOn
Operation: switchOn qRd, [<duration>]
Description: Switch on the optical path for color center with electron qubit qRd using the MEMS switches. Optional: specify how long the optical path should be active with the duration parameter.

Example: switchOn q0

switchOff
Operation: switchOff qRd
Description: Switch off the optical path for color center with electron qubit qRd using the MEMS switches.

Example: switchOn q0

A.2.4. Standard (Classical) Instructions

AND/OR/XOR/NOT
Operation: AND/OR/XOR/NOT Rs, Rd
Description: Applies the classical bit-wise operation on registers Rs and Rd. Stores the result in register Rd. For NOT, Rs is not used.

Example: AND/OR/XOR r0, r1 ; NOT r2

ADD(i)/SUB(i)
Operation: ADD(i)/SUB(i) Rs, Rt, Rd
Description: Add Rs with Rt and store in Rd. For immediate, Rs can be a integer.

Example: ADD/SUB r0, r1, r0 ; ADDi/SUBi 5, r0, r0

MUL/DIV
Operation: MUL/DIV Rs, Rt, Rd
Description: Multiply Rs with Rt and store in Rd or divide Rs by Rt and store in Rd.

Example: MUL r0, r1, r0

MOV
Operation: MOV Rs, Rd
Description: Move (copy) the contents from register Rs to register Rd.

Example: MOV r0, r1

LD(i)/ST(i)
Operation: LD(i)/ST(i) Rs, Rd($i)
Description: Load or store data from or to memory.

Example: LD/ST r0, r1($0) ; LDi/STi 0, r0

BR
Operation: BR <comp>, <register/address>
Description: Branch operation that, if the comparison flag is true, sets the output register to a value of ‘1’, or jumps to a defined label address.

Example: BR r0>r1, LAB1

jump
Operation: jump <address>
Description: Unconditional jump to the specified address.

Example: jump LAB1

B

Decomposition Microcode

B.1. Measurement

Input: measure q0

```
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0<R33, ResultReg0
```

B.2. Initialization

Input: initialize q0

```
LABEL LAB0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB0
```

B.3. qentangle

Input: qentangle q0, nuq15

```
qgate MX90, q0
qgate2 PMX90, q0, nuq15
qgate X90, q0
```

B.4. NVentangle

Input: NVentangle q0 q1

```
LDi 0, R2
LABEL LAB0
switchOn q0
switchOn q1
excite_MW 1, 100, 200, 0, 60, q0
excite_MW 1, 100, 200, 0, 60, q1
wait 100
```

```
mov R0, photonReg01
switchOff q0
switchOff q1
ADDi R2, 1
wait 50
BR R1>1, LAB1
qgate X, q0
qgate X, q1
mov R0, R1
JUMP LAB0
LABEL LAB1
```

B.5. memswap

Input: memswap q0, nuq1

```
qgate2 PMY90, q0, nuq1
qgate X90, q0
qgate2 PMX90, q0, nuq1
qgate MY90, q0
```

B.6. sweep_bias

Input: sweep_bias 10, 0, 0, 10, 100, 0

```
LDi 10, dacReg0
LDi 0, sweepStartReg0
LDi 10, sweepStepReg0
LDi 100, sweepStopReg0
LDi 0, memAddr0
LABEL LAB0
switchOn q0
excite_MW 1, 100, sweepStartReg0, 0, 60, q0
switchOff q0
mov photonReg0, R0
ST R0, memAddr0($0)
ST sweepStartReg0, memAddr0($0)
ADD sweepStartReg0, sweepStartReg0, sweepStepReg0
ADDi memAddr0, 4
BR sweepStartReg0<sweepStopReg0, LAB0
```

B.7. decouple

Input: decouple q0

```
excite_MW 0, 500, 200, 1.57, 60, q0
wait 50
excite_MW 0, 1000, 200, 1.57, 60, q0
wait 100
excite_MW 0, 1000, 200, 3.14, 60, q0
wait 100
excite_MW 0, 1000, 200, 1.57, 60, q0
wait 100
excite_MW 0, 1000, 200, 3.14, 60, q0
wait 100
excite_MW 0, 1000, 200, 3.14, 60, q0
wait 100
excite_MW 0, 1000, 200, 1.57, 60, q0
```

```
wait 100
excite_MW 0, 1000, 200, 3.14, 60, q0
wait 100
excite_MW 0, 1000, 200, 1.57, 60, q0
wait 50
excite_MW 0, 500, 200, 1.57, 60, q0
```

B.8. calculate_bias

Input: calculate_bias R0, R1, R0

```
calculate_current()
```

B.9. calculate_voltage

Input: calculate_voltage R0, R1, R0

```
calculate_voltage()
```

B.10. cal_meas

Input: cal_meas q0

```
LABEL LAB0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB0
LDi 0, photonReg0
LDi 1, R30
LABEL LAB1
switchOn q0
excite_MW 0, 1, 200, 0, 60, q0
switchOff q0
ST photonReg0, R1($0)
ST R30, R1($0)
ADDi R30, 1
BR R30<40, LAB1
LABEL LAB2
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB2
qgate X, q0
LDi 0, photonReg0
LDi 1, R30
LABEL LAB3
switchOn q0
excite_MW 0, 1, 200, 0, 60, q0
switchOff q0
ST photonReg0, R2($0)
ST R30, R3($0)
ADDi R30, 1
BR R30<40, LAB3
```

```
calculate_readouttime(R0, R1, R2, R3)
```

B.11. cal_pi

Input: cal_pi q0

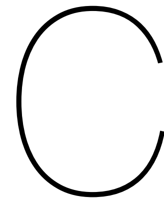
```
LABEL LAB0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB0
LDi 0, R0
LDi 0, R1
LDi 0, R2
LABEL LAB1
LABEL LAB2
switchOn q0
excite_MW 0, 1000, 200, 0, R0, q0
switchOff q0
ADDi R1, 1
BR R1<12, LAB1
measure_fidelity(R0)
ADDi R0, 0.1
ADDi R2, 1
BR R2>10, LAB2
calculate_minimum_fidelity()
```

B.12. cal_halfpi

Input: cal_halfpi q0

```
LABEL LAB0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB0
LDi 0, R0
LDi 0, R1
LDi 0, R2
LABEL LAB1
switchOn q0
excite_MW 0, 500, 200, 0, R0, q0
switchOff q0
ADDi R1, 1
BR R1<7, LAB1
measure_fidelity(R0)
LABEL LAB0
switchOn q0
LDi 0, photonReg0
excite_MW 1, 100, 200, 0, 60, q0
mov photonReg0, R0
switchOff q0
BR R0>0, LAB0
```

```
LDi 0, R0
LDi 0, R1
LDi 0, R2
LABEL LAB2
switchOn q0
excite_MW 0, 500, 200, 0, R0, q0
excite_MW 0, 1000, 200, 0, 60, q0
switchOff q0
ADDi R1, 1
BR R1<7, LAB2
measure_fidelity(R0)
```



List of Supported Functions and Gates of the Compiler

Below follows the documentation for the compiler part. It will go over every instruction that is found in `OpenQL/examples/diamond/test.py` found on GitHub [34].

C.1. Initialization

C.1.1. prep_z

Description: prepare the qubit in $|0\rangle$.
Syntax: `k.gate('prep_z', [qubit])`
cQASM syntax: `prep_z qubit`

Example: `k.gate('prep_z', [0])`

C.1.2. prep_x

Description: prepare the qubit in $|+\rangle$
Syntax: `k.gate('prep_x', [qubit])`
cQASM syntax: `prep_x qubit`

Example: `k.gate('prep_x', [0])`

C.1.3. prep_y

Description: prepare the qubit in $|+i\rangle$
Syntax: `k.gate('prep_y', [qubit])`
cQASM syntax: `prep_y qubit`

Example: `k.gate('prep_y', [0])`

C.1.4. initialize

Description: prepare the qubit in $|0\rangle$.
Syntax: `k.gate('initialize', [qubit])`
cQASM syntax: `prep_z qubit`

Example: `k.gate('initialize', [0])`

C.2. Measurement

C.2.1. measure

Description: measure a qubit in the z-basis and store the result in the corresponding classical register, breg.
Syntax: `k.gate('measure', [qubit])`
cQASM syntax: `measure_z qubit`

Example: `k.gate('measure', [0])`

C.2.2. measure_z

Description: measure a qubit in the z-basis and store the result in the corresponding classical register, breg.
Syntax: `k.gate('measure_z', [qubit])`
cQASM syntax: `measure_z qubit`

Example: `k.gate('measure_z', [0])`

C.2.3. measure_x

Description: measure a qubit in the x-basis and store the result in the corresponding classical register, breg.
Syntax: `k.gate('measure_x', [qubit])`
cQASM syntax: `measure_x qubit`

Example: `k.gate('measure_x', [0])`

C.2.4. measure_y

Description: measure a qubit in the z-basis and store the result in the corresponding classical register, breg.
Syntax: `k.gate('measure_y', [qubit])`
cQASM syntax: `measure_y qubit`

Example: `k.gate('measure_y', [0])`

C.3. Single Qubit Gates

Description: single qubit gates such as x, y, z, s, t, sdag, tdag, h, +/- x90, +/-y90,
Syntax: `k.gate('gatename', [qubit])` or `k.gatename(qubit)`
cQASM syntax: `gatename qubit`

Example: `k.x(0)`

C.4. Two Qubit Gates

Description: two qubit gates such as cnot, cz, swap, pmx90, pmy90, pmx180 and pmy180.
Syntax: `k.gate('gatename', [qubit1, qubit2])`
cQASM syntax: `gatename qubit1, qubit2`

Example: `k.gate('cnot', [0 1])`

C.5. Three Qubit Gate

Description: three qubit gate, such as toffili.
Syntax: `k.gate('gatename', [qubit1, qubit2, qubit3])`
cQASM syntax: `gatename qubit1, qubit2, qubit3`

Example: `k.gate('toffoli', [0, 1, 2])`

C.6. Diamond Calibration

C.6.1. cal_measure

Description: calibrate the readout time for readout with maximum fidelity on a qubit.
Syntax: `k.gate('cal_measure', [qubit])`
cQASM syntax: –

Example: `k.gate('cal_measure', [0])`

C.6.2. cal_pi

Description: calibrate the signal amplitude for pi rotation with maximum fidelity on a qubit.
Syntax: `k.gate('cal_pi', [qubit])`
cQASM syntax: –

Example: `k.gate('cal_pi', [0])`

C.6.3. cal_halfpi

Description: calibrate the signal amplitude for half pi rotation with maximum fidelity on a qubit.
Syntax: `k.gate('cal_halfpi', [qubit])`
cQASM syntax: –

Example: `k.gate('cal_halfpi', [0])`

C.6.4. decouple

Description: execute the xy8-decoupling protocol on a qubit.
Syntax: `k.gate('decouple', [qubit])`
cQASM syntax: –

Example: `k.gate('decouple', [0])`

C.6.5. Custom Rotations

C.6.6. rz

Description: rotate the qubit along the x axis with a specified angle.
Syntax: `k.gate('rz', [qubit], 0, angle`
cQASM syntax: `Rz qubit, angle`

Example: `k.gate('rz', [0], 0, 1.57`

C.6.7. rx

Description: rotate the qubit along the x axis with a specified angle.
Syntax: `k.gate('rx', [qubit], 0, angle`
cQASM syntax: `Rx qubit, angle`

Example: `k.gate('rx', [0], 0, 1.57`

C.6.8. ry

Description: rotate the qubit along the y axis with a specified angle.
Syntax: `k.gate('ry', [qubit], 0, angle`
cQASM syntax: `Ry qubit, angle`

Example: `k.gate('ry', [0], 0, 1.57`

C.6.9. cr

Description: rotate the target qubit along the z axis with a specified angle, based on the state of the control qubit.

Syntax: `k.gate('cr', [qubit1 qubit2], 0, angle`
cQASM syntax: `CR qubit1, qubit2, angle`

Example: `k.gate('cr', [0 1], 0, 1.57`

C.6.10. crk

Description: rotate the target qubit along the z axis with a specified angle, calculated as the angle $= \frac{\pi}{2^k}$, based on the state of the control qubit.
Syntax: `k.gate('crk', [qubit1 qubit2], 0, k`
cQASM syntax: `CRK qubit1, qubit2, k`

Example: `k.gate('crk', [0 1], 0, 1`

C.7. Diamond Protocols and Sequences

C.7.1. crc

Description: charge resonance check (crc) for a qubit. Take three arguments, qubit, a threshold value and a starting value for the voltage.
Syntax: `k.diamond_crc(qubit, threshold, value)`
cQASM syntax: –

Example: `k.diamond_crc(0,30,5)`

C.7.2. rabi_check

Description: determines the rabi frequency for a qubit. Takes as input the qubit, the duration of the signal, the amount of measurements and the end value for the duration.
Syntax: `k.diamond_rabi_check(qubit, duration, measurements, t_max)`
cQASM syntax: –

Example: `k.diamond_rabi_check(0, 100, 2, 3)`

C.7.3. excite_mw

Description: Excites the electron qubit of a color center with a custom laser pulse. The user can specify the envelope, duration, frequency, phase and amplitude of the signal.
Syntax: `k.diamond_excite_mw(envelope, duration, frequency, phase, amplitude, qubit)`
cQASM syntax: –

Example: `k.diamond_excite_mw(1, 100, 200, 0, 60, 0)`

C.7.4. qentangle

Description: Entangles a electron qubit with a nuclear spin qubit.
Syntax: `k.diamond_qentangle(qubit,nuqbit`
cQASM syntax: –

Example: `k.diamond_qentangle(0,15)`

C.7.5. nventangle

Description: Entangles two color centers according to a protocol.
Syntax: `k.gate('nventangle', [qubit1, qubit2])`
cQASM syntax: –

Example: `k.gate('nventangle', [0, 1])`

C.7.6. memswap

Description: Swaps the state between a electron spin qubit and a nuclear spin qubit of the same color center.
Syntax: `k.diamond_memswap(qubit,nuqubit)`
cQASM syntax: –

Example: `k.diamond_memswap(0,1)`

C.7.7. sweep_bias

Description: sweeps the frequency of the laser over a qubit, storing the data pair of detected photons and frequency. Is used for magnetic biasing, in combination with calculate_current.
Syntax: `k.diamond_sweep_bias(qubit, value, dacreg, start, step, max, memaddress)`
cQASM syntax: –

Example: `k.diamond_sweep_bias(0, 10, 0, 0, 10, 100, 0)`

C.8. Timing

C.8.1. wait

Description: wait for x amount of cycles.
Syntax: `k.gate('wait', [], value)`
cQASM syntax: `wait value`

Example: `k.gate('wait', [], 200)`

C.8.2. qnop

Description: the quantum no-operation. Equivalent to waiting 1 cycle.
Syntax: `k.gate('qnop', [0])`
cQASM syntax: `wait 1`

Example: `k.gate('qnop', [0])`

C.9. Classical Support Functions

C.9.1. calculate_current

Description: calculates the new value for the current. To be used in combination with sweep_bias for magnetic biasing.
Syntax: `k.gate('calculate_current', [0]`
cQASM syntax: –

Example: `k.gate('calculate_current', [0])`

C.9.2. calculate_voltage

Description: calculates the new value for the voltage. Is used in the charge resonance check.
Syntax: `k.gate('calculate_voltage', [0]`
cQASM syntax: –

Example: `k.gate('calculate_voltage', [0])`

Bibliography

[1] C. G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K. Bertels, “The engineering challenges in quantum computing,” in *Design, Automation Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 836–845.

[2] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.

[3] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC ’96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866

[4] D-Wave, “D-Wave Systems.” [Online]. Available: https://www.dwavesys.com/

[5] Google, “Quantum Supremacy Using a Programmable Superconducting Processor,” 23rd of October 2019. [Online]. Available: https://ai.googleblog.com/2019/10/quantum-supremacy-using-programmable.html

[6] QuTech, “Quantum inspire,” accessed 29-3-2021. [Online]. Available: https://www.quantum-inspire.com/

[7] DelftX, “DelftX: QTM2x - The Hardware of a Quantum Computer.” [Online]. Available: https://courses.edx.org/courses/course-v1:DelftX+QTM2x+3T2020/course/

[8] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th ed. USA: Cambridge University Press, 2011.

[9] S.-M. Wikipedia, “Bloch sphere,” accessed 29-3-2021. [Online]. Available: https://en.wikipedia.org/wiki/Bloch_sphere#/media/File:Bloch_sphere.svg

[10] V. Acosta and P. Hemmer, “Nitrogen-vacancy centers: Physics and applications,” *MRS Bulletin*, vol. 38, no. 2, p. 127–130, 2013.

[11] S. Prawer and A. D. Greentree, “Diamond for quantum computing,” *Science*, vol. 320, no. 5883, pp. 1601–1602, 2008. [Online]. Available: https://science.sciencemag.org/content/320/5883/1601

[12] M. S. Blok, N. Kalb, A. Reiserer, T. H. Taminiau, and R. Hanson, “Towards quantum networks of single spins: analysis of a quantum memory with an optical interface in diamond,” *Faraday Discuss.*, vol. 184, pp. 173–182, 2015. [Online]. Available: http://dx.doi.org/10.1039/C5FD00113G

[13] T. H. Taminiau, J. Cramer, T. van der Sar, V. V. Dobrovitski, and R. Hanson, “Universal control and error correction in multi-qubit spin registers in diamond,” *Nature Nanotechnology*, vol. 9, no. 3, pp. 171–176, Mar 2014. [Online]. Available: https://doi.org/10.1038/nnano.2014.2

[14] A. Mohtashami, M. Frimmer, and A. F. Koenderink, “Quantum efficiency of single nv centers in nanodiamonds,” in *2013 Conference on Lasers Electro-Optics Europe International Quantum Electronics Conference CLEO EUROPE/IQEC*, 2013, pp. 1–1.

[15] B. C. Rose, D. Huang, A. M. Tyryshkin, S. Sangtawesin, S. Srinivasan, D. J. Twitchen, M. L. Markham, A. M. Edmonds, A. Gali, A. Stacey, W. Wang, U. D. Johansson, A. Zaitsev, S. A. Lyon, and N. P. de Leon, “New color centers in diamond for long distance quantum communication,” in *2017 Conference on Lasers and Electro-Optics (CLEO)*, 2017, pp. 1–1.

[16] J. Wang, S. Paesani, R. Santagati, S. Knauer, A. A. Gentile, N. Wiebe, M. Petruzzella, A. Laing, J. G. Rarity, J. L. O'Brien, and M. G. Thompson, "Learning nitrogen-vacancy electron spin dynamics on a silicon quantum photonic simulator," in *2017 Conference on Lasers and Electro-Optics (CLEO)*, 2017, pp. 1–1.

[17] X. Fu, "Quantum Control Architecture: Bridging the Gap between Quantum Software and Hardware," Ph.D. dissertation, TU Delft, 2018.

[18] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, "eqasm: An executable quantum instruction set architecture," 2019.

[19] M. Serrao Morato Moreira, "QuTech Central Controller: A Quantum Control Architecture for a Surface-17 Logical Qubit," Master's thesis, TU Delft, 2019.

[20] A. Yadav, "CC-Spin: A Micro-architecture design for scalable control of Spin-Qubit Quantum Processor," Master's thesis, TU Delft, 2019.

[21] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. Oliveira, M. Papendrecht, J. Rabbie, F. Rozpedek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. T. Knoop, D. Elkouss, and S. Wehner, "Netsquid, a discrete-event simulation platform for quantum networks," 2020.

[22] W. Kozlowski, A. Dahlberg, and S. Wehner, "Designing a quantum network protocol," *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, Nov 2020. [Online]. Available: http://dx.doi.org/10.1145/3386367.3431293

[23] A. Dahlberg, M. Skrzypczyk, T. Coopmans, L. Wubben, F. Rozpedek, M. Pompili, A. Stolk, P. Pawełczak, R. Knegjens, J. de Oliveira Filho, and et al., "A link layer protocol for quantum networks," *Proceedings of the ACM Special Interest Group on Data Communication*, Aug 2019. [Online]. Available: http://dx.doi.org/10.1145/3341302.3342070

[24] K. Azuma, S. Bäuml, T. Coopmans, D. Elkouss, and B. Li, "Tools for quantum network design," 2020.

[25] A. Dahlberg and S. Wehner, "Simulaqron—a simulator for developing quantum internet software," *Quantum Science and Technology*, vol. 4, no. 1, p. 015001, Sep 2018. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/aad56e

[26] S. DiAdamo, J. Nötzel, B. Zanger, and M. M. Beşe, "Qunetsim: A software framework for quantum networks," 2020.

[27] B. Bartlett, "A distributed simulation framework for quantum networks and channels," 2018.

[28] T. Matsuo, "Simulation of a dynamic, ruleset-based quantum network," 2019.

[29] X. Wu, A. Kolar, J. Chung, D. Jin, T. Zhong, R. Kettimuthu, and M. Suchara, "Sequence: A customizable discrete-event simulator of quantum networks," 2020.

[30] QuTech, "OpenQL Documentation," accessed 25-03-2021. [Online]. Available: https://openql.readthedocs.io/en/latest/

[31] B. Hensen, H. Bernien, A. Dréau, A. Reiserer, N. Kalb, M. Blok, J. Ruitenberg, R. Vermeulen, R. Schouten, C. Abellan, W. Amaya, V. Pruneri, M. Mitchell, M. Markham, D. Twitchen, D. Elkouss, S. Wehner, T. Taminiau, and R. Hanson, "Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres," *Nature*, vol. 526, 10 2015.

[32] M. Pompili, S. L. N. Hermans, S. Baier, H. K. C. Beukers, P. C. Humphreys, R. N. Schouten, R. F. L. Vermeulen, M. J. Tiggelman, L. dos Santos Martins, B. Dirkse, and et al., "Realization of a multinode quantum network of remote solid-state qubits," *Science*, vol. 372, no. 6539, p. 259–264, Apr 2021. [Online]. Available: http://dx.doi.org/10.1126/science.abg1919

[33] S. D. Barrett and P. Kok, “Efficient high-fidelity quantum computation using matter qubits and linear optics,” *Physical Review A*, vol. 71, no. 6, Jun 2005. [Online]. Available: http://dx.doi.org/10.1103/PhysRevA.71.060310

[34] QE-Lab, “QE-Lab/OpenQL.” [Online]. Available: https://github.com/QE-Lab/OpenQL

[35] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, “Qx: A high-performance quantum computer simulation platform,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017.

[36] M. A. Ali Ahmed, G. A. Álvarez, and D. Suter, “Robustness of dynamical decoupling sequences,” *Physical Review A*, vol. 87, no. 4, Apr 2013. [Online]. Available: http://dx.doi.org/10.1103/PhysRevA.87.042309