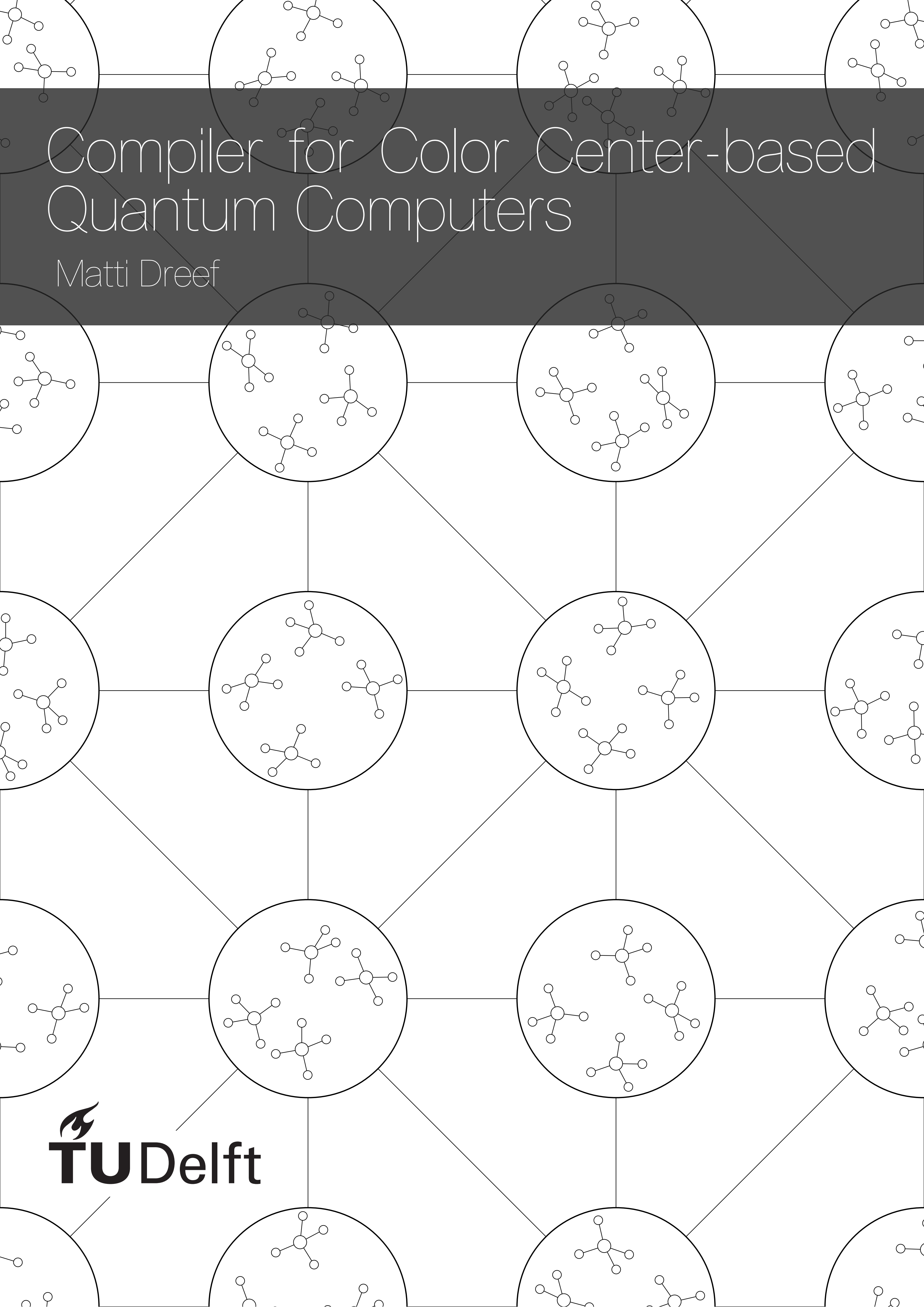
Compiler for Color Center-based Quantum Computers

Matti Dreef



Compiler for Color Center-based Quantum Computers

by

Matti Dreef

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday, April 25th, 2023 at 9:30 AM.

Student number:	4456122	
Project duration:	January, 2022 – April, 2023	
Thesis committee:	Prof. dr. ir. J.S.S.M. Wong,	TU Delft, supervisor
	Dr. D. Coronas Elkouss,	TU Delft
	Dr. S. Feld,	TU Delft

TU Delft

Abstract

Quantum computing is a promising means of satisfying the ever-increasing need for more and faster computations. While some specific applications can already benefit from quantum computing today, a large-scale general-purpose quantum computer is yet to be developed. Many different technologies are being explored to reach this goal, *color centers in diamond* being a promising candidate. This thesis focuses on a specific kind of color center: the nitrogen-vacancy center (NV center).

Building on top of existing work, a quantum instruction set architecture (QISA) for a quantum computer based on NV centers is designed. A compiler targeting this QISA is designed and implemented using the OpenQL framework. In this process alternative approaches are also explored and contributions useful outside the context of NV centers are made. The final compiler is able to take a quantum circuit operating on logical qubits and transform it into a sequence of QISA instructions operating on physical qubits. This functionality integrates seamlessly with the existing passes in OpenQL, allowing for further extension and the possibility to make use of future developments.

Additional functionality includes optimization, scheduling, and hardware-oriented features such as quantization of operands. Each step is easily configurable using a multitude of scripts and data files, and additional steps can be added in the future if needed.

The functionality of the developed compiler is demonstrated by compiling several circuits, some of which are validated by passing their output through a simulator. The compiler configuration as used in these tests is able to transform logical circuit into simple physical circuits, but the compiler provides all the functionality to use more complicated logical qubits which incorporate quantum error correction.

Preface

After my first thesis topic got stuck in finding what exactly I could contribute, Stephan offered me the possibility of working on a quantum computing project instead. While at the start of this project I had practically no knowledge about quantum computing, it turned out that the part I have been working on fits neatly between quantum algorithms on one side and quantum physics on the other, with basic quantum information theory being enough for me during most of the project. A continuing challenge has been the fact that this thesis focuses mostly on software, while my main interest is in hardware design. Nonetheless I think this project was a great opportunity for a slight sidetrack, and the field of quantum computing in general has definitely sparked my interest.

Completing this thesis would not have been possible without the support of the people around me. I would like to thank Fujitsu Limited and Delft University of Technology for setting up the project this thesis contributes to. I would like to thank Stephan Wong for all the opportunities he has opened up for me and his guidance throughout this project, the group of Sebastian Feld for their help with quantum algorithms and software, and David Elkouss for his advice early on in my thesis that eventually led to the final results. I would also like to thank Folkert de Ronde for his input in many aspects of this project and helping me out with the 'real' quantum physics when needed. My thanks also go out to all the fellow students and employees on the 10th and 11th floor for the fun times outside of studying, from group dinners to just sharing a coffee break.

Finally, I would like to thank my friends and family for their continuous support. Richard, you played a great part in inspiring me to pursue the direction of electrical and computer engineering, and I can always count on you for technical and personal support or just some fun and banter. Thank you for being an amazing friend for all these years. Jori, I've come to realize that by being the younger sibling I always had someone who had to figure out the next step in life before me. Thank you for making all those decisions a little less overwhelming. Mom and dad, your never-ending support and encouragement means the world to me. Thank you.

Matti Dreef
Delft, April 2023

Notation

The Glossary lists important terms and their definitions in the scope of this thesis. This is intended both as a reference while reading and to avoid any confusion when certain terms have a different meaning in other fields or inconsistent meaning within the field of quantum computing. When such a term is first introduced it is accompanied by a definition such as the following.

Definition: Example

Example of a definition. This text is also found in the glossary.

Furthermore, this thesis uses colored text to distinguish cross-references, citations, and URLs. In the electronic version these references can be followed by clicking the text.

Contents

1 Introduction

Quantum computing is a promising means of satisfying the ever-increasing need for more and faster computations. By utilizing superposition and entanglement of quantum mechanical systems, certain computations can be carried out exponentially faster than classical computers.

While some specific applications can already benefit from quantum computing today [1], a large-scale general-purpose quantum computer is yet to be developed. Many different technologies are being explored to reach this goal, *color centers in diamond* being a promising candidate. This thesis focuses on a specific kind of color center: the nitrogen-vacancy center (NV center).

1.1. Fujitsu Project

To further explore the possibilities of NV center-based quantum computing, Fujitsu Limited and Delft University of Technology have started a collaboration with the Fujitsu Project. The goal of this project is to develop a scalable quantum computer architecture using color centers. More specifically, the focus at this point in time is on NV centers, while in the future also tin-vacancy centers are to be utilized.

1.2. Problem Statement

As with classical computers, at the lowest level quantum computers work with basic operations that have practical use only when combined into more complex algorithms. Additionally, the qubits a quantum computer operates on are inherently analog. The inevitable noise limits the usability of a single qubit, requiring multiple to be combined for practical quantum algorithms.

This motivates the use of a compiler that can translate a high-level quantum algorithm to the corresponding low-level operations. The goal of this thesis is to design and implement a compiler capable of performing this translation for a quantum computer utilizing NV center qubits. Additionally, this requires defining a quantum instruction set architecture (QISA) serving as an interface between the quantum software and hardware. These requirements are summarized in the main research question of this thesis:

> **Can we design and implement a functional compiler that targets a quantum instruction set architecture for quantum computers based on NV centers?**

1.3. Methodology

In order to answer the research question the following subtasks have been identified:

1. **Identify how NV centers need to be controlled.**
   This encompasses defining the low-level operations that can be applied directly to NV centers.
2. **Design and extend a QISA for a quantum computer based on NV centers.**
   This requires defining high-level instructions that can be implemented using the available low-level operations.
3. **Design and implement a compiler targeting this QISA.**
   This consists of identifying NV center-specific features, formulating and designing the required

compiler functionality, and implementing it using a new or existing compiler framework.

4. **Verify the functionality of the compiler.**
   This involves comparing the compiler output to the expected output for simple cases, as well as evaluating the scalability of the implementation with more complex situations.

The focus in this thesis is on making the compiler *functional*, and not necessarily directly usable for large-scale applications. The compiler should however be able to adapt and scale to accommodate these applications in the future. A similar approach will be taken for the QISA; it should be designed for the *current* outlook of the quantum hardware while also taking the possibility of future extensions into consideration.

1.4. Thesis Overview

The remainder of this thesis is structured as follows; Chapter 2 introduces the quantum theory and NV center physics relevant to the scope of this thesis, as well as outlining the state-of-the-art on quantum compilers and control systems. In Chapter 3 the design of a new control architecture and compiler is discussed. The implementation of the compiler is described in Chapter 4, after which Chapter 5 shows how the compiler can be used in practice. Finally, Chapter 6 concludes this thesis and suggests potential improvements and future work.

2 Background

This chapter provides background information and introduces existing work relevant to this thesis. A theoretical background of quantum computing is given in Section 2.1. Section 2.2 introduces NV centers, the qubit technology that is the focus of this thesis. Section 2.3 describes existing work related to the classical digital control system surrounding a quantum computer. A comparison of existing compiler techniques, full compiler toolchains, and how these can be applied to NV centers is given in Section 2.4.

2.1. Quantum Computing Theory

The following sections first introduce individual qubits and their mathematical notation, followed by multiple qubits and quantum gates. Subsequently quantum circuits are discussed, and finally how qubits are used in practice with quantum error correction. Focus is given to the aspects relevant to this thesis. Many resources are available for a more complete background of quantum computing, as for example provided by QuTech Academy [2].

2.1.1. Qubits

Where a classical bit can represent either 0 or 1, a quantum bit (qubit) can represent not just these states but also a *superposition* of both. This can be seen as the probabilities that the qubit is in one specific state, such as 30% 0 and 70% 1. The possibility of being in a superposition is one of the key features that gives quantum computing an advantage over classical computing.

The state of a single qubit can be represented by a column vector $\begin{bmatrix}\alpha & \beta\end{bmatrix}^T$ with α and β complex numbers. The 'probabilities' of the qubit being in the 0 or 1 state are $P_0 = ||\alpha||^2$ and $P_1 = ||\beta||^2$. These probabilities must add up to 100% which translates to the vector having to be being normalized, i.e., it must have length 1.

Ket Notation & Basis Vectors

As will become clear later, when multiple qubits are introduced the vector notation becomes unwieldy very quickly. As alternative the *ket* notation is commonly used, which denotes a qubit state as $|\psi\rangle$ where ψ is a distinguishing identifier. Kets are actually only half of what is known as bra-ket notation, but only kets are of interest for the purpose of this thesis.

While the ket notation theoretically allows for any description to be used as identifier, a number of standard qubit states known as the basis vectors can be defined which are shown below. Each pair forms an orthogonal basis of the qubit state space. The pairs are each named after a Cartesian axis, the reason for which will become clear in the next section. Note how the identifiers for the basis vectors are based on the value of their second component, while the first component is kept as simple as possible.

Z-basis

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

X-basis

$$|+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix}$$

Y-basis

$$|+i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\i\end{bmatrix} \qquad |-i\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-i\end{bmatrix}$$

Arbitrary qubit states can now be written as linear combinations of these basis vectors, for example the qubit state shown previously can be written as

$$|\psi\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \alpha\begin{bmatrix}1\\0\end{bmatrix} + \beta\begin{bmatrix}0\\1\end{bmatrix} = \alpha\,|0\rangle + \beta\,|1\rangle\,.$$

It is also possible to use other basis vectors here, or use combinations of arbitrary qubit states. This suggests these other vectors can also be used as 'default' basis. Indeed, the qubit state $\begin{bmatrix}\alpha & \beta\end{bmatrix}^T$ is only a representation in the Z-basis. The same state can be written as a column vector with coefficients corresponding to the X- or Y-basis as well, although the Z-basis is the most common.

The Bloch Sphere

An alternative but very useful representation of a qubit state is the *Bloch vector*. Using the polar form of the complex numbers α and β, the representation used in the previous section can be written as

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = e^{i\delta_0} r_0\,|0\rangle + e^{i\delta_1} r_1\,|1\rangle\,.$$

As mentioned previously the length of the vector must be 1, which is the same as requiring the point (r_0, r_1) to lie on a unit circle. This constraint means the values of r_0 and r_1 can be represented using a single angle θ as $r_0 = \cos(\theta/2)$ and $r_1 = \sin(\theta/2)$. The reason for dividing θ by 2 will become clear later. This gives

$$|\psi\rangle = e^{i\delta_0}\cos(\theta/2)\,|0\rangle + e^{i\delta_1}\sin(\theta/2)\,|1\rangle\,.$$

Taking out the common factor in the complex angles allows them to be written as

$$e^{i\delta_0} + e^{i\delta_1} = e^{i\delta_0}(1 + e^{i(\delta_1 - \delta_0)}).$$

Applying this to the qubit state with $\phi = \delta_1 - \delta_0$ gives

$$|\psi\rangle = e^{i\delta}\left(\cos(\theta/2)\,|0\rangle + e^{i\phi}\sin(\theta/2)\,|1\rangle\right)$$

This finally leads to the Bloch vector, which represents a qubit state as a point on the surface of a radius-1 sphere with $\theta \in [0, \pi]$ the polar angle and $\phi \in [0, 2\pi]$ the azimuthal angle as shown in Figure 2.1. The common factor δ known as the *global phase* turns out to have no physical effect, so it is ignored in the Bloch vector representation. The range of θ together with the division by 2 limits the sine and cosine argument to a single quadrant; other quadrants are merely sign changes which get absorbed by the exponents.

Figure 2.2 shows the basis vectors introduced previously as Bloch vectors. It can be seen that each pair of vectors aligns with an axis which corresponds to their basis. More generally, any two opposing Bloch vectors are orthogonal to each other in Cartesian form, allowing such a pair to be used as orthogonal basis.

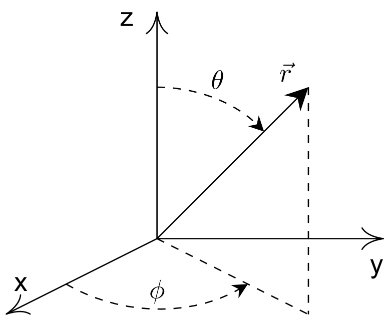


Figure 2.1: Angles representing a point on the surface of a sphere.

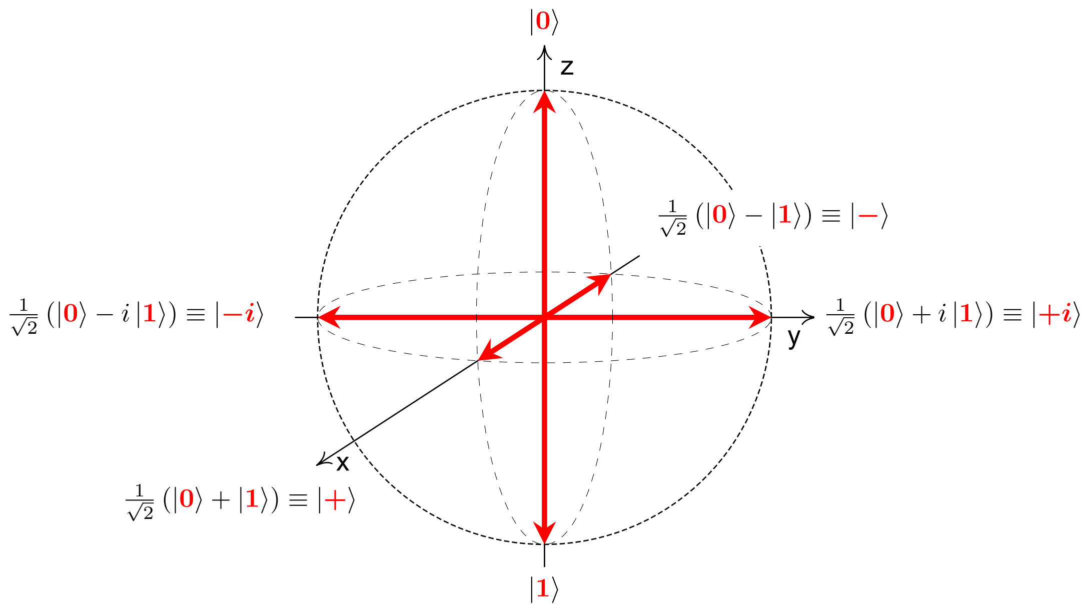

$|0\rangle$
z
$\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \equiv |-\rangle$
$\frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \equiv |-i\rangle$
y
$\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \equiv |+i\rangle$
x
$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \equiv |+\rangle$
$|1\rangle$

Figure 2.2: Bloch sphere visualization of a single qubit.

When applying these theoretical concepts to real physical qubits, the frame of reference can be chosen arbitrarily. Since the Z-basis is most often used it is usually chosen as the most convenient physical axis to work with.

Multiple Qubits

Just by itself, introducing another qubit to a quantum system does not do much. Without any interaction each qubit can still be represented individually using a vector, ket notation, or as Bloch vector. The two qubits can also be represented in a combined state, which can be obtained by taking the *tensor product* of their individual states. This is represented in ket notation as

$$(\alpha_0 |0\rangle + \beta_0 |1\rangle) \otimes (\alpha_1 |0\rangle + \beta_1 |1\rangle) = \alpha_0\alpha_1 |00\rangle + \alpha_0\beta_1 |01\rangle + \beta_0\alpha_1 |10\rangle + \beta_0\beta_1 |11\rangle$$

and in vector form as

$$\begin{bmatrix}\alpha_0\\ \beta_0\end{bmatrix} \otimes \begin{bmatrix}\alpha_1\\ \beta_1\end{bmatrix} = \begin{bmatrix}\alpha_0\alpha_1\\ \alpha_0\beta_1\\ \beta_0\alpha_1\\ \beta_0\beta_1\end{bmatrix}.$$

The Bloch vector representation can also be extended to multiple qubits but is less useful than in the single qubit case, so it will not be discussed here,

These examples show how adding a qubit to a quantum system multiplies the number of terms required to represent that system by 2. Thus, adding a third qubit results in a total of 8 terms, a fourth in 16 terms, and so on continuing with exponential growth.

So far the combined state of the qubits is nothing more than a very tedious and redundant way of representing the individual qubit states, since they can be separated again back into their original components. This changes however when qubits start to interact with each other.

2.1.2. Quantum Gates

To be able to perform computations using qubits it is necessary to manipulate them somehow using *quantum gates*.

The term ‘gate’ suggests a similarity with classical gates such as AND, OR, and NOT, but this is only partially true. First, a quantum gate can not consume or create qubits, and as such the number of outputs must be equal to the number of inputs. Secondly, a quantum gate must be *reversible*, meaning each input produces a unique output such that the input can be reconstructed.

Using the vector notation, quantum gates can be represented by a matrix that is multiplied with the quantum state to produce an output vector. Arbitrary gates can be represented by matrices, but to comply with the reversibility requirement the matrix must be *unitary*, meaning its conjugate transpose is equal to its inverse. That is, a matrix U is unitary if

$$UU^\dagger = UU^{-1} = I.$$

Definition: Gate

A reversible, unitary operation on any number of qubits. Depending on the context it can also be specifically referred to as a *quantum gate*.

Single-qubit Gates

Regardless of the physical implementation of a qubit and its control system, any quantum gate on a single qubit effectively ‘rotates’ the Bloch vector some angle along a specified axis. For this reason single-qubit gates are also called rotations, and in this thesis this term is used specifically for single-qubit gates only.

Definition: Rotation

A single-qubit gate, for example an 180° X or 90° Y rotation.

Using vector notation, the Pauli matrices $\sigma_x, \sigma_y, \sigma_z$ represent rotations of 180° along the X, Y, and Z axes, respectively.

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Besides the Pauli matrices a number of other single-qubit rotations are commonplace, a notable example being the *Hadamard* or H-gate. This gate performs a 180° rotation around the axis 45° between the +Z and +X directions. Its matrix representation is

$$\hat{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

As an example, applying the Hadamard gate to the $|0\rangle$-state gives

$$\hat{H}\,|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = |+\rangle\,.$$

Gates can also be applied based on some external condition, performing the gate only if the condition evaluates to true. These are referred to as *conditional gates*.

Definition: Conditional gate

A gate with classical control, likewise for a *conditional rotation*.

Multi-qubit Gates

As with single-qubit gates, multi-qubit gates can be represented by a matrix. The controlled NOT gate (CNOT) for example, is a 2-qubit gate that performs an X-rotation on one qubit based on the state of another with a matrix representation as shown in (2.1). The first qubit is also referred to as the 'target', while the second is the 'source' or 'control' qubit. This makes the CNOT a *controlled* gate, or more specifically a controlled X-gate. Controlled gates should not be confused with a conditional gates; a controlled gate has qubit control, while a conditional gate has classical control.

Definition: Controlled gate

A gate with qubit control, for example a CNOT.

Using a qubit as control works identical to classical control when the qubit is in the $|0\rangle$ or $|1\rangle$ state. When the control qubit is in a superposition however, performing the CNOT gives rise to *entanglement*. To illustrate what happens, we start with two qubits: one as control in the $|+\rangle$ state and one as target in the $|0\rangle$ state. Their combined state is

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\1\\0\end{bmatrix}.$$

Multiplying this with the matrix representation of the CNOT gate gives

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\1\\0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}. \tag{2.1}$$

Unlike the initial combined qubit state, this outcome can *not* be written as the product of two separate states anymore. At this point the qubits are said to be entangled. Extending entanglement to even more qubits, the number of terms required to represent the combined state grows exponentially as mentioned in Section 2.1.1. Additionally, a measure of exactly how entangled two qubits are can be defined. This is given by the *concurrence*

$$C\left(|\psi\rangle\right) = 2\left|\alpha_{00}\alpha_{11} - \alpha_{01}\alpha_{10}\right|,$$

which is nonzero for entangled states. When $C = 1$ the two qubits are *maximally entangled*, as for example is the case for the result of (2.1).

Where single qubits have the X-, Y-, and Z-basis, there is also a commonly used 2-qubit basis known as the *Bell basis*, the basis vectors of which represent pairs of maximally entangled qubits.

$$|\Phi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}^T \qquad |\Psi_+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix}^T$$

$$|\Phi_-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 0 & -1\end{bmatrix}^T \qquad |\Psi_-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}0 & 1 & -1 & 0\end{bmatrix}^T$$

Other single-qubit gates also have controlled counterparts, such as the controlled Y- or Z-gates or even controlled arbitrary rotations. Another common 2-qubit gate is the swap gate, which as the name implies simply swaps the states of two qubits.

Going beyond 2 qubits there are only few standard or common gates. A notable example is the 3-qubit *Toffoli* gate, also known as the CCNOT or CCX gate. This is an extension of the 2-qubit CNOT gate with multiple control qubits. Only if both control qubits are in the $|1\rangle$-state is the target qubit fully flipped. This can be extended to any number of control qubits.

2.1.3. Quantum Operations

Unitary gates are just one kind of operation that can be performed on qubits. There are also non-unitary operations which by definition are non-reversible and destroy (part of) the existing qubit state. Nevertheless they are vital for useful quantum algorithms.

Definition: Quantum operation

Any action that can be performed on qubits, including gates, non-unitaries, and platform-specific operations.

Initialization

For a quantum circuit to be useful the initial state of the qubits has to be defined. So far any qubits used in examples were already in a given state without further consideration of how they got there. In reality, physical qubits can often be initialized only in a limited number of states. Generally the reference frame is picked such that at least initialization in $|0\rangle$ can be performed natively. Initialization in other states can then be achieved by rotating the qubits from $|0\rangle$ into the desired state.

Initialization is not a unitary quantum gate since it forces the qubit into a specific state regardless of the previous state, making the operation irreversible.

Measurement

Unfortunately, the complete state of a qubit can not be measured directly. Instead it is only possible to measure a qubit in a chosen basis, and the measurement itself causes the qubit state to *collapse* into one of the basis vectors. The only way to get more information about a qubit state is to repeat the steps that got it there and measuring again, repeating this process until the desired accuracy is achieved.

Platform-specific Operations

While initialization and measurement are universally present in practical quantum computers, certain qubit technologies or even specific implementations may require their own platform-specific operations.

An example of such an operation is entanglement generation when using NV centers. Instead of performing 2-qubit gates to entangle the qubits in 2 different NV centers, a special procedure can generate a Bell-pair directly called an *ebit*.

Definition: Ebit

A pair of maximally $\underline{\text{e}}$ntangled qu$\underline{\text{bit}}$s.

Similar to single-qubit initialization, the previous qubit state has no effect on the outcome, making this a non-unitary operation. The ebit generation procedure for NV centers is explained in more detail in Section 2.2.3.

2.1.4. Quantum Circuits

As with classical computing, quantum operations can be performed in sequence to implement a quantum algorithm. This sequence of operations is referred to as a *quantum circuit*.

Definition: Quantum circuit

A specific type of quantum algorithm that consists of a sequence of quantum operations.

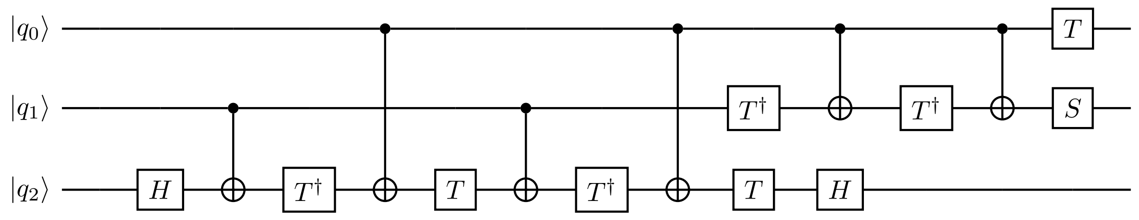


Figure 2.3: Decomposition of the 3-qubit Toffoli gate into rotations and CNOTs.

Since individual gates are reversible, a quantum circuit or portion thereof consisting purely of quantum gates is also reversible as a whole. Theoretically, these gate sequences are just a large and complex gate, with a single matrix representing its operation. This concept is the basis of *gate synthesis*, where a complex gate is broken down into (usually) 1- and 2-qubit gates that can be performed in practice.

More generally the processes of breaking down complex operations into simpler ones is called *decomposition*. An example of this is shown in Figure 2.3 for the 3-qubit Toffoli gate.

2.1.5. Quantum Error Correction

Real physical qubits are inherently analog. As a result, they suffer from errors caused by noise. This originates from multiple sources such as the control electronics or other qubits.

To counter these errors, quantum error correction (QEC) can be used. Much like classical error correction, QEC allows for detection and correction of errors. By making use of some peculiarities of quantum mechanics, this can even go beyond what is possible with classical analog signals.

Classical Error Correction

Consider the most basic classical digital error correcting code: the repetition code. A single bit can be encoded by repeating its value over for example 3 bits. This way, if one bit experiences an error, the other 2 can be used to detect and correct this error through a majority vote. Essentially, a single *logical* bit is represented by multiple *physical* bits.

With analog electronics a similar approach can be used, for example by applying the same input to multiple identical circuits and averaging the outputs. However, any common-mode noise will not be filtered out this way. The total error is reduced, but not completely eliminated.

Improved Quantum Error Correction

At first glance qubits might seem to face the same fate as classical analog electronics, but more pressing is the inherent property of qubits that they can not be copied or cloned [3]. Instead, a close alternative is to perform a controlled NOT gate from the source qubit onto the target qubits that have been initialized in the $|0\rangle$ state. While this does result in the target qubits being identical for purposes of measurement or as source qubits, it also means the 3 qubits may now be entangled depending on their initial state. As a result, measuring one qubit will not only collapse its own superposition but also (partially) collapse the superposition of the others, making a direct majority vote impossible. However, quantum error correction is still possible by making use of *indirect parity* measurements.

'Indirect' means that the qubits are not measured directly but rather through additional qubits. A distinction can be made between *data* qubits that store a quantum state for longer periods of time, and *auxiliary* qubits that are used for indirect measurements or other short-term procedures. Multiple data and auxiliary qubits then make up a single logical qubit.

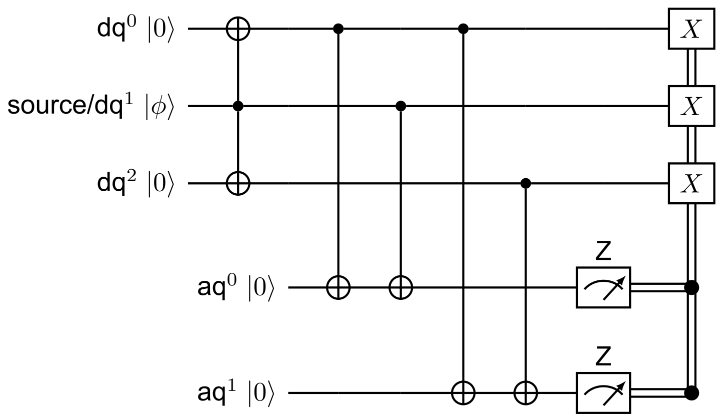


Figure 2.4: Quantum repetition code with 3 data qubits.

Definition: Auxiliary qubit

A qubit used to perform indirect operations such as measurements, as opposed to a data qubit. Also known as an *ancilla* qubit.

'Parity' means that instead of trying to measure a single qubit, a parity check of two qubits is performed. This process is illustrated in Figure 2.4. After initializing the data qubits, two CNOTs are performed on each of the auxiliary qubits after which they are measured in the Z-basis. Because these measurements only give information on the combined state of two or more qubits rather than of an individual one, the superposition of the data qubits is preserved.

Even without knowledge of the exact states of the data qubits, the parity measurements still provide enough information to detect and correct errors. For classical bits this is illustrated in the following table. Note that only single-bit errors can be correctly detected, since a 2- or 3-qubit error is indistinguishable from a 1-qubit error or no error at all.

aq^0	aq^1	Single-bit error
0	0	None
0	1	dq^2
1	0	dq^0
1	1	dq^1

This shows the process with digital values, however superposition of qubits is an analog phenomenon. This is where the peculiarities of quantum mechanics come to help: the auxiliary qubits are still entangled with the data qubits, and measuring them will discretize the error on the data qubits. After applying the conditional gates, this results in the error being fully corrected. It should be noted however that this simple repetition code can only correct unwanted X-rotations.

Better Quantum Error Correction

To correct *all* possible errors in a logical qubit, more advanced error correction is necessary. An example of such a QEC are the *surface codes* [4] , which perform both X and Z-parity measurements on the data qubits as opposed to only X-parity with the repetition code. Additionally, as the name suggest the data and auxiliary qubits are laid out in a 2D plane and interaction edges between qubits do not cross each other. This simplifies the requirements for the physical qubit system.

Different kinds of surface codes exist with different advantages and disadvantages, implementation requirements, and other properties that might make one more suitable for a specific use case. Some examples are color codes [5], holographic codes [6], and lattice surgery [7]–[9]. QEC is an active field of research with novel ideas, combinations of existing work, and specific implementations being introduced regularly.

2.2. NV Center Physics

While many different qubit technologies exist, the current focus of the Fujitsu project and therefore also of this thesis are nitrogen-vacancy center (NV center)s.

A nitrogen-vacancy center is a point defect in diamond, formed by a nitrogen substitution and carbon vacancy next to each other. The negatively charged NV$^-$ state, which occurs when an additional electron from the environment is captured, is of primary interest for this thesis. The neutrally charged NV^0 state also has its uses [10], but these are limited. A short description of the NV center qubits from a control perspective is provided next. For more details on the underlying physics, the reader is referred to [11]–[15].

2.2.1. Qubits

Central to the operation of an NV center-based quantum system is the qubit associated with the electron (e$^-$) spin in the NV center. This qubit is highly controllable through radio-frequency electromagnetic fields and photons. However, this comes at the cost of being susceptible to noise.

Additionally, nuclear spins from naturally occurring Carbon-13 (^{13}C) isotopes within the diamond can be used as qubits. These spins cannot be controlled directly, instead they interact with the electron spin and can be controlled indirectly. The carbon spin qubits have the advantage of being less susceptible to noise and decoherence. This motivates their use as data qubits, storing quantum states for longer periods of time while the e$^-$ qubit performs most operations [15]. Control of a single NV center with up to 8 of these qubits has been demonstrated [16], however an increasing number of carbon spin qubits is only useful up to a point due to increased runtime and decreased control. What amount of qubits is useful in practice remains to be seen.

Finally, the nuclear spin of the nitrogen atom itself can be used as qubit, in a way similar to the ^{13}C qubits [16], [17]. However, other types of color centers do not necessarily have a usable nuclear spin like nitrogen does, so for the purpose of generality this qubit is not treated as a special case.

A schematic overview of an NV center with 4 ^{13}C qubits is shown in Figure 2.5.

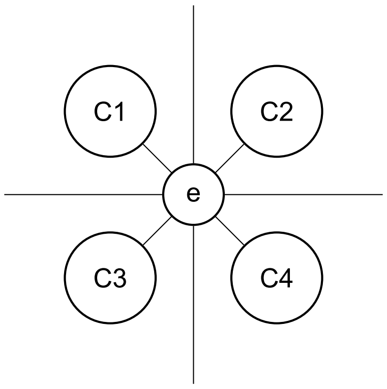


Figure 2.5: A nitrogen-vacancy center with 4 carbon spin qubits. The lines extending outwards from the electron spin qubit represent connections with other NV centers.

2.2.2. Control

As mentioned, only the electron spin qubit in an NV center can be controlled directly. This can be done using either an AC magnetic field or photons.

The magnetic field can be used to perform gates on a single NV center. Starting from single-qubit rotations on the e^- qubit, controlled and uncontrolled rotations of the ^{13}C qubit can also be achieved [13]–[15].

Different photon wavelengths can be used for different operations as described below. The NV center can also (re-)emit photons which have to be detected. This requires single-photon detectors, which can be designed using for example superconducting nanowires [18].

Initialization is done with a procedure known as charge pumping, where photons are continuously fired upon the NV center until they are not absorbed anymore, indicating initialization has been achieved. Measurement is done in a similar fashion, where the number of photons absorbed is indicative of the qubit state [11].

Photons can also be used to entangle the electron spin qubits in two different NV centers. This involves exciting the NV center with a photon, resulting in another photon being emitted. Interfering the photons emitted by two NV centers and measuring the outcome results in the e^- qubits to be entangled. Since this procedure is not 100% guaranteed to be successful, it can be repeated and combined with single-qubit rotations to improve the entanglement fidelity [13], [19], [20].

2.2.3. Connectivity

A major advantage of the photon-based entanglement described in the previous section is that photons can be easily transported. On a small scale, on-chip wave guides could be used to allow interactions not limited to neighboring NV centers. Ignoring photon losses any arbitrary network would be possible only under the constraints of the waveguides. On a larger scale, fiber optic cable could be used to connect NV centers across longer distances of several meters [21] or even hundreds of meters [22]. This would allow multiple quantum computers to work together using the same or similar entanglement procedures as used internally.

A peculiarity of this photon-based entanglement is that it does not allow one to perform unitary gates between NV centers. Instead, the only operation possible is to initialize the two e^- qubits in a maximally entangled Bell pair state. This means that if the desired action is to perform a unitary gate, some extra steps are required. However, if maximum entanglement itself is the desired result this can thus be performed as a single operation.

Figure 2.6 shows an example of 6 NV centers connected in a 2D nearest-neighbor grid, illustrating both connectivity within a single NV center and between multiple. A 2D grid is used here just for simplicity, but as mentioned the NV center connections can be arbitrary.

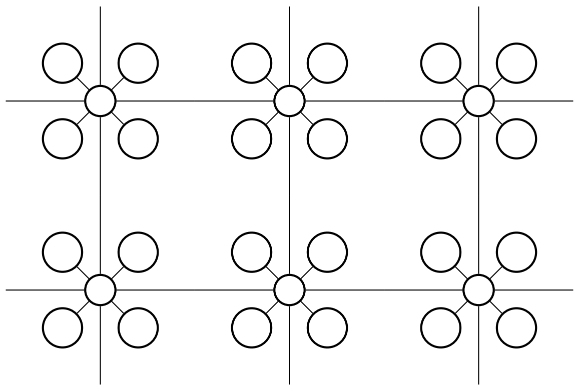


Figure 2.6: Connectivity of multiple NV centers, here shown with 4 ^{13}C qubits each.

2.2.4. Comparison with Other Technologies

Quantum hardware based on NV centers or other color centers has a number of features that distinguish it from other qubit technologies. This section discusses these differences and their implications on higher-level systems such as the digital control system and compiler.

Stability

The stability of NV centers make them useful at temperatures higher than other commonly used qubit technologies. Even at room temperature some features of NV centers are usable. This mainly applies to quantum sensing [23], [24], but also unitary gates and entanglement have been demonstrated at ambient conditions [25], [26].

Hybrid Qubits

Within a single NV center, the two qubit types have distinctly different properties resulting in a hybrid qubit system. The e^- qubits are highly controllable but vulnerable to decoherence, while the ^{13}C qubits are stable for relatively long periods of time. This motivates their use as auxiliary and data qubits respectively.

Additionally, the ^{13}C qubits cannot be controlled or interact with each other directly, only through the e^- qubit. Figure 2.6 shows an example for six NV centers. This contrasts with for example superconducting and trapped ion technologies where each qubit functions the same, and connectivity is more uniform [27], [28].

Connectivity

The possibility of arbitrary connections between NV centers as described in Section 2.2.3 is in stark contrast with for example superconducting qubits, which even in state-of-the-art implementations are limited to nearest-neighbor connectivity [29]. The prevalence of superconducting qubit systems is reflected in the large amount of work being done towards surface codes which are inherently planar. There is an advantage to be gained with higher-connectivity systems however [27], [30], and some possibilities for superconducting systems have already been explored [31], [32].

The native connectivity possibility of NV centers may give them an edge over superconducting qubits. While trapped-ion qubits provide high connectivity as well, the shared control electronics limits the number of simultaneous operations, while NV centerscan also operate individually when required.

2.3. Digital control

To execute a quantum program, the analog control hardware described in Section 2.2.2 needs to perform the required operations in the correct order. This can be handled by a digital control system which executes a quantum program as a sequence of instructions, much like a classical computer.

In the context of the Fujitsu project, a layered control architecture is envisioned. The lower layer consists of local controllers responsible for one NV center each, and the upper layer consists of a single global controller which orchestrates the operation of the lower layer. By separating the control architecture in this way, the local controllers can directly manage the analog control electronics of individual NV centers in a highly parallel manner, while these details are kept away from the global controller which can operate on much higher-level instructions.

This contrasts with existing work where a single controller is directly responsible for many physical qubits, as for example shown in [33]. An example of a more separated control architecture is presented in [34], but also here there is a certain degree of global control at the lower layers. A control architecture with 2 fully separated layers is presented in [35], which implements quantum error correction on the lower layer.

Based on the layered concept, in [36] a version of such a QISA has already been designed. However, this QISA does not clearly distinguish between the two levels. Additionally, further developments in other areas of the Fujitsu project have brought new possibilities to light.

2.4. Compiler

While the basic instructions described in the previous section provide all necessary control over the quantum hardware, they are very fine-grained. This makes writing a full algorithm in assembly a tedious and error-prone task. For this reason software is typically written in a high-level programming language.

Since the control electronics only understands the lower-level instructions, it is necessary to convert these high-level languages down to assembly language. This is done by a compiler. The process is typically divided into several *passes*, each responsible for a certain kind of transformation. These passes are performed on an intermediate representation (IR) that represents the code in an abstract way suitable for the compiler.

Passes can be categorized into being fully internal or being part of the *frontend* or *backend*. The internal passes are more generic, performing actions on the IR with little to no assumptions about how and where the program was written or what will happen to the output. Passes belonging to the frontend are specific to a certain type of input such as an input programming language. Similarly the backend passes perform actions specific to the target platform.

Definition: Execution platform

Something which can execute a quantum algorithm, be it a simulator or real quantum computer. In this thesis a platform is assumed to be compatible with quantum circuits.

A classical compiler can include passes such as optimization, analysis, and code generation. A quantum compiler can include passes with similar functionality, but will have some passes specific to quantum computing as well. The following sections describe several passes in detail, how they apply to NV centers, and the resulting compiler requirements.

2.4.1. Frontend

The frontend is how the main user input enters the compiler. This can be in the form of files written in a dedicated programming language such as cQASM [37], but also as a generic programming language extended with a library such as Qiskit[38]. In both cases the frontend converts the user input into an intermediate representation used by the internal compiler passes.

2.4.2. Decomposition

Decomposition is the process of ‘taking apart’ complex operations. This is needed if not all quantum operations can be applied directly to the hardware. For example, a certain quantum system might have CNOTs as the only 2-qubit gate, requiring other operations involving 2 or more qubits to be decomposed into a sequence of CNOTs and single qubit gates. Since many existing qubit technologies allow for 2-qubit gates at most, decompositions such as these can be generally applicable.

Decompositions can also target a certain technology or even specific implementation. For example, rotations could be limited to some specific axes, requiring rotations in other axes to be decomposed into multiple rotations over valid axes.

2.4.3. Optimization

Optimization is the process of transforming a quantum circuit into a 'better' equivalent, usually optimizing for execution time. This can be done by for example reorganizing and combining gates or removing redundant operations [39]. Similar to decomposition it can be generic or tuned to a specific technology or implementation.

A basic optimization for a quantum compiler is to combine Clifford gates which act on just a single qubit. Sequences of Clifford gates can sometimes be merged into fewer gates, although this does depend on the gates supported by the target platform.

2.4.4. Scheduling

If the output of one operation is used as an input for the next they must be executed sequentially. If there is no such dependency they can be executed in parallel, decreasing execution time. A scheduler pass performs this dependency analysis and schedules instructions accordingly.

Information about how long each instruction takes can help increase the effectiveness of this process. Additionally, there may be other constraints to consider, for example when some control electronics are shared between multiple qubits.

2.4.5. Mapping

A generic quantum algorithm with logical qubits has no knowledge of the platform it will be executed on. Especially hardware platforms however can have constraints that limit the possible operations. For example, connectivity can be limited and allow gates only on neighboring physical qubits. To be able to execute the algorithm these constraints have to be satisfied by mapping qubits and operations to their equivalents available on the platform.

Definition: Mapping

Transforming qubits and quantum operations into closer-to-hardware equivalents. In this thesis mapping specifically refers to transforming logical qubits and operations into their physical counterparts. Outside of this thesis 'mapping' often refers to just place & route.

The input of the mapping process is a sequence of instructions on logical qubits with no knowledge or assumptions about the hardware. The output is a sequence of instructions on physical qubits that conform to the available connectivity. Instructions that are not directly executable are still allowed as long as they can be decomposed into executable instructions without requiring any other qubits.

The mapping pass includes subpasses for initial placement and routing, and can encapsulate more subpasses such as logical to physical decomposition.

Place & Route

While theoretical quantum circuits can perform arbitrary gates on any selection of qubits, in practice connectivity between physical qubits is limited. This poses restrictions on which qubits can interact with each other [40]. Satisfying these constraints is handled by a place and route (P&R) pass.

Definition: Place & route

Initial placement and routing of qubits based on the required interaction and available topology.

The goal of a P&R pass is to first assign each qubit a location so that it is connected to the qubits it needs to interact with. It is however not always possible to find an exact placement that satisfies all interactions, in which case routing is also necessary. Routing can for example use swap to move qubits around during execution of the circuit. Moving qubits around to perform one operation takes time and can affect subsequent operations, so careful planning is necessary to ensure the resulting circuit is fast and correct.

Besides physical P&R it can also be useful to perform this procedure on logical qubits. The connectivity available here can be arbitrary, but can also originate from some underlying physical connectivity.

Physical Synthesis

As previously stated in Section 2.1.5, a single logical qubit can consist of multiple physical qubits. Even when no QEC is used there may not be a one-to-one correspondence between logical and physical qubits, for example due to physical qubits reserved for another purpose such as routing. Operations performed on the logical qubit thus need to be transformed into the corresponding operations on physical qubits.

This process can be seen as a form of decomposition, where in addition to new instructions also new qubit references are inserted. In this thesis, physical synthesis is considered part of mapping as mentioned in Section 2.4.5.

2.4.6. Code Generation

Once all internal passes are completed, the internal representation used by a compiler needs to be converted to a format suitable for the target platform. While instructions that directly act on data such as qubits or classical registers are usually present directly, features such as control flow, parallelism, and timing may all be represented in an abstract manner suitable for the compiler but not understood by the platform.

The code generation pass is thus responsible for extracting all information required by the target platform and writing it to a format such as assembly or binary machine code. This output can then be fed into another tool for further processing or directly into the platform.

2.4.7. NV Center-specific Details for Compiler Passes

While the previous sections introduced quantum compiler passes for generic platforms, the following sections discuss if and what NV center-specific adaptions need to be made for several passes.

Decomposition

With NV centers, 2-qubit gates have some limitations since they can only be performed with an e$^-$ qubit as control and ^{13}C qubit in the same NV center as target. Performing for example a CNOT from ^{13}C qubit to e$^-$ qubit requires a sequence of gates, in this case the direction of the CNOT can be reversed by surrounding it with Hadamard gates as shown in Figure 2.7

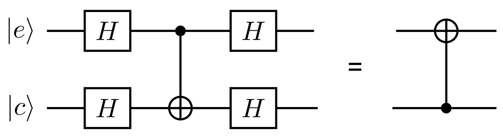


Figure 2.7: Reversing the direction of a CNOT by surrounding it with Hadamard gates.

Scheduling

The e$^-$ qubit and ^{13}C qubits belonging to the same NV center share control electronics, thus only one instruction can be executed on each of these qubits at any point in time.

Additionally, the execution time of some operations is not fixed. Initialization of an e^- qubit for example is probabilistic as mentioned in Section 2.2.2, although an expected upper bound could be determined. Furthermore, rotations are performed by applying a magnetic field, where the duration determines the angle of rotation.

Mapping

Connectivity between qubits in NV centers is not uniform. Within an NV center a ^{13}C qubit can only interact through the e^- qubit, while between NV centers it is possible to have flexible or even arbitrary connections as mentioned in Section 2.2.3. This contrasts with more common qubit technologies which are restricted to nearest-neighbor interactions across the whole system. On the other hand, the interactions within an NV center allow for 2-qubit gates to be performed, while between NV centers it is only possibly to generate an ebit.

This implies that 2-qubit gates between NV centers will always require a sequence of operations involving at least 2 each of e^- qubits and ^{13}C qubits, for example as shown in [41], [42]. The generation of an ebit also destroys the current state of the e^- qubits. If these are to be preserved, they have to be saved by swapping them to spare ^{13}C qubits. Possible implementations of NV center-specific mappers that take above properties into account are discussed in Section 3.2.

Additionally, single-qubit operations performed on a ^{13}C qubit take more instructions and time than the same gates on an e^- qubit. This might mean longer sequences of gates should preferably be executed on an e^- qubit.

2.4.8. Complete Compiler Toolchains

This thesis is centered around the instruction set level which ties in with the compiler backend. Thus, it is preferable to use an existing compiler for the frontend and intermediate steps to save on development costs, effort, and time. Table 2.1 lists a number of available quantum compilers and their features. Special attention is given to the creation of backends. Additionally, as Section 2.2.4 has shown, NV centers have a number of features that distinguish them from other qubit technologies. The compiler for NV centers should at least be able to work with these features, and preferably be able to make good use of them with for example specific mapping or optimization steps.

From these compilers, Qiskit and OpenQL were noticed to both promote modifications and be used frequently in existing research environments. For these reasons only the potential use of these two have been explored.

Qiskit

Qiskit is a compiler developed by IBM. Its main frontend is a Python SDK and it has backends for simulators and IBM's own hardware. This hardware uses superconducting qubits with grid-like connectivity. Only routing protocols based on swaps are available [39].

OpenQL (QuTech/TU Delft)

OpenQL is a modular quantum compiler framework developed by TU Delft. It provides frontends for C++ and Python, and allows compiling to any target using custom backends. Its modular design and configuration options allow for easy adaptation to different technologies and experimentation with new compiler passes. A wide variety of passes are readily available such as place and route, decomposition, optimization, and scheduling [59].

2.4.9. Comparison

From the current state of available compilers, none of them stand out as being particularly well-suited for the envisioned NV center architecture. Therefore, since multiple passes and a new backend has to

Name	Developer(s)	Frontends	Backends	References
Qiskit	IBM	Python	Simulator Superconducting	Source: [43] [38]
QuilC	Rigetti	Python Java JavaScript Haskell	Simulator Superconducting	Source: [44] [45], [46]
QDK	Microsoft	Q# Python	Simulator Superconducting Trapped-ion	Source: [47] [48]
ProjectQ	ETH Zurich	Python	Simulator Superconducting	Source: [49] [50], [51]
ScaffCC	University of Chicago	Scaffold	Simulator	Source: [52] [53]
Cirq	Google	Python	Simulator Superconducting Trapped-ion Generic	Source: [54] [55]
PennyLane	Xanadu	Python	Simulator Photonics	Source: [56] [57]
OpenQL	TU Delft	C++ Python	Simulator Superconducting Generic	Source: [58] [59]

Table 2.1: Overview of several open-source quantum compiler toolchains.

be created, the technical flexibility and ease of implementation is considered, including the availability of documentation. The **OpenQL compiler** stands out in providing a clearly defined modular framework with many highly configurable passes. Additionally, since OpenQL is being developed here at TU Delft, it is considered easier to get support and feedback on modifying and working with OpenQL than it would be for other compilers. This motivates the decision to use OpenQL as the compiler to work with for this thesis. OpenQL has also been used by the prior work in [36] on a compiler for NV centers.

2.5. Conclusion

This chapter introduced the reader to quantum information theory, NV center physics, existing control systems, and quantum compilers relevant to the scope of this thesis.

A number of existing quantum compilers have been compared, after which the OpenQL compiler framework has been selected as base for the compiler for NV centers for its generality and ease of extensibility.

3 Design

To execute a quantum program, the analog control hardware described in Section 2.2.2 needs to perform the required operations in the correct order. This can be handled by a digital control system which executes a quantum program as a sequence of instructions, much like a classical computer. Before the digital control hardware can be implemented, a quantum instruction set architecture (QISA) needs to be defined which acts as the interface between hardware and software.

The envisioned digital control hardware operates only on physical qubits. However, an algorithm to be executed on a quantum computer is often written for logical qubits. This means some way of mapping logical qubits to physical qubits is necessary. Given an algorithm that operates on logical qubits, the mapping pass needs to be able to generate a program that implements this algorithm with valid operations on physical qubits. *Valid* operations refers to the operations being actually possible in the target system, satisfying for example connectivity and control system constraints.

This chapter starts with describing the design of the QISA of the digital control system in Section 3.1. Subsequently, an exploration of compiler design choices is described in Section 3.2. Details of the chosen method are then worked out in Section 3.3.

3.1. Hardware Architecture

This section describes the design of the quantum instruction set architecture for a quantum computer based on NV centers. In the scope of this thesis, the primary purpose of designing the QISA is to provide a compilation target for the compiler. The underlying hardware has been explored and partially designed to aid in designing the QISA but no hard decisions have been made in this area. The work presented in this thesis can be used as a starting point for future work however.

It should be noted that the layered architecture concept has already been conceived earlier in the Fujitsu project, and the design process in this thesis builds on top of this concept. For this reason, no extensive exploration of alternatives has been performed.

3.1.1. Overview

The envisioned control hardware architecture is divided into two layers as depicted in Figure 3.1. Closest to the quantum system, each NV center is controlled by a dedicated local controller (LC). The LCs in turn receive their instructions from the global controller (GC) which executes the quantum algorithm.

The following sections go through these layers from the quantum instruction set architecture (QISA) in the GC at the top, through the micro-ISA (μISA) in the LC, down to the physical quantum system at the bottom.

3.1.2. Global Controller QISA

The global controller is the central component from which the rest of the system is controlled. As input it takes a program written in the QISA, and based on these instruction it performs computations or sends commands to the local controllers.

A first version of a QISA designed for NV centers is presented in [36]. This work improves on this in

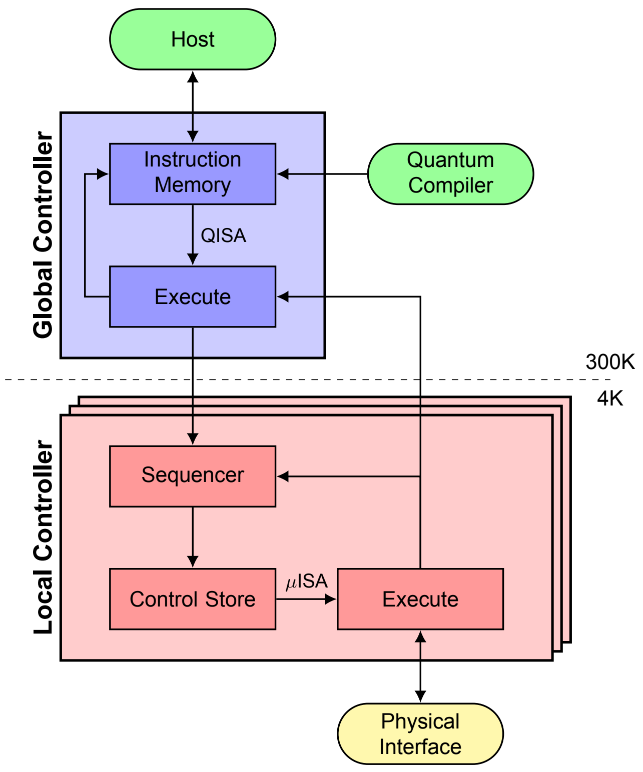


Figure 3.1: Overview of the envisioned hardware architecture.

several ways, most importantly by establishing a clear distinction between the QISA and μISA. How this has been achieved is illustrated in the following sections, each focussing on a specific group of instructions. A full listing of the QISA instructions can be found in Appendix B.1.

Single-qubit Gates

In the new QISA, just 4 instructions can represent all single-qubit gates. The `qgateE` instruction performs a rotation on an e^- qubit with an arbitrary angle around a rotation vector anywhere on the XY plane. The instruction specifies both the vector and angle as illustrated in Figure 3.2. The `qgateUC` instruction (from *Uncontrolled Carbon*) provides the same functionality for the ^{13}C qubits. Both are implemented in the LC with a sequence of micro-instructions that control the physical system.

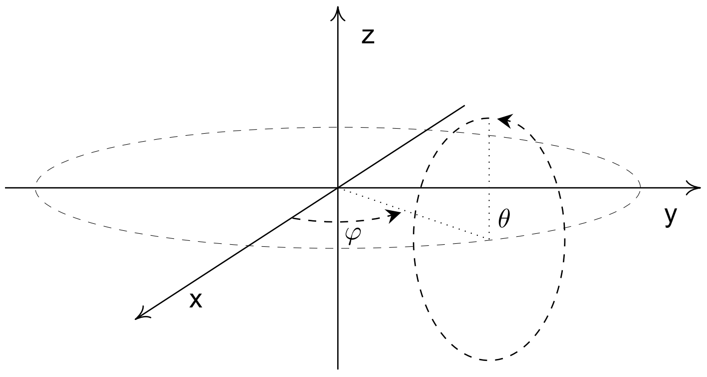


Figure 3.2: Rotation around a vector on the XY plane.

This contrasts with old QISA which specifies a large number of specific gates in both the QISA and μISA, as well as exposing the low-level `excite_mw` instruction to the QISA. The new QISA abstracts these implementation details away.

Z-rotations are implemented in the new QISA using the `qgateZE` and `qgateZC` instructions for the e^- qubits and ^{13}C qubits, respectively. These are dedicated instructions because in the envisioned LC hardware Z-rotations can be performed by incrementing a classical register rather than physically affecting the qubit. In the time the old QISA was designed this possibility had not been considered yet.

Old QISA

```
x
y
z
```

New QISA

```
qgateE 0, pi
qgateE pi/2, pi
qgateZE pi
```

Figure 3.3: Comparison of the old and new QISA single-qubit instructions on an e^- qubit.

Two-qubit Gates

A pattern similar to the single-qubit gates can be observed here. The primary 2-qubit gate in new QISA is the `qgateCC` instruction, from *Controlled Carbon*. This instruction performs a controlled rotation on the specified ^{13}C qubit with the corresponding e^- qubit as control qubit.

On a lower level, the `qgateCC` instruction can actually be constructed from an uncontrolled `qgateUC` instruction followed by a controlled gate. This more primitive gate is also useful on its own and is thus made available in the QISA as the `qgateDIR` instruction.

Additionally, to facilitate the common operation of swapping the state of two qubits, the `swapEC` and `swapCE` perform a 'half-swap' from the e^- qubit to the ^{13}C qubit and vice-versa. Contrasting with a full swap, the half-swap moves the state from the source qubit to the target qubit but leaves the source qubit in a maximally mixed state. A full swap can be performed using additional instructions. While the two swap instructions can be implemented using the `qgate` instructions as well, using a dedicated instruction can reduce the total number of instructions in a program, reducing the load on the global controller.

Other Quantum Operations

Besides quantum gates, other operations need to be performed as well. Initialization of an e^- qubit is achieved with the `initialize` instruction. Initialization a ^{13}C qubit can be done by first initializing its corresponding e^- qubit and then performing a swap using the `swapEC` instruction.

Measurement follows a similar pattern; the state of an e^- qubit can be measured using the `measureE` instruction, and measuring a ^{13}C qubit requires preceding this with a `swapCE` instruction.

Calibration

To accommodate for the variation in physical qubit properties between NV centers or even the same NV center at another time, the analog control electronics have to be calibrated. Collecting the data necessary for these calibrations is done by the instructions `detectCarbon`, `magbias`, `rabicheck`, and `crc`. Details of the operation of these instructions can be found in [60].

From the data collected through these calibration instructions, either the global controller or host system can calculate the required calibration parameters. These can then be sent to the local controllers using the `set` instruction.

Classical Instructions

In addition to controlling the quantum system, the global controller also needs to be able to perform classical calculations, for example for error-correcting codes. No specific operations have been identified here yet, thus a RISC-like ensemble of classical instructions is assumed to be available. This includes functionality to load from and store to memory, perform basic arithmetic, and branch and jump to different sections of the program.

Finally, the global controller needs to be able to communicate with the local controllers. A generic `send` and `receive` instruction provide this functionality.

3.1.3. Local Controller Micro-ISA

The local controller is responsible for controlling a single NV center. It receives commands from the global controller and executes the corresponding microcode, i.e., a sequence of microinstructions.

Controlling the analog control electronics is done in the local controller by making use of the micro-ISA. As with the QISA the μISA consists of a few groups of instructions.

- The **quantum** micro-instructions directly correspond to an action performed by the analog control electronics. This consists of the `mw` and `rf` instructions for controlling the variable electromagnetic fields, and the `switchOn` and `switchOff` instructions for routing photons both from lasers to the NV centers as well as between NV centers for entanglement.
- The **communication** micro-instructions allow each LC to communicate with the GC using the `send` and `receive` instructions. Additionally, the `sync` instruction has been reserved for potential use in the entangling scheme. Its function is to synchronize two LCs to allow for interference between photons emitted by the corresponding NV centers.
- Similar to the QISA, a number of **classical** micro-instructions are required to handle for example arithmetic and control flow. As with the QISA no definitive decisions have been made in this regard.

A full listing of the micro-instructions can be found in Appendix B.3.

Microcode

A sequence of micro-instructions can be stored as microcode to implement more complex functionality. Microcode implementing all the above QISA instructions has been designed in [60]. A full listing of these sequences can be found in Appendix B.2.

In the hardware design as currently envisioned, each local controller has its own control store memory where the microcode is stored. This does mean the microcode is duplicated for every LC and will always be stored even if some sequences are not used for a certain algorithm. Other options to be explored in future work are using a smaller control store which only contains microcode that will be used, or sharing memory blocks between multiple LCs.

Ports and Registers

To interface with the analog control electronics connected to the LC, a number of input and output ports have been defined. Some ports are register-mapped and can be written to and read from as such, while other ports are accessible only indirectly through a microinstruction.

A full listing of the ports and registers currently envisioned can be found in Appendix B.4.

3.2. NV Mapping Exploration

In the process of mapping logical qubits to physical qubits in NV centers, two actions must be performed: place and route (P&R) and logical to physical transformation.

P&R is the process of assigning qubits a location, as previously described in Section 2.4.5. It has also been mentioned that NV centers may require a different kind of P&R method than other types of qubits due to their unique properties.

Logical to physical transformation has two sides. First, the *topology* of logical and physical qubits is not the same, and secondly a single operation on a logical qubit may consist of many operations on the underlying physical qubits.

What is not established is in what order to perform these steps. This section explores the following mapping methods:

Section 3.2.1 - NVP: first performs P&R on logical qubit, then logical to physical transformation.
Section 3.2.2 - NVD: first performs logical to physical transformation, then P&R on physical qubits.
Section 3.2.3 - NVC: is a hybrid approach that performs a partial transformation, then P&R, and then a final transformation.

As the names suggest the methods originate from NV centers which are the main point of interest. Some of the ideas presented can be useful to other qubit technologies as well however, so more general applicability is also considered.

3.2.1. NVP

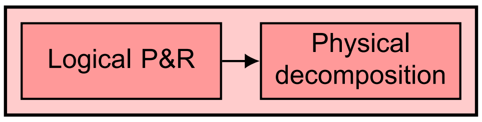


Figure 3.4: Passes in the NVP method.

The NVP method gets its name from patch-based mapping. This is inspired by existing work such as lattice surgery [7]–[9]. An overview of the compiler passes for this method is shown in Figure 3.4.

Passes

NVP starts with a logical P&R pass. This can be a generic algorithm or a more advanced method, as long as it respects the connectivity of the logical qubits. This pass may output specialized operations, for example as sometimes required by the routing for lattice-surgery gates. The workload of the P&R pass is proportional only to the number of logical qubits used, regardless of physical qubits and the QEC used.

This is followed by a decomposition pass, which translates the logical operations into physical operations. These decompositions have to specify the operations down to assignment of physical qubits. This allows for high flexibility in the physical structure of a logical qubit at the cost of having to supply highly detailed decompositions. Since the decomposition rules themselves contain all information related to NV centers, the compiler pass itself can be generic and also work with other qubit technologies.

Notation

The exact functioning of the NVP method depends on the size and internal layout of the logical qubits. To distinguish between these specializations the notation NVPS-C is proposed where C specifies the

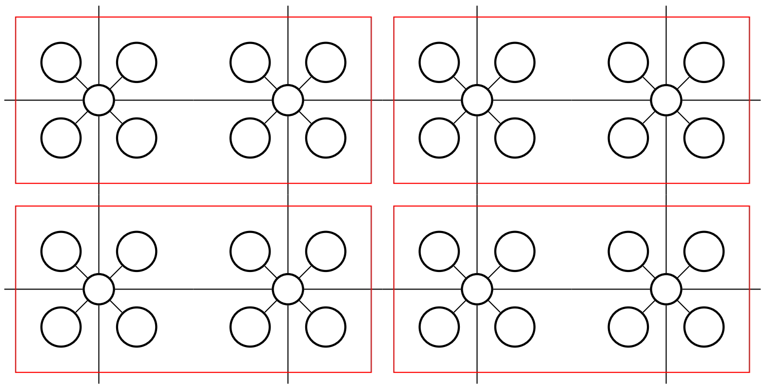


Figure 3.5: Qubit organization for an NVP2x1-4 system. Logical qubits (patches) are outlined in red.

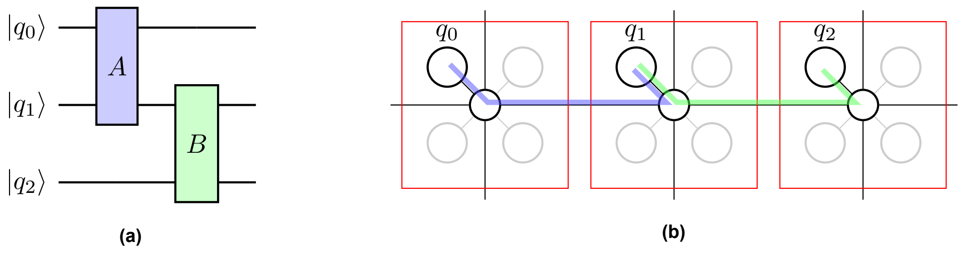


Figure 3.6: An example of NVP mapping of a circuit.

number of ^{13}C qubits per NV center and S specifies the number and configuration of NV centers per logical qubit. The simplest case would be $S = 1$ and $C = 1$, which specifies logical qubits consisting of 1 NV center with 1 ^{13}C qubit. Using this notation this will be written as NVP1-1. The value of S is not limited to a single number however. The notation S=MxN is proposed for a 2D nearest-neighbor connected patch consisting of M by N NV centers. Figure 3.5 illustrates this for NVP2x1-4, i.e., patches of 2 by 1 NV centers with 4 ^{13}C qubits per NV center.

Note that this notation does not specify the physical *operations* necessary to implement a logical operation. Which (if any) QEC method is used and if any ^{13}C qubits are reserved for other purposes such as routing is left to the decomposition rules. These rules can be specified outside of the compiler, allowing for quick changes without having to completely recompile the compiler. This notation also does not reflect the possibility of different amounts of ^{13}C qubits per NV center.

Examples

To illustrate the NVP mapping process, the circuit shown in Figure 3.6a is mapped using an NVP1-1 configuration. As the result shows in Figure 3.6b, the ^{13}C qubits store the logical qubit state while the e$^-$ qubits are used to perform the 2-qubit gates. Note that the figure shows a physical topology with 4 ^{13}C qubits per NV center, illustrating how NVP cannot make use of extra physical qubits without explicitly being configured to do so.

Conclusion

As long as the physical NV center connections always align with the logical qubits it is possible to use a single, generic NVP implementation for many different qubit technologies. This comes at the cost of having to manually specify all possible logical to physical decompositions. Irregular physical connectivity or mixed types of logical qubits may also be mapped using NVP, but require a more advanced implementation.

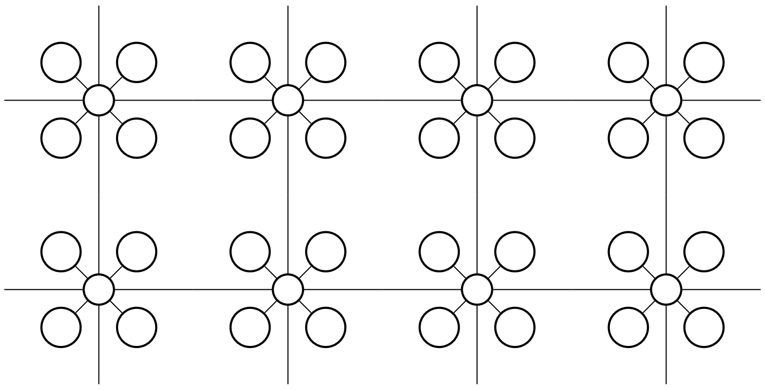


Figure 3.8: Qubit organization for an NVD-4 system. Note the lack of logical qubit boundaries.

3.2.2. NVD

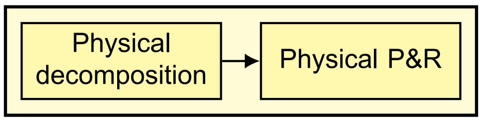


Figure 3.7: Passes in the NVD method.

With the NVD method, first the logical operations are decomposed into physical operations, and only then is P&R performed on a physical level. This gives the mapper direct control over both the e$^-$ qubits and ^{13}C qubits, lending this method its name. An overview of the compiler passes for this method is shown in Figure 3.7.

Passes

NVD starts with a decomposition pass with decomposition rules very similar to those used for NVP. The main difference is the lack of physical qubit assignment, which is to be handled by the the P&R pass instead. The decomposition rules therefore also do not need to specify routing operations, meaning this pass can be skipped if no QEC is used.

Placement of the physical qubits can be performed freely without patch boundaries to adhere to. This allows for irregular physical topology and a logical layout optimized to the quantum circuit. Which e$^-$ qubit and ^{13}C qubits should be assigned to the same NV center is strongly implied by the decomposition, but not necessarily made explicit. Some means of explicitly assigning qubits could be advantageous however.

Since the decomposition rules do not specify the locations of the physical qubits, they also can not specify full ebit generation schemes. Instead, ebits are generated by the routing pass when necessary. This poses two problems however. First, if the e$^-$ qubit state is needed again at a later point in time, it has to be stored in an available ^{13}C qubit. Second, routing *through* an NV center with long ebits also requires a ^{13}C qubitfor the intermediary state. This requires either reserving up to 2 ^{13}C qubit in each NV center for routing even if it turns out to be unnecessary, or a more advanced implementation where the need and availability of ^{13}C qubits is tracked per NV center.

Notation

Similar to the NVP method, the notation NVD-C is proposed to specify an NVD method configured for C ^{13}C qubits per NV center. Since NVD makes no assumptions about the physical layout of a logical qubit, there is no need for a specification similar to S.

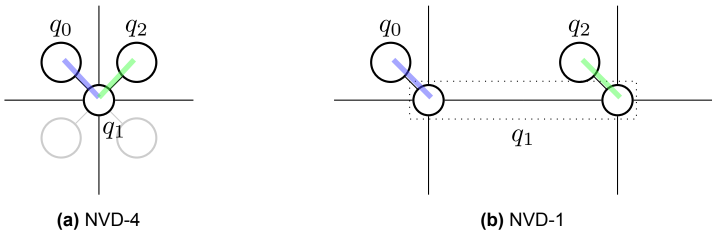


Figure 3.9: An example of NVD mapping of a circuit.

Examples

To illustrate the NVD mapping process, the circuit shown in Figure 3.6a is mapped again using NVD this time. The result shown in Figure 3.9a shows how NVD is able to fit this circuit in just a single NV center by assigning q_1 to an e$^-$ qubit. If just a single ^{13}C qubit is available, the result will be as shown in Figure 3.9b. In this case still only 2 NV centers are necessary since q_1 can be mapped to the *combined* state of the e$^-$ qubits after initializing them as ebit.

Conclusion

The NVD method is highly flexible and can optimize physical placement based on the input circuit. It does however require a specialized P&R implementation to handle the specifics of NV centers.

3.2.3. NVC

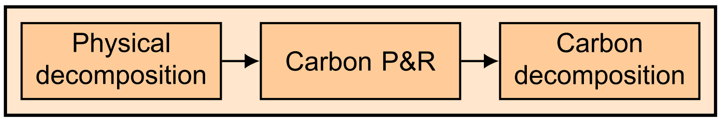


Figure 3.10: Passes in the NVC method.

The NVC method moves the focus to individual carbon qubits. A distinction is made between data qubits and auxiliary qubits as introduced in Section 2.1.5. Following the NV center properties described in Section 2.2.1, data qubits are assigned to ^{13}C qubits and auxiliary qubits to e$^-$ qubits. An overview of the compiler passes for this method is shown in Figure 3.10.

Passes

The first pass is decomposition of logical operations into operations on the *data* qubits of a QEC. This level can contain higher-level operations such as parity measurement without explicitly specifying the use of auxiliary qubits. Simply the presence of such an operation already implies the use of an auxiliary qubit and the required connectivity. As with the NVD method, the decomposition rules in this pass do not need to specify routing operations, meaning it can be skipped if no QEC is used.

Next is a P&R pass which assigns data qubits to ^{13}C qubits. Assuming the physical data qubits that make up a logical one frequently interact with each other, they are expected to cluster together forming 'patches' similar to the NVP method. However, just like the NVD method, NVC allows for an irregular logical qubit layout without being bound to the regular structure enforced by the NVP method.

Also like NVD, it is possible for a single NV center to contain data qubits from more than one logical qubit, as illustrated in Figure 3.12a.

Routing in the NVC method is again similar to what is required for NVD. While NVC lacks explicit e$^-$ qubit operations, they are still necessary for both routing and performing qubit operations. Tracking the availability of e$^-$ qubits and ^{13}C qubits is thus still necessary.

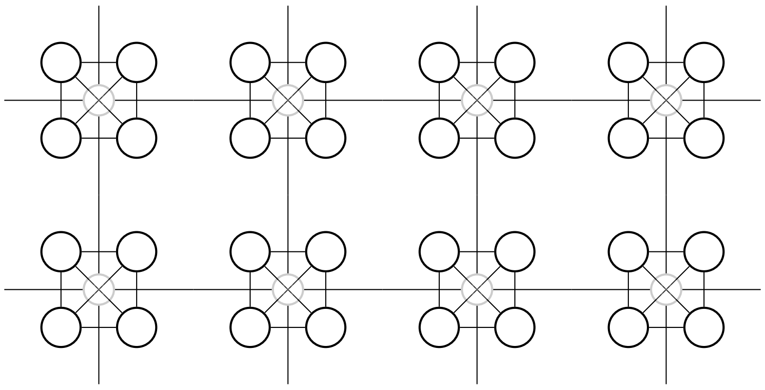


Figure 3.11: Qubit organization for an NVC-4 system. Note the direct connections between ^{13}C qubits.

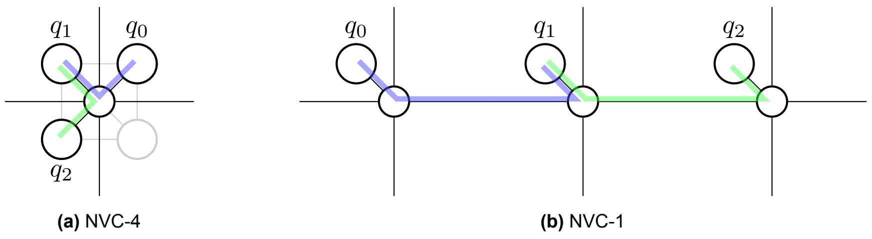


Figure 3.12: An example of NVC mapping of a circuit.

Finally, another decomposition pass is performed which expands the high-level operations into the required operations on the e$^-$ qubit and ^{13}C qubits. These decompositions can be shared among different QECs given the similar kinds of operations performed such as basic gates and parity measurement.

Notation

Similar to the NVD method, the notation NVC-C is proposed to specify an NVC method configured for with C ^{13}C qubits per NV center. Again, since there is no predefined number of NV centers per logical qubit there is no need for a specification similar to S.

Examples

The circuit from Figure 3.6a is mapped again, this time using the NVC method. In the case of NVC-4 shown in Figure 3.12a, the result is close to NVD-4, but using an additional ^{13}C qubit since input qubits must be mapped to ^{13}C qubits. For the same reason, having only a single ^{13}C qubit available per NV center implies using 3 NV centers as shown in Figure 3.12b, with the final result being the same as the NVP method.

Conclusion

The NVC method provides similar flexibility as the NVD method while allowing for higher-level decomposition rules, simplifying the required input to the compiler. This does require the decompositions to be tailored specifically to NV centers, in addition to also requiring an NV center-specific P&R pass.

3.2.4. Comparison

Having explored some of the possibilities of NV mapping, this section compares the methods discussed. The figures of the topology and mapping examples can also be found side-by-side in Appendix A.1 and Appendix A.2 to easily compare the methods visually.

Scalability

NVP performs P&R on the level of logical qubits. This makes the runtime complexity of the this pass independent of the QEC used. In contrast, both NVC and NVD perform some form of decomposition before P&R, requiring this pass to handle more instructions and more qubits. This makes NVP more suitable for future quantum computers with large QECs and thousands of logical qubits such as for example envisioned in [61]. Targeting fault-tolerant applications rather than smaller NISQ applications also aligns with the interests of the Fujitsu project which this work is a part of.

External input

All methods require the decomposition from logical to physical qubits to be specified somehow. In the case of NVP this decomposition needs to described in detail, down to which operations have to be performed between which physical qubits. The decomposition for NVD needs to be just as detailed but without assignment of physical qubits. NVC allows the use of higher-level operations in the specification while also not requiring physical qubit assignment. In general, NVC thus requires the least amount of external input.

For all methods the amount of external input depends on the QEC complexity. Simple or no QEC for example will require only little input, as less physical operations are necessary to implement a logical operation. It is also expected that the decomposition rules can be at least partially generated for highly structured or repetitive QECs. Similarly, symmetry in the qubit layout can be taken advantage of.

Implementation

The NVP method can use existing P&R algorithms, requiring only the NV center-aware decomposition to be added for a basic implementation. The OpenQL compiler used for this work already features a compatible P&R pass, as well as a decomposition pass which can be used as starting point for the logical to physical decomposition. Using an existing P&R pass also makes it possible to benefit from its future improvements.

NVD on the other hand requires both a decomposition pass as well as a specialized P&R algorithm. The NVC method lies somewhere in between, but still closer to NVD. For initial placement it might be possible to reuse an existing algorithm, but routing will again require a specialized implementation. NVC also requires a second decomposition pass.

The NVP method is thus considered the best choice for implementation.

Flexibility

The available physical topology may not always be the best case for the desired logical topology. Both NVD and NVC can adapt the shape of individual logical qubits to match the underlying physical qubits, making them more suitable for irregular topologies. NVP on the other hand assumes a fixed layout of patches to which the logical qubit can be assigned, although differently-sized patches can be used.

While NVD and NVC are more flexible than NVP in this regard, most current work on QEC assumes a regular physical topology anyway [62] . This can be seen with for example surface codes and its usage in lattice surgery on which NVP is based.

Generality

The target technology of this work is NV centers, but more generic applicability can also have its advantages. Reuse of algorithms and implementations can be beneficial both ways, also allowing advances in other areas to be useful for an NV center architecture.

The NVP method is the most general option, as it can easily work with different types of P&R algorithms and the logical to physical decomposition can be implemented in a generic way as well. NVD uses a similarly generic decomposition pass but requires specialized P&R, and for NVC all passes have some NV center-specific functionality.

3.2.5. Conclusion

Table 3.1 summarizes the differences discussed above. Since the Fujitsu project targets large-scale quantum computers and not necessarily NISQ applications, scalability is an important factor to consider making NVP a good option. Simultaneously the fine-grained flexibility offered by NVD and NVC has less utility in large systems. While NVP does require the most manual input, it is also expected to be the least complex to implement. Finally, while not of direct interest to this project, the NVP method is the most generic and can on its own benefit other quantum technologies.

Based on this comparison, NVP mapping is chosen to be implemented for this work. More specifically, the initial goal is to implement the features necessary for an NVP1-1 configuration but with future extensions in mind.

	Scalability	External input	Implementation	Flexibility	Generality
NVP Patches	+	-	+	0	+
NVD Direct	-	0	-	+	0
NVC Carbon	-	+	-	+	-

Table 3.1: Comparison of the 3 different NV mapping methods.

The NVD and NVC methods can be useful as well, especially for NISQ systems. The Fujitsu project aims for fault-tolerant quantum computing however, making the more suitable method. It might also prove useful to combine two methods; for example by specifying an unmapped QEC, using NVC to convert this into more detailed decompositions, and then making use of those in an NVP mapper.

3.3. NVP Mapping Design

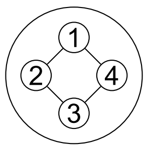
This section describes the detailed design of the NVP method outlined in Section 3.2.1. During the design process it became clear that while inspired by NV centers, the concept of NVP mapping can be applied to other qubit technologies as well. For this reason, first the generic design is described and only then are NV center-specific details introduced. The generic design starts with describing the logical to physical topology transformation, which is not an actual compiler pass but explains the NVP concept without the details of qubit operations. Next the logical P&R and decomposition pass are explained, followed by how NVP can be applied to NV centers and some practical examples.

3.3.1. Topology Transformation

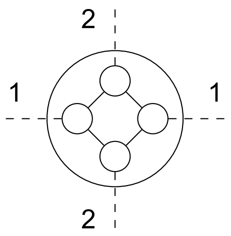
This section describes the process of transforming a logical topology into a physical topology. Parts of this topology transformation have their use in the compiler passes, either directly or by generating some external input. The primary input of this transformation is a logical topology. An example of a 2x2 grid is shown in Figure 3.14a, where each circle represents a logical qubit. The qubits are numbered for reference, and the edge labels will become useful later.

Internal Layout

The first step of the topology transformation is to specify the internal physical layout of the logical qubits. An example of this is shown in Figure 3.13a. *Note that this represents a generic logical qubit, not specifically one using NV centers.* For NVP, the only information the compiler needs from this is the number of physical qubits in a logical qubit. The decompositions must comply with the physical topology, but they are themselves an input for the compiler. It is up to the user or program that generates the decompositions outside of the NVP mapping process to guarantee they are valid. The physical topology

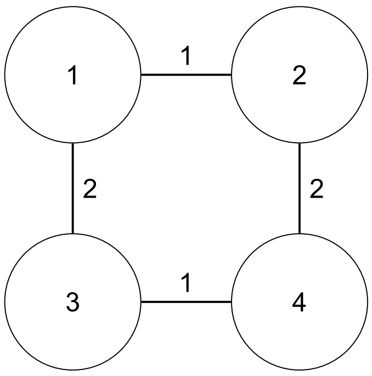


(a) Internal layout.

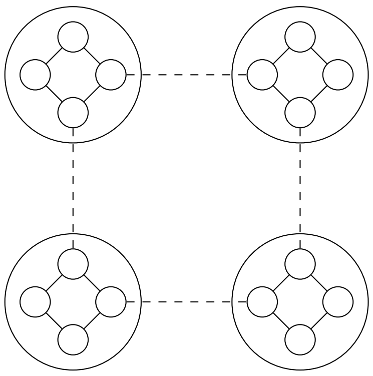


(b) External connections.

Figure 3.13: Physical qubits in a logical qubit.



(a) Logical



(b) Physical

Figure 3.14: Logical 2x2 grid and corresponding physical topology.

is still shown here for illustration purposes and is also required for for example NVD.

External Connectivity

Connections between logical qubits also need to be transformed into their physical counterparts. In doing so, a distinction needs to be made between different connection types. For example, in a 2D grid only the physical qubits on the edge of a logical qubit connect with the neighboring logical qubit. Which physical qubits these are depends on in which cardinal direction the logical connection is pointed.

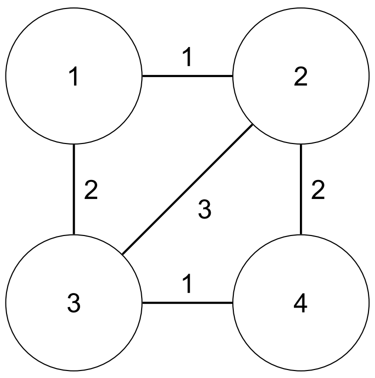
Edge types are introduced to distinguish these different connections, indicated by the numbers 1 and 2 along the edges in Figure 3.14a. In this example an edge is considered to be bidirectional, but it can be split up into unidirectional edges with different edge types if required.

Figure 3.13b shows which physical connections are made for which edge type. As with the internal layout, the compiler actually does not need any information about the exact physical connectivity of edges. This information is instead implied by the decomposition rules, and it is up to the provider of these rules to assure they conform to the physical connectivity.

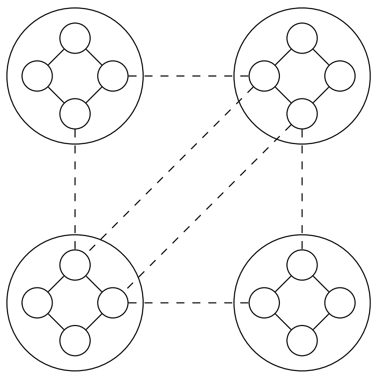
Constructing the full topology

With both the internal layout and external connectivity of logical qubits specified, the full physical topology can be constructed. Figure 3.14b shows this for the 2x2 grid mentioned previously. To illustrate more different edge types, Figure 3.15a shows a logical topology with a diagonal edge. This is then transformed into the physical topology shown in Figure 3.15b.

Using different edge types also allows for different *logical qubit types*. Figure 3.16a extends the logical topology with a larger logical qubit and additional edge types that specify a connection between a small and large qubit. A corresponding physical topology is shown in Figure 3.16b.

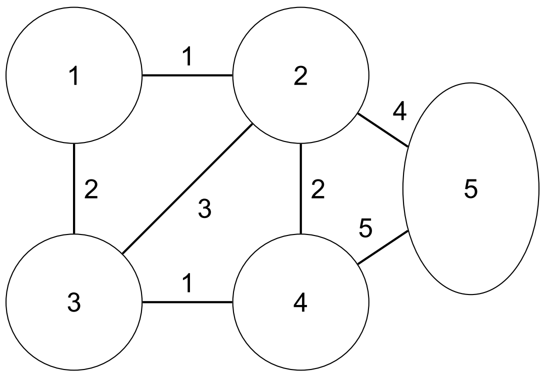


(a) Logical

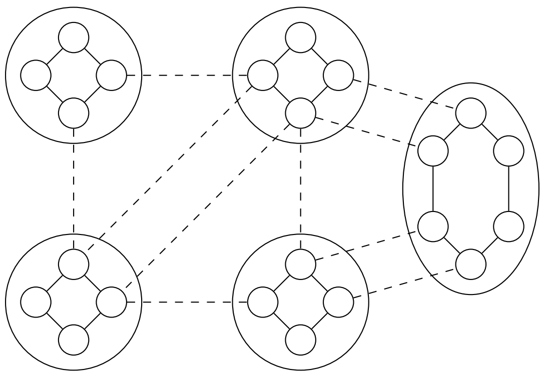


(b) Physical

Figure 3.15: Logical to physical topology with a diagonal connection.



(a) Logical



(b) Physical

Figure 3.16: Logical to physical topology with a larger qubit.

3.3.2. Logical P&R

The actual NVP mapping process starts with a logical place and route (P&R) pass. Due to its design, NVP mapping does not require a specific logical P&R method to produce a valid output. While target-specific optimizations may be possible here, they are considered to be out of the scope of this thesis.

3.3.3. Instruction Decomposition

The second pass performs decomposition from logical to physical operations. Each logical instruction *must* be decomposed into its corresponding physical instructions. For multi-qubit gates, a different decomposition can be specified for each edge type.

3.3.4. Application to NV centers

Applying the NVP mapping method to NV centers, a natural grouping is already present in the form of an individual NV center. Handling logical qubit boundaries within a single NV centerdoes not align well with the NVP method, leading to the assumption that logical qubits consist of one or more NV centers. This results in the structure introduced in Section 3.2.1.

Figure 3.17 shows an example of a full NVP mapping process from logical circuit to physical operations.

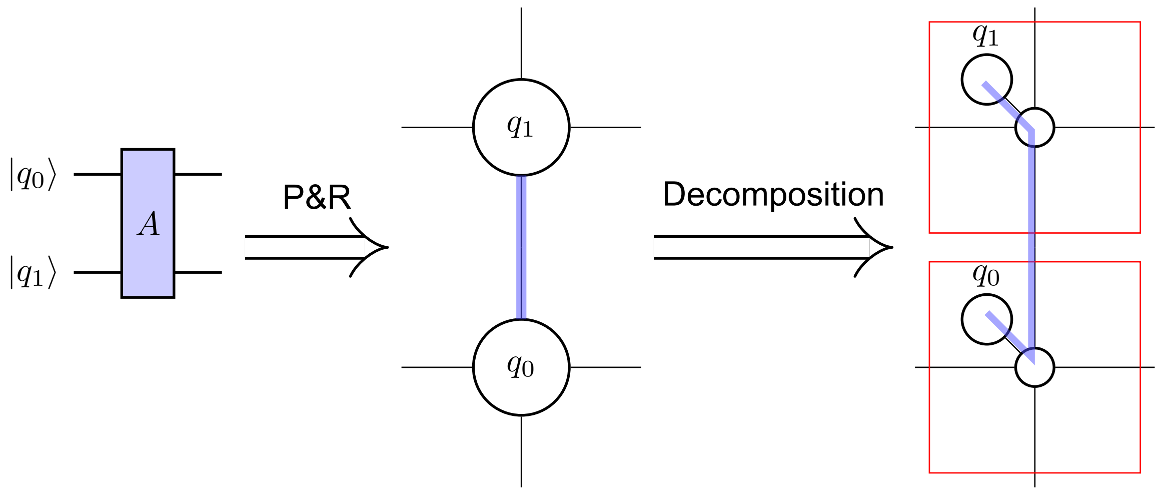


Figure 3.17: NVP1-1 mapping steps.

3.4. Conclusion

In this chapter the design of the quantum instruction set architecture (QISA), micro-ISA, and a compiler for NV centers has been presented. The μISA is designed to provide low-level control over the NV center qubits, to be executed by local controllers each responsible for one NV center. The QISA is designed as a layer between this low-level control and high-level algorithms, and is to be executed on a single global controller. It provides instructions for basic quantum gates as well as NV center-specific instructions for for example calibration.

Three methods of performing logical to physical qubit mapping and transformation have been explored. The NVP method was considered most suitable for this work and has been discussed and designed with greater detail. The NVD and NVC methods may also have a use in the future as a means of generating NVP decomposition rules.

4 Implementation

This chapter describes the implementation of the compiler as designed in Chapter 3. As determined in Section 2.4.9 the compiler OpenQL framework is used for this. The compiler passes are organized in 3 groups:

1. Section 4.1 describes the implementation of the **NV mapping** pass group, which entails the NVP mapping passes as designed in Section 3.3. These transform the logical instructions into physical instructions. These passes can be replaced by the NVD and NVC mapping passes if those are implemented in the future.
2. In Section 3.3 the design and implementation of the **common** passes is described. These passes perform further processing of instructions on the quantum level. These are common to each of the NV mapping methods.
3. Section 4.3 describes the development of the **backend**. The backend passes translate the generic quantum instructions to the QISA designed in Section 3.1. For reasons to be discussed in this section the backend is implemented outside of OpenQL.

4.1. NVP Mapping Implementation

This section describes the implementation of the NVP mapping approach as designed in Section 3.3 in OpenQL. First the qubit organization in OpenQL is defined, and then the implementations of the place and route and decomposition passes are described.

4.1.1. Internal Qubit Organization

OpenQL has no direct option to distinguish between logical and physical qubits; all qubits are organized in a single list with a single index to specify the qubit. One option to make this distinction is to add native support for it in OpenQL, using two different lists for the different types of qubits. This would however require adapting all existing passes and other components in OpenQL to this new representation.

While this option might be the most scalable and adaptable in the long term, for the purpose of this thesis an alternative solution is used where the distinction between logical and physical qubits is made purely by index: for a quantum system with N logical qubits each consisting of M physical qubits, the first N qubits are designated as logical and the next $N \cdot M$ as physical. Truly distinguishing between the different qubits is left to how they are used by the passes. Figure 4.1 gives an example of how the qubits are numbered.

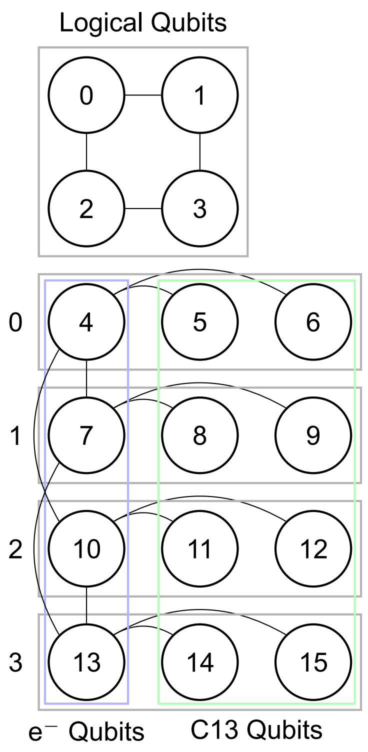
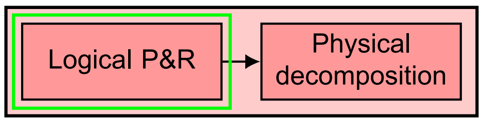


Figure 4.1: Qubit organization in OpenQL for logical qubits consisting of 1 NV center with 2 ^{13}C qubits. The 4 bottom rows each correspond to a logical qubit.

4.1.2. Place & Route



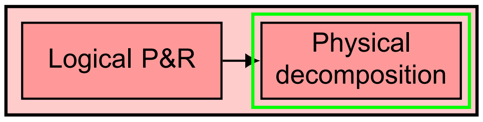
As mentioned in Section 3.3.2, an existing P&R method can be used for the NVP approach. The built-in `map.qubits.Map` pass of OpenQL can perform initial placement and routing for 2-qubit gates on arbitrarily connected qubits, making it suitable for the purpose of NVP mapping.

The existing P&R requires as input the full connectivity graph of the qubits, specifying both qubit locations and connecting edges. The 'physical' qubit locations can be used by OpenQL to more easily determine which qubits are close together. For NVP the locations are specified for the logical qubits corresponding to the logical topology, since that is what P&R is performed on.

Since P&R only happens on logical qubits, location and connectivity information is not necessary for the physical qubits as far as NVP is concerned. OpenQL does however require these values for a valid platform configuration. In the current implementation, the locations given for physical qubits are placeholders, but the connectivity is fully specified. Specifying the full physical connectivity as well allows for inserting specific physical operations in the input circuit. This has shown to be useful during development and may continue to be used for similar purposes.

To facilitate the construction of a regular 2D grid with nearest-neighbor connectivity, a script has been created which generates exactly this given just the grid size. This script generates the logical topology with location and connectivity, and the physical qubits with placeholder location and full physical connectivity. The layout generated by this script is as shown in Figure 4.1.

4.1.3. Decomposition



The NVP decomposition pass translates operations on logical qubits into operations on the physical qubits present in NV centers as described in Section 3.3.3.

The existing decomposition pass `dec.Instructions` from OpenQL has been used as a starting point for this pass. The existing pass provides the `op(n)` function to refer to instruction operands from within a decomposition. The example in Figure 4.2 shows how a Hadamard gate can be decomposed into two other gates using this notation. Note that operand counting starts at 0.

Input instruction

```
h q[2]
```

Decomposition rule

```
y90 op(0)
x op(0)
```

Output instructions

```
y90 q[2]
x q[2]
```

Figure 4.2: Applying a decomposition rule in OpenQL.

For the logical to physical decomposition pass, several modifications have been made to the built-in decomposition pass to make it suit the needs of NVP mapping.

To start with, the input operations refer to *logical* qubits, while the output operations need to refer to *physical* qubits. To facilitate this the function `phys(n,m)` has been added to OpenQL, which refers to the mth physical qubit of the logical qubit provide as nth operand. Note that due to the nature of the NVP method this function is not specific to NV centers and can thus be easily adopted for generic usage.

If the nth operand refers to logical qubiist l, then for a quantum system with N logical qubits each consisting of M physical qubits the internal index of the physical qubit referred to by `phys(n,m)` can be computed as

$$p = N + M \cdot l + m.$$

Figure 4.3 shows an example of how the qubit indices are computed when using a qubit organization as shown in Figure 4.1. In this situation $N = 4$ and $M = 3$.

Input instruction

```
h q[2]
```

Decomposition rule

```
y90 phys(0,1)
x phys(0,1)
y90 phys(0,2)
x phys(0,2)
```

Output instructions

```
y90 q[11]
x q[11]
y90 q[12]
x q[12]
```

Figure 4.3: Applying an NVP decomposition rule on the system shown in Figure 4.1.

Next, different decompositions for each edge type can be specified in the platform configuration. The original decomposition pass can select which decomposition rule to use based on a predicate function taking just the rule itself into account. This has been extended to also include the instruction parameters to be able to determine the edge type of each instruction. The implemented predicate function accepts a decomposition rule if it is labelled as ‘nvp’ and the rule edge type matches the instruction edge type.

Since especially for simple or regular logical qubit layouts some amount symmetry is expected, a script has been written that can generate a complete specification given the decomposition for one edge and duplicate it to other edges.

Finally, the original decomposition pass feeds the decomposition output back into itself to allow for further decomposition if possible. Since for NVP the decomposition rules always convert logical operations into physical operations, the resulting physical operations do not need further processing by this pass therefore this recursive approach is not necessary. In fact, disabling recursion completely allows for using the same instruction name for both logical and physical operations without their meanings getting mixed up.

NVP1-1 Decompositions

A full listing of the implemented logical to physical decomposition rules for an NVP1-1 system with 1 NV center with 1 ^{13}C qubit per logical qubit can be found in Appendix C.1. More rules can easily be added using a configuration file.

4.2. Common Compiler Passes

The program generated by the NV mapping passes operates on physical qubits. Several steps still need to be taken to arrive at the hardware level though. This starts with a number of passes that operate purely on a quantum system level, without taking into account control hardware details. These passes are common to all the NV mapping methods described in Section 3.2.

This section describes the design and implementation of 3 such compiler passes shown in Figure 4.4: decomposition (Section 4.2.1), optimization (Section 4.2.2), and scheduling (Section 4.2.3).

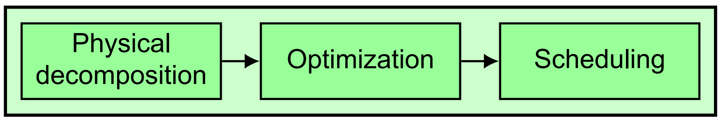


Figure 4.4: Overview of the common passes.

4.2.1. Physical Decomposition

This pass performs decomposition of physical operations, bringing them closer to the gates supported by NV centers. This allows prior passes to work on operations at a higher level, thus for example reducing the amount of input required in the NVP decomposition pass. The platform configuration needs to specify the possible decompositions.

To implement this pass, the improved version of the built-in decomposition pass as modified for NVP decomposition in Section 4.1.3 is reused. In the code itself both of these passes use the same base class but with a different configuration.

First, the original `op(n)` function for referring to input operands can be used again in this pass, since both the input as output instructions operate on physical qubits. Next, while for NVP decomposition recursion was not desired, for this pass again the original setting is used. This allows the physical decomposition rules to be expanded recursively.

Finally, a different kind of predicate function is required to select the correct decomposition rule. For this pass this is determined by the type (e$^-$ qubits and ^{13}C qubits) and ordering of physical qubits used by the instruction. Each decomposition rule specifies which types of qubits it can be applied to. The implemented predicate function then accepts a decomposition rule if it is labelled as ‘phys’ and the rule qubit types matches the instruction qubit types.

Examples

The native controlled gate in an NV center always has the e$^-$ qubit as control and ^{13}C qubit as target. If a CNOT in the other direction is required it has to be decomposed into a CNOT from e$^-$ qubit to ^{13}C qubit surrounded by Hadamard gates as shown in Figure 4.5.

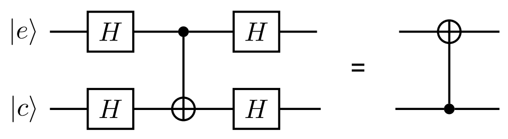


Figure 4.5: Reversing the direction of a CNOT using Hadamard gates.

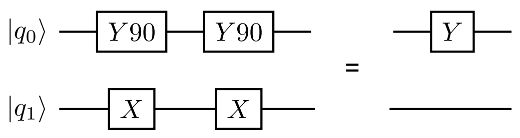


Figure 4.6: Optimization of single-qubit Clifford gates.

4.2.2. Optimization

This pass performs optimization of operations. Currently only the built-in `opt.clifford.Optimize` pass is used to implement this functionality, minimizing the amount of single-qubit Clifford gates.

Examples

The example illustrated in Figure 4.6 shows how two Y90 gates will merge into a single Y gate with a rotation angle of 180°, and two X gates will cancel each other out.

4.2.3. Scheduling

This pass performs scheduling of instructions, arranging them to be executed in parallel where possible and inserting delays where needed. This is implemented using the built-in `sch.ListSchedule` pass, which requires the platform configuration to specify the duration of each instruction.

Additionally, the scheduler can manage shared resources and schedule instructions without clashing. As described in Section 3.1, the envisioned hardware architecture has one local controller and other control electronics per NV center, making these a shared resource between the e$^-$ qubit and ^{13}C qubits. OpenQL allows for defining a resource which is utilized by the scheduler. The resource is required to define a function that is called for every instruction schedule considered, allowing the resource to keep track of what is in use at which point in time. If the schedule results in overlapping resource usage it is discarded.

A scheduling task that has not been solved with the current implementation has been found to occur with initialization of the ^{13}C qubits. On the hardware level, this involves first initializing the corresponding e$^-$ qubit and then swapping that with the ^{13}C qubit. If an ^{13}C qubit initialization is scheduled in between some e$^-$ qubit operations in the same NV center, this leaves the e$^-$ qubit in an unknown state.

4.3. Classical Backend

With processing of instructions on the quantum level complete, the next step is to transform them into the QISA instructions described in Section 3.1.2 making the program suitable for execution on a hardware controller or with a simulator. This process is handled by the backend, for which a custom framework has been developed as will be discussed in Section 4.3.1. Inspired by OpenQL, this framework is modular and allows for custom compiler passes to be defined. Figure 4.7 gives an overview of the implemented backend passes.

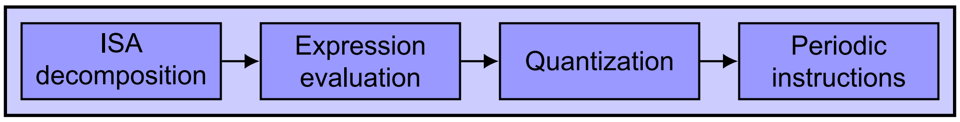


Figure 4.7: Passes in the classical backend.

4.3.1. Design and Functionality

While OpenQL has so far been very useful for implementing passes acting on quantum operations, it has been found to be less flexible than desired when it comes to instructions at the hardware architecture level.

For this reason, a custom 'mini compiler framework' has been developed in the Lua programming language to function as backend. The overall design of this backend is based heavily on OpenQL, but is more flexible and where needed more specifically targeted to handling classical instructions. It should be noted however that, as a custom non-production tool, the increased flexibility comes at the price of less strict data structures and error handling.

The backend framework is centered around the `backend` function, as shown in Listing 4.1. This function can be invoked with two arguments; the initial contents of the IR and an order list of passes. The passes themselves are again functions taking the IR as single argument. This function can be specified directly as illustrated by `pass1`. All the implemented however use an indirect approach such as `pass2`, using a generator function that takes additional arguments to configure the pass.

Listing 4.1: Invoking the backend function.

```
backend{
    ir = {
        -- initial IR contents
    },
    passes = {
        pass1 = function (ir)
            -- pass implementation
        end,

        pass2 = pass2_gen(arguments)
    }
}
```

4.3.2. ISA Decomposition

The backend starts with yet another decomposition pass, converting the quantum operations into the QISA instructions as specified in Appendix B.1. Similar to the physical decomposition pass described in Section 4.2.1, the qubit type and ordering in the instruction determines which decomposition rule is selected. The decomposition rules in turn need to specify on which input qubit types they can be applied.

Due to the hardware organization with a single local controller per NV center, the qubit index needs to be adjusted to this format. Every quantum instruction has an index that specifies the NV center and corresponding local controller, and instructions operating on a ^{13}C qubit have an additional index that specifies the ^{13}C qubit.

To specify these values in a decomposition rule, the functions `nv(n)` and `ci(n)` are introduced. In both cases n is the instruction operand index just as with the `op(n)` and `phys(n,m)` function used in earlier decomposition passes. In the ISA decomposition pass `nv(n)` is then replaced by the index of the NV center the qubit is part of, and `ci(n)` is the ^{13}C qubit index within its NV center.

Due to the way OpenQL represents conditional instructions, the `phys(n,m)` function as implemented in Section 4.1.3 can not be used to refer to measurement results of physical qubits. As a workaround, for every instruction accepted by the ISA decomposition pass, a conditional version is generated. The conditional version's name is prefixed by `if_` and has an additional argument specifying the controlling physical qubit. Converting from the native OpenQL conditional instruction to this format has to be done in prior passes.

Examples

Listing 4.2 shows how the ISA decomposition pass can be instantiated in the backend framework. The `decompositions` argument is a list of decomposition rules structured as shown in Listing 4.3. This example is of an X-gate, for which two different decompositions are specified depending on the qubit type. This decomposition also shows how the `nv(n)` and `ci(n)` functions are used in practice.

Listing 4.2: ISA decomposition pass instantiation.

```
pass.dec_qtype{
    decompositions = dec_isa
},
```

Listing 4.3: ISA decomposition rule for an X-gate.

```
{
    name = "x",
    dec  = {{
        qtype = {"e"},
        into  = "qgatee nv(0), 0.0, pi"
    },{
        qtype = {"c"},
        into  = "qgateuc nv(0), ci(0),
            0.0, pi, 1"
    }}
};
```

Figure 4.8 shows how a `y` instruction operating on the system shown in Figure 4.1 is decomposed into a single `qgateUC` instruction.

A notable ISA decomposition is that for a `cnot` operation. The NV center-native controlled gate implemented by the `qgateCC` instruction, besides performing the desired X/Y rotation, also introduces a phase shift (i.e., a Z-rotation) on the ^{13}C qubit. Thus, to implement a CNOT as desired, this phase shift has to be removed by an opposing Z-rotation.

A full listing of the implemented ISA decomposition rules can be found in Appendix C.3, for each rule listing the instruction format, qubit types, and decomposition. A single example of the generated conditional instructions is given with `if_x`; others follow the same pattern.

Input

```
y q[8]
```

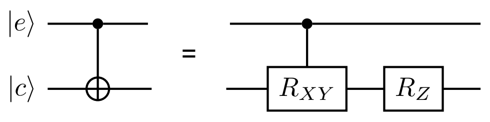
Decomposition rule

```
qgateUC nv(0), ci(0), pi/2, pi, 1
```

Output

```
qgateUC 1, 0, pi/2, pi, 1
```

Figure 4.8: Applying a single-qubit ISA decomposition rule on the system shown in Figure 4.1.



```
qgateCC nv(0), ci(1), 0.0, pi
qgateZC nv(0), ci(1), pi/2
```

Figure 4.9: QISA decomposition of a controlled NOT gate.

4.3.3. Operand Evaluation & Quantization

Qubit rotations are most often performed with angles that are fractions of π. For this reason it is advantageous to be able to use a symbolic expression for operands, allowing for example a value of $\pi/2$ to be represented exactly. The implemented operand evaluation pass computes the value of a symbolic expression including constants such as `pi` and stores it as a single double-precision floating-point number.

Furthermore, in a hardware controller only a limited number of bits can be used to represent each operand. The envisioned design represents operands as a 16-bit 'integer-like' value, which requires quantizing the 64-bit floats. This quantization step is implemented in another pass.

To make use of this pass, the QISA instruction definition is required to specify a quantization function. An example of such a function is limiting the possible values to a fixed number of options spaced equally between a minimum and maximum. A generator function is provided to easily construct such quantization functions. The pass can also be configured to replace operands by the 16-bit integer value that will be used by the hardware.

In the backend as currently configured, the operand evaluation and quantization passes directly follow each other, and the angle operands are quantized to steps of $\pi/2^{-15}$ in the range $\langle -\pi; \pi]$. Since in practice the symbolic expressions are usually of the form π/n with $n = 1, 2, 4$, the two passes effectively cancel each others' effect. It would be possible to merge the passes into one and potentially increase accuracy and compilation performance. However, the current approach is more modular and allows for more complex expressions and non-aligned quantization if so desired. Additionally, in practice performance has not been a concern and the double-precision floats provide enough precision to not cause any problems.

Examples

Listing 4.4 shows how the expression evaluation passes can be instantiated. The evaluation pass takes as argument a list of constants to replace, in this example instances of the symbolic `pi` are replaced by the floating-point constant. The quantization pass makes use of the quantization functions specified in the QISA.

Listing 4.4: Expression evaluation and quantization pass instantiation.

```
pass.eval_expr{
    constants = {pi = math.pi}
},
pass.quantize{
    instructions = isa_ucode
},
```

Figure 4.10 shows how both passes operate on an instruction. In this example, and what is also expected in practice, the operands are quantized to some multiple of π resulting in no effect for most gates.

Input

```
qgateE q[3], pi/2, pi
```

Output

```
qgateE q[3], 1.5707963267949, 3.1415926535898
```

Figure 4.10: Applying expression evaluation and quantization to an instruction.

4.3.4. Periodic Instructions

The original compiler implementation from [36] has the ability to insert instructions periodically, more specifically it inserts a charge-resonance check every 10 instructions. While this exact functionality is in fact not desired since the check destroys the quantum state of the e$^-$ qubit, it has been reimplemented for this work primarily as a proof of concept.

Improving on the original implementation, this version inserts instruction after a specific amount of *time* rather than instructions, taking into account the duration of each instruction. This information is already required by OpenQL but has been extended for the purpose of NV centers. Instead of allowing only fixed instruction durations, a function can be specified that takes the actual instruction operands and returns a duration based on those values. For example, in the case of qubit rotations on NV centers, larger rotation angles directly correspond to a longer instruction duration, which this system can take into account.

Examples

Listing 4.5 shows how the periodic instruction pass can be instantiated. The pass requires information about the instruction duration from the QISA. The instructions to be inserted can then be specified together with their period.

Listing 4.5: Periodic instruction pass instantiation.

```
pass.periodic{
    instructions = isa_ucode,
    scheduled = {{
        period = 200,
        into = "crc"
    }}
},
```

4.3.5. Output

As the very last step, the finalized list of instructions has to be written to an output. Depending on which passes were executed the output may still contain symbolic expressions which are handled by the writer pass as well. Three output passes have been implemented for different purposes.

The first is closest to the internal representation and thus doubles as debugging output. Other output generators can be derived from this by disabling output that is not needed by the target, or for example renaming or reordering information.

One such derived writer pass has been made tailored for a quantum simulator implemented in [60]. The simulator expects a certain formatting for operands and does for example not handle parallel instructions. Output produced by this pass will be discussed in Section 5.1.

Finally, a 'non-functional' writer pass has been implemented that writes the microcode corresponding to each instruction to the output. This output is not functional as-is, but is intended as an aid in more precisely defining the hardware requirements of the local controller as outlined in Section 3.1.3.

4.3.6. Future Extensions

While the implemented passes are deemed sufficient for use with the simulator, a number of changes are required to make it functional for an actual hardware implementation.

Firstly, the writer passes currently implemented are for either human readability or the simulator. When the hardware from Section 3.1.2 is implemented, a form of binary output will be required as well.

Next, since yet another decomposition step has been performed in the backend, rescheduling of the new instructions might be beneficial. This will require a scheduler that takes into account not just the quantum system and local control electronics, but the global control system as well.

Finally, implementation of the global controller and communication system will most likely lead to a need for additional passes and modifications. Specifically the communication is expected to be a shared resource that the instruction scheduling must consider or at least can optimize for.

4.4. Conclusion

In this chapter the NVP mapping method has been implemented in OpenQL. This consits of an exsiting place and route pass applied to the logical qubits, followed by a custom decomposition pass that transforms the logical operations into physical operations. This required adding new functionality to OpenQL and modifying it to work with both logical and physical qubits simultaneously.

Additionally, several compiler passes common to the different NV mapping methods have been identified. A physical decomposition pass has been implemented, an existing optimization pass reused, and a scheduling pass has been extended with NV center-specific functionality.

Finally, a backend has been implemented separate from OpenQL. The backend transforms generic quantum instructions to QISA instructions making use of symbolic decomposition specifications. Its modular design has been demonstrated by the implementation of an operand quantization and periodic scheduling pass.

Figure 4.11 illustrates the complete compilation flow with all utilized passes.

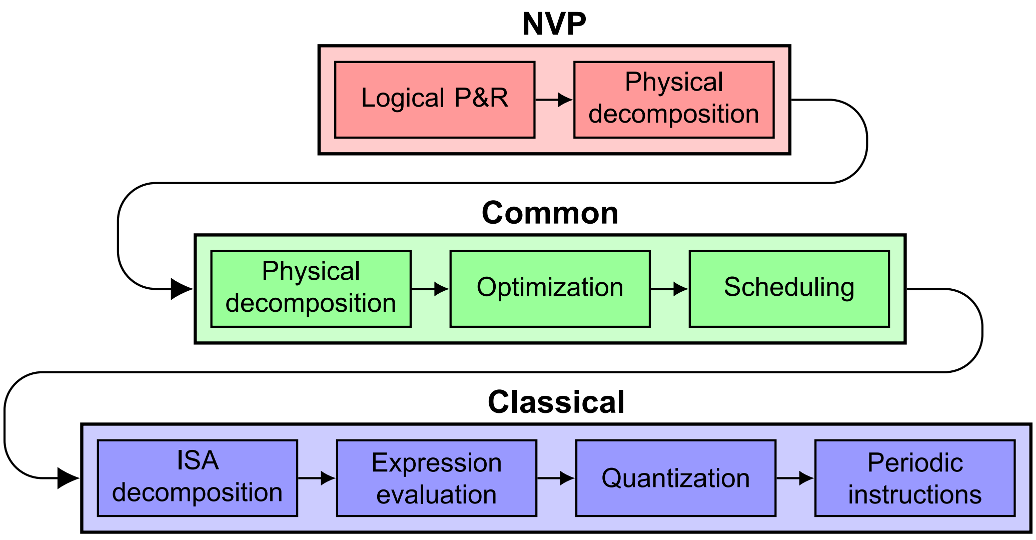


Figure 4.11: Overview of the implemented compiler passes.

5 Results

With the compiler designed and implemented, its functionality can be verified. The complete compilation process for is discussed for multiple circuits in Section 5.1, followed by the simulation results of two simple circuits in Section 5.2.

5.1. Compilation Results

This section presents the compilation process of 3 circuits from algorithm input to QISA output. First two simple circuits are compiled, both consisting of just 2 logical operations. Next, a circuit implementing Grover's algorithm is compiled to demonstrate the compiler can handle larger circuits as well.

5.1.1. Single-Qubit Circuit

The first circuit consists of just two operations on a single logical qubit. Figure 5.1 shows the logical circuit consisting of initializing the qubit and applying a Hadamard gate.

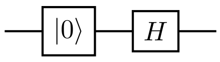


Figure 5.1: Logical circuit performing initialization and a H-gate.

Listing 5.1 shows the Python code implementing this circuit. For brevity, the configuration and initialization code has been left out. Complete Python inputs and their corresponding outputs can be found in Appendix D.

Listing 5.1: Python code for the H-gate circuit.

```
kernel.prepz(0)
kernel.gate('h', 0)
```

To compile this using the NVP method, details of the physical system have to be specified as well. This circuit is compiled for a physical system consisting of 2 NV centers with one ^{13}C qubit each, with each NV center representing 1 logical qubit. The corresponding qubit organization of this system in OpenQL is illustrated in Figure 5.2.

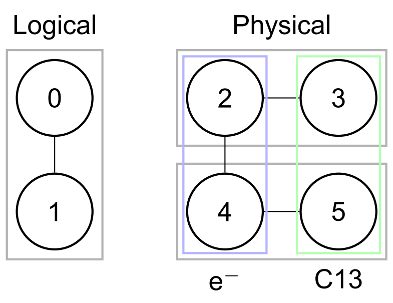


Figure 5.2: Qubit organization for a small NV center system.

The cQASM output produced by OpenQL is shown in Listing 5.2. As with the Python code, the header and other non-instruction lines have been omitted for brevity. Note the qubit index is 3 here, corresponding to the first ^{13}C qubit of the first NV center as seen in Figure 5.2.

Listing 5.2: cQASM code for the H-gate circuit.

```
prep_z q[3]
y90 q[3]
x180 q[3]
skip 1
```

Figure 5.3 shows a visual representation of the physical circuit. Note how the ^{13}C qubit is initialized by first initializing the e$^-$ qubit and then moving the $|0\rangle$ state using the `swapEC` instruction.

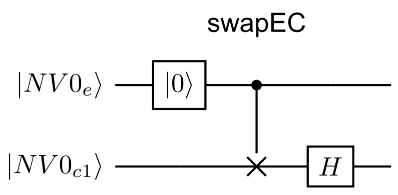


Figure 5.3: Physical circuit implementing the H-gate circuit.

Finally, passing this through the backend gives the NV assembly shown in Listing 5.3.

Listing 5.3: NV assembly for the H-gate circuit.

```
initialize q0
swapec q0 0
qgateuc q0 0 1.5707963267949 1.5707963267949 1
qgateuc q0 0 0.0 3.1415926535898 1
```

5.1.2. Two-Qubit Circuit

As a more complex example, the next circuit involves a Hadamard gate and a controlled NOT gate. Depending on the initial conditions this will entangle the two logical qubits to form a Bell pair. Since this circuit involves a two-qubit gate, place and route on the logical qubit level is required. The logical circuit is visualized in Figure 5.4.

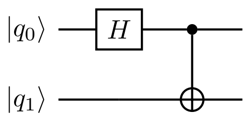


Figure 5.4: Logical circuit performing a H-gate and CNOT.

The corresponding abbreviated Python code is shown in Listing 5.4. The full code can again be found in Appendix D.

Listing 5.4: Python code for the entangle circuit.

```
kernel.gate('h', 0)
kernel.cnot(0, 1)
```

This logical circuit is compiled to the same physical system using the same configuration as the previous circuit. The intermediate cQASM output from OpenQL is shown in Listing 5.5.

Listing 5.5: cQASM code for the entanglement circuit.

```
nventangle q[2], q[4]
y90 q[2]
x180 q[2]
cnot q[2], q[3]
y90 q[2]
x180 q[2]
measure q[2]
if_x q[4], q[2]
cnot q[4], q[5]
y90 q[4]
x180 q[4]
measure q[4]
y90 q[3]
x180 q[3]
if_z q[3], q[4]
```

The physical circuit is visualized in Figure 5.5. This visualization shows how the CNOT between two logical qubits is performed; first the ^{13}C qubit in the source qubit is entangled with the e^{-} qubit in the target qubit, then the actual 2-qubit gate is performed, and finally the physical qubits are disentangled again to complete the circuit.

Additionally, the entangling procedure introduces multiple H-gates, one of which is applied to the same physical qubit that the logical H-gate directly maps to. These two gates outlined in gray in Figure 5.5 cancel each other out, which is reflected in the compiler output.

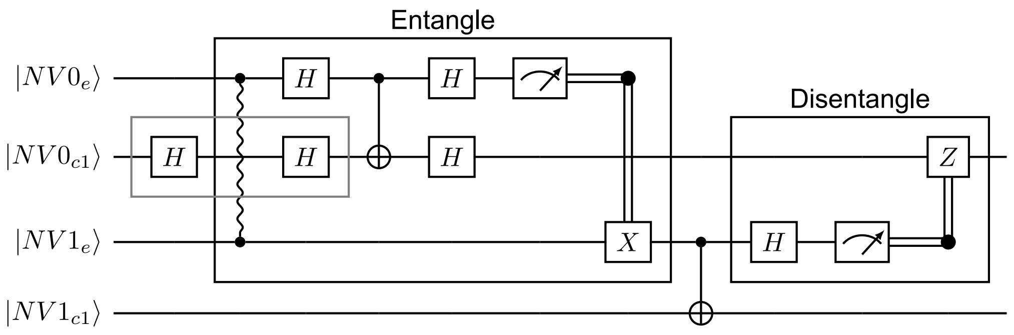


Figure 5.5: Physical circuit implementing the entanglement circuit.

Finally, the decomposed and optimized circuit can be passed through the backend, producing the NV assembly shown in Listing 5.6. Note how the CNOT is further decomposed into QISA instructions as detailed in Section 4.3.2. On the other hand the `nventangle` instruction is not decomposed as its implementation is still uncertain at this point. The required QISA instructions are available however, most importantly the `entangle` instruction.

Listing 5.6: NV assembly for the entanglement circuit.

```
nventangle q0 q1
qgatee q0 1.5707963267949 1.5707963267949
qgatee q0 0.0 3.1415926535898
qgatecc q0 0 0.0 3.1415926535898
qgatezc q0 0 1.5707963267949
qgatee q0 1.5707963267949 1.5707963267949
qgatee q0 0.0 3.1415926535898
measuree q0
br MeasureResultRegNVnode0 > 0 x_11
qgatee q1 0.0 3.1415926535898
x_11:
qgatecc q1 0 0.0 3.1415926535898
qgatezc q1 0 1.5707963267949
qgatee q1 1.5707963267949 1.5707963267949
qgatee q1 0.0 3.1415926535898
measuree q1
qgateuc q0 0 1.5707963267949 1.5707963267949 1
qgateuc q0 0 0.0 3.1415926535898 1
br MeasureResultRegNVnode1 > 0 z_18
wait 2
z_18:
```

5.1.3. Grover's Algorithm

To verify the compiler also works with larger scale quantum algorithms, a circuit implementing Grover's search algorithm has been compiled, of which the full Python code can be found in Appendix D. The original code is taken from the example circuits included with OpenQL. To facilitate the use of Toffoli gates in the input circuit, the built-in decomposition pass has been instantiated in OpenQL to decompose these gates into sequences of 1- and 2-qubit gates prior to the NVP mapping passes. This shows how the newly developed passes seamlessly integrate into the existing OpenQL framework.

In the code for the quantum algorithm itself, the only partially functional change made was the removal of the 'display' gates, which produced an error in OpenQL for unknown reasons. The 'display' gate is however non-functional in a real quantum computer as it prints out the full quantum state. While it can be useful to debug circuits using a simulator, it has been removed for this test to focus on the compilation results.

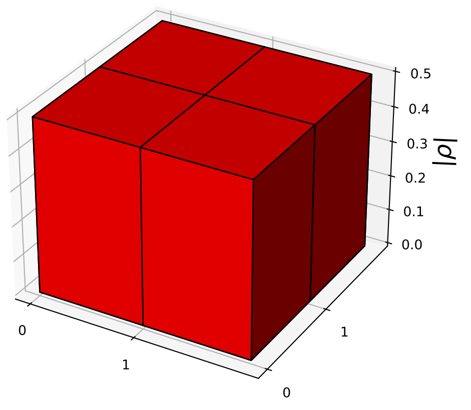
With this slight modification and several organizational changes, the code compiles into a program consisting of nearly 4000 NV assembly instructions on 18 qubits across 9 NV centers. The exact number of instructions varies between 3500 to 4100 instructions on different compilation runs due to variations in the P&R pass in OpenQL. This compilation result exposed a simplification taken in the backend where different subprograms are combined into one without taking into consideration their order. This can however be remedied outside of the compiler by either compiling the subprograms separately or merging them in the desired order beforehand.

5.2. Simulation of Compiled Programs

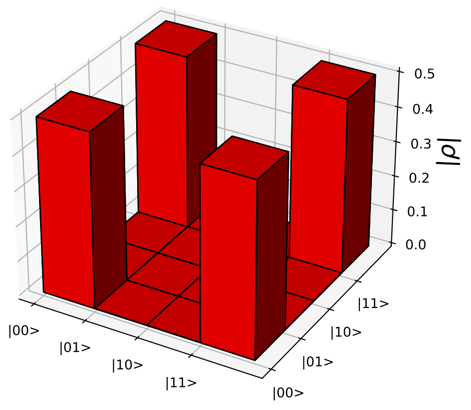
Using the simulator developed in [60], the first two smaller circuits have been simulated.

The simulator result of the Hadamard circuit from Section 5.1.1 is shown in Figure 5.6a. The figure only shows the state of the ^{13}C qubit, as this directly corresponds to the state of the logical qubit. The outcome is the expected result of the ^{13}C qubit being in the $|+\rangle$ state.

For the entanglement circuit from Section 5.1.2 the qubits have been initialized in the $|0\rangle$ state. The expected outcome is thus a $|\Phi_+\rangle$ state of the logical qubits, which as with the Hadamard circuit corresponds to the state of the ^{13}C qubits. Figure 5.6b shows the simulation result, which is again the expected outcome.



(a) Simulator output for the H-gate circuit.



(b) Simulator output for the entanglement circuit.

Figure 5.6

5.3. Conclusion

In this chapter the functionality of the compiler has been verified by compiling several circuits. Two simple circuits have been compiled and simulated showing the expected results across all steps in the process. A more complex circuit implementing Grover's search algorithm has been compiled as well demonstrating the scalability of the implemented compiler.

6 Conclusion

This chapter concludes this thesis. Section 6.1 summarizes the work discussed in each chapter, followed by the main contributions of this thesis in Section 6.2. Finally, Section 6.3 gives several recommendations for future work that can further contribute to the field of quantum computing.

6.1. Summary

- Chapter 2 introduced the reader to quantum information theory, NV center physics, existing control systems, and quantum compilers relevant to the scope of this thesis.

  A number of existing quantum compilers have been compared, after which the OpenQL compiler framework has been selected as base for the compiler for NV centers for its generality and ease of extensibility.

- In Chapter 3 the design of the quantum instruction set architecture (QISA), micro-ISA, and a compiler for NV centers has been presented. The μISA is designed to provide low-level control over the NV center qubits, to be executed by local controllers each responsible for one NV center. The QISA is designed as a layer between this low-level control and high-level algorithms, and is to be executed on a single global controller. It provides instructions for basic quantum gates as well as NV center-specific instructions for for example calibration.

  Three methods of performing logical to physical qubit mapping and transformation have been explored. The NVP method was considered most suitable for this work and has been discussed and designed with greater detail. The NVD and NVC methods may also have a use in the future as a means of generating NVP decomposition rules.

- In Chapter 4 the NVP mapping method has been implemented in OpenQL. This consits of an exsiting place and route pass applied to the logical qubits, followed by a custom decomposition pass that transforms the logical operations into physical operations. This required adding new functionality to OpenQL and modifying it to work with both logical and physical qubits simultaneously.

  Additionally, several compiler passes common to the different NV mapping methods have been identified. A physical decomposition pass has been implemented, an existing optimization pass reused, and a scheduling pass has been extended with NV center-specific functionality.

  Finally, a backend has been implemented separate from OpenQL. The backend transforms generic quantum instructions to QISA instructions making use of symbolic decomposition specifications. Its modular design has been demonstrated by the implementation of an operand quantization and periodic scheduling pass.

- In Chapter 5 the functionality of the compiler has been verified by compiling several circuits. Two simple circuits have been compiled and simulated showing the expected results across all steps in the process. A more complex circuit implementing Grover's search algorithm has been compiled as well demonstrating the scalability of the implemented compiler.

6.2. Contributions

The main contributions of this thesis are:

1. **The specification of a micro-ISA that can perform quantum operations on NV centers.**
   The specification of a μISA has been achieved by making an inventory of how the physical quantum system and control electronics must be controlled. Micro-instructions have been defined for physical operations as well as supporting instructions.
2. **The specification of a high-level QISA targeting NV centers that acts as an interface between software and hardware.**
   The QISA has been specified on top of the μISA, making use of the microcode from [60] and building upon the work from [36]. While the QISA has been designed for NV centers, many aspects of it are directly applicable to other types of colors centers as well.
3. **A generic method of transforming a logical quantum circuit into a physical quantum circuit.**
   The NVP mapping method introduced in Section 3.2.1 can transform a logical quantum circuit into a physical quantum circuit. While it has been designed with NV centers in mind, the general approach is applicable to any kind of qubit technology. Two other mapping methods have also been explored.
4. **A functional compiler that can transform a quantum algorithm written in Python into a sequence of QISA instructions.**
   Making use of NVP mapping and a newly designed backend, a functional compiler has been implemented that can transform a logical circuit into a sequence of QISA instructions. Making use of the OpenQL compiler framework, **new passes have been implemented which seamlessly integrate with existing passes.** This allows the compiler to take advantage of prior and future work on OpenQL, such as the possibility to specify input circuits in Python, C++, and cQASM.
5. **Direct and indirect improvements of the OpenQL compiler framework.**
   Finally, implementing the compiler in OpenQL revealed several limitations for which a solution or workaround had to be found. Some of the implemented solutions can be incorporated to directly contribute to the development of OpenQL, and the workarounds can guide future developments to help resolve them.

6.3. Future Work

While this thesis focuses on NV centers, significant portions of the work done in this thesis is applicable to other types of color centers as well. However, the context in which which this thesis is done focuses on NV centers. Whether a certain idea or implementation works for NV centers specifically is thus of the highest importance. Any coincidental applicability to other types of color centers is taken as a positive result, but considered not worth validating for every idea and for all types of color centers. Since the future plans of the Fujitsu project include tin-vacancy centers, this is the first candidate for verifying the more generic applicability of this work.

The goal of this thesis as defined in Chapter 1 is to create a compiler capable of handling larger logical qubits, possibly with error detection and correction. The implemented NVP method has only been verified using simple logical qubit layouts however. Creating the required configuration files and verifying the compiler with such a larger qubit is thus still to be done. Additionally, given the often regular structure of large logical qubits, the configuration files can most likely be partially or even fully computer-generated. Identifying the structure and creating such a generator can prove useful for future large-scale quantum computers.

Of the 3 different mapping methods discussed in Section 3.2, only the NVP method has been implemented. However, both the NVC and NVD methods may have their uses in certain situations as well. In particular, using one of these methods to optimize the physical operations *within* a logical qubit may be a promising use case. The NVC and NVD methods may also be useful in the short term for NISQ applications where little to no quantum error correction is used.

A generator script that has already proven its worth in this project is one for creating regular qubit topologies. This has been used in Section 4.1.1 to generate a combined logical and physical qubit topology. Providing similar functionality natively in OpenQL would simplify the topology specification required for the platform configuration files.

As mentioned in Section 4.3.1, the custom compiler framework designed for the backend has been made mostly for easier and faster prototyping by removing several restrictions that OpenQL poses. The limitations found can however be used as starting point for further extending the capabilities of OpenQL, at which point the functionality of the backend could be moved into the OpenQL framework.

Finally, verifying the design of the QISA and μISA will require the hardware discussed in Section 3.1 to be designed and implemented. While both ISAs have been carefully designed to match the hardware on one side and software on the other, there is the possibility that implementing them will expose shortcomings in the ISAs and compiler. Parallelism and other hardware optimizations can be explored as well, possibly requiring further modifications to the compiler as well.

References

[1] P. Hauke, H. G. Katzgraber, W. Lechner, H. Nishimori, and W. D. Oliver, “Perspectives of quantum annealing: Methods and implementations,” *Reports on Progress in Physics*, vol. 83, no. 5, p. 054 401, May 2020. DOI: 10.1088/1361-6633/ab85b8.

[2] *Qutech school of quantum*, Accessed 2023-04-17. [Online]. Available: https://www.qutube.nl/.

[3] W. K. Wootters and W. H. Zurek, “A single quantum cannot be cloned,” *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982. DOI: 10.1038/299802a0.

[4] A. G. Fowler, A. M. Stephens, and P. Groszkowski, “High-threshold universal quantum computation on the surface code,” *Physical Review A*, vol. 80, no. 5, p. 052 312, Nov. 2009. DOI: 10.1103/physreva.80.052312.

[5] A. J. Landahl and C. Ryan-Anderson, “Quantum computing by color-code lattice surgery,” Jul. 2014. DOI: 10.48550/arXiv.1407.5103.

[6] F. Pastawski, B. Yoshida, D. Harlow, and J. Preskill, “Holographic quantum error-correcting codes: Toy models for the bulk/boundary correspondence,” *Journal of High Energy Physics*, vol. 2015, no. 6, Jun. 2015. DOI: 10.1007/jhep06(2015)149.

[7] C. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, “Surface code quantum computing by lattice surgery,” *New Journal of Physics*, vol. 14, no. 12, p. 123 011, Dec. 2012. DOI: 10.1088/1367-2630/14/12/123011.

[8] A. G. Fowler and C. Gidney, “Low overhead quantum computation using lattice surgery,” Aug. 2019. DOI: 10.48550/arXiv.1808.06709.

[9] D. Litinski, “A game of surface codes: Large-scale quantum computing with lattice surgery,” *Quantum 3, 128 (2019)*, Aug. 2019. DOI: 10.22331/q-2019-03-05-128.

[10] S. Baier, C. Bradley, T. Middelburg, V. Dobrovitski, T. Taminiau, and R. Hanson, “Orbital and spin dynamics of single neutrally-charged nitrogen-vacancy centers in diamond,” *Physical Review Letters*, vol. 125, no. 19, p. 193 601, Nov. 2020. DOI: 10.1103/physrevlett.125.193601.

[11] F. Jelezko, T. Gaebel, I. Popa, A. Gruber, and J. Wrachtrup, “Observation of coherent oscillations in a single electron spin,” *Physical Review Letters*, vol. 92, no. 7, p. 076 401, Feb. 2004. DOI: 10.1103/physrevlett.92.076401.

[12] M. V. G. Dutt, L. Childress, L. Jiang, E. Togan, J. Maze, F. Jelezko, A. S. Zibrov, P. R. Hemmer, and M. D. Lukin, “Quantum register based on individual electronic and nuclear spin qubits in diamond,” *Science*, vol. 316, no. 5829, pp. 1312–1316, Jun. 2007. DOI: 10.1126/science.1139831.

[13] H. Bernien, “Control, measurement and entanglement of remote quantum spin registers in diamond,” PhD thesis, Feb. 2014. DOI: 10.4233/uuid:75130C37-EDB5-4A34-AC2F-C156D377CA55.

[14] F. Dolde, V. Bergholm, Y. Wang, I. Jakobi, B. Naydenov, S. Pezzagna, J. Meijer, F. Jelezko, P. Neumann, T. Schulte-Herbrüggen, J. Biamonte, and J. Wrachtrup, “High-fidelity spin entanglement using optimal control,” *Nature Communications*, vol. 5, no. 1, Feb. 2014. DOI: 10.1038/ncomms4371.

[15] 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, Feb. 2014. DOI: 10.1038/nnano.2014.2.

[16] C. Bradley, J. Randall, M. Abobeih, R. Berrevoets, M. Degen, M. Bakker, M. Markham, D. Twitchen, and T. Taminiau, “A ten-qubit solid-state spin register with quantum memory up to one minute,” *Physical Review X*, vol. 9, no. 3, p. 031 045, Sep. 2019. DOI: 10.1103/physrevx.9.031045.

[17] S. Pezzagna and J. Meijer, “Quantum computer based on color centers in diamond,” *Applied Physics Reviews*, vol. 8, no. 1, p. 011 308, Mar. 2021. DOI: 10.1063/5.0007444.

[18] O. Kahl, S. Ferrari, P. Rath, A. Vetter, C. Nebel, and W. H. P. Pernice, “High efficiency on-chip single-photon detection for diamond nanophotonic circuits,” *Journal of Lightwave Technology*, vol. 34, no. 2, pp. 249–255, Jan. 2016. DOI: 10.1109/jlt.2015.2472481.

[19] N. Kalb, A. A. Reiserer, P. C. Humphreys, J. J. W. Bakermans, S. J. Kamerling, N. H. Nickerson, S. C. Benjamin, D. J. Twitchen, M. Markham, and R. Hanson, “Entanglement distillation between solid-state quantum network nodes,” *Science*, vol. 356, no. 6341, pp. 928–932, Jun. 2017. DOI: 10.1126/science.aan0070.

[20] C. Liu, M. V. G. Dutt, and D. Pekker, “Single-photon heralded two-qubit unitary gates for pairs of nitrogen-vacancy centers in diamond,” *Physical Review A*, vol. 98, no. 5, p. 052 342, Nov. 2018. DOI: 10.1103/physreva.98.052342.

[21] H. Bernien, B. Hensen, W. Pfaff, G. Koolstra, M. S. Blok, L. Robledo, T. H. Taminiau, M. Markham, D. J. Twitchen, L. Childress, and R. Hanson, “Heralded entanglement between solid-state qubits separated by three metres,” *Nature*, vol. 497, no. 7447, pp. 86–90, Apr. 2013. DOI: 10.1038/nature12016.

[22] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M. S. Blok, J. Ruitenberg, R. F. L. Vermeulen, R. N. Schouten, C. Abellán, W. Amaya, V. Pruneri, M. W. Mitchell, M. Markham, D. J. Twitchen, D. Elkouss, S. Wehner, T. H. Taminiau, and R. Hanson, “Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres,” *Nature*, vol. 526, no. 7575, pp. 682–686, Oct. 2015. DOI: 10.1038/nature15759.

[23] M. W. Doherty, J. Michl, F. Dolde, I. Jakobi, P. Neumann, N. B. Manson, and J. Wrachtrup, “Measuring the defect structure orientation of a single nv$^-$ centre in diamond,” *New Journal of Physics*, vol. 16, no. 6, p. 063 067, Jun. 2014. DOI: 10.1088/1367-2630/16/6/063067.

[24] P. Rembold, N. Oshnik, M. M. Müller, S. Montangero, T. Calarco, and E. Neu, “Introduction to quantum optimal control for quantum sensing with nitrogen-vacancy centers in diamond,” *AVS Quantum Science*, vol. 2, no. 2, p. 024 701, Jun. 2020. DOI: 10.1116/5.0006785.

[25] F. Dolde, I. Jakobi, B. Naydenov, N. Zhao, S. Pezzagna, C. Trautmann, J. Meijer, P. Neumann, F. Jelezko, and J. Wrachtrup, “Room-temperature entanglement between single defect spins in diamond,” *Nature Physics*, vol. 9, no. 3, pp. 139–143, Feb. 2013. DOI: 10.1038/nphys2545.

[26] X. Rong, J. Geng, F. Shi, Y. Liu, K. Xu, W. Ma, F. Kong, Z. Jiang, Y. Wu, and J. Du, “Experimental fault-tolerant universal quantum gates with solid-state spins under ambient conditions,” *Nature Communications*, vol. 6, no. 1, Nov. 2015. DOI: 10.1038/ncomms9748.

[27] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver, “Superconducting qubits: Current state of play,” *Annual Review of Condensed Matter Physics*, vol. 11, no. 1, pp. 369–395, Mar. 2020. DOI: 10.1146/annurev-conmatphys-031119-050605.

[28] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, “Trapped-ion quantum computing: Progress and challenges,” *Applied Physics Reviews*, vol. 6, no. 2, p. 021 314, Jun. 2019. DOI: 10.1063/1.5088164.

[29] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019. DOI: 10.1038/s41586-019-1666-5.

[30] E. T. Campbell, B. M. Terhal, and C. Vuillot, “Roads towards fault-tolerant universal quantum computation,” *Nature*, vol. 549, no. 7671, pp. 172–179, Sep. 2017. DOI: 10.1038/nature23460.

[31] H. Paik, A. Mezzacapo, M. Sandberg, D. McClure, B. Abdo, A. Córcoles, O. Dial, D. Bogorin, B. Plourde, M. Steffen, A. Cross, J. Gambetta, and J. M. Chow, “Experimental demonstration of a resonator-induced phase gate in a multiqubit circuit-QED system,” *Physical Review Letters*, vol. 117, no. 25, p. 250 502, Dec. 2016. DOI: 10.1103/physrevlett.117.250502.

[32] D. Rosenberg, D. Kim, R. Das, D. Yost, S. Gustavsson, D. Hover, P. Krantz, A. Melville, L. Racz, G. O. Samach, S. J. Weber, F. Yan, J. L. Yoder, A. J. Kerman, and W. D. Oliver, “3d integrated superconducting qubits,” *npj Quantum Information*, vol. 3, no. 1, Oct. 2017. DOI: 10.1038/s41534-017-0044-0.

[33] J. M. Hornibrook, J. I. Colless, I. D. C. Lamb, S. J. Pauka, H. Lu, A. C. Gossard, J. D. Watson, G. C. Gardner, S. Fallahi, M. J. Manfra, and D. J. Reilly, “Cryogenic control architecture for large-scale quantum computing,” *Physical Review Applied*, vol. 3, no. 2, p. 024 010, Feb. 2015. DOI: 10.1103/physrevapplied.3.024010.

[34] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, “A microarchitecture for a superconducting quantum processor,” *IEEE Micro*, vol. 38, no. 3, pp. 40–47, May 2018. DOI: 10.1109/mm.2018.032271060.

[35] S. S. Tannu, Z. A. Myers, P. J. Nair, D. M. Carmean, and M. K. Qureshi, “Taming the instruction bandwidth of quantum computers via hardware-managed error correction,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, Oct. 2017. DOI: 10.1145/3123939.3123940.

[36] Q. van Wingerden, “Quantum computer microarchitecture for color centers in diamond,” Master’s thesis, TU Delft, Aug. 2021. [Online]. Available: http://resolver.tudelft.nl/uuid:c643181f-4e60-40cc-a61c-ad59b825dd8a.

[37] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels, *Cqasm v1.0: Towards a common quantum assembly language*, 2018. DOI: 10.48550/ARXIV.1805.09607.

[38] *Qiskit documentation*, Accessed 2023-04-16. [Online]. Available: https://qiskit.org/documentation/index.html.

[39] *Qiskit transpiler passes tutorial*, Accessed 2023-04-16. [Online]. Available: https://qiskit.org/documentation/tutorials/circuits_advanced/04_transpiler_passes_and_passmanager.html.

[40] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, “On the qubit routing problem,” Feb. 2019. DOI: 10.4230/LIPIcs.TQC.2019.5.

[41] A. Yimsiriwattana and S. J. Lomonaco, *Generalized ghz states and distributed quantum computing*, 2004. DOI: 10.48550/ARXIV.QUANT-PH/0402148.

[42] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi, “Compiler design for distributed quantum computing,” *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–20, 2021. DOI: 10.1109/tqe.2021.3053921.

[43] *Qiskit: An open-source framework for quantum computing*, 2022. DOI: 10.5281/zenodo.7080365.

[44] *Quilc: The optimizing quil compiler*, Accessed 2023-04-16. [Online]. Available: https://github.com/quil-lang/quilc.

[45] *Forest quil documentation*, Accessed 2023-04-16. [Online]. Available: https://pyquil-docs.rigetti.com/en/latest/.

[46] R. S. Smith, M. J. Curtis, and W. J. Zeng, “A practical quantum instruction set architecture,” Aug. 2016. DOI: 10.48550/arXiv.1608.03355.

[47] *Q# compiler*, Accessed 2023-04-16. [Online]. Available: https://github.com/microsoft/qsharp-compiler.

[48] *Azure quantum documentation*, Accessed 2023-04-16. [Online]. Available: https://docs.microsoft.com/en-us/azure/quantum/.

[49] *Projectq framework*, Accessed 2023-04-16. [Online]. Available: https://github.com/ProjectQ-Framework/ProjectQ.

[50] D. S. Steiger, T. Häner, and M. Troyer, "Projectq: An open source software framework for quantum computing," *Quantum 2, 49 (2018)*, Dec. 2016. DOI: 10.22331/q-2018-01-31-49.

[51] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *Quantum Sci. Technol. 3 (2018) 020501*, Apr. 2016. DOI: 10.1088/2058-9565/aaa5cc.

[52] *Scaffcc compiler framework*, Accessed 2023-04-16. [Online]. Available: https://github.com/epiqc/ScaffCC.

[53] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: Scalable compilation and analysis of quantum programs," *Parallel Comput. 45, C (June 2015), 2-17*, Jul. 2015. DOI: 10.1016/j.parco.2014.12.001.

[54] *Cirq compiler*, Accessed 2023-04-16. [Online]. Available: https://github.com/quantumlib/cirq.

[55] *Cirq documentation*, Accessed 2023-04-16. [Online]. Available: https://quantumai.google/cirq.

[56] *Pennylane compiler*, Accessed 2023-04-16. [Online]. Available: https://github.com/PennyLaneAI/PennyLane.

[57] *Pennylane documentation*, Accessed 2023-04-16. [Online]. Available: https://docs.pennylane.ai/en/stable/.

[58] *Openql compiler*, Accessed 2023-04-16. [Online]. Available: https://github.com/QuTech-Delft/OpenQL.

[59] *Openql documentation*, Accessed 2023-04-16. [Online]. Available: https://openql.readthedocs.io/en/latest/.

[60] F. W. M. de Ronde, "Quantum simulator for electronical control of diamond spin qubits," Master's thesis, TU Delft, Aug. 2022. [Online]. Available: http://resolver.tudelft.nl/uuid:6e56a4c9-a248-4fc0-8d63-425688c06b4d.

[61] C. Gidney and M. Ekerå, "How to factor 2048-bit rsa integers in 8 hours using 20 million noisy qubits," *Quantum 5, 433 (2021)*, May 2021. DOI: 10.22331/q-2021-04-15-433.

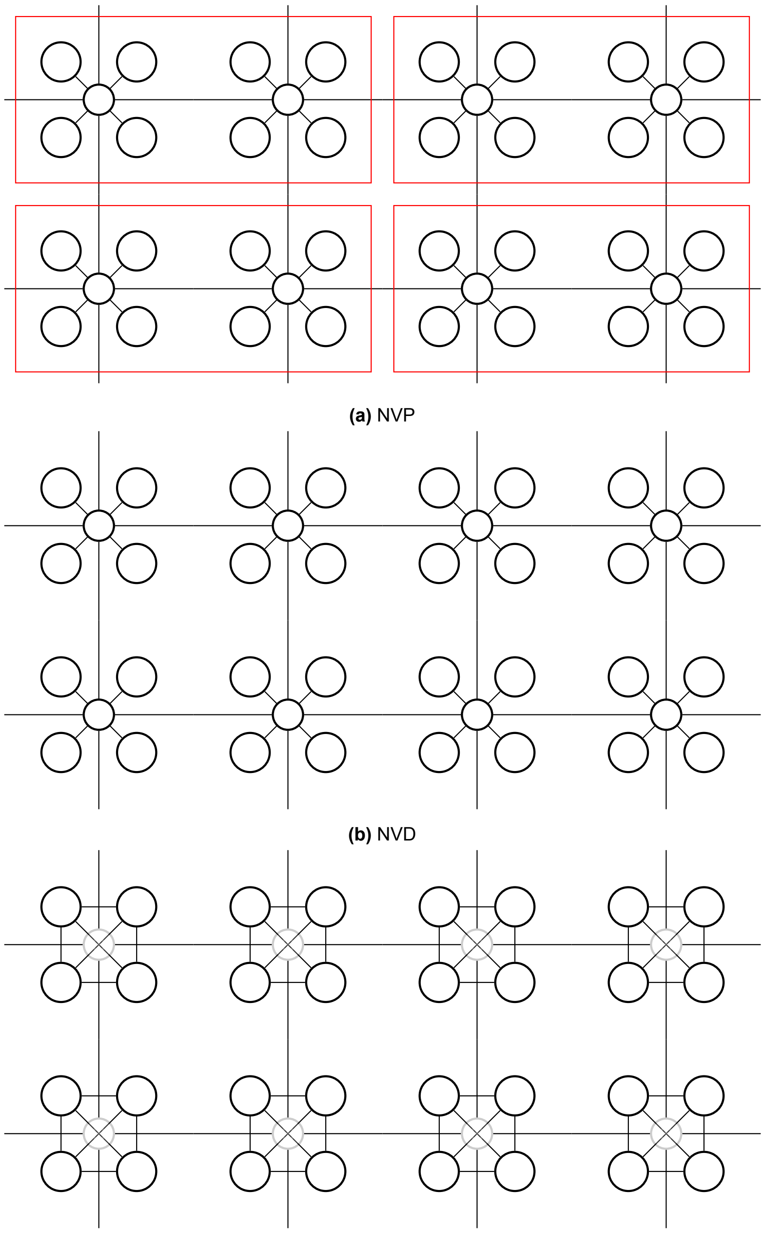
[62] J. Roffe, "Quantum error correction: An introductory guide," *Contemporary Physics*, vol. 60, no. 3, pp. 226–245, Jul. 2019. DOI: 10.1080/00107514.2019.1667078.

Glossary

Auxiliary qubit p9	A qubit used to perform indirect operations such as measurements, as opposed to a data qubit. Also known as an *ancilla* qubit.
Conditional gate p6	A gate with classical control, likewise for a *conditional rotation*.
Controlled gate p7	A gate with qubit control, for example a CNOT.
Ebit p16	A pair of maximally entangled qubits.
Execution platform	Something which can execute a quantum algorithm, be it a simulator or real quantum computer. In this thesis a platform is assumed to be compatible with quantum circuits.
Gate p5	A reversible, unitary operation on any number of qubits. Depending on the context it can also be specifically referred to as a *quantum gate*.
Instruction p13, p19, p37	What a compiler operates on. An instruction can be on the hardware level such as what is executed by control electronics, including quantum operations but also for example classical addition and comparison or waiting for a specific time. An instruction can also be on on a higher level, representing a sequence of lower-level quantum and classical instructions.
Kernel	A sequence of instructions. In this thesis a kernel is assumed to implement a quantum circuit and surrounding necessary classical computations.
Mapping p15	Transforming qubits and quantum operations into closer-to-hardware equivalents. In this thesis mapping specifically refers to transforming logical qubits and operations into their physical counterparts. Outside of this thesis 'mapping' often refers to just place & route.
Place & route	Initial placement and routing of qubits based on the required interaction and available topology.
Quantum algorithm	An algorithm that uses any type of quantum computation.
Quantum circuit p8	A specific type of quantum algorithm that consists of a sequence of quantum operations.
Quantum computer	A device capable of performing quantum computations. In general this can be any quantum algorithm, but in this thesis it is used for quantum circuits specifically.
Quantum operation p8	Any action that can be performed on qubits, including gates, non-unitaries, and platform-specific operations.
Rotation p6	A single-qubit gate, for example an 180° X or 90° Y rotation.

A Figures

A.1. NVx Topology



(a) NVP

(b) NVD

(c) NVC

Figure A.1: Qubit organization of a 4x2 physical topology for the different compiler approaches.

A.2. NVx Mapping Example

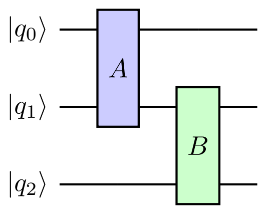


Figure A.2: A quantum circuit with 2 generic 2-qubit gates.

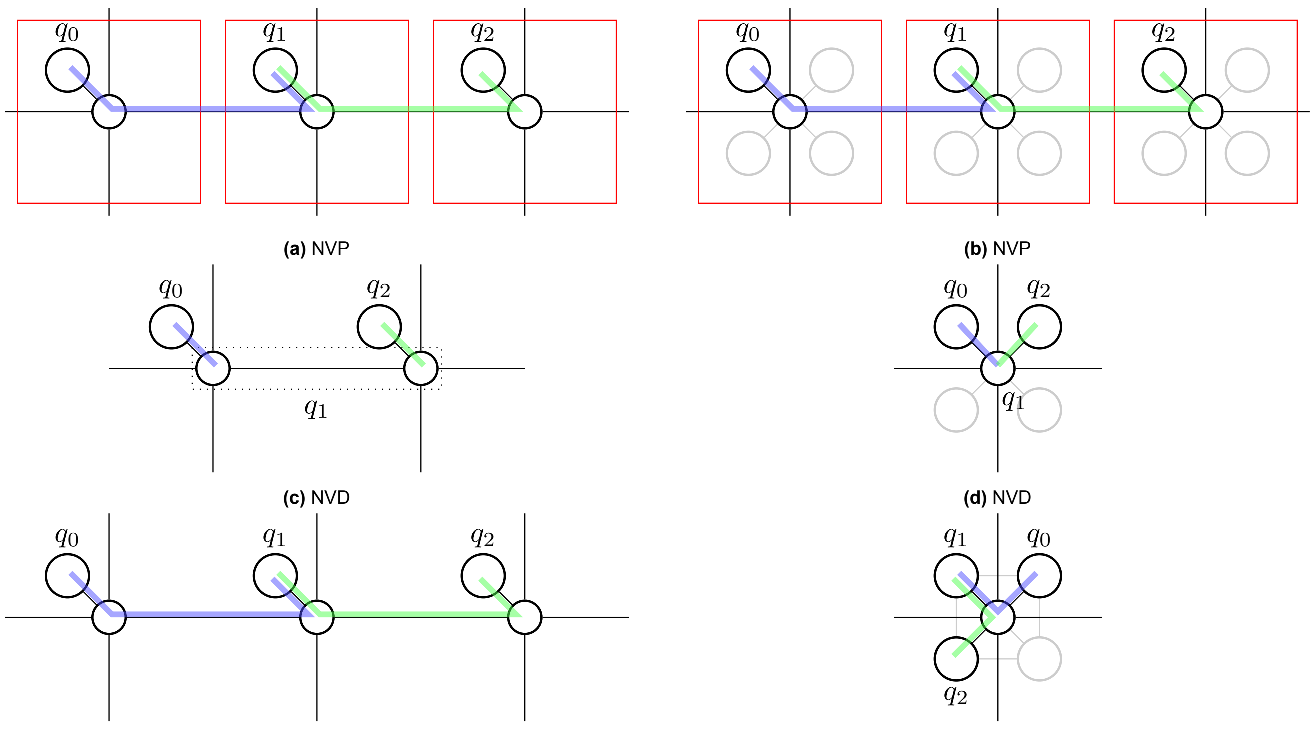


Figure A.3: NVx-1 codes mapped to an NVx-1 (left) and NVx-4 (right) physical topology.

B ISA Reference

This chapter describes the proposed quantum instruction set architecture (QISA) for a quantum computer based on NV centers. Included are the top-level QISA itself, microcode stored on the local controller, the micro-ISA, and the local controller registers.

B.1. Global Instruction Set

The following table contains all instructions supported by the global controller. Following that a detailed description of each instruction is provided.

Classical		
`ldi`	`dest, value`	Load immediate
`addi`	`dest, source, value`	Add immediate
`st`	`source, address`	Store in memory
`ld`	`dest, address`	Load from memory
`br`	`(condition), label`	Branch conditionally
`jump`	`label`	Jump unconditionally
Communication		
`send`	`NV, source`	Send data to local controller
`receive`	`NV, dest`	Receive data from local controller
Quantum		
`set`	`NV, register, value`	Set local register
`qgateE`	`NV, phase, angle`	Apply electron gate
`qgateZE`	`NV, angle`	Apply Z-rotation on electron
`qgateZC`	`NV, C13, angle`	Apply Z-rotation on C13
`qgateCC`	`NV, C13, phase, angle`	Apply conditional C13 gate
`qgateUC`	`NV, C13, phase, angle, preserve`	Apply unconditional C13 gate
`qgateDIR`	`NV, C13, phase, angle, dir`	Rotate C13 in conditional direction
`initialize`	`NV`	Initialize NV center
`measureE`	`NV`	Measure electron state
`swapEC`	`NV, C13`	Half-swap of electron to C13 qubit
`swapCE`	`NV, C13, basis`	Half-swap of C13 to electron qubit
`entangle`	`NV, direction`	Entangle with other NV center
`detectCarbon`	`NV, sweepStart, sweepStep, sweepStop, measMax`	Perform carbon detection
`magbias`	`NV, sweepStart, sweepStep, sweepStop`	Perform magnetic biasing check
`rabicheck`	`NV, sweepStart, sweepStep, sweepStop, measMax`	Perform Rabi check
`crc`	`NV`	Perform charge resonance check

ldi

Format: `ldi dest, value`

Load immediate value in register.

addi

Format: `addi dest, source, value`

Add immediate value to register.

st

Format: `st source, address`

Store value from register in memory.

ld

Format: `ld dest, address`

Load value from memory in register.

br

Format: `br (condition), label`

Branch to label or position based on condition.

jump

Format: `jump label`

Jump unconditionally to label or position.

send

Format: `send NV, source`

Send data from register `source` to local controller `NV`.

receive

Format: `receive NV, dest`

Receive data from local controller `NV` and store in register `dest`. If the destination register is omitted the received value is discarded (like r0 in MIPS; something does still have to be received).

TBD: results are read per NV using a FIFO on global controller side, in the same order as the corresponding `send` instruction. It is up to the global controller/microprograms to balance sends and receives.

set

Format: `set NV, register, value`

Send command `set` and its arguments to the specified local controller. Receive confirmation and return data.

Set the value of a local register. Intended use is setting local calibration values that are calculated on the global controller/host.

qgateE

Format: `qgateE NV, phase, angle`

Send command `qgateE` and its arguments to the specified local controller. Receive confirmation and return data.

Apply a gate on the e^- qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

qgateZE

Format: `qgateZE NV, angle`

Send command `qgateZE` and its arguments to the specified local controller. Receive confirmation and return data.

Apply a Z-rotation to the specified e^- qubit. `angle` specifies the angle of rotation.

qgateZC

Format: `qgateZC NV, C13, angle`

Send command `qgateZC` and its arguments to the specified local controller. Receive confirmation and return data.

Apply a Z-rotation to the specified ^{13}C qubit. `angle` specifies the angle of rotation.

qgateCC

Format: `qgateCC NV, C13, phase, angle`

Send command `qgateCC` and its arguments to the specified local controller. Receive confirmation and return data.

Apply a conditional gate on the specified ^{13}C qubit based on the e^- qubit state. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

qgateUC

Format: `qgateUC NV, C13, phase, angle, preserve`

Send command `qgateUC` and its arguments to the specified local controller. Receive confirmation and return data.

Apply an unconditional gate on the specified ^{13}C qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

If `preserve` is true, the e^{-} qubit state is preserved.

qgateDIR

Format: `qgateDIR NV, C13, phase, angle, dir`

Send command `qgateDIR` and its arguments to the specified local controller. Receive confirmation and return data.

Apply a gate on the specified ^{13}C qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis. Based on `dir` the qubit is rotated in the positive or negative direction.

initialize

Format: `initialize NV`

Send command `initialize` and its arguments to the specified local controller. Receive confirmation and return data.

Initialize the e^{-} qubit in the $|0\rangle$-state.

measureE

Format: `measureE NV`

Send command `measureE` and its arguments to the specified local controller. Receive confirmation and return data.

Measure the e^{-} qubit state and send the result to the global controller.

swapEC

Format: `swapEC NV, C13`

Send command `swapEC` and its arguments to the specified local controller. Receive confirmation and return data.

Swap the quantum state of the e^{-} qubit to the specified ^{13}C qubit. After the operation the e^{-} qubit is left in the maximally mixed state.

swapCE

Format: `swapCE NV, C13, basis`

Send command `swapCE` and its arguments to the specified local controller. Receive confirmation and return data.

Swap the state of the specified ^{13}C qubit to the e^{-} qubit in the specified basis. After the operation the ^{13}C qubit is left in an unknown non-entangled state.

entangle

Format: `entangle NV, direction`

Send command `entangle` and its arguments to the specified local controller. Receive confirmation and return data.

Perform an optical entanglement routine between the e^{-} qubit of this NV center with that of another. The `direction` argument determines which neighboring NV center is targeted. The global controller is expected to send the same instruction with opposite `direction` to the other NV center.

detectCarbon

Format: `detectCarbon NV, sweepStart, sweepStep, sweepStop, measMax`

Send command `detectCarbon` and its arguments to the specified local controller. Receive confirmation and return data.

Perform measurements to detect ^{13}C qubit frequencies.

magbias

Format: `magbias NV, sweepStart, sweepStep, sweepStop`

Send command `magbias` and its arguments to the specified local controller. Receive confirmation and return data.

Perform measurements to calibrate the magnetic biasing.

rabicheck

Format: `rabicheck NV, sweepStart, sweepStep, sweepStop, measMax`

Send command `rabicheck` and its arguments to the specified local controller. Receive confirmation and return data.

Perform a Rabi oscillation check to calibrate the duration of MW pulses.

crc

Format: `crc NV`

Send command `crc` and its arguments to the specified local controller. Receive confirmation and return data.

Perform a charge resonance check.

B.2. Local Microcode

The following microcode provides higher-level functionality to the local controller. These programs are present in a hardware control store and get executed according to commands received from the global controller.

`set`	`NV, register, value`	Set local register
`qgateE`	`NV, phase, angle`	Apply electron gate
`qgateZE`	`NV, angle`	Apply Z-rotation on electron
`qgateZC`	`NV, C13, angle`	Apply Z-rotation on C13
`qgateCC`	`NV, C13, phase, angle`	Apply conditional C13 gate
`qgateUC`	`NV, C13, phase, angle, preserve`	Apply unconditional C13 gate
`qgateDIR`	`NV, C13, phase, angle, dir`	Rotate C13 in conditional direction
`initialize`	`NV`	Initialize NV center
`measureE`	`NV`	Measure electron state
`swapEC`	`NV, C13`	Half-swap of electron to C13 qubit
`swapCE`	`NV, C13, basis`	Half-swap of C13 to electron qubit
`entangle`	`NV, direction`	Entangle with other NV center
`detectCarbon`	`NV, sweepStart, sweepStep, sweepStop, measMax`	Perform carbon detection
`magbias`	`NV, sweepStart, sweepStep, sweepStop`	Perform magnetic biasing check
`rabicheck`	`NV, sweepStart, sweepStep, sweepStop, measMax`	Perform Rabi check
`crc`	`NV`	Perform charge resonance check

set

Format: `set NV, register, value`

Set the value of a local register. Intended use is setting local calibration values that are calculated on the global controller/host.

Requires indexing a register with a variable, ie like memory. This might require a dedicated local instruction.

```
.alias register = r1
.alias value    = r2

receive $register, $value

mov [$register], $value
```

qgateE

Format: `qgateE NV, phase, angle`

Apply a gate on the e^- qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

Microwave pulse parameters are calculated based on local calibration.

Rotation time is calculated with the `angle` and `erabifreq`.

```
.alias phase = r1
.alias angle = r2
.alias time  = r2

receive $phase, $angle

# Calculate rotation time, to be replaced with mult by inverse
div $time, $angle, $erabifreq

mw $time, $phase
```

qgateZE

Format: `qgateZE NV, angle`

Apply a Z-rotation to the specified e$^-$ qubit. `angle` specifies the angle of rotation.

No physical gate is applied, instead only the phase tracking oscillator is shifted by the specified angle.

```
.alias angle = r1

receive $angle

% Convert to numerical phase?

add $ephase, $ephase, $angle
```

qgateZC

Format: `qgateZC NV, C13, angle`

Apply a Z-rotation to the specified ^{13}C qubit. `angle` specifies the angle of rotation.

No physical gate is applied, instead only the phase tracking oscillator is shifted by the specified angle.

```
.alias C13   = r1
.alias angle = r2

receive $C13, $angle

% Convert to numerical phase?

add $cphase[$C13], $cphase[$C13], $angle
```

qgateCC

Format: `qgateCC NV, C13, phase, angle`

Apply a conditional gate on the specified ^{13}C qubit based on the e^{-} qubit state. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

RF pulse parameters are calculated based on local calibration.

```
.alias C13   = r1
.alias phase = r2
.alias angle = r3
.alias ctime = r3
.alias etime = r4
.alias phaseppi = r5

receive $C13, $phase, $angle

# Calculate carbon rotation time, to be replaced with mult by inverse
div $ctime, $angle, $crabifreq[$C13]

# Calculate electron rotation time, to be replaced with mult by inverse
div $etime, PI, $erabifreq

add $phaseppi, $phase, PI

rf $C13, $ctime, $phase
mw $etime, 0
rf $C13, $ctime, $phase
mw $etime, 0

rf $C13, $ctime, $phase
mw $etime, 0
rf $C13, $ctime, $phaseppi
mw $etime, 0
```

qgateUC

Format: `qgateUC NV, C13, phase, angle, preserve`

Apply an unconditional gate on the specified ^{13}C qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis.

If `preserve` is true, the e$^-$ qubit state is preserved.

Microwave parameters are calculated based on local calibration. The optional e$^-$ qubit state preservation is implemented by use of flips on the e$^-$ qubit.

```
.alias C13   = r1
.alias phase = r2
.alias angle = r3
.alias ctime = r3
.alias presv = r4
.alias etime = r5

receive $C13, $phase, $angle, $presv

# Calculate carbon rotation time, to be replaced with mult by inverse
div $ctime, $angle, $crabifreq[$C13]

# Calculate electron rotation time, to be replaced with mult by inverse
div $etime, PI, $erabifreq

br $presv == 1, preserve

# QgateUC, not preserved
initialize
mw $etime, 0
rf $ctime, $carbonfreq[C13], $phase

jump end

# QgateUC, preserved
preserve:
rf $C13, $ctime, $phase
mw $etime, 0
rf $C13, $ctime, $phase
mw $etime, 0

end:
```

qgateDIR

Format: `qgateDIR NV, C13, phase, angle, dir`

Apply a gate on the specified ^{13}C qubit. `phase` specifies the axis of rotation in the XY plane. `angle` specifies the angle of rotation around that axis. Based on `dir` the qubit is rotated in the positive or negative direction.

Microwave parameters are calculated based on local calibration.

```
.alias C13    = r1
.alias phase1 = r2
.alias angle  = r3
.alias ctime  = r3
.alias dir    = r4
.alias etime  = r5
.alias phase2 = r6

receive $C13, $phase1, $angle, $dir

# Calculate carbon rotation time, to be replaced with mult by inverse
div $ctime, $angle, $crabifreq[$C13]

# Calculate electron rotation time, to be replaced with mult by inverse
div $etime, PI, $erabifreq

# Rotate other direction with phase2
add $phase2, $phase1, PI

# Swap phase1 and phase2 if dir == 1
br $dir == 0, noswap
mov $r7, $phase1
mov $phase1, $phase2
mov $phase2, $r7
noswap:

rf $C13, $ctime, $phase1
mw $etime, 0
rf $C13, $ctime, $phase2
mw $etime, 0
```

initialize

Format: `initialize NV`

Initialize the e$^-$ qubit in the $|0\rangle$-state.

```
loop:
ldi $phcount, 0
switchOn INIT
wait (time)
switchOff
br $phcount > 0, loop
(send done)
```

measureE

Format: `measureE NV`

Measure the e$^-$ qubit state and send the result to the global controller.

```
ldi $phcount, 0
switchOn RO
wait (time)
switchOff
compare $phcount > $phthres
(send result)
```

swapEC

Format: `swapEC NV, C13`

Swap the quantum state of the e$^-$ qubit to the specified ^{13}C qubit. After the operation the e$^-$ qubit is left in the maximally mixed state.

```
.alias C13   = r1
.alias etime = r2
.alias ctime = r3

receive $C13

# To be replaced with mult by inverse
div $ctime, PI, $crabifreq
div $etime, PI, $erabifreq

mult $etime2, $etime, 0.5

mw $etime2, PI/2
rf $C13, $ctime, 0
mw $etime, 0
rf $C13, $ctime, PI
mw $etime, 0
mw $etime2, 0
rf $C13, $ctime, PI*3/2
mw $etime, 0
rf $C13, $ctime, PI/2
mw $etime, 0
```

swapCE

Format: `swapCE NV, C13, basis`

Swap the state of the specified ^{13}C qubit to the e^{-} qubit in the specified basis. After the operation the ^{13}C qubit is left in an unknown non-entangled state.

The different basis swaps are implemented separately.

```
.alias C13   = r1
.alias etime = r2
.alias ctime = r3

receive $C13, basis

# To be replace by mult with inverse
div $etime, PI, $erabifreq
div $ctime, PI, $crabifreq

mult $etime2, $etime, 0.5
mult $ctime2, $ctime, 0.5

branch (basis = y), swapy
branch (basis = z), swapz

# Swap in X basis
swapx:
mw $etime2, PI/2
rf $C13, $ctime2, PI
mw $etime, 0
rf $C13, $ctime2, 0
mw $etime, 0
mw $etime2, 0

jump end

# Swap in Y basis
swapy:
mw $etime2, PI/2
rf $C13, $ctime2, PI*3/2
mw $etime, 0
rf $C13, $ctime2, PI/2
mw $etime, 0
mw $etime2, 0

jump end

# Swap in Z basis
swapz:
rf $C13, $ctime2, 0
mw $etime, 0
rf $C13, $ctime2, PI
mw $etime, 0
mw $etime2, PI/2
rf $C13, $ctime2, PI*3/2
mw $etime, 0
rf $C13, $ctime2, PI/2
mw $etime, 0
mw $etime2, 0

end:
```

entangle

Format: `entangle NV, direction`

Perform an optical entanglement routine between the e$^-$ qubit of this NV center with that of another. The `direction` argument determines which neighboring NV center is targeted. The global controller is expected to send the same instruction with opposite `direction` to the other NV center.

```
(sync with other Q)
ldi $phent?  # phent corresponding to other Q
switchOn ENTANGLE, (to other Q)
wait 100
switchOff
wait 50
send $phent?
(send entangleReg value)
```

detectCarbon

Format: `detectCarbon NV, sweepStart, sweepStep, sweepStop, measMax`

Perform measurements to detect ^{13}C qubit frequencies.

```
.alias sweep      = r1
.alias sweepStep  = r2
.alias sweepStop  = r3
.alias measMax    = r4
.alias phtotal    = r5
.alias meascount  = r6

receive $sweep, $sweepStep, $sweepStop, $measMax

# To be replace by mult with inverse
div $etime, PI, $erabifreq
div $ctime, PI, $crabifreq

mult $etime2, $etime, 0.5

ldi $meascount, 0
loop:
ldi $phcount, 0
switchOn INIT
wait (time)
switchOff
br $phcount > $phthres, loop

mov $sweep, $cresfreq[0]

mw $etime2, PI/2
rf 0, $ctime, 0
mw $etime, 0
rf 0, $ctime, PI
mw $etime, 0
mw $etime2, PI*3/2

ldi $phcount, 0
switchOn RO
wait (time)
switchOff
add $phtotal, $phtotal, $phcount
add $meascount, $meascount, 1
br $meascount < $measMax, loop
send $phtotal
send $sweep          # should not be necessary since sweep is known in global

add $sweep, $sweep, $sweepStep
ldi $meascount, 0
br $sweep < $sweepStop, loop
```

magbias

Format: `magbias NV, sweepStart, sweepStep, sweepStop`

Perform measurements to calibrate the magnetic biasing.

```
.alias sweep     = r1
.alias sweepStep  = r2
.alias sweepStop  = r3

receive $sweep, $sweepStep, $sweepStop

loop:
ldi $phcount, 0
switchOn RO
mov $eresfreq, $sweep
mw (time), 0
switchOff
send $phcount
send $sweep          # should not be necessary since sweep is known in global
add $sweep, $sweep, $sweepStep
br $sweep < $sweepStop, loop
```

rabicheck

Format: `rabicheck NV, sweepStart, sweepStep, sweepStop, measMax`

Perform a Rabi oscillation check to calibrate the duration of MW pulses.

```
.alias count      = r1
.alias measMax    = r2
.alias sweep      = r3
.alias sweepStep  = r4
.alias sweepStop  = r5
.alias measureSum = r6

receive $sweep, $sweepStep, $sweepStop, $measMax

ldi $count, 0
ldi $measureSum, 0

loop:
# initialize
initloop:
ldi $phcount, 0
switchOn INIT
wait (time)
switchOff
br $phcount > 0, initloop

mw $sweep, 0

# measure
ldi $phcount, 0
switchOn RO
wait (time)
switchOff
add $measureSum, $measureSum, $phcount

send $measureSum
send $sweep          # should not be necessary since sweep is known in global

addi $count, 1
br $count < $countMax, loop
addi $sweep, $sweepStep
br $sweep < $sweepStop, loop
```

crc

Format: `crc NV`

Perform a charge resonance check.

```
loop:
ldi $phcount, 0
switchOn RO, INIT
wait (time)
switchOff
br $phcount > $phthres, done
switchOn PUMP
wait (time)
switchOff
jump loop
done:
(send done)
```

B.3. Local Microinstruction Set

The following table contains all microinstructions supported by the local controller. Following that a detailed description of each instruction is provided.

Classical		
`ldi`	`dest, value`	Load immediate
`compare`	`(condition)`	Compare values
`br`	`(condition), label`	Branch conditionally
`jump`	`label`	Jump unconditionally
`wait`	`n`	Wait
Communication		
`sync`	`unknown`	Synchronize local controllers
`send`	`source`	Send data to global
`receive`	`dest`	Receive data from global
Quantum		
`mw`	`duration, phase`	Apply a microwave pulse
`rf`	`c13, duration, phase`	Apply a radio-frequency pulse
`switchOn`	`sw...`	Enable optical switches
`switchOff`	`sw...`	Disable optical switches

ldi

Format: `ldi dest, value`

Load immediate value in register.

compare

Format: `compare (condition)`

Compare register contents.

TBD: what comparisons, use of immediates

br

Format: `br (condition), label`

Branch to label based on condition.

jump

Format: `jump label`

Jump unconditionally to label.

wait

Format: `wait n`

Wait for `n` clock cycles.

sync

Format: `sync unknown`

Synchronize this local controller with another one. Necessary for entanglement.

TBD: how will this be implemented in actual hardware?

send

Format: `send source`

Send data from register `source` to the global controller.

receive

Format: `receive dest`

Receive data from the global controller and store in register `dest`.

mw

Format: `mw duration, phase`

Apply a microwave pulse to the NV center to rotate the e^- qubit.

Set registers `mwdur`, `mwphase`. Implicit use of registers `elarmorfreq`, `mwenv`, `mwampl`.

rf

Format: `rf c13, duration, phase`

Apply a radio-frequency pulse to the NV center to apply a 2-qubit gate on the e^- qubit and specified ^{13}C qubit.

Set registers `rfindex`, `rfdur`, `rfphase`. Implicit use of registers `rfenv`, `rfampl`.

switchOn

Format: `switchOn sw...`

Enable the specified optical switches. Allowed values are:

RO	Readout laser
INIT	Initialize laser
PUMP	Charge pump laser
ENTANGLE	Variable-frequency laser
(directions)	Optical links to other (neighboring) NV centers

Set `sw`.

switchOff

Format: `switchOff sw...`

Switch off the specified optical switches. Allowed values are the same as for `switchOn`, or none to disable all switches.

Set `sw`.

B.4. Local Registers

The following table lists all registers and register-mapped IO ports in the local controller. Following that a detailed description of each register/port is provided.

General Purpose	
`r0`	Constant 0 register
`r1-15`	General purpose registers
Calibration	
`erabifreq`	Electron Rabi frequency
`crabifreq[16]`	Carbon Rabi frequency
`eresfreq`	Electron resonance frequency
`cresfreq[16]`	Carbon resonance frequencies
`efield`	Electric field voltage
`bfield`	Magnetic field strength
`voa[4]`	Variable optical attenuator tuning
`mwenv`	Microwave envelope
`mwampl`	Microwave pulse amplitude
`rfenv`	RF pulse envelope
`rfampl`	RF pulse amplitude
`phcurrent`	SNSPD bias current
`phthres`	Photon threshold
Control	
`ephaseinc`	Electron phase increment
`cphaseinc[16]`	Carbon phase increments
`mwdur`	Microwave pulse duration
`mwphase`	Microwave pulse phase
`rfindex`	RF ^{13}C qubit index
`rfdur`	RF pulse duration
`rfphase`	RF pulse phase
`sw`	Optical switches
Readout	
`phcount`	Photon count

r0

Reading from this register always returns 0. Writing to this register has no effect.

r1-15

Registers for general purposes.

erabifreq

Rabi frequency of the e$^-$ qubit. This is the frequency at which the qubit oscillates between the $|0\rangle$ and $|1\rangle$ states.

This parameter is calibrated by the `rabicheck` instruction.

crabifreq[16]

Rabi frequency of the ^{13}C qubit. This is the frequency at which the qubit oscillates between the $|0\rangle$ and $|1\rangle$ states.

For this parameter no calibration instruction is implemented yet.

eresfreq

Resonance (ie. Larmor) frequency of the e$^-$ qubit.

This parameter is calibrated by the `magbias` instruction.

cresfreq[16]

Resonance (ie. spin precession) frequencies for the ^{13}C qubits.

This parameter is calibrated by the `detectCarbon` instruction.

efield

Constant voltage that determines the electric field strength. This electric field separates the energy states of the e$^-$ qubit.

For this parameter no calibration instruction is implemented yet.

bfield

Controls the local magnetic field strength.

For this parameter no calibration instruction is implemented yet.

voa[4]

Tuning of optical power allowed to pass through the VOA of different laser sources.

For this parameter no calibration instruction is implemented yet.

mwenv

Envelope (ie. shape) of the microwave generator output.

For this parameter no calibration instruction is implemented yet.

mwampl

Amplitude of the microwave generator output.

For this parameter no calibration instruction is implemented yet.

rfenv

Envelope (ie. shape) of the radio-frequency generator output.

For this parameter no calibration instruction is implemented yet.

rfampl

Amplitude of the radio-frequency generator output.

For this parameter no calibration instruction is implemented yet.

phcurrent

Bias current for the superconducting nanowire single-photon detecter (SNSPD).

For this parameter no calibration instruction is implemented yet.

phthres

Threshold for the photon count to determine whether a $|0\rangle$ or $|1\rangle$ is measured.

For this parameter no calibration instruction is implemented yet.

ephaseinc

cphaseinc[16]

mwdur

Pulse duration of the microwave generator output.

mwphase

Phase of the microwave generator output.

rfindex

Index to select a specific ^{13}C qubit for the radio-frequency generator.

rfdur

Pulse duration of the radio-frequency generator output.

rfphase

Phase of the radio-frequency generator output.

sw

State of the optical switches. Each bit represents one switch.

phcount

Photon counter coupled to the single-photon detector.

C Decomposition Reference

This chapter describes the decomposition rules proposed for a compiler targeting a quantum computer based on NV centers. Included are the logical to physical decomposition for an NVP1 system, NV center-specific physical decomposition rules, and finally the translation from generic quantum instructions to architecture-specific ones.

C.1. NVP1 Decomposition Rules

The following decomposition rules transform operations on logical qubits to operations on the underlying physical qubits, following an 'NVP1' logical qubit layout. The `phys(n,m)` syntax refers to physical qubit *m* of the logical qubit referred to by the *n*th instruction operand.

x

Format: `x q`

Decomposition

```
x phys(0,1)
```

x90

Format: `x90 q`

Decomposition

```
x90 phys(0,1)
```

mx90

Format: `mx90 q`

Decomposition

```
mx90 phys(0,1)
```

x180

Format: `x180 q`

Decomposition

```
x180 phys(0,1)
```

y

Format: `y q`

Decomposition

```
y phys(0,1)
```

y90

Format: `y90 q`

Decomposition

```
y90 phys(0,1)
```

my90

Format: `my90 q`

Decomposition

```
my90 phys(0,1)
```

y180

Format: `y180 q`

Decomposition

```
y180 phys(0,1)
```

z

Format: `z q`

Decomposition

```
z phys(0,1)
```

s

Format: `s q`

Decomposition

```
s phys(0,1)
```

sdag

Format: `sdag q`

Decomposition

```
sdag phys(0,1)
```

t

Format: `t q`

Decomposition

```
t phys(0,1)
```

tdag

Format: `tdag q`

Decomposition

```
tdag phys(0,1)
```

h

Format: `h q`

Decomposition

```
h phys(0,1)
```

cnot

Format: `cnot control, target`

Based on martinez2019

Decomposition: symmetry=all

```
nventangle phys(0,0), phys(1,0)

cnot phys(0,1), phys(0,0)
measure phys(0,0)
if_x phys(1,0), phys(0,0)

cnot phys(1,0), phys(1,1)

h phys(1,0)
measure phys(1,0)
if_z phys(0,1), phys(1,0)
```

prep_z

Format: `prep_z q`

Decomposition

```
prep_z phys(0,1)
```

measure

Format: `measure q`

Decomposition

```
measure phys(0,1)
```

move

Format: `move q0, q1`

Decomposition: symmetry=all

```
nventangle phys(0,0), phys(1,0)

cnot phys(0,1), phys(0,0)
measure phys(0,0)
if_x phys(1,0), phys(0,0)

cnot phys(1,0), phys(1,1)
cnot phys(1,1), phys(1,0)

h phys(1,0)
measure phys(1,0)
if_z phys(0,1), phys(1,0)
```

swap

Format: `swap q0, q1`

Decomposition: symmetry=all

```
nventangle phys(0,0), phys(1,0)

cnot phys(0,1), phys(0,0)
measure phys(0,0)
if_x phys(1,0), phys(0,0)

cnot phys(1,0), phys(1,1)
cnot phys(1,1), phys(1,0)
cnot phys(1,0), phys(1,1)

h phys(1,0)
measure phys(1,0)
if_z phys(0,1), phys(1,0)
```

C.2. Physical Decomposition Rules

The following decomposition rules allow for higher-level instructions in prior compiler passes such as the NVP decomposition. They are decomposed by the compiler into their corresponding sequences of instructions.

h

Format: `h q`

Hadamard gate into y90 x180

Decomposition: qtype=[e]

```
y90 op(0)
x op(0)
```

Decomposition: qtype=[c]

```
y90 op(0)
x op(0)
```

cnot

Format: `cnot control, target`

Reverse CNOT-direction by surrounding it with Hadamard gates.

Decomposition: qtype=[c, e]

```
h op(0)
h op(1)
cnot op(1), op(0)
h op(0)
h op(1)
```

C.3. ISA Decomposition Rules

The following decomposition rules convert generic quantum instructions and higher-level instructions into the ISA as required by the hardware and simulator platforms. They are decomposed by the compiler into their corresponding sequences of instructions.

A conditional version of each decomposition is also generated. This renames an instruction from *instr* to `if_`*instr*, adds an extra argument which indicates the measured qubit, and wraps the decomposition in a branch that tests for this qubit. An example is provided for `x` and `if_x` and other conditional decompositions are constructed similarly.

x

Format: `x q`

Decomposition: qtype=[e]

```
qgatee nv(0), 0.0, pi
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), 0.0, pi, 1
```

if_x

Format: `if_x q_test, q`

Decomposition: qtype=[e, e]

```
br MeasureResultRegNVnode[nvi(1)] > 0, line(x)
qgatee nv(0), 0.0, pi
line(x):
```

Decomposition: qtype=[c, e]

```
br MeasureResultRegNVnode[nvi(1)] > 0, line(x)
qgateuc nv(0), ci(0), 0.0, pi, 1
line(x):
```

x90

Format: `x90 q`

Decomposition: qtype=[e]

```
qgatee nv(0), 0.0, pi/2
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), 0.0, pi/2, 1
```

mx90

Format: `mx90 q`

Decomposition: qtype=[e]

```
qgatee nv(0), 0.0, -pi/2
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), 0.0, -pi/2, 1
```

x180

Format: `x180 q`

Decomposition: qtype=[e]

```
qgatee nv(0), 0.0, pi
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), 0.0, pi, 1
```

rx

Format: `rx q, angle`

Decomposition: qtype=[e]

```
qgateee nv(0), 0.0, op(1)
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), 0.0, op(1), 1
```

y

Format: `y q`

Decomposition: qtype=[e]

```
qgatee nv(0), pi/2, pi
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), pi/2, pi, 1
```

y90

Format: `y90 q`

Decomposition: qtype=[e]

```
qgatee nv(0), pi/2, pi/2
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), pi/2, pi/2, 1
```

my90

Format: `my90 q`

Decomposition: qtype=[e]

```
qgatee nv(0), pi/2, -pi/2
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), pi/2, -pi/2, 1
```

y180

Format: `y180 q`

Decomposition: qtype=[e]

```
qgatee nv(0), pi/2, pi
```

Decomposition: qtype=[c]

```
qgateuc nv(0), ci(0), pi/2, pi, 1
```

z

Format: `z q`

Decomposition: qtype=[e]

```
qgateze nv(0), pi
```

Decomposition: qtype=[c]

```
qgatezc nv(0), ci(0), pi
```

s

Format: `s q`

Decomposition: qtype=[e]

```
qgateze nv(0), pi/2
```

Decomposition: qtype=[c]

```
qgatezc nv(0), ci(0), pi/2
```

sdag

Format: `sdag q`

Decomposition: qtype=[e]

```
qgateze nv(0), -pi/2
```

Decomposition: qtype=[c]

```
qgatezc nv(0), ci(0), -pi/2
```

t

Format: `t q`

Decomposition: qtype=[e]

```
qgateze nv(0), pi/4
```

Decomposition: qtype=[c]

```
qgatezc nv(0), ci(0), pi/4
```

tdag

Format: `tdag q`

Decomposition: qtype=[e]

```
qgateze nv(0), -pi/4
```

Decomposition: qtype=[c]

```
qgatezc nv(0), ci(0), -pi/4
```

cnot

Format: `cnot control, target`

Decomposition: qtype=[e, c]

```
qgatecc nv(0), ci(1), 0.0, pi
qgatezc nv(0), ci(1), pi/2
```

prep_z

Format: `prep_z q`

Decomposition: qtype=[e]

```
initialize nv(0)
```

Decomposition: qtype=[c]

```
initialize nv(0)
swapec nv(0), ci(0)
```

measure

Format: `measure q`

Decomposition: qtype=[e]

```
measuree nv(0)
```

Decomposition: qtype=[c]

```
swapce nv(0), ci(0), z
measuree nv(0)
```

nventangle

Format: `nventangle eq0, eq1`

Decomposition: qtype=[e, e]

```
nventangle nv(0), nv(1)
```

D Circuits

D.1. Circuit: hgate

A single logical qubit is initialized in the $|0\rangle$ state, followed by a Hadamard gate.

Python Input

```python
import openql as ql

ql.initialize()

ql.set_option('output_dir', 'output')
ql.set_option('log_level', 'LOG_DEBUG')

platform = ql.Platform('basic_platform', 'nvhwconf.json', 'nvcconfig.json')

nqubits = 2
program = ql.Program('hgate', platform, nqubits)
kernel = ql.Kernel('basic_kernel', platform, nqubits)

kernel.prepz(0)

kernel.gate('h', 0)

program.add_kernel(kernel)

program.compile()
```

cQASM Intermediate

```
# Generated by OpenQL 0.11.1 for program hgate
version 1.2

pragma @ql.name("hgate")


.basic_kernel
    prep_z q[3]
    y90 q[3]
    x180 q[3]
    skip 1
```

NV Assembly Output

```
initialize q0
swapec q0 0
qgateuc q0 0 1.5707963267949 1.5707963267949 1
qgateuc q0 0 0.0 3.1415926535898 1
```

D.2. Circuit: bpair

A Hadamard and controlled NOT gate are applied to two logical qubits. No initialization is performed prior.

Python Input

```python
import openql as ql

ql.initialize()

ql.set_option('output_dir', 'output')
ql.set_option('log_level', 'LOG_DEBUG')

platform = ql.Platform('basic_platform', 'nvhwconf.json', 'nvcconfig.json')

nqubits = 2
program = ql.Program('bpair', platform, nqubits)
kernel = ql.Kernel('basic_kernel', platform, nqubits)

kernel.gate('h', 0)
kernel.cnot(0, 1)

program.add_kernel(kernel)

program.compile()
```

cQASM Intermediate

```
# Generated by OpenQL 0.11.1 for program bpair
version 1.2

pragma @ql.name("bpair")


.basic_kernel
    nventangle q[2], q[4]
    y90 q[2]
    x180 q[2]
    cnot q[2], q[3]
    y90 q[2]
    x180 q[2]
    measure q[2]
    if_x q[4], q[2]
    cnot q[4], q[5]
    y90 q[4]
    x180 q[4]
    measure q[4]
    y90 q[3]
    x180 q[3]
    if_z q[3], q[4]
    skip 1
```

NV Assembly Output

```
nventangle q0 q1
qgatee q0 1.5707963267949 1.5707963267949
qgatee q0 0.0 3.1415926535898
qgatecc q0 0 0.0 3.1415926535898
qgatezc q0 0 1.5707963267949
qgatee q0 1.5707963267949 1.5707963267949
qgatee q0 0.0 3.1415926535898
measuree q0
br MeasureResultRegNVnode0 > 0 x_11
qgatee q1 0.0 3.1415926535898
x_11:
qgatecc q1 0 0.0 3.1415926535898
qgatezc q1 0 1.5707963267949
qgatee q1 1.5707963267949 1.5707963267949
qgatee q1 0.0 3.1415926535898
measuree q1
qgateuc q0 0 1.5707963267949 1.5707963267949 1
qgateuc q0 0 0.0 3.1415926535898 1
br MeasureResultRegNVnode1 > 0 z_18
qgatezc q0 0 3.1415926535898
z_18:
```

D.3. Circuit: Grover's Algorithm

Python Input

```python
import os
import unittest
from openql import openql as ql
import numpy as np

rootDir = os.path.dirname(os.path.realpath(__file__))
curdir = os.path.dirname(__file__)
output_dir = curdir #os.path.join(curdir, 'test_output')
num_qubits = 9


'''
  grover algorithm implementation
'''

def init(platform):
    init_k = ql.Kernel('init', platform, num_qubits)

    # oracle qubit
    oracle = 4

    # init
    init_k.x(oracle)
    init_k.hadamard(oracle)

    # Hn search space
    for i in range(4):
        init_k.hadamard(i)

    return init_k


def grover(platform):
    grover_k = ql.Kernel('grover', platform, num_qubits)
    # oracle qubit
    oracle = 4
    # name search space
    s0 = 0
    s1 = 1
    s2 = 2
    s3 = 3

    # oracle
    grover_k.x(s0)
    grover_k.x(s1)
    grover_k.toffoli(s0,s1, 5)
    grover_k.toffoli(s1, 5, 6)
    grover_k.toffoli(s2, 6, 7)
    grover_k.toffoli(s3, 7, 8)
    grover_k.cnot(8, oracle)
    grover_k.toffoli(s3,7,8)
    grover_k.toffoli(s2,6,7)
    grover_k.toffoli(s1,5,6)
    grover_k.toffoli(s0,1,5)

    # restore ss
    grover_k.x(s0)
    grover_k.x(s1)

    # diffusion
    for i in range(4):
        grover_k.hadamard(i)

    for i in range(4):
        grover_k.x(i)

    # controlled-phase shift
    grover_k.hadamard(s3)
    grover_k.toffoli(s0,s1,5)
    grover_k.toffoli(s1,5,6)
    grover_k.toffoli(s2,6,7)
    grover_k.cnot(7,s3)
    grover_k.toffoli(s2,6,7)
    grover_k.toffoli(s1,5,6)
    grover_k.toffoli(s0,s1,5)
    grover_k.hadamard(s3)

    # restore search space
    for i in range(4):
        grover_k.x(i)
    for i in range(4):
        grover_k.hadamard(i)

    # grover_k.gate('display',[])

    return grover_k



def grover_algorithm():
    ql.set_option('output_dir', output_dir)
    ql.set_option('optimize', 'no')
    ql.set_option('scheduler', 'ASAP')
    ql.set_option('log_level', 'LOG_DEBUG')
    ql.set_option('write_qasm_files', 'yes')

    platform  = ql.Platform('grover_platform', 'nvhwconf.json', '../nvcconfig.json')
    # num_qubits = 9
    # oracle qubit
    oracle = 4

    # create a grover program
    p = ql.Program('grover', platform, num_qubits, 0)

    # kernels
    init_k   = init(platform)
    grover_k = grover(platform)
    result_k = ql.Kernel('result', platform, num_qubits)

    # result
    result_k.hadamard(oracle)
    result_k.measure(oracle)
    # result_k.gate("display",[])

    result_k.measure(0)
    result_k.measure(1)
    result_k.measure(2)
    result_k.measure(3)
    # result_k.gate("display_binary",[])

    # build the program
    p.add_kernel(init_k)
    p.add_kernel(grover_k)
    p.add_kernel(result_k)
    ql.set_option('decompose_toffoli', 'NC')
    p.compile()

if __name__ == '__main__':
    grover_algorithm()
```