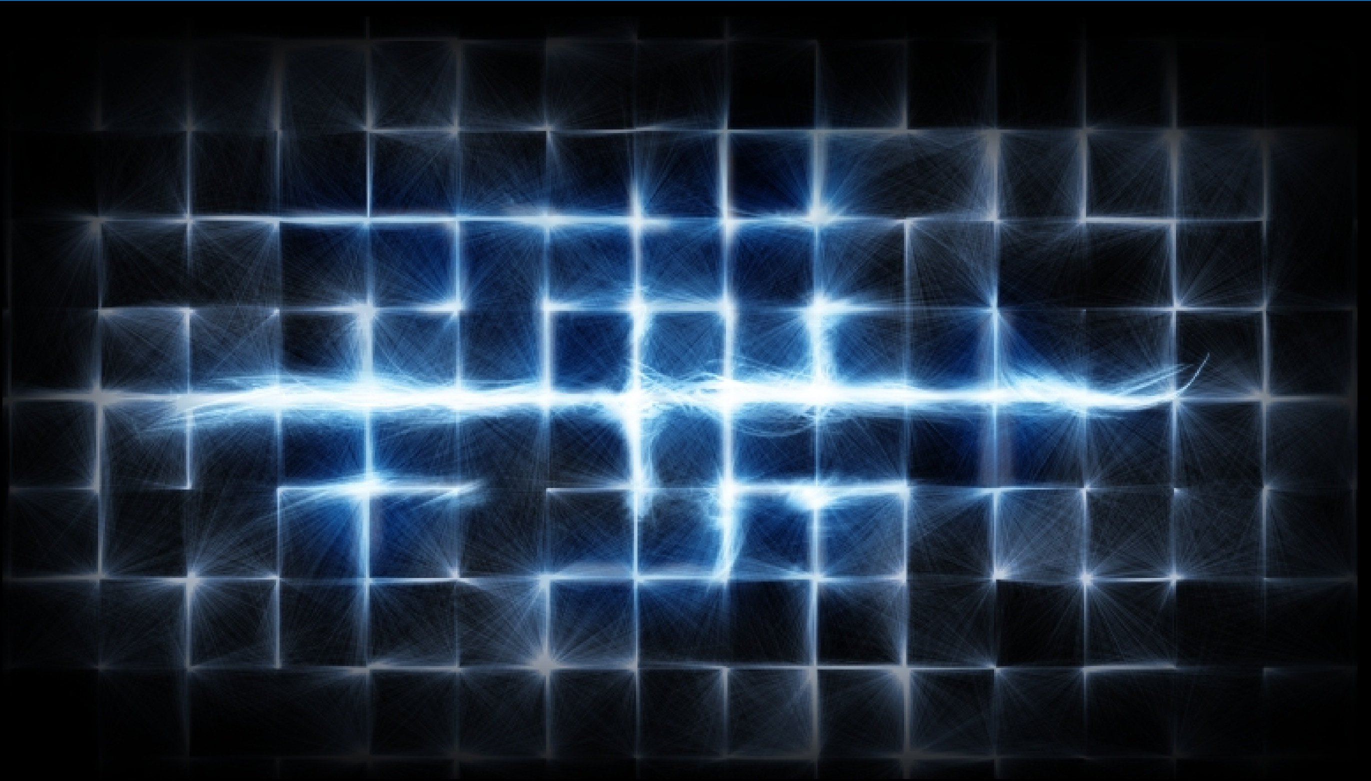
Mapping quantum algorithms

in a crossbar architecture

Alejandro Morais



TU Delft

Mapping quantum algorithms

in a crossbar architecture

by

Alejandro Morais

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on the 25th of September 2019.

Student number:	4747240	
Project duration:	November 1, 2018 – July 1, 2019	
Thesis committee:	Dr. ir. C.G. Almudever,	QCA, TU Delft
	Dr. ir. Z. Al-Ars,	CE, TU Delft
	Dr. ir. F. Sebastiano,	AQUA, TU Delft
	Dr. ir. M. Veldhorst,	QuTech, TU Delft
Supervisor:	Dr. ir. C.G. Almudever,	QCA, TU Delft

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

TU Delft Quantum & Computer Engineering

Abstract

In recent years, Quantum Computing has gone from theory to a promising reality, leading to quantum chips that in a near future might be able to exceed the computational power of any current supercomputer. For this to happen, there are some problems that must be overcome. For example, in a quantum processor, qubits are usually arranged in a 2D architecture with limited connectivity between them and in which only nearest-neighbour interactions are allowed. This restricts the execution of two-qubit gates and requires qubit to be moved to adjacent positions. Quantum algorithms, which are described as quantum circuits, neglect the quantum chip constraints and therefore cannot be directly executed. This is known as the mapping problem.

This thesis focuses on the problem of mapping quantum algorithms into a quantum chip based on spin qubits, called the crossbar architecture. In this project we have developed the required compiler support (mapping) for making quantum circuits executable on the crossbar architecture based on the tools provided by OpenQL. Using this compiler, we have analyzed the mapping overhead of the crossbar architecture and studied how it relates to the characteristics of quantum algorithms. In addition, we have developed a verification program that checks the output of the compiler and provides a visualisation tool for debugging.

Acknowledgements

This work would not have been possible without the support of many people.

Firstly, I would like to thank all of those who have been involved in my education, helping me find my passion for quantum computing and computer engineering. Especially to my supervisor, Dr. Carmina G. Almudéver, for her guidance and for giving me the opportunity to work in the field of quantum computing. I would also like to thank Lingling Lao for her additional supervision during this thesis and to Hans Van Someren for his help during the technical aspect of this thesis.

In addition, I would like to mention the rest of people in the Quantum Computer Architecture group: Prof. Koen Bertels, Abid Moueddene, Amitabh Yadav, Anneriet Krol, Aritra Sarkar, Diogo Valada, Imran Ashraf, Jeroen van Straten, Matthijs Brobbel, Savvas Varsamopoulos and Yaoling Yang; and people who have already left: Daniel Moreno Manzano, Leon Riesebos, Miguel Serrao and Xiang Fu.

Also, I would like to send a special thanks to all of those who have contributed to open source projects from which millions of people benefit today.

Finally, I would like to thank my family for their financial and emotional support throughout these years and, essentially, for putting up with me every day even a thousand kilometers away.

Alejandro Morais
Delft, September 2019

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Problem definition . . . 1

1.3 Structure . . . 3

2 Background **4**

2.1 Quantum Computing . . . 4

2.1.1 Qubits . . . 4

2.1.2 Quantum Gates . . . 5

2.1.3 Quantum Circuits . . . 6

2.2 Mapping problem in quantum computing . . . 7

2.2.1 Gate Decomposition . . . 8

2.2.2 Initial Placement . . . 8

2.2.3 Routing . . . 9

2.2.4 Scheduling . . . 9

2.2.5 State of the art . . . 10

3 Crossbar Architecture **12**

3.1 Layout and constraints . . . 12

3.1.1 Layout . . . 12

3.1.2 Layout constraints . . . 13

3.2 Model . . . 14

3.3 Operations . . . 16

3.3.1 Shuttling . . . 16

3.3.2 One-qubit gate . . . 19

3.3.3 Two-qubit gate . . . 22

3.3.4 Measurement . . . 24

3.4 Conflicts . . . 28

3.4.1 Side effects . . . 28

3.4.2 Undecidable configurations . . . 29

3.5 Gate Set Decomposition . . . 31

4 Mapping Implementation 33

4.1 Initial Placement . . . 33

4.2 Routing . . . 34

4.2.1 Crossbar Topology . . . 34

4.2.2 Crossbar Configuration. . . 35

4.2.3 Routing Strategy . . . 36

4.2.4 Routing Implementation . . . 37

4.3 Scheduling . . . 39

4.4 Mapping Decomposition . . . 39

5 Simulation framework 42

5.1 Framework overview . . . 42

5.2 Verification Program . . . 43

5.2.1 Parameters . . . 44

5.2.2 Conflict Checker . . . 44

5.2.3 Visualisation Tool . . . 45

5.3 Additional Crossbar Parameters . . . 46

5.3.1 Ancillary qubits . . . 46

5.3.2 Phase shift gates . . . 46

5.3.3 Crossbar Configuration. . . 47

6 Experiments and Results 50

6.1 Benchmarks . . . 50

6.2 Mapping Results . . . 51

6.2.1 Mapping Results with Trivial Initial Placement . . . 51

6.2.2 Mapping Results with Initial Placement . . . 53

6.3 Comparison of Different Mappers . . . 56

6.4 Comparison with the Surface 17 chip . . . 57

6.5 Mapping Overhead Analysis . . . 59

6.5.1 Characteristics . . . 59

6.5.2 Experiments . . . 60

6.6 Scalability . . . 62

7 Conclusions and future work 66

7.1 Conclusions. . . 66

7.2 Future work 67

A Gate decomposition **69**

A.1 CPHASE Decomposition 69

A.2 CNOT Decomposition 69

B Benchmarks **71**

Bibliography **75**

1

Introduction

This chapter explains the motivation behind the mapping of quantum algorithms in quantum processors in Section 1.1 with its definition in Section 1.2. Section 1.3 describes the organization of this thesis.

1.1. Motivation

Since the birth of the modern computer in the early 1930s, the main goal of a computer program was to execute a task as fast as possible. This meant, directing the majority of the efforts towards improving the electronics that handle the logic. The speed of progress was so high that a few years after, *Moore*, co-founder of Intel, successfully predicted that the number of transistors in a processor would double every two years. Unfortunately, there is a physical limit in the number of transistors that one can fit inside a certain amount of space. Due to this fact, nowadays many chip manufacturers include more than one core in their processors. This allows the software to increase its efficiency by doing calculations in parallel. But even with parallelization, this hardware has a hard limit in terms of computational power.

During this period, a new field of physics emerged, called quantum mechanics, which brought a new way of thinking and interacting with nature. In fact, it brought a new paradigm of computation which had the potential to overcome the limits of classic computation and increase the computational power exponentially. But analogous to what happened in the 1950s, nowadays there is a wide range of quantum computer technologies and all of them are still in a very early stage. And since every technology has different constraints, one can not execute a quantum algorithm directly into any quantum processors. There are some restrictions that must be respected to be able to execute the algorithm.

Therefore, there is clearly a gap between the software and the quantum hardware; we need to make quantum algorithms executable on given quantum processors. However, as we will see in the next chapters, there is no straight forward solution to this problem. And it is at this point in which this thesis will focus on; that is, in the process of mapping quantum algorithms to a specific chip architecture, the crossbar architecture.

1.2. Problem definition

In classical computing, a task is executed by dividing it into specific and discrete steps. This means describing the task in a language that the computer can understand: machine code. Initially, these programs were written in assembly language which was then translated into machine code. In the early days, this translation to machine code was dependant on

the underlying hardware implementation. In other words, a compiled program could not be executed in different hardware than its target. Fortunately, nowadays, this problem is solved by using a compatible instruction set between the most used processors of the market (AMD and Intel), although there are still some architectures (like ARM) that need a different compiler to run the program.

A similar problem is found in the execution of quantum algorithms in a quantum architecture. To tackle this and other related problems, the Quantum Computer Architecture Lab at TU Delft [9] proposed the system stack for a quantum computer as shown in Figure 1.1. The highest layer of the stack is the quantum algorithm, which is represented by a quantum circuit. This circuit describes the number of qubits and the gates that should be executed. Note that this description is hardware agnostic, so there are no limits in the types of gates or the interactions between qubits. But at the lowest level, the quantum chip might not support every gate and interaction. This is why it is necessary to have intermediate layers that translate the circuit input into an equivalent one that can be executed on the quantum chip.

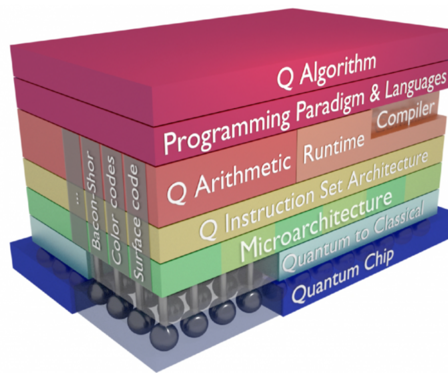


Figure 1.1: The full-stack design of quantum computer architecture [10]

One of the main constraints of a quantum chip is the interaction between qubits. These can be represented as graph, where each node is a qubit and each edge is a possible interaction between two qubits. This means that if two qubits not connected by an edge need to interact, they must be moved or swapped until they are adjacent. However, this routing is not trivial, since there are more constraints of the architecture that can obstruct some paths. And, as one might expect, this causes overhead in the final compiled circuit, since it will require more gates and time to execute the same algorithm.

In general, minimizing the overhead of a program is usually a good practice, however, in this particular, scenario it is crucial. This is because the qubits used nowadays are very fragile and the time a qubit can maintain its state is short [31]. Depending on the underlying quantum technology the qubits of the quantum chip will have a shorter or longer lifespan. But even with the longest decoherence time, it is just a matter of seconds [34] until the qubit is useless for computation. Thus, it is important to do a translation in which the time necessary to execute the algorithm (*latency* of the circuit) is as short as possible. In addition, gates are faulty; and the more faulty gates applied, the more errors the computation will have. So to minimize these two problems, we need not only to focus on the amount of gates, but also on the gates themselves; since the duration and fidelity can vary from gate to gate. So the aim of this intermediate layer is to map any quantum circuit into a quantum chip, while being compatible with the hardware constraints and minimizing overhead.

In addition, unlike classical computing, the translation from assembly language to native instructions is not *one-to-one* and there is not a common set of gates for all chips. It requires a program that takes into account the constraints of the target architecture, in order to add or remove the necessary quantum gates to execute the algorithm as fast and accurate as possible.

1.3. Structure

This thesis is organized as follows:

- Chapter 2 gives a brief introduction of quantum computing, while also describing the mapping problem in quantum hardware architectures.
- Chapter 3 introduces the quantum crossbar architecture and describes its operations and constraints.
- Chapter 4 describes the challenges encountered through the process of mapping quantum algorithms into the crossbar architecture and proposes techniques to deal with them.
- Chapter 5 illustrates the framework developed to run the experiments and verify their compatibility with the crossbar constraints.
- Chapter 6 shows all the experiments and explains how the results show new insights into the crossbar architecture.
- Chapter 7 summarizes the results and ideas drawn from this thesis. And, finally, future lines of work are introduced.

2

Background

This chapter introduces in a brief manner the fundamental concepts of quantum computing used throughout the thesis. In addition, it describes the main parts of mapping a quantum algorithm.

2.1. Quantum Computing

To explain the mapping problem in depth it is important to have a good understanding of the fundamentals of quantum computing. In this section we will introduce the concepts of qubits, gates and circuits. As we have done in the previous chapter, it is easier to understand these kind of concepts by using a analogy with classical computing.

2.1.1. Qubits

In classical computing, the basic unit of information is a bit: 0 or 1, high voltage or low voltage. However, quantum mechanics have shown that small particles, like an electron, can have a state of different nature. For example, the electron has a spin which can be used to represent these new type of states, called *quantum state* or *qubit*. This differs from a classical state in the deterministic feature. A quantum state can hold the value of "0" and "1" at the same time but with different or equal probabilities. This is called *superposition* and is one of the characteristics that make quantum mechanics so interesting [30]. So to define a qubit with this superposition we can make use of linear combination as shown in Figure 2.1.

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

Figure 2.1: Generalized representation of a qubit in bases $|0\rangle$ and $|1\rangle$

In this formula, α and β are arbitrary complex numbers that give the probabilistic property to the qubit state ϕ, and they are called *amplitudes*. In this example, the probability of measuring a "0" is $|\alpha|^2$ and the probability of measuring a "1" is $|\beta|^2$. As one might expect the sum of all the probabilities is 1. Thus, we can express this as a formula, shown in Figure 2.2.

$$|\alpha|^2 + |\beta|^2 = 1$$

Figure 2.2: Formula that relates the amplitudes of a qubit

To measure a qubit, one must use a specific measurement axis (or measurement operator). The most common is the Z axis, with eigenvectors $|0\rangle$ and $|1\rangle$. The measurement does not return the superposition state, it only returns one of the eigenvalues of the basis. Thus, after measuring, the quantum state *collapses* into one of the eigenstates of the measurement operator. In our example of Figure 2.1, the qubit will collapse to either $|0\rangle$ or $|1\rangle$ depending on the amplitudes [30]. However, when the qubit collapses it stays in that base and the superposition is destroyed. So the qubit does not reverse back to the superposition state. Although this might seem as an disadvantage, we will see that we leverage this property in some cases.

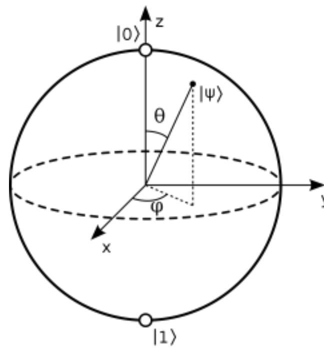


Figure 2.3: The Bloch sphere

In the previous example, we have used only one basis, in particular the *computational basis*, but there are infinite other bases from which we can choose to represent a quantum state, as well as to measure it. Figure 2.3 shows a visual representation of the possible quantum states that we can have. Each point in the bloch sphere represents a quantum state. Thus, we can define it by two parameters: θ and φ, where $\theta \in [0, \pi)$ and $\varphi \in [0, 2\pi)$. We can see these variables as the polar angle (θ) and azimuth (φ).

This explanation only takes into account one qubit. However, we can have any number of qubits. For example, assuming we are using the bases $|0\rangle$ and $|1\rangle$ then, a system of two qubits means that now these are the possible outcomes of our quantum system: $|00\rangle$, $|01\rangle$, $|10\rangle$ or $|11\rangle$. Again, we can define a general state, as shown in Figure 2.4.

$$|\phi\rangle = \alpha_1 |00\rangle + \alpha_2 |01\rangle + \alpha_3 |10\rangle + \alpha_4 |11\rangle$$

Figure 2.4: General quantum state for two qubits in bases $|0\rangle$ and $|1\rangle$

2.1.2. Quantum Gates

In general, having information (bits or qubits) is useful, but being able to do a computation is the whole purpose of a computer. So in this section we will see how to use a qubit.

As explained, quantum states can be seen as a complex unit normal vector with 2^n dimensions. To manipulate this information we can make use of a transformation that changes the qubit state into another one. To maintain a valid outcome after applying this operation, it needs to preserve the normalization of the quantum state. To do this, the transformation

is a unitary matrix U where $U = (U^*)^T$. In these types of transformations we can distinguish two types of transformations: one-qubit gates and two-qubit gates.

One-qubit gate: This type of gates can be seen as a rotation over any axis of the Bloch sphere. So any one-qubit gate can be expressed as: $R_x(\theta)\ R_y(\varphi)\ R_z(\lambda)$, rotations along the X, Y and Z axes, respectively. The matrices for the gates X, Y and Z, shown in Table 2.1, are called *Pauli matrices* (note that the identity operation is just the application of the identity matrix). There are other common gates, such as the H gate, which applies the rotation $R_x(\pi/2)R_y(\pi/2)$. This gate is commonly use for creating a superposition state in a qubit. So is important to be able to execute this gate in a quantum chip [26].

I	X	Y	Z	H	S	T
$-\boxed{I}-$	$-\boxed{X}-$	$-\boxed{Y}-$	$-\boxed{Z}-$	$-\boxed{H}-$	$-\boxed{S}-$	$-\boxed{T}-$
$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$	$\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$	$\begin{pmatrix}0 & -i\\i & 0\end{pmatrix}$	$\begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}$	$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$	$\begin{pmatrix}1 & 0\\0 & i\end{pmatrix}$	$\begin{pmatrix}1 & 0\\0 & e^{\frac{i\pi}{4}}\end{pmatrix}$

Table 2.1: Examples of single-qubit gates with their corresponding matrices

Two-qubit gate: This type of gates can create entanglement between qubits (but not all two-qubit gates). This means that they can connect two qubits in such a way that by measuring any of them it will affect the measurement of the other. The only way to achieve this is by applying a two-qubit gate. So in order to take advantage of this property is essential that the quantum chip we want to use supports a two-qubit gate that is capable of creating entanglement. In addition, if our set of quantum gates only contains single-qubit gates then there is no point in using a quantum computer, because the available computations could be simulated in a classical computer. An example of a two-qubit gate is the $CNOT$, where the *target qubit* is flipped in the X axis (from $|0\rangle$ to $|1\rangle$ and viceversa), if the *control qubit* is in the state $|1\rangle$. Otherwise the target qubit will be unchanged. This type of gates that use a *control-target* scheme are called control gates and they can execute a one-qubit gate based on the control qubit [30]. Finally, we want to introduce another two two-qubit gates which will be repeatedly used through this thesis, called SWAP and $\sqrt{SWAP}$ (or $\sqrt{SWAP}$). The SWAP gate is used to "exchange" or swap the quantum states of two qubits. And the $\sqrt{SWAP}$ is just the square root of the first, so by applying two $\sqrt{SWAP}$ gates consecutively we will have done a SWAP gate. Its main difference (compared to the SWAP gate) is that is universal - together with single-qubit gates, we can use it to run any quantum computation. Table 2.2 shows some examples of the matrices of two-qubit gates.

$CNOT$	$CPHASE$	$SWAP$	$\sqrt{SWAP}$
[circuit: control dot, target ⊕]	[circuit: control dot, $\boxed{Z}$]	[circuit: × — ×]	[circuit: × — (1/2) — ×]
$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&\frac{1}{2}(1+i)&\frac{1}{2}(1+i)&0\\0&\frac{1}{2}(1-i)&\frac{1}{2}(1+i)&0\\0&0&0&1\end{pmatrix}$

Table 2.2: Examples of two-qubit gates with their corresponding matrices

2.1.3. Quantum Circuits

A *quantum circuit* can be seen as a unitary transformation done in n number of qubits. For example, if we want to apply a Z gate in qubit 0 and a X gate in qubit 1, then the transfor-

mation will be $Z \otimes X$. Note that there are multiple ways of realizing a circuit. For example, Figure 2.5 shows four different quantum circuits which executes the same algorithm.

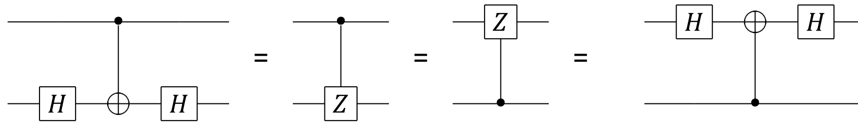


Figure 2.5: Example of equivalent quantum circuits

The fact that is possible to execute any quantum algorithm using a set of one-qubit gates and a two-qubit gate was shown in [3]. In particular, it used a technique, known as *QR decomposition*, that decomposed any quantum gate into one-qubit gates and CNOT gates. So a quantum circuit can be defined as a list of gates executed in order, affecting one, two or three qubits. An example of these circuits is shown in Figure 2.5.

Finally, it is worth defining the concepts of *depth* and *latency* of a circuit, since we will use them in the following chapters. The depth-d of a quantum circuit consists of d time steps, each time step contains one- and two-qubit gates acting on disjoint qubits [6]. And the latency of a quantum circuit is the time needed to execute the whole circuit.

2.2. Mapping problem in quantum computing

This section will describe the mapping problem and then it will explain the stages of mapping a quantum algorithm into a quantum chip.

As we have already introduced in chapter 1, each quantum chip has its own constraints. The main one is the limitation in the interactions between physical qubits. In many quantum chips, not all the qubits can interact with every other qubit. This means that, in those types of architectures, it is not possible to execute a two-qubit gate between any two qubits. Usually, this limitation in connectivity is based on nearest neighbour - i.e. two qubits must be adjacent to each other to execute a two-qubit gate. On the other hand, the quantum chip must be able to execute any quantum algorithm, so there needs to be a way to execute two-qubit gates between two qubits that are not connected. Depending on the quantum technology, the solution requires to move the qubits in different ways. In superconducting architectures, the qubits are moved by applying SWAP gates - this will move the virtual qubit through physical qubits. In semiconducting architectures, the physical qubit is move until it is adjacent to the other qubit required to perform the two-qubit gate. In chapter 4, we will explain in detail this difference.

On top of the connectivity constraints, there are other constraints depending on the technology. For example, in superconducting chips, the physical qubits can be operated with electromagnetic microwaves to execute single-qubit gates; and depending on the frequency used they can apply it to different qubits. Thus, a new constraint based on the frequency pulses is added to the quantum chip.

Regarding the mapping process, Figure 2.6 introduces the four main steps necessary to map a quantum algorithm into a quantum architecture. This process transforms a hardware-agnostic circuit specification into a hardware-dependent circuit. The input circuit is usually specified using a *Quantum Assembly Language* (QASM). Nowadays there are many different QASM languages, such as *cQASM* [17], *OpenQASM* [2] and *Quil* [40]. So, in theory, we could use any of these QASM languages as the input for our compiler. However, as we will explain later, for this thesis we will use cQASM due to its off-the-shelf implementation for the compiler framework we will use, called OpenQL [2]. This circuit specification is then passed through the four stages: gate decomposition, initial placement, routing and scheduling. Each of

these stages solves a different problem As a result, the compiling process outputs a list of instructions that can execute the input circuit into the target architecture. As we will see in the next sections, these stages can be implemented in a simple way; however, the problem arises when we want to optimize the number of gates and latency of the circuit.

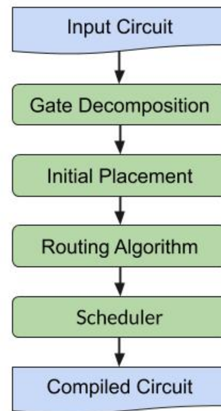


Figure 2.6: The compilation stages of mapping a quantum algorithm

2.2.1. Gate Decomposition

In practice, quantum computers do not support all types of two-qubit gates e.g. some ion trap quantum chips only supports $CNOT$ gates. However, to be able to execute any quantum algorithm, a quantum computer must support a two-qubit gate that can produce entanglement. Fortunately, as shown in the previous section, it is possible to execute any quantum algorithm if the set of gates is universal. In other words, if the quantum computer supports a universal set of gates, we can execute the rest of gates based on a decomposition of the supported gates. Thus, this first step that transforms any gate into a supported gate is called *gate decomposition*.

For example, if the quantum chip does not support natively the SWAP gate but it supports the CNOT gate then, it can execute such gate between qubit a and qubit b by applying: $CNOT_{a,b} \times CNOT_{b,a} \times CNOT_{a,b} = SWAP_{a,b}$. This is a common decomposition for moving the qubit state through SWAP gates in superconducting architectures [4].

2.2.2. Initial Placement

After the gate decomposition, the next step is to map each virtual qubit (from the circuit) to a physical qubit (in the quantum chip). This process is called *initial placement*. Although, the limitation of the qubit connectivity in the quantum chip will be addressed in the routing algorithm, the initial placement is, essentially, the first step into optimizing the routing. This is because the purpose of the initial placement is to reduce the number of movement operations (e.g. SWAPs) needed in the routing stage. This problem, also called *qubit assignment*, is NP-complete; so there is no straight forward solution.

Figure 2.7 shows an example, where the virtual qubits 1, 2 and 3 are mapped to the physical qubits A, D and G, respectively. This initial placement is optimal, since the circuit respects the connectivity of the physical qubits in such a way that no additional gates are needed.

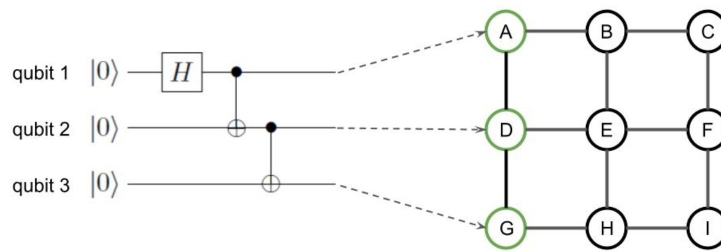


Figure 2.7: An example of an initial placement. The virtual qubits numbered (at the left) are mapped to physical qubits in a lattice topology (at the right). The dashed arrows represent the mapping from virtual to physical qubits. Virtual qubits 1, 2 and 3 are mapped to the physical qubits A, D and G, respectively.

2.2.3. Routing

After the initial placement (or initial mapping) is done, the routing algorithm checks if all the two-qubit gates can be executed based on the connectivity constraints. If not, then this process moves the qubit states or the physical qubits (depending on the technology used) through the topology in order to execute all the two-qubit gates.

For example, using the initial placement of Figure 2.7 and a new quantum circuit, Figure 2.8 shows an example where qubits A and G are not directly connected, so the CNOT gate can not be executed. But, after running the routing algorithm, a SWAP gate is added between the physical qubits A and D, in such a way that the physical qubits A and G are now connected and the CNOT gate can be executed.

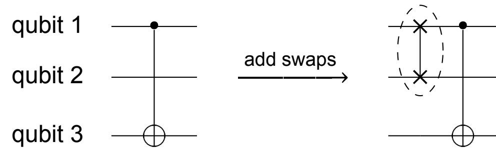


Figure 2.8: An example of a mapped quantum circuit. This example uses the initial mapping from Figure 2.7. The routing algorithm adds a SWAP gate to execute the CNOT between adjacent physical qubits in the topology of Figure 2.7.

2.2.4. Scheduling

Finally, the scheduler receives a quantum circuit that respects the connectivity constraints of the architecture. In theory, this quantum circuit can be executed directly into the quantum chip in a sequential order. However, for relative large circuits this might result in a random result due to the short lifespan of the physical qubits. This means that it is necessary to reduce the latency of the circuit as much as possible to be able to successfully execute large quantum circuits. This can be done by enabling parallel operations in the quantum chip. To do this, the *scheduler* is responsible for planning parallel operations while respecting the dependencies between them and the constraints of the quantum chip. This scheduling of gates can be done As Soon As Possible (ALAP) or As Last As Possible (ALAP).

For example, Figure 2.9 shows a quantum circuit that can be directly executed into the quantum architecture. To schedule this circuit, the scheduler must first create a dependency graph and then try to schedule as many parallel operations as possible. In this example, there is a dependency between the first two CNOT gates which prevents them from being scheduled in parallel. However, the third CNOT can be scheduled in parallel with the first one, if the constraints of the chip allows it.

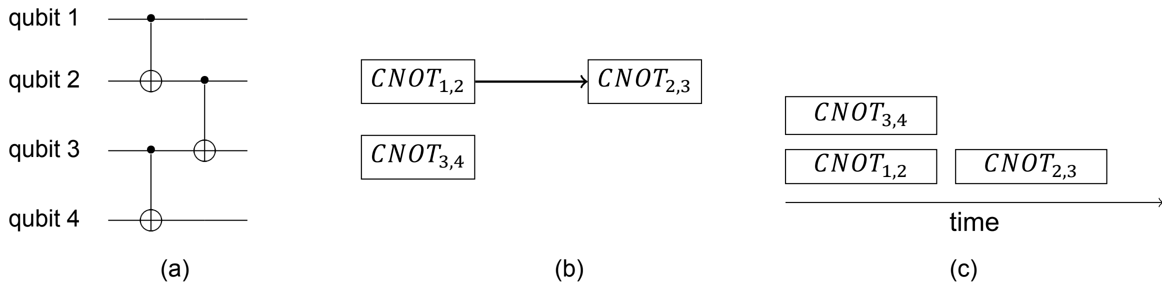


Figure 2.9: An example of a quantum circuit scheduling. **(a)** The quantum circuit to be scheduled. **(b)** The dependency graph of the gates. **(c)** A representation in time of the scheduled gates, assuming compatibility of constraints between $CNOT_{1,2}$ and $CNOT_{3,4}$.

2.2.5. State of the art

Quantum computing is starting to become a new revolution in fields such as physics and computer engineering. Nowadays, big companies such as Google [11], IBM [16] and Microsoft [27] are leading the quantum hardware. Some companies have even announced quantum chips with 128 qubit [32]. However, as with any new technology, the prototypes have tight constraints. For example, although IBM Q20 Tokyo already has 20 qubits and another with 50 qubits [18], the connectivity between qubits and decoherence time is still highly restrictive. Fortunately, there are error-correcting protocols that encode a logical qubit into multiple physical qubits; that way it provides a way to obtain a fault-tolerant quantum chip [33]. However, for now, the amount of physical qubits is not enough to encode these *error-correcting codes* (QEC). The period of time in which quantum chips will only be able to handle up to a few hundred qubits was defined as *Noisy Intermediate Scale Quantum* (NISQ) era by John Preskill [31]. In this period, the noise and decoherence will produce a negative effect on the reliability of quantum algorithms. To tackle these issues it is necessary to create a layer of abstraction between the quantum algorithms and the quantum chip that allows to overcome or, at least, minimize these problems. As explained in the previous section, this layer is called *mapping* and it is also responsible for making sure the constraints of the quantum chip are being respected.

In the process of mapping a quantum algorithm into a quantum chip, is necessary to take into account four things: the connectivity of qubits, the additional constraints of the architecture, the error rates and the decoherence time of the qubits. At this point, a high amount of research has only focused on the connectivity constraints of the architecture [15] [1] [7]. Another part of the research in mapping focuses on the fact that not all physical qubits have the same error rates [41] [29]. In particular, [41] proposed noise-aware mapping policies which tries to select paths with the lowest probability of failure and use strong links for performing two-qubit gates.

As we have mentioned, the qubit mapping problem has been proved to be NP-complete [39]. This means that we can distinguish two types of approaches in current research. The first one uses exact software solvers to find the optimal solution to the routing problem [36] [37]. These approaches need a long time to run, so they are only feasible for a small number of qubits. The second method uses a search to find the best mapping based on a heuristic-cost function [46] [45] [22]. The initial placement of these algorithms and their cost function are what differentiates these approaches between them. Although the heuristic method is currently the only reasonable approach for real applications, some algorithms can take even an hour to be compiled [46].

Most of the work done in mapping focuses in the superconducting devices, and specially in IBM chips. Regarding spin qubits, there is not much work being done in mapping quantum algorithms to quantum chips based on spin qubits. Mainly because the current prototypes of these quantum chips are only based on a one-dimensional array [28]. Although some research has been done in creating a 2D array of quantum dots [14] [42], including [23] which

is the architecture to which we will map quantum algorithms in this thesis. Lately, there has been developments around the creation of a more general compiler to target more quantum technologies. For example, [29] built a tool to map quantum algorithms to superconducting and ion-trap architectures. Yet, this tool can not be used to map quantum algorithms into spin qubits, since these quantum chips use *shuttles* to move the physical qubits [23], unlike superconducting architectures which use SWAP gates [4]. Finally, [43] proposed a compiler based on time-planning to model a general quantum architecture. However, this type of compiler is more limited than other compilers based in C++, like *OpenQL* [2], which allows to define more complex constraints. In summary, there is a gap yet to be filled regarding mapping quantum algorithms into spin qubits.

3

Crossbar Architecture

Multiple processor architectures have been proposed for building quantum computers. However, many all of these architecture do not scale up easily in terms of number of qubits e.g. some approaches such as ion traps controlled by microwaves will require $100x100m^2$ for two billion qubits [21] and superconducting qubits will require $5x5m^2$ [8]. Although this number of qubits seems high, it is approximately the amount needed to perform a Shor factorization in a 2048-bit number [21]. Thus, there a need to search for a better approach to handle the size and complexity of the control of qubits in a more compact way. This is where the crossbar architectures might be a good step towards this direction. In particular, this thesis will study the crossbar network proposed by Li et al. [23].

In this chapter we will explain how the crossbar architecture works, from the perspective of the mapping problem. Firstly, Section 1.3 will describe the structure and the supported operations. Then, to be able to implement this new architecture in the compiler, Section 3.2 will propose a model of the crossbar and Section 3.3 will discuss the operational constraints based on this model. Finally, Section 3.4 will show the consequences of violating these constraints and Section 3.5 will define the gate set decomposition to run any quantum algorithm.

3.1. Layout and constraints

Traditionally, the term *crossbar* is referred to a type of switch architecture, where the interconnection of horizontal and vertical lines creates a circuit controlled by a switch. Although the crossbar network proposed in this quantum chip might resemble the old idea of a crossbar, it is still very different in terms of functionality and operations. In this section, we will discuss the structure of this architecture and its supported operations.

3.1.1. Layout

Firstly, it is worth describing the physical layer of the crossbar in a brief way. Starting from the qubit itself, this architecture is based on spin qubits, that is, the quantum state of the qubit is encoded in the spin of an electron. This spin can be manipulated by applying a magnetic field to the electron. In the crossbar architecture shown in Figure 3.1a, a grid is made by using direct current (DC) lines that makes the horizontal and vertical lines, called row lines (RL) and column lines (CL), respectively. These lines, through their magnetic field, divide the grid into sites where a spin qubit can be placed. Hence, these lines are called *barriers*. Each of these barriers can be open (lowered) or closed (raised). For example, if a vertical barrier (CL) is open then the electron at its left can move to the right site (if empty)

and vice versa. This technique to move electrons from one site to another will be explained in the next section.

Moreover, there is an important difference between the horizontal and vertical lines: the DC currents of the column lines are placed in an alternating direction. This creates a difference in Zeeman energy between the odd columns and the even columns. This difference between columns changes the resonance frequency of the electrons. In the next sections we show see how this effect will influence the execution of sequential and parallel operations.

Lastly, the diagonal lines, called qubit lines (QL), cross the grid diagonally (from the bottom left to the top right). They are also DC lines, but unlike the rest of lines, these ones are not barriers. By setting a certain voltage through the diagonal lines, they are used to make two qubits interact with each other or to move qubits from site to site.

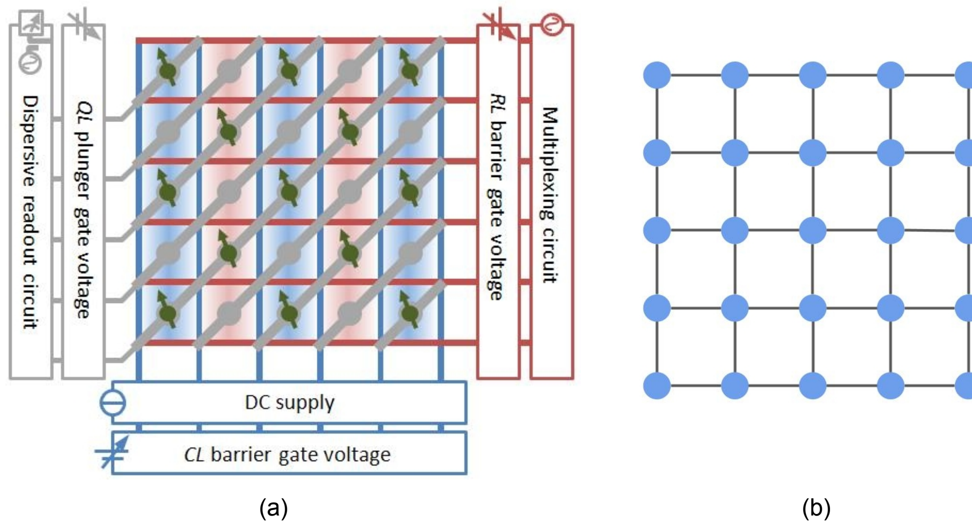


Figure 3.1: The crossbar architecture **(a)** [23] the control lines for performing different operations. The green dots are the electrons with arrows representing their spin. The red horizontal lines are the row lines and the blue lines are the column lines. Finally, the gray diagonal lines, that pass over the electrons, are the qubit lines. The blue columns have a higher Zeeman energy than the red columns. **(b)** Representation of the topology of the crossbar architecture. Each node (blue circle) represents a site of the grid and each edge (black line) represents the possibility of interaction between the sites connected to that edge.

In order to study this architecture, we need to make a level of abstraction where this physical crossbar architecture can be seen as a grid where each position of the grid can be occupied by a qubit (an electron). In this particular architecture, in order to execute a qubit gate between two qubits, they must be horizontally or vertically adjacent to each other. The interaction between two qubits diagonally adjacent is not possible. In other words, this crossbar implementation uses near-neighbour interaction between qubits in the same row or in the same column. Figure 3.1b shows a visual representation of this topology, where qubits in the borders can interact with 2 or 3 neighbours, whereas the rest of qubits can interact with 4 neighbours. Note that some sites represented in Figure 3.1b can be empty, so to make two non-adjacent qubits interact they must be moved until they are adjacent.

3.1.2. Layout constraints

Logically, the definition of each operation in the crossbar is limited by the structures that define the architecture. Therefore, it is necessary to define the constraints of the architecture before we execute any operation. This section describes the common constraints of the architecture for all operations. In the next section we will define the different constraints of each operation.

Qubit sites: Firstly, as explained before, the crossbar is composed of a grid where each site can be occupied at most by two qubits. Although, in general, we should only allow one qubit per site, in the first phase of the measurement operation we will show that for some cases we might have two qubits in the same site.

Qubit positions: Secondly, the number of qubits that can be used in the crossbar is limited by the number of sites. For example, a crossbar of size $4x4$ can only have a maximum of 16 qubits. However, in general, having a qubit per site is not practical due to the following reasons: 1) it will increase the number of shuttles to make two far away qubits interact; 2) it will increase the crosstalk between qubits; 3) it will reduce the number of control gates per qubit. Helsen et al. [13] defines different configurations for the positions of the qubits based on repetitive operations, such as multiple measurements. Since these configurations are made for error-correcting protocols, as an initial approach, we will use a configuration where half of the sites are empty and the qubits are placed in a checkerboard structure. It is worth mentioning that this amount of qubits includes both the data qubits and ancillary qubits and there is no constraint in the ratio of data qubits to ancilla qubit.

3.2. Model

A simplified visual representation of the crossbar layout is shown in Figure 3.2a. The colored lines represent the control lines used to execute the operations while the square sites represent the quantum dots. The number of wires required for this square crossbar is $\approx 4\sqrt{2N}+1$, where N is the number of qubits [23]. But note that the control lines at the edges of the crossbar are not used, these barriers are considered to always be raised. On the other hand, Figure 3.2b shows how the different types of columns of the crossbar are laid out. We will use this visual representation for the rest of the thesis.

Although the crossbar network proposed in [23] does not have a limit in terms of size, we will consider the crossbar to have a size $N \times N$ (where N is both the number of rows and the number of columns), since it is the optimal shape for the number of control gates per qubit when using the *idle* configuration. Based on the layout and control of this architecture, it seems reasonable to model it as a $N \times N$ matrix. The vertical axis is labeled with i and the horizontal axis is labeled with j. The indices are numbered from 0 to $N-1$ from bottom to top and from left to right, respectively.

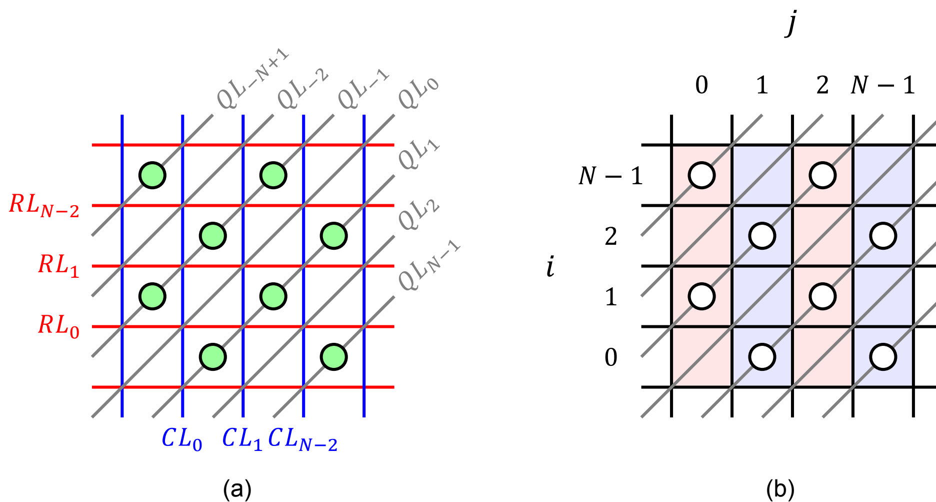


Figure 3.2: **(a)** The column lines (CL) are the blue vertical lines. The row lines (RL) are the red horizontal lines. The qubit lines (QL) are the diagonal gray lines. And the qubits are the green dots inside a site. Note that the column and row lines at the edges of the layout are ignored. In other words, they are considered to be always raised. **(b)** The vertical axis of the crossbar is label with i and the horizontal axis with j. The columns in blue have a higher Zeeman frequency than the red columns.

In this section, we will describe the crossbar layout and control lines. Firstly, we define the parameters of a crossbar:

$$N = \text{size of the crossbar}$$

$$Q = \text{number of qubits}$$

Note this number Q also contains the ancillary qubits necessary to do the measurement operation. Secondly, we define the data structures used:

- S stores the quantum state of all qubits (as shown in Equation 3.1).

$$\begin{gathered} S[q] = |\phi\rangle_q \\ Q_b = \{0, .., Q-1\} \\ q \in Q_b \end{gathered} \tag{3.1}$$

- P stores the position (i, j) (row i, column j) of all qubits. While C is exactly the opposite, is a matrix that stores the set of qubit identifiers of the specified site. Note that we use a set to be able to represent the cases where there are two qubits in the same site during the first phase of a measurement. And this set must be unique to be compatible with the Pauli Spin Blockade (PSB) (explained in section 3.3.4) These two structures are shown in Equation 3.2 and 3.3.

$$P[q] = (i, j) \tag{3.2}$$

$$\begin{gathered} C[i,j] = \begin{cases} \{x \mid x \in q, S[x] \text{ is unique}\}, & \text{when a site is occupied} \\ \emptyset, & \text{when a site is empty} \end{cases} \\ i, j \in [0 \mathinner{..} N-1], \quad q \in Q_b \end{gathered} \tag{3.3}$$

- RL and CL are vectors which store the status of the row lines and column lines, respectively (as shown in Equation 3.4 and 3.5).

$$RL[k] = \begin{cases} 0, & \text{barrier raised} \\ 1, & \text{barrier lowered} \end{cases} \tag{3.4}$$

$$\begin{gathered} RL[k] = \begin{cases} 0, & \text{barrier raised} \\ 1, & \text{barrier lowered} \end{cases} \\ CL[k] = \begin{cases} 0, & \text{barrier raised} \\ 1, & \text{barrier lowered} \end{cases} \end{gathered} \tag{3.5}$$

$$k \in [0 \mathinner{..} N-2]$$

- Finally QL stores the current voltages of all qubit lines (as shown in Equation 3.6). T is the maximum allowed voltage specified by the physical device. We assume the voltage is an integer, although the definitions are still valid for real numbers.

$$\begin{gathered} QL[d] = t \\ d \in [-N+1 \mathinner{..} N-1], \quad t \in [0 \mathinner{..} T] \end{gathered} \tag{3.6}$$

Now that we have the necessary abstraction to deal with the characteristics of the crossbar architecture, we are ready to start defining each operation and its constraints. In the next section we will use the above definitions to explain how the operations are implemented while being compatible with the constraints.

3.3. Operations

As explained, this architecture uses the available control lines to execute each operation. Therefore, we can define an operation as a set of *instructions* (or *grid operations*, as called in [13]) that use the control lines to execute a specific action in the crossbar. For example, a change in the voltage of a qubit line is an instruction which is part of the shuttling operation.

On top of the limits that the architecture imposes (from Section 3.1), there are other constraints which must be respected in order to correctly run quantum algorithms. If any of these constraints is not followed, we call this a *conflict of constraints*. Logically, a conflict produces an invalid configuration or invalid result in the given quantum algorithm. We will explain these consequences in detail in section 3.4.

For each of the operations we first explain how the operation is implemented, then we define the main characteristics of the operation and show how to execute the operation by using primitive instructions. Finally, we describe the constraints that need to be followed to correctly execute the operation.

3.3.1. Shuttling

Firstly, it is important to point out that, to execute a two-qubit gate both qubits have to be adjacent. Therefore, if we want to execute a two-qubit gate between two non-adjacent qubits, we must have a way to bring them closer. And, as we have mentioned, in other quantum technologies, such as superconducting qubits [4], the method they use to solve this is through SWAP gates. This gate will, effectively, swap the states of two qubits. Thus, moving the qubit state through the topology. This is possible because each qubit is connected to, at least, another qubit. However, in the crossbar architecture, not all the sites are occupied, there can be an empty site without a qubit anywhere in the grid. So it is not possible to always use SWAP gates to move qubits between sites. In order to execute a two-qubit gate between two non-adjacent quits, it is necessary to actually move the qubit through empty sites. This movement is called *shuttling* and can be implemented horizontally or vertically. And, as we will see, it is faster than the SWAP operation. In the next chapter, we will show how the shuttle operation directly influences the scheduling of operations and routing of qubits.

3.3.1.1. Horizontal Shuttling

This operation is based on moving an electron to its left or right adjacent site. We need to lower the adjacent (left or right) vertical barrier and change the voltage of the qubit lines. By doing so, the voltage of the qubit line that passes through the destination site is higher than the qubit line that passes through the origin.

Note that in this shuttling, the qubit is changing to a different column. Therefore, due to a difference in the Zeeman energy between the even and odd columns, the qubit will start to precess around its Z axis. However, we can be mitigate this phase shift by timing the next operation correctly. By waiting enough time, a full Z rotation will happen and the original quantum state will be recovered.

Definition: The horizontal shuttling moves a qubit from site (i, j) to the site $(i, j + k)$, where $k \in \{+1, -1\}$ (shuttle to the right or left, respectively). Based on [13] we expect a latency of $10ns$.

Instructions: To execute the horizontal shuttling these instructions must be followed:

1. Lower the vertical barrier between the origin and destination sites: $CL[j-(1/2)+(k/2)] = 1$

2. Set a higher voltage in the qubit line passing through the destination site than the one passing through the origin: $QL[j-i] < QL[(j+k)-i]$
3. Rise the previous vertical barrier: $CL[j-(1/2)+(k/2)] = 0$

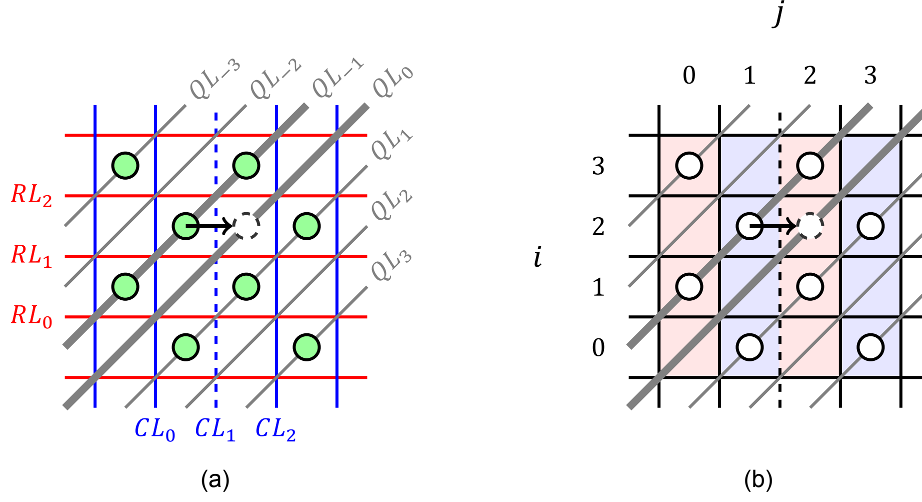


Figure 3.3: In this example a qubit in site $(2,1)$ is shuttled to site $(2,2)$. To do this QL_0 is set higher than QL_{-1} and the barrier CL_1 is lowered.

Constraints: To correctly execute this operation, the following conditions must be met:

- **Sites:** The destination site must be empty (as shown in Equation 3.7). Otherwise, the qubit will collapse.

$$C[i, j+k] = \emptyset \tag{3.7}$$

- **Qubits:** There can not be two qubits horizontally adjacent to each other between the columns j and $j+k$ (except in the row that we want to execute the operation, as shown in Equation 3.8). Otherwise, those horizontally adjacent qubits will interact by performing a two-qubit gate or even collapse (this will be explained in section 3.4).

$$C[x, j] = \emptyset \vee C[x, j+k] = \emptyset \text{ where } \forall x \in [0..N-1] \setminus \{i\} \tag{3.8}$$

- **Barriers:** The vertical barrier between the sites (i, j) and $(i, j+k)$ must be lowered (as shown in Equation 3.9). Otherwise, the qubit will not be able to shuttle.

$$CL[j-(1/2)+(k/2)] = 1 \tag{3.9}$$

  The barriers around the sites (i, j) and $(i, j+k)$ sites (left, right, top and bottom) must be raised. Otherwise, the movement of the qubit can not be predicted (as shown in Equation 3.10, 3.11, 3.12 and 3.13).

$$CL[j-1-(1/2)+(k/2)] = 0 \tag{3.10}$$
$$CL[j+(1/2)+(k/2)] = 0 \tag{3.11}$$
$$RL[i-1] = 0 \tag{3.12}$$
$$RL[i+1] = 0 \tag{3.13}$$

- **Qubit lines:** The voltage from the qubit line that passes through the site $(i, j+k)$ must be higher than the one passing through the site (i, j) (as shown in Equation 3.14). Otherwise, the qubit will not shuttle to the destination site.

$$QL[j-i] < QL[(j+k)-i] \tag{3.14}$$

To keep the rest of qubits in columns j and $j+k$ from shuttling, we must apply a higher voltage in the qubit line that passes through their site than the one that passes through its adjacent empty site (as shown in Equation 3.15, 3.16 and 3.17).

$$\text{if } C[j-x] \neq \emptyset \Rightarrow QL[(j+k)-x] < QL[j-x] \quad (3.15)$$
$$\text{if } C[(j+k)-x] \neq \emptyset \Rightarrow QL[j-x] < QL[(j+k)-x] \quad (3.16)$$
$$\forall x \in [0..N-1] \setminus \{i\} \quad (3.17)$$

3.3.1.2. Vertical Shuttling

This operation is based on the same ideas as the horizontal shuttling. In this case, the electron is move in the same column, up or down. To do this, the horizontal barrier adjacent to the target site need to be lowered. Then the voltage of the qubit line that passes through the target site need to be higher than the voltage of the qubit lines that passes through the origin. Then, the electron shuttles to its destination and the horizontal barrier can be raised again. The main difference from the horizontal shuttling is that the vertical shuttling preserves the qubit state without taking into account the time. So, in general, this shuttling is preferred since it will require less time.

Definition: A vertical shuttling moves a qubit in site (i, j) to the site $(i+k, j)$, where $k \in \{+1, -1\}$ (shuttle to the right or left, respectively). Based on [13] we expect a latency of $10ns$.

Instructions: To execute this operation these instructions must be followed:

1. Lower the horizontal barrier between the origin site and the destination site: $RL[i-(1/2)+(k/2)] = 1$.
2. Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[j-i] < QL[(j+k)-i]$.
3. Rise the previous horizontal barrier: $RL[i-(1/2)+(k/2)] = 0$.

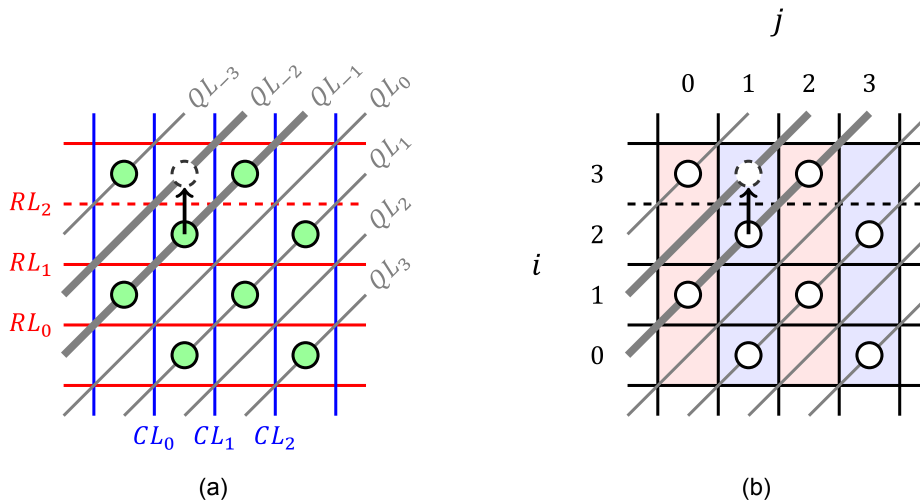


Figure 3.4: In this example a qubit in site $(2,1)$ is shuttled to site $(3,1)$. To do this QL_{-2} is set higher than QL_{-1} and the barrier RL_2 is lowered.

Constraints: To correctly execute this operation, the following conditions must be met:

- **Sites:** The destination site must be empty. Otherwise, the qubit will collapse (as shown in Equation 3.18).

$$C[i+k,j] = \emptyset \tag{3.18}$$

- **Qubits:** There can not be two qubits vertically adjacent to each other between the rows i and $i+k$ (except in the column that we want to execute the operation, as shown in Equation 3.19). Otherwise, those vertically adjacent qubits will interact and change of state or even collapse.

$$C[i,x] = \emptyset \vee C[i+k,x] = \emptyset \text{ where } \forall\, x \in [0..N-1] \setminus \{j\} \tag{3.19}$$

- **Barriers:** The horizontal barrier between the sites (i,j) and $(i+k,j)$ must be lowered (as shown in Equation 3.20). Otherwise, the qubit will not be able to shuttle.

$$RL[i-(1/2)+(k/2)] = 1 \tag{3.20}$$

  The barriers around the sites (i,j) and $(i+k,j)$ must be raised (as shown in Equation 3.21, 3.22, 3.23 and 3.24). Otherwise, the movement of the qubit can not be predicted.

$$RL[i-1-(1/2)+(k/2)] = 0 \tag{3.21}$$
$$RL[i+(1/2)+(k/2)] = 0 \tag{3.22}$$
$$CL[j-1] = 0 \tag{3.23}$$
$$CL[j+1] = 0 \tag{3.24}$$

- **Qubit lines:** The voltage from the qubit line that passes through the $(i+k,j)$ site must be higher than the one passing through the (i,j) site (as shown in Equation 3.25). Otherwise, the qubit will not shuttle to the destination site.

$$QL[j-x] < QL[j-(i+k)] \tag{3.25}$$

  To keep the rest of qubits in rows i and $i+k$ from shuttling, we must apply a higher voltage in the qubit line that passes through their site than the one passing through its adjacent empty site (as shown in Equation 3.26, 3.27 and 3.28).

$$\text{if } C[i,x] \neq \emptyset \text{ then } QL[x-(i+k)] < QL[x-i] \tag{3.26}$$
$$\text{if } C[i+k,x] \neq \emptyset \text{ then } QL[x-i] < QL[x-(i+k)] \tag{3.27}$$
$$\forall\, x \in [0..N-1] \setminus \{j\} \tag{3.28}$$

As mentioned previously, there is no need to wait in this type of shuttling, since there is no phase shift in the quantum state.

3.3.2. One-qubit gate

Regarding the one-qubit gates, there are two methods of executing this operation based on the gate and the amount of qubits that we can apply to.

3.3.2.1. Semi-global rotation

The first method uses the magnetic field from the column lines to change the spin of the electrons. As mentioned before, the direction of the current of these lines are placed in an alternating order. This means that if we want to change the spin of an electron, not only it will affect the entire column, but it will also affect all the electrons in the columns with the

same parity. This clearly has the advantage of executing parallel one-qubit operations, but it makes it harder to schedule it with other parallel operations.

The problem arises when we want to apply a one-qubit gate (for example an X gate) to only one qubit. To solve this we need to take advantage of the shuttling operation defined previously. Since is not possible to directly apply a one-qubit gate to particular qubit, we first, apply the desired gate to the qubits that are in the columns with the same parity as the target qubit. Then, after shuttling the target qubit to an adjacent, a new one-qubit gates is apply to the same columns, but this time it will apply the inverse gate of the first one. By doing this, we make sure that the target qubit is the only one with the desired one-qubit gate applied. Finally, we can shuttle the target qubit back to its original column, if necessary. This procedure is called *semi-global rotation*. A visual example can be seen at Figure 3.5.

Definition: This operation makes a single qubit rotation in the special unitary group $SU(2)$. By default, this operation is applied to all the qubit on the odd or even columns, but with additional shuttles we can apply it to the qubit in site (i, j). Based on [13] we expect a latency of $1000ns$.

Instructions: To execute this operation these instructions must be followed:

1. Through the column lines, apply the unitary global rotation to all the target columns TC, where their parity matches the parity of j. This can be expressed as: $TC = \{x \mid \forall x \in [0..N-1], x \bmod 2 = j \bmod 2\}$.
2. Shuttle the target qubit from (i, j) to a column with a different parity $j + k$.
   (a) Lower the barrier: $CL[j - (1/2) + (k/2)] = 1$.
   (b) Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[j - i] < QL[(j + k) - i]$.
   (c) Rise the barrier: $CL[j - (1/2) + (k/2)] = 0$.
3. Through the column lines, apply the inverse of the unitary rotation applied in step 1 to the same columns in TC.
4. (Optional) Shuttle the target qubit back to its original column in j.
   (a) Lower the barrier: $CL[j - (1/2) + (k/2)] = 1$.
   (b) Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[(j + k) - i] < QL[j - i]$.
   (c) Rise the barrier: $CL[j - (1/2) + (k/2)] = 0$.

Constraints: To correctly execute this operation, the following conditions must be met:

- **Sites:** The destination site $(i, j + k)$ must be empty (as shown in Equation 3.29). Otherwise, the qubit will collapse.

$$C[i, j + k] = \emptyset \tag{3.29}$$

- **Qubits:** During the shuttle to the ancilla site, there can not be two qubits horizontally adjacent to each other between the columns j and $j+k$ (except in the row that we want to execute the shuttle, as shown in Equation 3.30). Otherwise, those horizontally adjacent qubits will interact and change of state or even collapse.

$$C[x, j] = \emptyset \lor C[i, x + k] = \emptyset \text{ where } \forall x \in [0..N-1] \setminus \{j \tag{3.30}$$

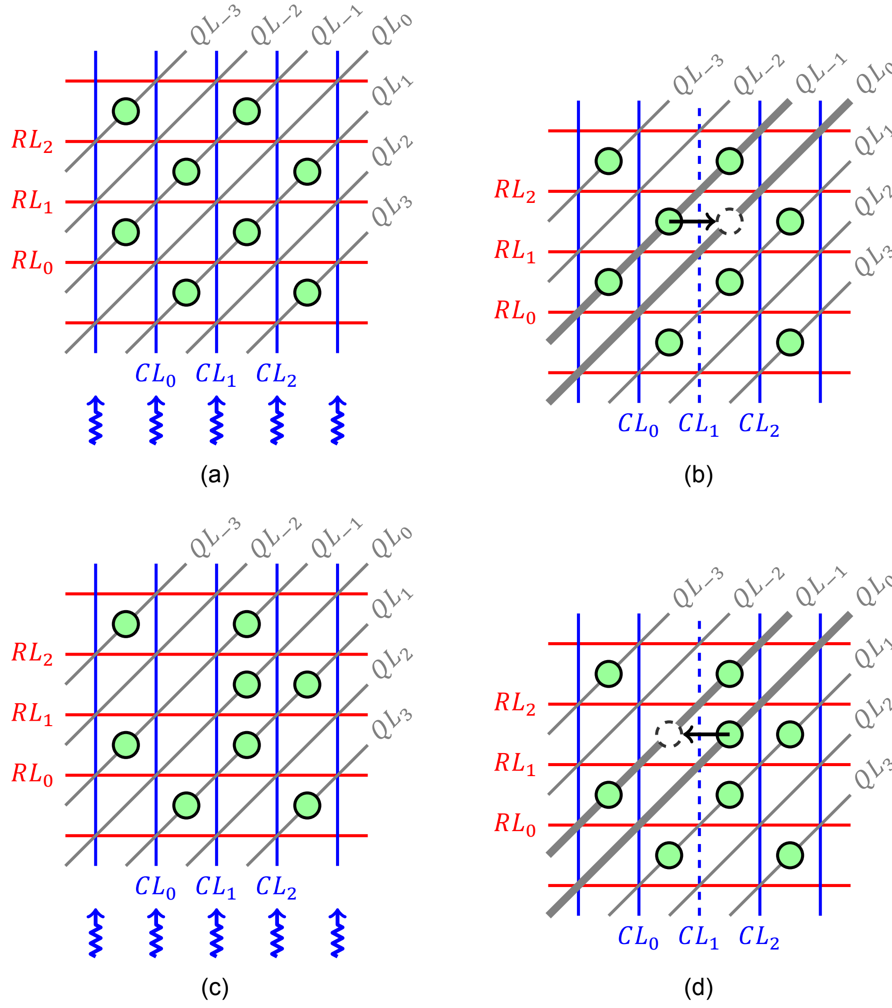


Figure 3.5: Example of how the semi-global rotation scheme works. Firstly, **(a)** it applies an RF pulse through the column lines and affecting only the electrons in the odd columns (1 and 3). Secondly, **(b)** the target qubit is shuttled to an adjacent column. Then **(c)** the inverse gate is applied to the same columns. Finally, **(c)** the target qubit is shuttled back to its original site.

- **Barriers:** During the shuttle of the target qubit from and to the adjacent column, the barriers around the sites (i, j) and $(i, j+k)$ must be raised (as shown in Equation 3.31, 3.32, 3.33 and 3.34). Otherwise, the movement of the qubit can not be predicted.

$$RL[j-1] = 0 \tag{3.31}$$
$$RL[j+1] = 0 \tag{3.32}$$
$$CL[j-1-(1/2)+(k/2)] = 0 \tag{3.33}$$
$$CL[j+(1/2)+(k/2)] = 0 \tag{3.34}$$

- **Qubit lines:** During the shuttle to the ancilla site, the voltage from the qubit line that passes through the $(i, j+k)$ site must be higher than the one passing through the (i, j) site (as shown in Equation 3.35). Otherwise, the qubit will not be able to shuttle to the ancilla site.

$$QL[j-i] < QL[(j+k)-i] \tag{3.35}$$

To keep the rest of qubits in columns j and $j+k$ from shuttling, we must apply a higher voltage in the qubit line that passes through their site than the one passing through its

adjacent empty site (as shown in Equation 3.36, 3.37 and 3.38).

$$\text{if } C[x, j] \neq \emptyset \text{ then } QL[(j + k) - i] < QL[j - x] \tag{3.36}$$
$$\text{if } C[x, j + k] \neq \emptyset \text{ then } QL[j - i] < QL[(j + k) - i] \tag{3.37}$$
$$\forall\, x \in [0..N - 1] \setminus \{i\} \tag{3.38}$$

3.3.2.2. Phase shift gates

The second method is based on the difference in Zeeman energy between the columns. As mentioned before, the even columns have a different magnetic field than the odd columns. Therefore, when an electron moves from one column to another its quantum state will start to phase shift. This means that by timing the next operation correctly, we can apply a Z gate to the qubit. The advantage of this method is the low latency compared to the previous method. However, there are two disadvantages. The main disadvantage is that any horizontal shuttling will start to produce a phase shift in the state of the qubit. The second one is that this method can only apply gates of the form $U(\phi) = e^{i\phi Z}$ (where ϕ is the state of the qubit).

Definitions: This operation applies a gate of the form $U(\Phi) = e^{i\Phi Z}$, where Z = Pauli-Z to the qubit in site (i, j). The auxiliary site used for this gate is $(i, j + k)$, where $k \in \{-1, +1\}$.Based on [13] we expect a latency of $100ns$.

Instructions: To execute this operation these instructions must be followed:

1. Shuttle the qubit to an adjacent column $j + k$
   (a) Lower the barrier: $CL[j - (1/2) + (k/2)] = 1$.
   (b) Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[j - i] < QL[(j + k) - i]$.
   (c) Rise the barrier: $CL[j - (1/2) + (k/2)] = 0$.
2. Wait $50ns$ [13] for the qubit state to shift its phase π.
3. Shuttle the qubit back to one of the adjacent columns.
   (a) Lower the barrier: $CL[j - (1/2) + (k/2)] = 1$.
   (b) Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[(j + k) - i] < QL[j - i]$.
   (c) Rise the barrier: $CL[j - (1/2) + (k/2)] = 0$.

Constraints: To correctly execute this operation, the following conditions must be met:

- Apply all constraints from the *horizontal shuttling*.
- Note: we can shuttle the qubit to any of the adjacent columns. So, if the destination site is empty, we are free to choose the value of the parameter k without affecting the overall quantum algorithm.

3.3.3. Two-qubit gate

Regarding the two-qubit gates, there are two methods of executing this operation based on the position of the qubits. As mentioned before, to make two qubits interact we need to make them adjacent to each other. Therefore we have two methods: a vertical method that executes a $\sqrt{SWAP}$ and the horizontal method that executes a CPHASE.

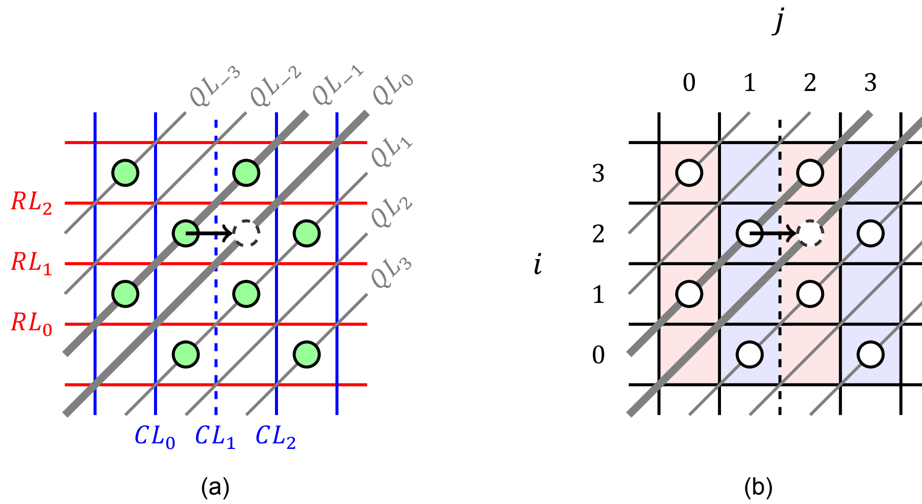


Figure 3.6: Similar to figure 3.3, these figures show the shuttling of a qubit to a different column, where it will gain a different phase based on the time it stays in that column.

3.3.3.1. $\sqrt{SWAP}$ gate

The first method works by having two qubits vertically adjacent to each other, lowering the horizontal barrier between them and setting the qubit lines that passes through those two qubits at the same voltage. However, this interaction can only produce a $\sqrt{SWAP}$ gate, and based on how long we leave the qubit in that configuration, we can execute a $\sqrt{SWAP}$ gate or SWAP gate.

Definitions: This operation executes a $\sqrt{SWAP}$ gate to the qubits in sites (i, j) and $(i + k, j)$, where $k \in \{+1, -1\}$. Based on [13] we expect a latency of $20ns$.

Instructions: To execute this operation these instructions must be followed:

1. Lower the row barrier between the qubits: $RL[i - (1/2) + (k/2)] = 1$.
2. Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[j - i] = QL[j - (i + k)]$.
3. Rise the row barrier between the qubits: $RL[i - (1/2) + (k/2)] = 0$.

Constraints: To correctly execute this operation, the following conditions must be met:

- **Qubits:** There can not be two qubits vertically adjacent to each other between the rows i and $i + k$ (except in the column that we want to execute the operation, as shown in Equation 3.39). Otherwise, those vertically adjacent qubits will interact and change of state or even collapse.

$$C[i, x] = \emptyset \lor C[i + k, x] = \emptyset \text{ where } \forall x \in [0..N - 1] \setminus \{j\} \tag{3.39}$$

- **Barriers:** The horizontal barrier between the sites (i, j) and $(i + k, j)$ must be lowered (as shown in Equation 3.40). Otherwise, the gate can not be executed.

$$RL[i - (1/2) + (k/2)] = 1 \tag{3.40}$$

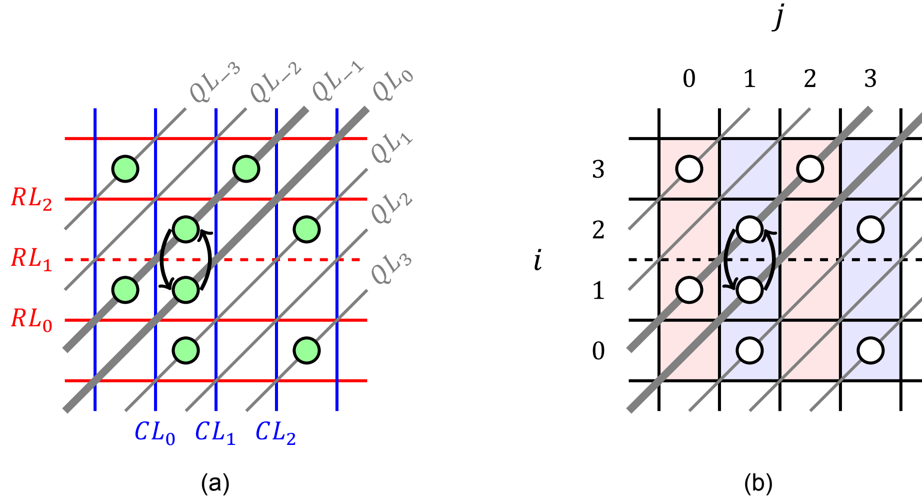


Figure 3.7: In this example a qubit in site $(1,1)$ is swapped with the qubit in site $(2,1)$. To do this QL_{-1} is set equal to the QL_0 and the barrier RL_1 is lowered.

During the operation the barriers around the sites (i, j) and $(i+k, j)$ must be raised (as shown in Equation 3.41, 3.42, 3.43 and 3.44). Otherwise, the movement of the qubit can not be predicted.

$$RL[i-1-(1/2)+(k/2)] = 0 \quad (3.41)$$
$$RL[i+(1/2)+(k/2)] = 0 \quad (3.42)$$
$$CL[j-1] = 0 \quad (3.43)$$
$$CL[j+1] = 0 \quad (3.44)$$

- **Qubit lines:** There should not be a difference in voltage between the qubit lines that passes through the sites (i, j) and $(i+k, j)$ (as shown in Equation 3.45). Otherwise, an unintended shuttle could happen and even a collapse of the state.

$$QL[j-i] = QL[j-(i+k)] \quad (3.45)$$

To keep the rest of qubits in rows i and $i+k$ from shuttling, we apply a higher voltage in the qubit line that passes through their site than the one passing through its adjacent empty site (as shown in Equation 3.46, 3.47 and 3.48).

$$\text{if } C[i,x] \neq -1 \text{ then } QL[x-(i+k)] < QL[x-i] \quad (3.46)$$
$$\text{if } C[i+k,x] \neq -1 \text{ then } QL[x-i] < QL[x-(i+k)] \quad (3.47)$$
$$\forall\, x \in [0..N-1] \setminus \{j\} \quad (3.48)$$

3.3.3.2. CPHASE gate

The second method works similarly by letting two qubits horizontally adjacent to each other interact. As in the previous method, the vertical barrier between them must be lowered and the qubit line that pass through both sites must be set to the same voltage. In this case, due to the difference in the Zeeman energy between columns, this method executes a CPHASE gate. However, since the fidelity of $\sqrt{SWAP}$ is higher than CPHASE [23] we will only use the $\sqrt{SWAP}$ to execute two-qubit gates. This means that for running any arbitrary quantum algorithm, a decomposition of the CPHASE gate into the supported gates is necessary.

3.3.4. Measurement

Based on the *Pauli exclusion principle*, two fermions (in this case electrons) can not be in the same quantum states in a quantum system. This means that two qubits with the same

quantum state can not be in the same site of the crossbar. We can take advantage of this principle to measure the state of a qubit. The technique we are going to use is based on the comparison of quantum states so, firstly, we need a qubit with a known quantum state. Traditionally, this qubit is called *ancilla* or *ancillary qubit* because is only used as an auxiliary qubit; unlike the rest of qubits which are called *data qubits* because they contain the information we need to measure.

The measurement has a different structure than the already defined operations. In particular, this operation is divided in *two phases*. By using the previous principle, these two phases use a technique called Pauli Spin Blockade (PSB) spin-to-charge conversion to measure the state of a qubit. From now on, the qubit to be measured will be referred as *target qubit*.

3.3.4.1. Phase One

In the first phase, to measure the quantum state of the data qubit we need to force a shuttle to the site of the ancillary qubit through the shuttle operation. If both qubits have different states, the shuttling will be successful. If their quantum states are the same, the data qubit will not be able to shuttle. In order for this whole procedure to work correctly, depending on the column in which the ancilla qubit is, it must have a predefined state of $|0\rangle$ or $|1\rangle$ [13]. Figure 3.8 shows an example of this phase.

Definitions: This first phase of the measurement shuttles the target qubit (the qubit we want to measure) in site (i, j) to a horizontal adjacent site $(i, j+k)$, where $k \in \{+1, -1\}$ (right site or left site, respectively). Based on [13] we expect a latency of $100ns$

Instructions: To execute this operation these instructions must be followed:

1. Lower the column barrier: $CL[j-(1/2)+(k/2)] = 1$.
2. Set voltage of the qubit line that passes through the destination site higher than the one that passes through the origin site: $QL[j-i] < QL[(j+k)-i]$.
3. Rise the column barrier: $CL[j-(1/2)+(k/2)] = 0$.

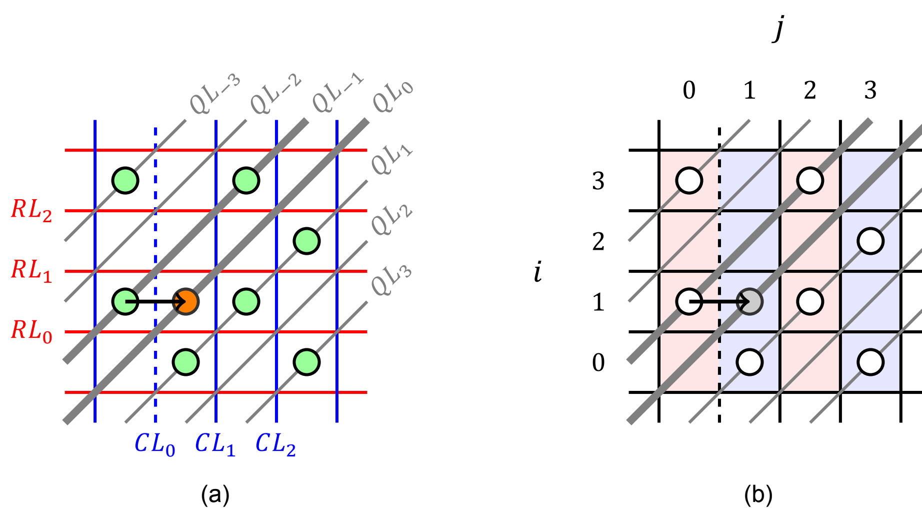


Figure 3.8: In this example a qubit in site $(1,0)$ is forced to shuttle into a site with an ancilla qubit in site $(1,1)$. To do this, the ancilla qubit must be in the spin down state ($|1\rangle$). In addition, the qubit line QL_{-1} is set lower than QL_0 and the barrier CL_0 is lowered.

Constraints: To correctly execute this operation, the following conditions must be met:

- **Qubits:** The ancilla qubit must be in the state $|0\rangle$ or $|1\rangle$ if it is in a odd or even column, respectively (as shown in Equation 3.49 and 3.50).

$$\text{if } (j+k) \bmod 2 = 0 \text{ then } S[i, j+k] = |1\rangle \tag{3.49}$$

$$\text{if } (j+k) \bmod 2 = 1 \text{ then } S[i, j+k] = |0\rangle \tag{3.50}$$

- **Barriers:** The vertical barrier between the sites (i, j) and $(i, j+k)$ must be lowered (as shown in Equation 3.51). Otherwise, the qubit will not be able to shuttle.

$$CL[j - (1/2) + (k/2)] = 1 \tag{3.51}$$

  During the operation the barriers around the sites (i, j) and $(i, j+k)$ must be raised (as shown in Equation 3.52, 3.53, 3.54 and 3.55). Otherwise, the movement of the qubit can not be predicted.

$$CL[j - 1 - (1/2) + (k/2)] = 0 \tag{3.52}$$

$$CL[j + (1/2) + (k/2)] = 0 \tag{3.53}$$

$$RL[i - 1] = 0 \tag{3.54}$$

$$RL[i + 1] = 0 \tag{3.55}$$

- **Qubit lines:** The voltage from the qubit line that passes through the $(i, j+k)$ site must be higher than the one passing through the (i, j) site (as shown in Equation 3.56). Otherwise, the qubit will be able to shuttle.

$$QL[j - i] < QL[(j+k) - i] \tag{3.56}$$

  To keep the rest of qubits in columns j and $j+k$ from shuttling, we must apply a higher voltage in the qubit line that passes through their site than the one passing through its adjacent empty site (as shown in Equation 3.57, 3.58 and 3.59).

$$\text{if } C[x, j] \neq \emptyset \text{ then } QL[(j+k) - x] < QL[j - x] \tag{3.57}$$

$$\text{if } C[x, j+k] \neq \emptyset \text{ then } QL[j - x] < QL[(j+k) - x] \tag{3.58}$$

$$\forall\, x \in [0..N-1] \setminus \{i\} \tag{3.59}$$

3.3.4.2. Phase Two

The result of the shuttle can be retrieved by checking if the data qubit is still in its original site. To do this, in the second phase, an RF carrier signal is sent through a qubit line while a horizontal barrier is lower. This will make the data qubit to shuttle back and fourth, from its site to a vertically adjacent site. This movement of the qubit can then be measured by the qubit line as a change in the signal. This means that if the qubit did not previously shuttled to the ancilla site, it will be detected through the qubit line and vice versa. A visual example is shown in Figure 3.9.

Definitions: This second phase of the measurement checks the presence the qubit in site (i, j) by sending an RF carrier signal through the qubit line that passes over the empty site $(i, j+k)$, where$k \in \{+1, -1\}$ (top site or bottom site, respectively). Based on [13] we expect a latency of $100ns$.

Instructions: To execute this operation these instructions must be followed:

1. Lower the row barrier: $RL[i] = 1$ or $RL[i-1] = 1$.

2. Send RF carrier signal through $QL[j-(i+1)]$ or $QL[j-(i-1)]$.
3. Read the RF carrier signal: if the reflected signal is different then the state of the qubit will be the same as the state of the ancilla qubit. If not, the state of the qubit will be different. (Note the state can only collapse to $|0\rangle \, or \, |1\rangle$)
4. Rise the row barrier: $RL[i] = 1$ or $RL[i-1] = 1$.

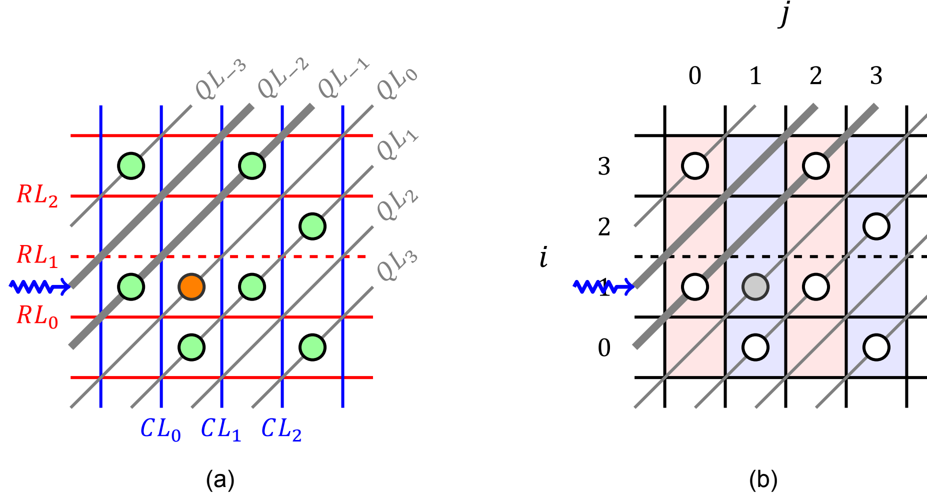


Figure 3.9: In this example the site in $(2,0)$ is used to check if the qubit shuttled to the site of the ancilla qubit or not. To do this a RF carrier signal is sent through QL_{-2} while RL_1 is lowered. In this case, the reflected signal will be different from the original; this means that the qubit did not shuttle to the site $(1,1)$, thus it must be in the same state as the ancilla ($|1\rangle$).

Constraints: To correctly execute this operation, the following conditions must be met:

- **Sites:** The site (above or below) adjacent to the site of the qubit been measurement must be empty (as shown in Equation 3.60). Otherwise, we will not be able to check the presence of the qubits.

$$C[i+1,j] = \emptyset \vee C[i-1,j] = \emptyset \tag{3.60}$$

- **Qubits:** There can not be two qubits vertically adjacent to each other between the rows i and $i+k$ (except in the column that we want to execute the operation, as shown in Equation 3.61). Otherwise, those vertically adjacent qubits will interact and change of state or even collapse.

$$C[i,x] = \emptyset \vee C[i+k,x] = \emptyset \text{ where } \forall\, x \in [0..N-1] \setminus \{j\} \tag{3.61}$$

- **Barriers:** The horizontal barrier between the sites (i,j) and $(i+k,j)$ must be lowered (as shown in Equation 3.62). Otherwise, the qubit will not be able to shuttle.

$$CL[j-(1/2)+(k/2)] = 1 \tag{3.62}$$

The barriers around the sites (i,j) and $(i+k,j)$ must be raised (as shown in Equation 3.63, 3.64, 3.65 and 3.66). Otherwise, the movement of the qubit can not be predicted.

$$RL[i-1-(1/2)+(k/2)] = 0 \tag{3.63}$$
$$RL[i+(1/2)+(k/2)] = 0 \tag{3.64}$$
$$CL[j-1] = 0 \tag{3.65}$$
$$CL[j+1] = 0 \tag{3.66}$$

- **Qubit lines:** To keep the rest of qubits in rows i and $i+k$ from shuttling, we must apply a higher voltage in the qubit line that passes through their site than the one passing through its adjacent empty site (as shown in Equation 3.67, 3.68 and 3.69).

$$\text{if } C[x, j] \neq \emptyset \text{ then } QL[(j+k)-x] < QL[j-x] \tag{3.67}$$
$$\text{if } C[x, j+k] \neq \emptyset \text{ then } QL[j-x] < QL[(j+k)-x] \tag{3.68}$$
$$\forall\, x \in [0..N-1] \setminus \{i\} \tag{3.69}$$

3.4. Conflicts

A *conflict* happens when two or more constraints are incompatible with each other. As mentioned previously, these constraints exist to maintain a valid configuration. This means that if a constraint is violated then two types of consequences could happen: a *side effect* and an *undecidable configuration*.

Side effects are events related to the qubits that were not intended to happen. Whereas an *undecidable configuration* is a specific combination of controls and qubit positions that does not allow a way to predict the behaviour of the qubits. Both of these concepts are explained in detail in the following sections.

Based on these types of consequences, we can differentiate two groups of conflicts:

- **Soft conflicts**: these conflicts happen when their consequence affects a qubit in a way that can be reversible. Inside this group we have defined the *solvable side effects*.
- **Hard conflicts**: these conflicts creates a consequence that leads to an unintended collapse of state or unpredictable qubit movement. Inside this group we have defined the *unsolvable side effects* and *undecidable configurations*.

At the stage of scheduling or mapping a quantum algorithm into an architecture, we need to distinguish these two types of conflicts. This is because, in some cases, it might be useful to suffer a conflict of constraints in favor of creating a better execution time. In particular, it might be beneficial to suffer a *soft constraint* which would be solved in a future step. The hard conflicts must be avoided.

3.4.1. Side effects

There are three different types of side effects that could happen if the constraints are not followed. It is worth mentioning that any of the following side effects can happen concurrently due to the nature of the crossbar architecture.

3.4.1.1. Solvable: Shuttling

A qubit unrelated to the current operation could be accidentally shuttled to another site. Figure 3.10a shows an example where a qubit (which is not part of the operation) is affected. From section 3.3.1 we can see that the violated constraint is: $QL_1 > QL_2$. Note that in the case where the qubit is accidentally shuttled to another column, depending on the timing it could also produce a change of state.

3.4.1.2. Solvable: Change of state

As already mentioned in the previous section, one of the ways we can have an accidental change of state in a qubit is by shuttling it to an adjacent column. However, we can also have a change of state if a qubit is shuttled in the middle of the semi-global rotation scheme.

Another change of state can be done through a two-qubit gate operation. This situation can happen when two adjacent qubits have equal QL voltages and the barrier between them is lowered. This could execute a $\sqrt{SWAP}$ or a $CPHASE$ gate when the qubits are vertically adjacent or horizontally adjacent, respectively.

3.4.1.3. Unsolvable: Collapse of state

As explained in section 3.3.4, by using the *spin-to-charge* conversion scheme [13], the target qubit needed to be measured is collapsed. However, we can also have a collapse of a qubit due to unintended shuttles. For example, Figure 3.10b shows a case of shuttling which, as a side effect, shuttles another qubit into a site that is occupied by a qubit.

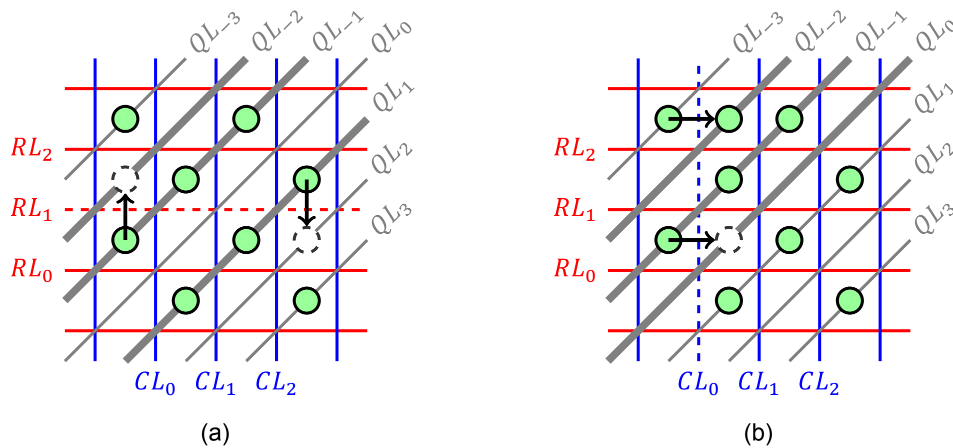


Figure 3.10: **(a)** In this example a qubit in site $(1,0)$ is shuttled to site $(2,0)$. But since RL_1 is lowered and the $QL_1 < QL_2$, as a side effect, the qubit in site $(2,3)$ is shuttled to the bottom. The constraint $QL_1 > QL_2$ was violated. **(b)** In this example a qubit in site $(1,0)$ is shuttled to site $(1,1)$. But since CL_0 is lowered and the $QL_{-3} < QL_{-2}$, as a side effect, the qubit in site $(3,0)$ is shuttled to the right. Based on the constraints, this shuttling is not be permitted because there are two qubits in the same row between the involved columns.

3.4.2. Undecidable configurations

Not every combination of controls can give a valid and decidable configuration in the crossbar. There are some cases in which one can not predict what might happen on a qubit. These cases are called *undecidable configurations*. Note that these configurations take place when lowering more than one barrier line (vertical or horizontal). Thus, if the instructions for each operation are followed correctly, these unpredictable situations should only happen when trying to execute two or more operations in parallel.

However, there are cases in which the lowering of two or more barrier lines do not cause an undecidable configuration. The following examples explain the most simple undecidable configurations. To recognise whether a crossbar is in an undecidable configuration one must identify if any of the following examples are part of the configuration.

3.4.2.1. Undecidable configuration 1: Rectangle

When two vertical or two horizontal barriers adjacent to each other are lowered at the same time, they produce a space of 3 horizontally adjacent empty sites, we will call this group of sites a *rectangle*. None of the operations mentioned previously require lowering more than one barrier. Hence, a rectangle is never produced when executing operations sequentially. Having this amount of sites open to each other can produce undecidable configurations, such as the one shown in Figure 3.11a. In this example, since the voltages of QL_{-2} and QL_0 is higher than the one passing through the site $(2,1)$, then we can not predict the behaviour of

the qubit in that site, since it can shuttle to the left site or to the right site. In addition, this example shows a situation where two qubits (in sites $(3, 0)$ and $(3, 2)$) are forced to shuttle to the same site $(3, 1)$. Since we can predict the movement of the qubits this not an undecidable configuration, but is a situation that needs to be avoided. Otherwise, these qubits will collapse.

3.4.2.2. Undecidable configuration 2: Square

When a vertical and a horizontal barriers are lowered at the same time, the place in which they intersect creates a space of 4 sites, we will call this group of sites a *square*. In this case, we can not predict the movement a qubit if the voltages at its direct neighbours (top, left, right or left) inside the square are equal. For example, Figure 3.11b has two qubits of which two adjacent qubit lines have the same voltage. Analogous to the rectangle example, the movement of the qubits are not predictable. In addition, in the case where both qubits shuttle to the same site, they would both collapse, losing their states.

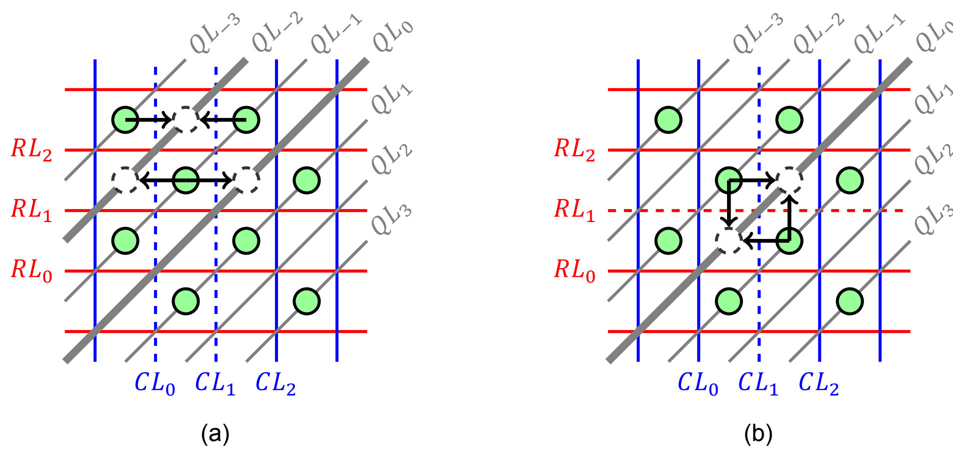


Figure 3.11: **(a)** Example of an undecidable configuration with a rectangular shape. In this example there is more than one case of ambiguity. If $QL_{-2} > QL_{-1}$ and $QL_{-1} < QL_0$, then we can not predict where the qubit in $(2, 1)$ is going to shuttle. **(b)** Example of an undecidable configuration with a squared shape. In this example the behaviour of two qubits are not predictable. Since $QL_{-1} < QL_0$ and $QL_1 < QL_0$ both qubits (at $(1, 2)$ and $(2, 1)$) could shuttle to two different sites.

In both of these cases, note that if the sites inside these “shapes” were empty there would not be any ambiguity, since there is no qubit to predict its movement. Figure 3.12b shows such example with two horizontal parallel shuttles. In this case, even with two vertically adjacent lowered barriers, the movement of the qubit and still be predicted. However, for simplicity, we will also consider these cases as an invalid configuration.

From the two undecidable configurations that we have defined, we can see that any crossbar with two lowered adjacent barriers can be categorized on the “rectangle” shape and, any crossbar with a lowered horizontal and vertical barrier can categorized in the “square” shape. Therefore, any configuration that has any of these two “shapes” can be labeled as undecidable. If the execution of the operations follow all the constraints defined previously, we should never encounter these undecidable configurations.

3.4.2.3. Decidable configurations

As mentioned before, there are trivial configurations in which the electron’s shuttling can be predicted even with more than 2 barriers lowered. Thanks to these cases, it is possible to correctly execute operations in parallel. As an example, Figure 3.12a shows how a parallel horizontal shuttling can be executed with a decidable configuration.

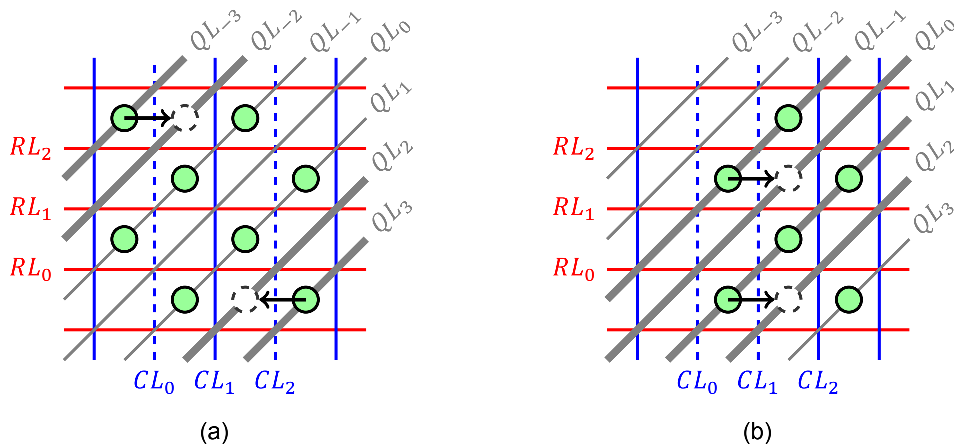


Figure 3.12: Two examples of two horizontal parallel shuttles with two qubits. **(a)** In this case, neither of the barriers lowered are crossing or adjacent to each other, so the movement of the qubit can be predicted. For clarity, the qubit lines follow this condition: $QL_{-3} < QL_{-2} < QL_2 < QL_3$. **(b)** In this scenario, lowering two adjacent barriers can still give a decidable configuration. For clarity, the qubit lines follow this condition: $QL_{-1} < QL_0 < QL_1 < QL_2$.

3.5. Gate Set Decomposition

The crossbar architecture does not natively support every quantum gate. As explained in the previous sections, this architecture is capable of executing any single-qubit rotation, but it can not directly execute any two-qubit gate. This section shows the decomposition of unsupported two-qubit gates into native gates. From the gate decompositions shown in Figures 3.15 and 3.17, there is a dependency on the decomposition of the CNOT gate. Thus, an efficient decomposition of the CNOT is important in order to have a fast and high fidelity execution of an algorithm.

- **CNOT (Controlled-X):** For the CNOT decomposition (in Figure 3.13), note that it is not possible to only use one $\sqrt{SWAP}$. One of the reasonings behind this is that the $\sqrt{SWAP}$ can not perform a maximal entangling state on two-qubits without adding an additional $\sqrt{SWAP}$ gate, whereas, the CNOT gate can, by applying it only once [35]. And the use of Z, S & T gates is preferred over other single-qubit gates. This decision has been made for two reasons: firstly, these gates can be easily parallelized; secondly, the time needed to execute phase shift gates is shorter than the rest.

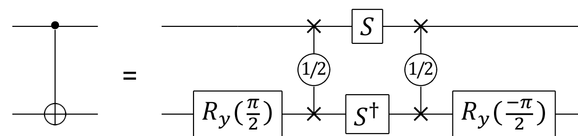


Figure 3.13: A quantum circuit for executing the $CNOT$ gate based on supported gates in the crossbar architecture.

- **CPHASE (Controlled-Z):** As mentioned previously, we have given priority to the use of $\sqrt{SWAP}$ over the CPHASE gate because it has higher fidelity, so it is relevant to give a decomposition for the CPHASE. However, if the overall fidelity of this decomposition (in Figure 3.14) is lower than the fidelity of the CPHASE gate, it would be more efficient to use the native CPHASE.

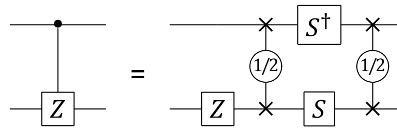


Figure 3.14: A quantum circuit for executing the $CPHASE$ gate based on supported gates in the crossbar architecture.

- **Toffoli (Controlled-CNOT):** For the $Toffoli$ decomposition (in Figure 3.15) we have use the decomposition from [38] based on the CNOT gates, H gates and T gates. Note that the CNOT gates are decomposed into native gates based on the decomposition from Figure 3.13.

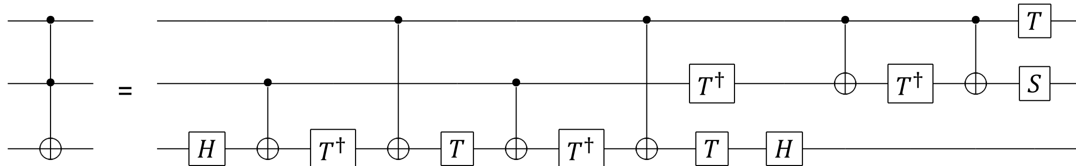


Figure 3.15: A quantum circuit for executing the Toffoli gate based on supported gates in the crossbar architecture.

- **SWAP:** The decomposition of the SWAP gate (in Figure 3.16) is straight forward. Since we have the support for the $\sqrt{SWAP}$ gate, we can apply two of them to obtain the SWAP gate.

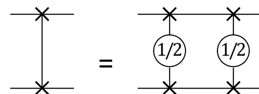


Figure 3.16: A quantum circuit for executing the SWAP gate based on supported gates in the crossbar architecture.

- **Fredkin (Controlled-SWAP):** Finally, the decomposition for the Fredkin gate (Figure 3.17) depends on the decomposition of the Toffoli gate. However, there might be a better decomposition for the Fredkin gate where it uses less gates by using directly the $\sqrt{SWAP}$ gate instead of relying on the decomposition of the Toffoli gate.

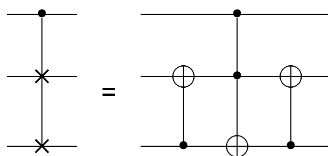
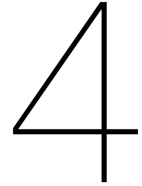


Figure 3.17: A quantum circuit for executing the CSWAP gate based on supported gates in the crossbar architecture.



Mapping Implementation

In this chapter we will introduce the mapping and routing strategies based on the constraints mentioned in chapter 3. In section 4.1 we will describe the initial placement strategy. In section 4.2 we will discuss the routing algorithm implemented in the compiler. Finally, in section 4.4 we will explain an additional layer of decomposition that is needed to comply with the crossbar constraints.

4.1. Initial Placement

As mentioned in chapter 2, the initial placement is an NP-complete problem [39]. Generally, there are three possible methods to address this problem.

Firstly, the most basic strategy is to do a random initial placement. In other words, this method will assign a virtual qubit to a random physical qubit. Although this is a naive method, it is the fastest way to test other components of the compiler, such as the routing algorithm. On top of this, it can also be used as a baseline to test against other placement methods.

Secondly, a popular approach is to model the initial placement problem as a mathematical optimization problem and then use software solvers to find an exact solution. This approach has two disadvantages: firstly, it takes a long time to obtain the optimal solution; secondly, it will only find a solution in a relative short amount of time if the quantum architecture and circuit is small enough. For relative big quantum circuits using more than 50 qubits, this approach is not feasible.

Thirdly, like with many NP-complete problems, the most suitable approach is to use an heuristic algorithm. To implement this approach it is common to start with a random initial placement and then use a search algorithm with an heuristic-cost function to find a better placement. This heuristic-cost function is what differentiates an heuristic algorithm from another [22].

Finding a new and better heuristic approach for the initial placement problem is out of scope of this thesis. In the experiments of chapter 6, we will use a random initial placement and an exact algorithm. For the exact algorithm, we will use an Integer Linear Programming (ILP) algorithm to find the optimal placement of the qubits. This algorithm minimizes the number of qubit movements required to run the circuit with the topology constraints. As previously mentioned, this type of approach will take a relative long time for large quantum circuits. We have tested that, in general, for circuits with more than 10 qubits the algorithm takes more than 5 minutes to find a solution. For this reason, we will only use this exact

algorithm for quantum circuits with maximum 10 qubits. Thus, the maximum crossbar size for these circuits is 5x5. And for the rest of experiments with a higher number of qubits, we will use a random initial placement.

4.2. Routing

Based on the initial placement, in general, it is unlikely that all the two-qubit gates of the quantum circuit can be directly performed without the need of moving qubits. Routing is required to find the set of movement operations to perform a two-qubit gate between non-adjacent qubits.

In this section, we will first describe the routing method for the crossbar architecture. Then we will introduce some ideas from the routing in superconducting architectures to use them in the routing implementation. Then we will discuss different approaches to implement the routing in the crossbar architecture. Finally, we will explain the approach taken to route qubits in our experiments.

4.2.1. Crossbar Topology

The topology of the crossbar can be represented as a graph where each node is a site of the grid and each edge is a possible shuttle between sites. A visual representation of this graph is shown in Figure 3.1b, which is, essentially, a 2-D upright square lattice. However, note that the two-qubit gates can only be performed between two vertically adjacent qubits (for the $\sqrt{SWAP}$ gate) and two horizontally adjacent qubits (for the CPHASE gate). Thus, the coupling graph that represents the possible two-qubit interactions between sites is different than the topology graph. Figure 4.3 shows two examples of a coupling graph (based on $\sqrt{SWAP}$ gates and CPHASE gates) in a crossbar of size 5x5. Since we are going to use only the $\sqrt{SWAP}$ gates for this thesis, because of a higher fidelity, we will only focus on the coupling graph for the $\sqrt{SWAP}$ gate (Figure 4.3a). Unlike other architectures, such as IBM QX2 [46] (as shown in Figure 4.1), the coupling graph and the topology (routing) graph of the crossbar architecture are different. Furthermore, the coupling graphs shown in Figure 4.3 uses undirected edges, this means that the direction of the two-qubit gates does not matter. This is evident, since the $\sqrt{SWAP}$ and CPHASE gates can be executed in reverse (exchanging the control qubit with the target qubit) and the result will be the same. However, the situation is different when the supported two-qubit operation is the CNOT gate and the execution of such gate is restricted in directions. For example, Figure 4.1 shows the coupling graph of the IBM QX2, where the direction of the CNOT gates is restricted based on the direction of the edges.

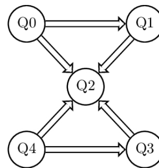


Figure 4.1: The coupling graph of the IBM QX2 [46]. The nodes represent the physical qubits and the edges represent the possible interactions. The direction of the edge represents the direction in which the two-qubit gates must be applied.

Moreover, in superconducting architectures, such as the IBM QX2, the coupling graph is the same as the topology graph because the method used to move qubits is based on on the SWAP gates. Thus, the two qubits must interact in order to move a qubit state through the topology. However, as previously explained, in spin qubits, the method used to move qubits around is the *shuttle* operation. In this case, the qubit is physically moved through each site without applying any quantum gate. The shuttling method has some advantages and disadvantages compared to the swap method.

On one hand, the shuttling method can be seen as a more complex scenario, since the

qubit must be physically moved to be able to execute a two-qubit gate, whereas, in the swap method, the qubit can execute a two-qubit gate with its neighbours. In addition, with the swap method, one can always find a path from one qubit to its target. But with the shuttle method, the qubit must find a path that has no other qubits. If there is a qubit blocking the path, it is necessary to move at least an additional qubit to unblock it. This additional qubit must find a new path to unblock the path of the previous qubit. Figure 4.2 shows an example of a qubit that can not be moved to its target site without moving another qubit. On top of this, like the rest of quantum operations, the shuttle operation also has errors.

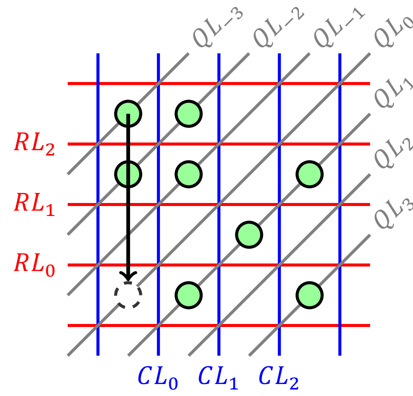


Figure 4.2: An example of a blocked path in the crossbar architecture

On the other hand, the shuttling method is faster (e.g. $10ns$ [13]) than applying a two-qubit gate such as a CNOT gate in superconducting architecture (e.g. $80ns$ [4]). In addition, the shuttle method allows to execute parallel operations easier than the swap method. This is because the shuttle operation does not block the use of another qubit, whereas, the SWAP gate is applied on two qubits. Thus, without taking other constraints into account, the swap method could potentially impose more limits on the parallelism of quantum operations.

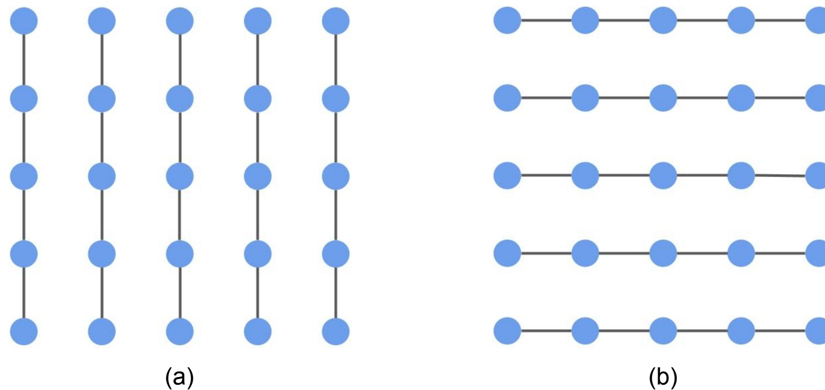


Figure 4.3: Two coupling graphs in a crossbar of size 5x5 **(a)** based on $\sqrt{SWAP}$ gates and **(b)** based on CPHASE gates. The nodes represent the sites and the edges represent the possible two-qubit gate interaction.

4.2.2. Crossbar Configuration

Although the crossbar has more configurations proposed in [13], analysing the different configurations and the ratio between data qubits and ancilla qubits is beyond the scope of this thesis. In this thesis, we will only use the *idle* configuration, where the crossbar is half full of qubits and there is one empty site between every qubit and its nearest neighbouring qubit. As explained before, having the crossbar half full is a good approach since it optimizes the number of control lines per qubit. In addition, it allows the qubits to be shuttled more easily

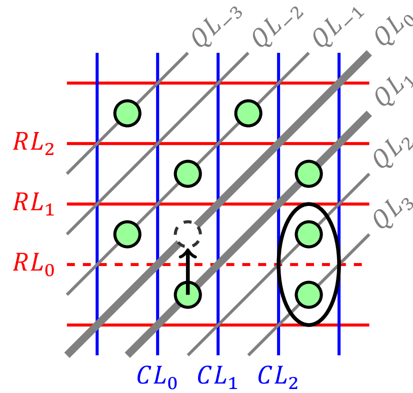


Figure 4.4: Example of a deadlock situation

than if they were all packed in one corner.

4.2.3. Routing Strategy

Based on the previous explanation of the shuttling operation, we now elaborate the strategies to move qubits around on the idle configuration. These strategies must take into account all the restrictions mentioned in section 4.2.1.

There are many routing approaches for a 2-D upright square lattice. However, the routing based on shuttles is a complex problem and can be modeled based on the *multi-agent path finding*. Where each qubit needs to be moved to a different site through a path that does not block the other moving qubits. This problem is NP-complete and there is no straight forward solution.

Besides, even with a good strategy, a deadlock might happen in the scheduling process. To illustrate this problem, Figure 4.4 shows a qubit that needs to be shuttled to the top site. However, there are two vertically adjacent qubits in the same row. Based on the shuttling constraints, this qubit is not able to be shuttled. Otherwise, the two vertically adjacent qubits will interact and their quantum state will change. If this scenario occurs and this shuttle instruction is the only one available, the scheduler will enter into a deadlock. These scenarios depend not only on the routing strategies but also on the scheduling process. A solution to deal with this issue is needed.

In the scheduler, we can check if there is a deadlock by checking if a conflict of constraints happens when we try to schedule an instruction sequentially. If this happens the only conflict that could happen is with the crossbar state itself. In other words, the position of the qubits or the state of the control lines are conflicting with the constraints of the instruction. Based on this “deadlock check” function, there are some approaches that we can take to solve this problem:

Avoid the deadlock: This deadlock can be avoided by maintaining always the idle configuration after each operation and schedule the problematic instructions that produces that deadlock in a sequential manner.

Backtrack: If the scheduler finds a deadlock, it can unschedule the previous instructions until the last routing was made. Then retries to schedule the instructions in a different way. If it encounters with another deadlock then it will repeat the same process. If after repeating this process the scheduler has backtracked to the first instruction, then the routing process must change the routing of the circuit.

Suffer a side effect: In chapter 3 we have explain the consequences of not following the constraints. So instead of backtracking we can go forward in time, suffering a side effect. If the constraints allow us, we can suffer a solvable side effect by executing a $\sqrt{SWAP}$ between the qubits. Naturally, this side effect gate can be reverted by applying 3 more times the $\sqrt{SWAP}$ gates between those qubits, since $SWAP \times SWAP = I$. If the constraints do not allow us to suffer this solvable side effect, we can suffer a collapse of the qubit states if those qubits were no longer used in the rest of the circuit. If none of these are allowed, then we could try the backtrack approach.

Although avoiding the deadlock seems a naive approach, this might be the only alternative if the constraints do not allow to use the other strategies. Also the backtrack method seems like the best approach but it is also the most computational intensive and the most complex to implement.

4.2.4. Routing Implementation

In this thesis, we have only implemented the most straight forward solution: to avoid the deadlock. To do this, as mentioned previously, we need two make tow changes: firstly, we need to maintain the idle configuration after each operation and, secondly, we need to execute the instructions that can produce a deadlock in a sequential manner. The problematic instructions are: $\sqrt{SWAP}$ and shuttles, since they are the only ones that can produce a similar situation to the one shown in Figure 4.4.

Firstly, to maintain the idle configuration, we need to use a new topology (shown in Figure 4.5) on top of the current one.

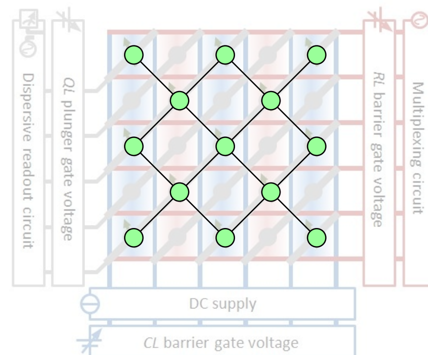
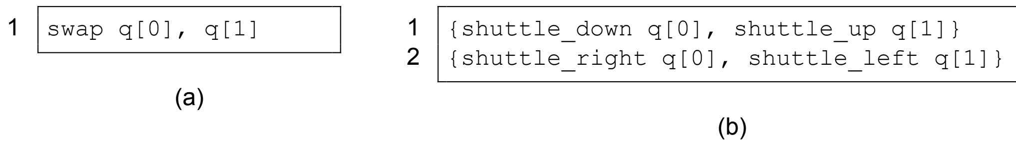


Figure 4.5: The new topology used for our routing strategy

Note that the new topology has a qubit in each site/node but is still equivalent to the old topology. In this new topology, we can route the qubits by physically swapping adjacent qubits. This means that by doing a SWAP we would be performing effectively 4 shuttles in the crossbar (as shown in Figure 4.6). For example, Figure 4.7a shows the shuttle operations required to swap two adjacent qubits in the new topology. Although this solution creates an overhead in terms of depth and gates, the shuttles required to execute this swap can be done in parallel, as seen in Figures 4.7b and 4.7c. So the additional depth overhead is only 2 shuttles.

Secondly, to sequentially schedule the instructions that produce a deadlock we need to force the scheduler to not schedule instructions in parallel with the problematic ones. To do this, we divide the kernel of the quantum algorithm into three kernels every time a problematic instructions is found. In such a way that the problematic instruction is isolated in one kernel. Thus, no other instruction can be scheduled at the same time. By doing this, the scheduler essentially schedules three different programs but maintaining the crossbar state across them. We have called this strategy the “division of kernels”. For example, Figure 4.8



```
1 swap q[0], q[1]
```

(a)

```
1 {shuttle_down q[0], shuttle_up q[1]}
2 {shuttle_right q[0], shuttle_left q[1]}
```

(b)

Figure 4.6: An example of the decomposition of the virtual swap into 4 shuttles. **(a)** The swap instruction. **(b)** The decomposed shuttle instructions.

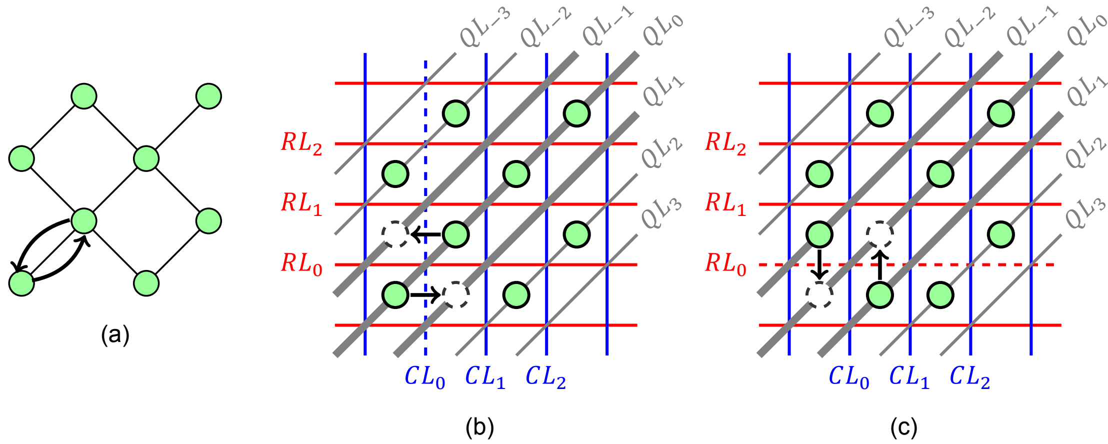
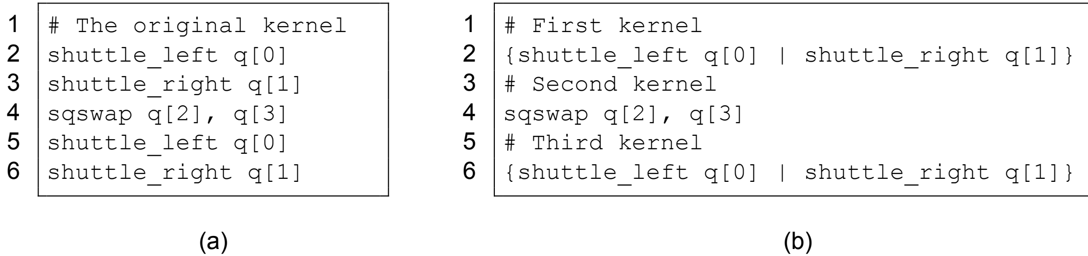


Figure 4.7: An example of the virtual swap (or swap-like shuttle) in the crossbar layout. **(a)** Representation of the new topology executing the swap-like shuttle. **(b)** First step in the swap-like shuttle. **(c)** Second step in the swap-like shuttle.

shows how this strategy would divide the kernel when it encounters a $\sqrt{SWAP}$. Note that in the example, the shuttles at line 4 and 5 from the input are not scheduled while the $\sqrt{SWAP}$ happens since they are in a different kernel.



```
1 # The original kernel
2 shuttle_left q[0]
3 shuttle_right q[1]
4 sqswap q[2], q[3]
5 shuttle_left q[0]
6 shuttle_right q[1]
```

(a)

```
1 # First kernel
2 {shuttle_left q[0] | shuttle_right q[1]}
3 # Second kernel
4 sqswap q[2], q[3]
5 # Third kernel
6 {shuttle_left q[0] | shuttle_right q[1]}
```

(b)

Figure 4.8: An example of division of kernels with a $\sqrt{SWAP}$ gate. **(a)** The input instructions. **(b)** The output instructions after diving the kernel.

For our experiments, we will use the routing algorithms implemented in *OpenQL* to find the shortest path [2]. There are three routers:

- The **base** router will find the shortest paths based on the Manhattan distance and then it will select a random one.
- The **minextend** router will also find the shortest paths, but in this case it will select the one which minimally increases the circuit depth.
- The **minextend-rc** router will also find the shortest paths, but it will consider the constraints of the architecture when selecting the path that minimally increases the circuit depth.

Note that the crossbar architecture uses two qubits for measurement: a data qubit and an ancilla qubit. Therefore, the ancilla qubit that will be used needs to be specified. This means

that, for any measurement operation, the compiler must find an ancilla qubit available and then move the data qubit and the ancilla to be adjacent. To simplify the implementation, whenever the compiler encounters a measurement instruction, it will use the ancilla qubit provided by the user. For example, to measure qubit 1, the user of the compiler must include the ancilla qubit that wants to use like this: `measure q[1], q[8]`. In this case, qubit 8 is the specified ancilla qubit.

4.3. Scheduling

The next step in the compiler is the scheduling of instructions. Although the scheduler that we are going to use in this thesis is already built in *OpenQL* [2], there are some points worth mentioning regarding its implementation.

Firstly, is important to highlight how the scheduler works. The constraints of the crossbar architecture are translated into resources for each operation. This means that an instruction can be scheduled only if its required resources are available. The scheduler makes sure these resources are available before scheduling to comply with the constraints. Each quantum chip has different resources. In the case of the crossbar, we have defined a resource for each type of constraint, including: qubits, sites of the crossbar, vertical barriers, horizontal barriers, qubit lines and RF pulses for single-qubit gates.

Note that if we need to schedule multiple single-qubit gates with the same rotation, we can use the same RF pulse to execute these instructions. For example, the same RF pulse can be used for performing the instructions `X q[0]` and `X q[1]`, allowing one-qubit gates to be executed in parallel.

Regarding the implementation of the scheduler, there are two main issues that we need to tackle in order to compile a circuit into the crossbar architecture.

Firstly, the scheduler does not have any context of the operation. This means that if it is trying to schedule a gate, it will check if the resources are available but only for that gate. For example if the scheduler is scheduling an X gate from the semi-global rotation, it will not know that the gate is part of the semi-global rotation. It is the reason why we can not divide the input instructions (cQASM code) into the opcodes defined in [13]. For example, if we wanted to schedule the opcode $V[0]$ (to lower the vertical barrier 0), we would lose the context that the opcode is part of a two-qubit gate (or shuttling). Thus, we will lose the constraints of these instructions and there will be no way to check if the instruction can be scheduled. This is an important point because, for this reason, the output of the compiler is cQASM code with additional gates (such as `shuttle_left`).

Secondly, *OpenQL* has not implemented a way to check if there is a deadlock in the scheduling process. The scheduler assumes that the only conflict that can happen is between instructions. However, as previously explained, the deadlock is a conflict between an instruction and the crossbar state. This means that, the scheduler will try to schedule an instruction indefinitely if it finds a deadlock. In the previous section, we have mentioned that we can implement a function to detect a deadlock and use one of the three proposed solutions. However, there is no easy way to implement a *backtrack* approach based on the current version of the scheduler.

4.4. Mapping Decomposition

During the mapping process, depending on the target architecture, it might need additional steps before the output can be sent for execution on the quantum chip. In the case of the crossbar architecture, there is a step that transforms and decomposes the gates so that they follow the constraints. In addition, we must do other decompositions, due to the new topology added in section 4.2. Based on their respective operations, these decompositions

can be divided into the following:

One-qubit gates: As explained previously, when we execute a one-qubit gate (except the phase shift gate) we must apply the semi-global rotation scheme. This process requires to add more gates in the circuit. Figure 4.9 shows an example of applying an X gate to qubit 1, which is then decomposed into 4 gates to complete the semi-global rotation scheme. Thus, each one-qubit gate (except the phase shift gate) produces an overhead of 3 more gates. Also note that in line 3 of Figure 4.9b, the inverse gate of the `X q[1]` gate is applied to a qubit that belongs to the same column (`q[0]`).

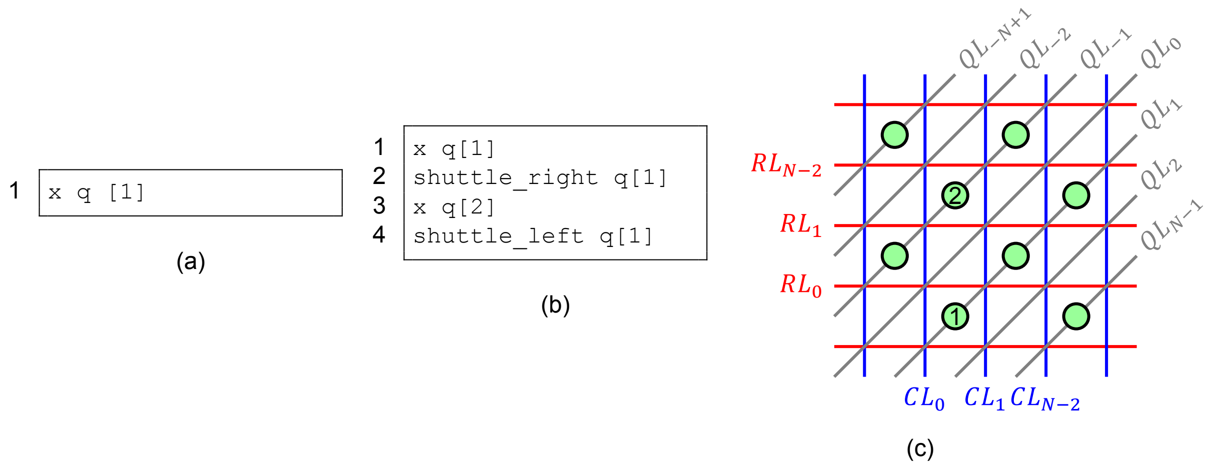


Figure 4.9: An example of decomposing a one-qubit gate. **(a)** The input instruction. **(b)** The output instructions after decomposition. **(c)** The crossbar configuration for the execution of the semi-global-rotation.

Phase shift gates: We have seen that the semi-global rotation method can also be used to execute phase shift gates. It is necessary to identify which phase shift gates that are being executed using the shuttling method. This can be done just by simply transforming the gate `z q[1]` into `z_shuttle q[1]` .

$\sqrt{SQSWAP}$ **gates:** Additional shuttles will be added in order to execute a two-qubit gate, resulting in mapping overhead. In particular, we need to add two shuttles: the first one makes the qubits adjacent and the second one moves the qubits back to their sites. Figure 4.10 shows an example of a $\sqrt{SWAP}$ gate being executed on qubits 1 and 2. Note that these shuttles are another source of overhead.

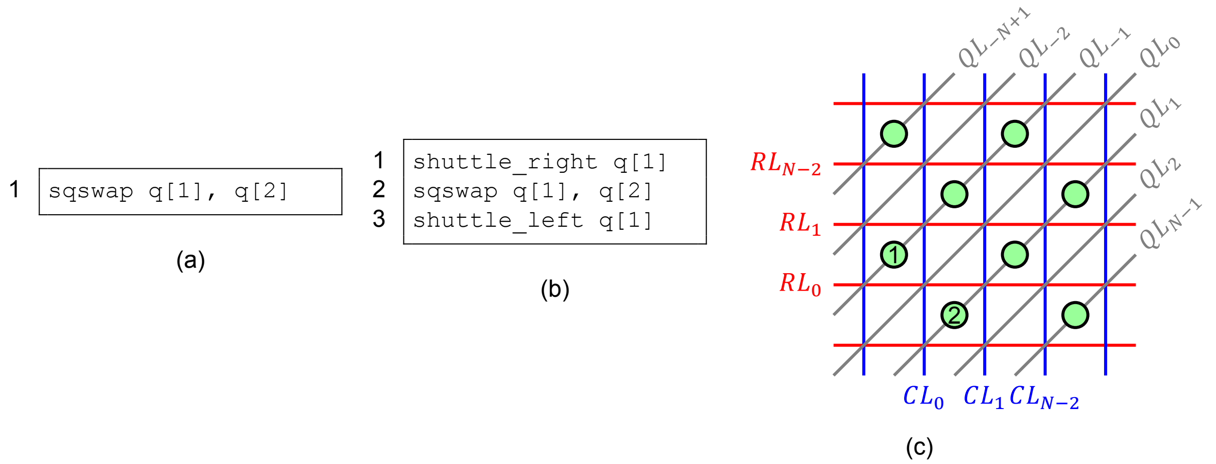


Figure 4.10: An example of decomposing a $\sqrt{SWAP}$. **(a)** The input instruction. **(b)** The output instructions after the decomposition. **(c)** The crossbar configuration for the execution of the $\sqrt{SWAP}$.

Measurement: Similarly to the $\sqrt{SWAP}$ gate decomposition, the measurement needs shuttles to align the data qubit with the ancilla qubit. To do this we need to shuttle the qubit or the ancilla in the same row and then move them back to their original sites. Figure 4.11b shows an example of a measurement on qubit 1 using qubit 2 as the ancilla. Note that if the state of the data qubit is different than the state of the ancilla, then the final `shuttle_up` instruction in line 4 of Figure 4.11b will not move the data qubit up, since it is now in the site of the ancilla.

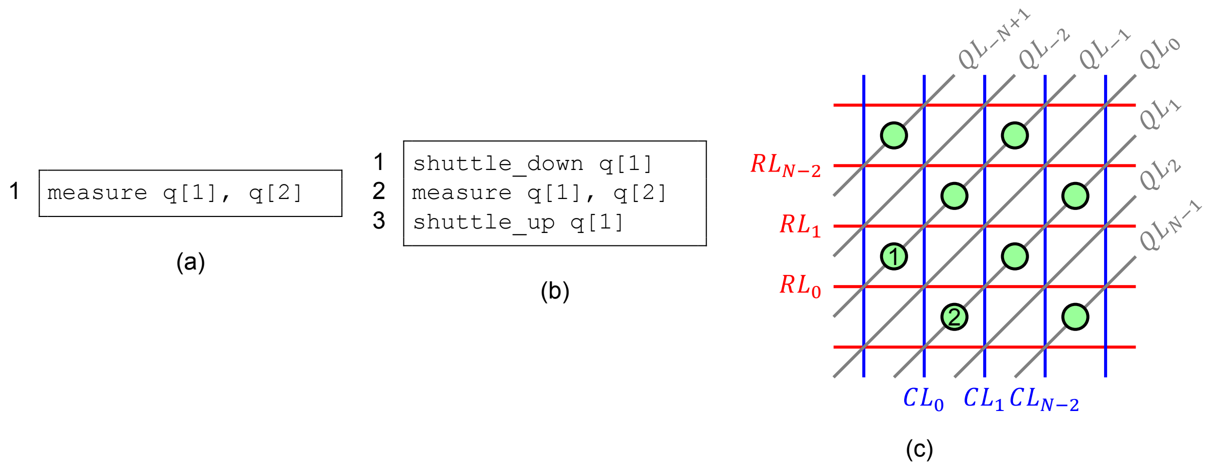


Figure 4.11: An example of decomposing a measurement gate. **(a)** The input instruction. **(b)** The output instruction after the decomposition. **(c)** The crossbar configuration for the execution of the measurement gate.

5

Simulation framework

In this chapter, we will go through the environment needed to run the experiments in chapter 6. Since there is a large amount of features that can be analyzed in this architecture, this section will clarify what are the characteristics to be analyzed and explain the decisions made during the building of this framework.

5.1. Framework overview

In this section, we will give an overview of the framework used in the experiments.

This framework can be divided into two modules: the compiler and the verification program. The compiler takes any quantum circuit coded in cQASM and outputs QASM code for the crossbar architecture. Unlike cQASM, the compiler output contains specific instructions used for mapping, such as the *shuttle* instruction. On the other hand, the verification program takes this compiled QASM and outputs an error if there is a conflict of constraints, an undecidable configuration or a parsing error. More details will be explained in section 5.2.

Regarding the compiler, Figure 5.1 shows a visual representation of the compilation. This process requires 3 types of input:

- **Circuit input**: a quantum algorithm is described by the cQASM language and this representation is hardware-agnostic.
- **Crossbar parameters**: these parameters include the number of qubits, the size of the crossbar, the position of the qubits and the gate decomposition, and they are encoded in a JSON file.
- **Resource constraints**: the resource constraints of the operations decoded in C++. In this way, the scheduler can use them without needing to do any additional parsing of files.

Note that, in every step of the compilation process, a new QASM file is generated. For example, the gate decomposition module outputs a QASM file with the gates decomposed into the supported primitive operations of the crossbar. These files serve two purposes: the main one is to use it as input for the next stage, and the second one is to analyze it in the experiments. In other words, having a different QASM file for each step, will allow us to have a more comprehensive study of the compilation process. The final output is the compiled cQASM that consists of the primitive operations (e.g. shuttling supported by the crossbar) and can be passed to the verification program.

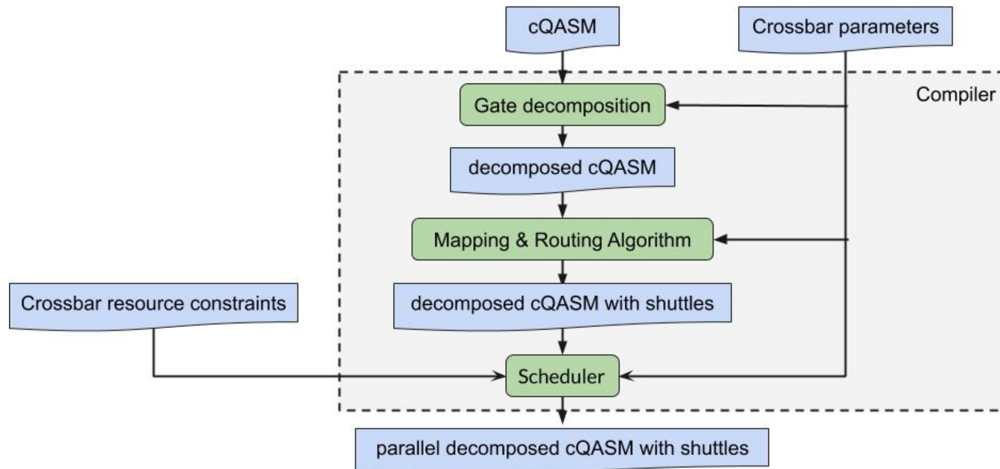


Figure 5.1: A diagram of the compiler overview

5.2. Verification Program

In this section, we will explain the additional program we have developed to check that the output of the compiler is compatible with the constraints of the crossbar. At first, one might think that if the compiler is correctly implemented then one can trust its output. However, there are two main reasons why we should not blindly trust the output of the compiler.

Firstly, in such a complex program, like a compiler, one could have a bug and not cover all the edge cases. For example, if we allow more than two physical qubits in the crossbar, none of the mapping stages will throw an error, since this bug is in the crossbar model, and the output of the compiler would be not compatible with the crossbar architecture. As previously explained, the compiler is composed of 4 different stages: the gate decomposition, the initial placement, the routing and the scheduler; and if in any of these stages there is a bug, it will propagate to the following stages, making harder to find the bug. On top of this, even in a relative small architecture with 10 or 20 qubits, the amount of possible qubit placements and configurations is so high that is not possible to cover them manually. In other words, it would not be possible to test every configuration manually. So if there is no way to verify all cases, one will be risking having a bug in an edge case.

Secondly, we should not trust the output of the compiler because the constraints of a quantum chip usually are complex. Any small error in the definition of the constraints will lead to compiled quantum circuits incompatible with the quantum chip. Besides, the constraints of a quantum chip can change over time if the physical quantum chip is improved and modified. Thus, it is necessary to automate the verification process and reduce the time used in this step.

In addition, the verification should be able to handle different parameters based on the target architecture. Logically, the most reliable way to verify the output of the compiler is by checking it experimentally in a real quantum chip. Unfortunately, there is not yet an implementation of the crossbar architecture. So, for this thesis, the verification process that we can use is based on software. We have built a program to verify that the constraints of the operations are not violated and the crossbar is always in a decidable configuration.

5.2.1. Parameters

The verification program of this compiler must take into account the flexibility of the crossbar configuration. For this reason, it is important to be able to modify these settings. Figure 5.2 shows the setup window where one can initialize the crossbar based on the following parameters:

- The size of the crossbar
- The number of data qubits
- The number of ancilla qubits
- The position of each qubit

There are two ways to initialize these parameters. The first one, is through the inputs in that window. If the user decides to use this method, the crossbar will use the idle configuration for the positions of the qubits. The second way is to encode these parameters in a *JavaScript Object Notation* (JSON) file. This method allows to create a custom configuration of the crossbar. In other words, the positions of any qubit can be defined independently, providing the flexibility to investigate different configurations.

Note that the compiled cQASM already defines the number of physical qubits in the crossbar. Since the user defines the number of data qubits and ancilla qubits before introducing the compiled cQASM in the verification program, we will use the number of physical qubits defined by the user instead of the one defined in the compiled cQASM.

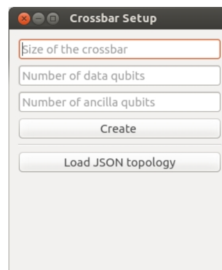


Figure 5.2: Setup window of the verification program

5.2.2. Conflict Checker

To build such a program it is necessary to be able to understand the output of the compiler. As we have explained, this output uses the language specification *cQASM* with additional instructions based on the mapping decomposition. To read this new specification, we can use library *libqasm* [19] built for parsing cQASM files and modified so that it can accept the new instructions, such as the `shuttle` operation.

Moreover, as previously mentioned, the scheduler checks if the resources are available for each instruction. To verify that the scheduler is working correctly, in this verification process, we will check the constraints in each cycle. By doing this, we also make sure that the scheduler is using the correct duration for each gate. Yet, if the compiler is using longer durations than the correct ones and the constraints are still being respected, this program will not be able to detect that issue, since is not violating any constraints, it is just wasting cycles.

Furthermore, it is important to note how the constraints are being checked. The constraint checker goes through each cycle of the scheduled circuit and verifies that the constraints are not violated. At each cycle, it checks the instructions the are starting or in the middle of

execution. For example, Figure 5.3 shows two shuttle operations scheduled one after the other. In this case, we only check the constraints of `shuttle_left q[0]` in the interval $[t-3, t)$ and we only check the constraints of `shuttle_right q[1]` in the interval $[t, t+3)$. This approach is realistic because, at cycle t, the previous operation has already left the crossbar in a valid configuration, this means that the shuttle operation has raised the barrier used before it reaches the new cycle t. Thus, the crossbar configuration respects the constraints of the operations.

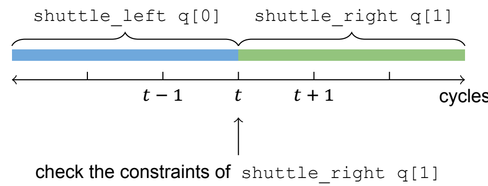


Figure 5.3: An example of checking the verification program.

Regarding the flow of execution, the verification program can detect three types of errors. The first type is an error during the parsing of the cQASM file. If it encounters a gate not supported by the crossbar or the number of qubits used is incorrect, it will throw an error. The second and third types of errors are raised from the constraint checker. The second type can be any conflict of constraints and the third type is any error due to undecidable configurations. If the program encounters any of these errors, it will stop executing and it will show an alert to the user, the flowchart of this process is shown in Figure 5.4. Note that we need the "crossbar parameters" to know the number of qubits in the cQASM parser and the position of the qubits in the constraint checker.

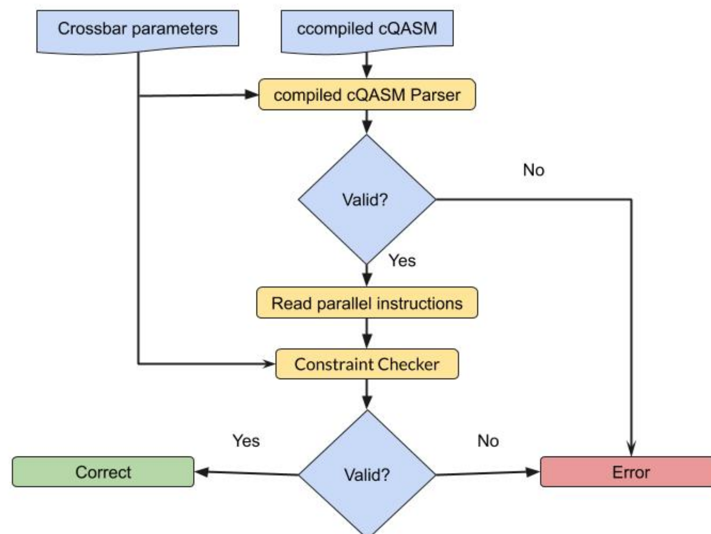


Figure 5.4: Flowchart of the verification program

5.2.3. Visualisation Tool

Finally, besides detecting errors, this program also provides a visualisation of the crossbar layout. If the cQASM code passes successfully the constraint checks, then one can visualize, through animations, how each instruction is performed in the crossbar. Figure 5.5 shows an example. This visualisation is certainly useful when dealing with a relative high number of qubits (between 10 and 40), it allows the user to see where are the qubits in each cycle. Moreover, this visualisation tool is also useful to debug new routing techniques. Because of

the built-in editor it is easier to modify the QASM code based on the visual feedback.

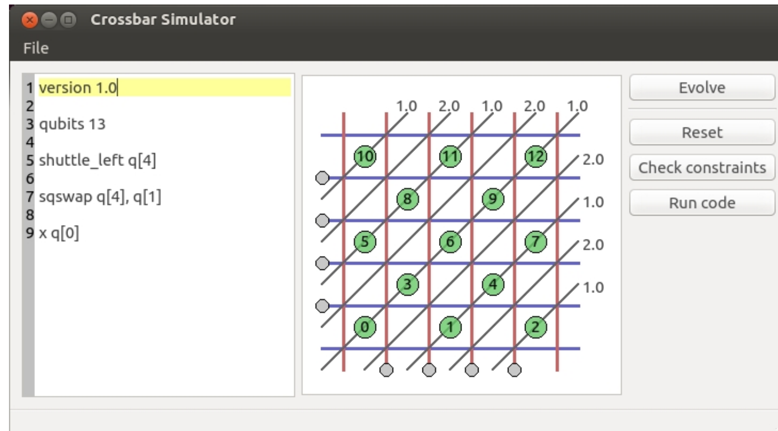


Figure 5.5: Screenshot of the verification program. At the left side, the built-in editor lets the user edit the QASM code. In the middle, the visualisation tool shows the current state of the crossbar (position of the qubits and control lines). And at the right, there is a button for each function. The "evolve" button lets manually change the control lines and see how the crossbar behaves. The "check constraints" button starts the constraint checker program. The "run code" button checks the constraints and, if the constraints are valid, it will run the animations in the visualisation tool. Finally, the "reset" button resets the crossbar state to its initial state.

5.3. Additional Crossbar Parameters

Beside the defined user parameters, there are some additional parameters which will affect in how the compilation of the quantum algorithm. To start with, we will explain the decisions made regarding the ancilla qubits. Then we will explain some alternative implementations of a quantum circuit. Finally, we will explain the crossbar configuration.

5.3.1. Ancillary qubits

As explained previously, the measurement uses ancilla qubits to execute the charge-to-state conversion. So it is necessary to find a way to know how many ancilla qubits are needed and where to place them. However, since this thesis does not focus on the analysis of this parameter, we will ignore this problem by not measuring the data qubits. Thus, despite the implementation of the measurement in the compiler, the experiments that we will run will not use any ancilla qubits.

5.3.2. Phase shift gates

Regarding the phase shift gate, in chapter 3, we declared the parameter k that defines the direction in which the shuttle-based gate is performed. For example: `z_shuttle_left` will use its left empty site to perform a Z gate by using shuttles. The compiler must decide in which direction (left or right) this gate must be performed. There are several approaches that one could use to tackle this problem and we can distinguish two. The first one chooses the same direction for every single phase shift gate. The second approach checks if the sites are empty before choosing a direction; if both sites are empty, it randomly chooses one of those.

In our case, the implementation of the scheduler in OpenQL, does not allow an easy way to implement the second approach. In addition, since our routing strategy maintains the idle configuration, it means that at some point both of the sites (left and right) will be empty. Thus, for our implementation, the first approach is more feasible and simple. For this reason, this parameter k is not tested in the experiments and is declared as $k = +1$ for all of them. In other words, all of the phase shift gates are performed by shuttling to the right site.

Note a similar issue happens for performing the measurement operation. For example: `measure_left_up` means to use the ancilla qubit of the left site for the first phase of the measurement and then use the above empty site for the second phase. Similarly to the phase shift gate, we always use the `measure_left_up` operation.

5.3.3. Crossbar Configuration

In chapter 3 we have explained how the crossbar architecture works and mentioned that this architecture can be scaled up to a higher number of qubits by increasing the number of rows and columns. So it is reasonable to simulate benchmarks different crossbar sizes before implementing a crossbar architecture.

The crossbar architecture is nearly homogeneous in terms of control lines. This means that for every site there are four barriers (that define the borders) and one qubit line (that passes over it), except for the four corners and sites at the border. This homogeneity allows one to simulate a small crossbar inside a bigger one, regardless of the position in the bigger one. For example, we can use the corner of a 5x5 crossbar to simulate a crossbar of 3x3. However, in chapter 4 we have defined a routing strategy that does not require or use external sites outside its crossbar size to move qubits. On top of this, running the compiler in a crossbar size bigger than the minimal size required would need more CPU and memory resources. Due to these two reasons, we will use the crossbar size just enough to allocate the qubits with the idle configuration. We can use the formula shown in Equation 5.1 to obtain the minimal crossbar size; where $size$ is the number of rows or columns of a squared crossbar and $qubits$ is the number of physical qubits in the crossbar.

$$size = \lceil \sqrt{(2 * qubits) - 1} \rceil \tag{5.1}$$

Note that there are cases when the physical qubits do not fill the whole crossbar with the idle configuration. For example, for a crossbar of size 5x5 the possible number of qubits are: 9, 10, 11, 12 and 13. But 13 is the maximum amount of qubits can be allocated in the crossbar of size 5x5 (following the idle configuration). So there are 4 cases in which the idle configuration can have alternative qubit positions. For example, Figure 5.7b shows a crossbar of size 5x5 with 9 physical qubits. In this case, the optimal crossbar size is 5x5 and there are still some sites with no qubits. This allows more flexibility, in terms of the positions of physical qubits. A possible alternative for positioning 9 qubits is shown in Figure 5.7c. The number of the possible qubit positions is based on the number of empty sites that allow such positions. For example, in the crossbar of size 5x5 there are 5 qubit sites that allow the 5 possible qubit positions we have mentioned (9, 10, 11, 12 and 13 qubits), so these alternatives can produce a 38% change in the number of nodes of the topology (5 empty sites over 13 possible qubits). Figure 5.6 shows how the percentage of empty sites that allow the alternative qubit position scale with the crossbar size.

Although at first it seemed that it might be interesting to analyze how the different alternative qubit positions behave, the impact that these alternatives produce decreases with the crossbar size. This means that for a crossbar of size 100x100 there can be maximum 100 empty sites, which is 1% of the total size. So there would not be a notable difference in the resulting routing overhead. Thus, we do not see this parameter relevant enough to analyze. For all the experiments in chapter 6 the physical qubits are placed following the idle configuration from bottom to top and left to right.

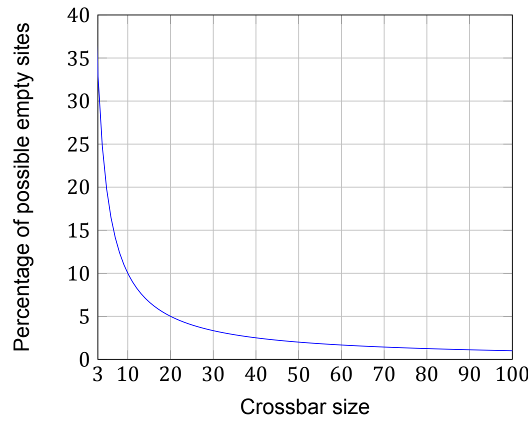


Figure 5.6: Graph that shows how the percentage of empty sites that allow alternative qubit positions scale with the crossbar size (using an idle configuration).

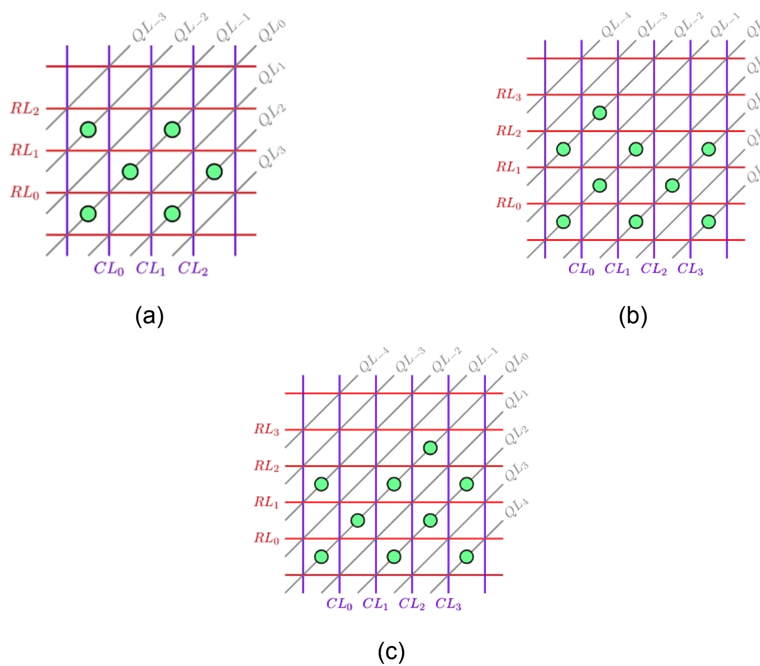


Figure 5.7: **(a)** An example of how the qubits are positioned. **(b)** and **(c)** Two alternative positions of the physical qubits in a crossbar of size 5x5.

On top of this, is important to highlight that a crossbar (in the idle configuration) with the maximum possible of qubits can have two possible qubit positions. For configurations with even size this is not a problem, since both of them are symmetrical, thus the swaps between qubits are the same. Examples are shown in Figure 5.9a and Figure 5.9b. Despite the difference in connections, the same routing can be applied by using an equivalent initial placement.

However, for configurations with odd size, the routing of qubits changes in a significant way. Examples are shown in Figure 5.9c and 5.9d. In this example we are using a crossbar of size 5x5, but the number of qubits that can fit inside is different (a difference of 1 qubit). And since we are looking for the minimal crossbar size, if we need 13 qubits we will use the crossbar with size 5x5. In addition, the topology graph of the crossbar in Figure 5.9d is denser than its opposite in Figure 5.9c. This is due to the difference of topology at the corners of the crossbar, and this difference in density decreases exponentially with the crossbar size,

as shown in Figure 5.8.

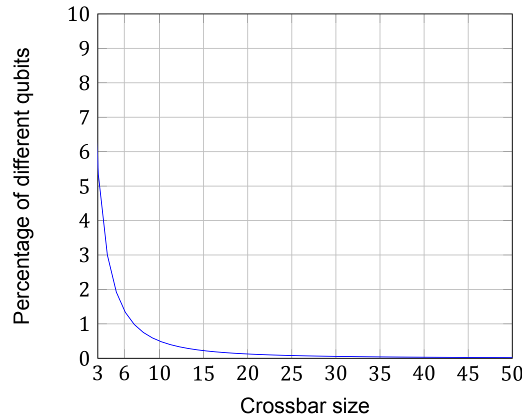


Figure 5.8: Graph that shows how the percentage of different qubits between the two possible qubit positions of odd size crossbars scale with the crossbar size

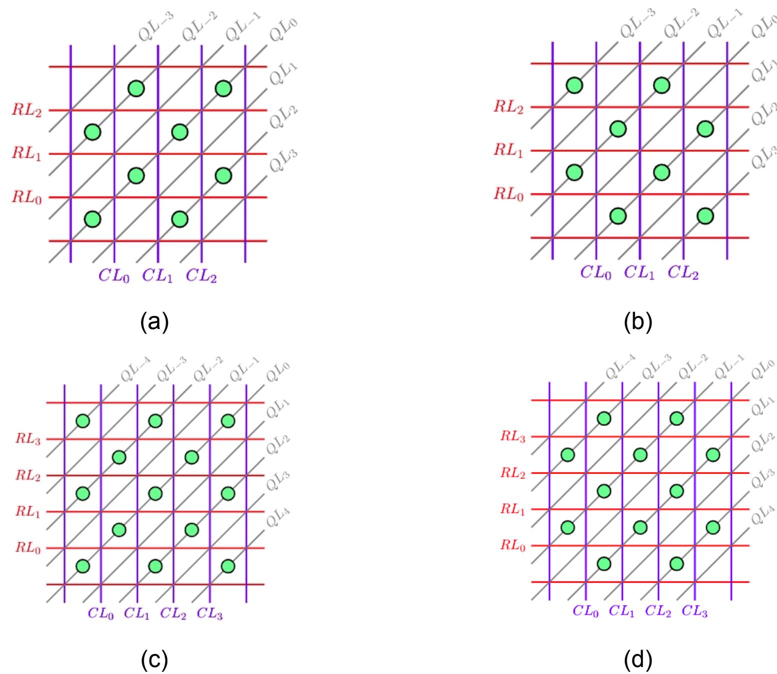


Figure 5.9: **(a)** and **(b)** Two possible layouts for a configuration with even size (4x4). **(c)** and **(d)** Two possible configurations with odd size (5x5).

Logically, depending on the algorithm to be compiled, one of the two possible qubit positions (for odd sizes) can give a better result; since the number of shuttles needed to route the qubits might be different. However, as mentioned previously, this difference is reduced as the crossbar size increases. In other words, there is only a significant difference in the routing of quantum algorithms with a low number of qubits. Thus, in the experiments of chapter 6 we will not analyze this difference since it is not relevant for a high number of qubits. The compiler will only use an idle configuration similar to the one shown in Figure 5.9c.

6

Experiments and Results

This chapter will show how the proposed techniques in the previous chapters behave with state of the art benchmarks. Firstly, in Section 6.1 we will present the benchmarks selected for the following experiments. In Section 6.2, we will analyze the overhead caused by the mapping for the crossbar architecture. In Section 6.3, we will compare the results of the different mappers from the OpenQL compiler. In Section 6.4, we will compare the results with a quantum chip that has similar topology. Finally, in Section 6.6, we will discuss the scalability of these benchmarks in the crossbar architecture.

6.1. Benchmarks

Before running the experiments, we need to collect a relative high amount of quantum algorithms to compile. These quantum algorithms should be described in cQASM, or any other hardware-agnostic quantum language, and should be a representative set of real-world algorithms. This means that there should not be any bias towards a particular benchmark. Since this set of quantum algorithms is what we are going to use to compare with other research, from now on we may call them *benchmarks*.

The benchmarks have been gathered from other research papers such as [44] for the reversible circuits, [24] for automated generated circuits and [45] for known quantum circuits. As an example, in the set of benchmarks that we are going to use we can find: Quantum Fourier Transform (QFT) and ripple adders.

Since these algorithms have been selected to test the performance of compilers in a wide range of quantum architectures, one might expect to have benchmarks with a high number of qubits. However, it is important to highlight that some of the compilers proposed, such as [12], uses optimal SAT solvers, which makes the compilation of quantum algorithms with a high number of qubits and gates not feasible. Due to this reason and since current research focuses on NISQ-era quantum computers, the majority of the benchmarks used have less than 100 qubits. We think that 100 qubits are enough to analyze how the quantum algorithm behaves when the number of qubits is increased.

Table B of Appendix 6.1 shows the list of the benchmarks used with their main characteristics: number of qubits, number of gates and circuit depth. For readability purposes, a number is assigned to each benchmark. This will help identify which benchmark is referred to in the figures of the next sections.

6.2. Mapping Results

In this section we will analyse the gate and depth overhead of the mapping process.

Regarding the setup, we are going to use the version *base* of the mapper. Note that the base mapper chooses randomly the shortest path between two physical qubits of the routing topology. Thus, as expected, the results are going to have a non-deterministic factor, since it will produce more or less SWAPs in the future, depending on the chosen path. So for this reason, in this thesis we decided to run the experiments 5 times to calculate the average gate and depth overhead of each benchmark.

In addition, we are going to run the compiler through all the benchmarks with the default parameters. As explained previously, we are going to use the same configuration (the *idle* configuration) and the same procedure of placing the physical qubits (from the bottom left to the top right), in order to focus on the parameters that can be analyzed without changing the compiler implementation.

Moreover, we are going to use the minimal size of the crossbar that can allocate the necessary amount of qubits (depending on the algorithm). It is also worth mentioning that none of the selected benchmarks have a measurement instructions. Thus, there is no need to arrange any ancilla qubit in this architecture.

6.2.1. Mapping Results with Trivial Initial Placement

In this section, we have used the trivial initial placement, called *one-to-one*, where each virtual qubit is mapped to a physical qubit. They are placed in the crossbar layout from left to right and bottom to top (just like the numbering of physical qubits). This way the virtual qubit 1 is mapped to the physical qubit 1, the virtual qubit 2 to the physical qubit 2, and so on.

6.2.1.1. Gate Overhead

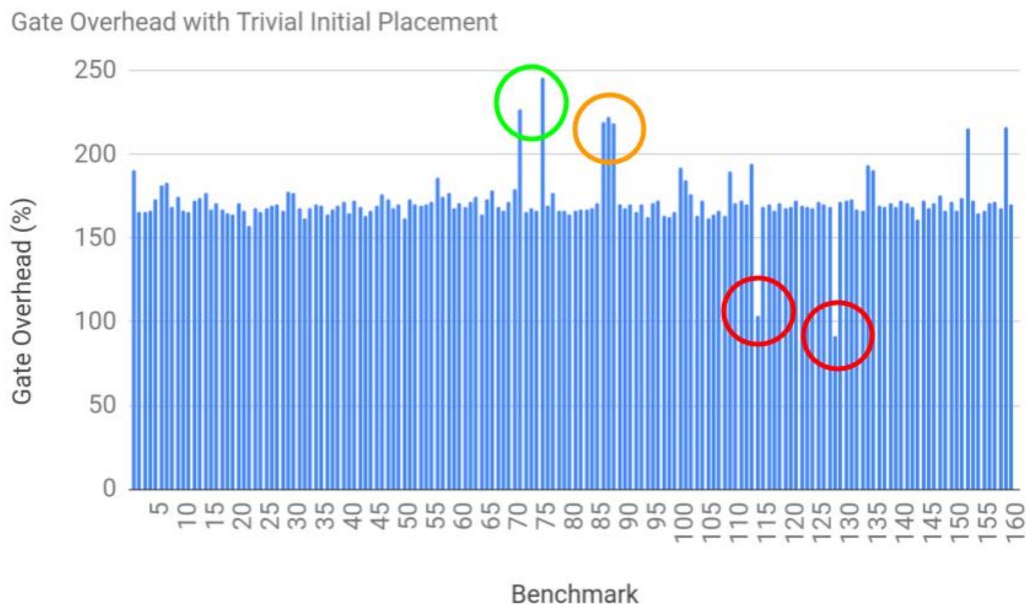


Figure 6.1: This column graph shows the percentage overhead of gates added by the mapper for each benchmark

Figure 6.1 shows the resulting gate overhead after running the experiment. The average gate overhead of the mapping process is around 170%. Note that this gate overhead is calcu-

lated based on the *difference* between the total number of gates before and after the mapping process. This means that the gate overhead is composed of the following gates:

- Swap-like shuttles, added for each swap between two physical qubits.
- Single-qubit gates used in the semi-global rotation scheme.
- Shuttles used for the semi-global rotation scheme.
- Shuttles used to execute the $\sqrt{SWAP}$ gate.

As one can see, in this list, both single-qubit gates and two-qubit gates produce an overhead in terms of gates. However, it is clear that the single-qubit gates produce more overhead. For example, an X gate will produce an extra X gate and two shuttles, but an $\sqrt{SWAP}$ gate will only require to add two shuttles (as explained in previous chapters). Note that the additional swap-like shuttles to route the physical qubits might not be present (because routing is not always required). So, for now, it seems that it is not trivial to predict what kind of algorithm will have bigger or smaller overhead. However, Figure 6.1 shows that some benchmarks clearly stand out from the rest, either due to a high or a low gate overhead.

On one hand, the benchmarks marked with a orange circle correspond to the *ising_model* algorithm. This family of algorithms have a high percentage of single-qubit gates - around 81% of the total number of gates. As mentioned previously, for each single-qubit gate (which is not Z, S or T) two shuttles and one extra single-qubit gate are added. Hence, the number of gates increases slightly more than the rest, since the overhead of two-qubit gate is lower (only two shuttles).

In addition, the algorithms marked with a green circle are *ex1_226* and *graycode6_47*, both of them have a low amount of gates: 7 and 5 gates, respectively. And only *ex1_226* has 2 single-qubit gates, the rest of them are two-qubit gates. Therefore, it is clear that, without an initial placement, the additional swap-like shuttles used for routing, in these cases, increases rapidly the overhead of the whole algorithm.

On the other hand, the benchmarks marked with a red circle, *qft_16* and *rd84_142*, have a relative low overhead. These cases are essentially the opposite of what happens in the *ising_model* algorithm. In this case, half of the gates are single-qubit. This means that if we assume 70% of the two-qubit gates need one SWAP, then the 30% which does not need a SWAP is the difference in overhead (compared to the average gate overhead) that we see in the chart.

6.2.1.2. Depth Overhead

As defined in chapter 2, the depth of a quantum circuit consists of d time steps, each time step contains one- and two-qubit gates acting on disjoint qubits [6]. To analyze the depth overhead caused by the mapping, we are going to use the same experiments from section 6.2.1.1. Figure 6.1 shows that the average depth overhead is around 368%. Again, we can not compare it with compilers of totally different architectures, but the compiler from [45] achieves a depth overhead of around 202%.

It is worth mentioning the possible causes of depth overhead that we will find in the results:

- $\sqrt{SWAP}$ gates: due to the division of kernels.
- SWAPs: because they are decomposed in swap-like shuttles and added to route qubits.
- Waiting instructions: it is necessary to add these instructions to wait while the gates are being executed.

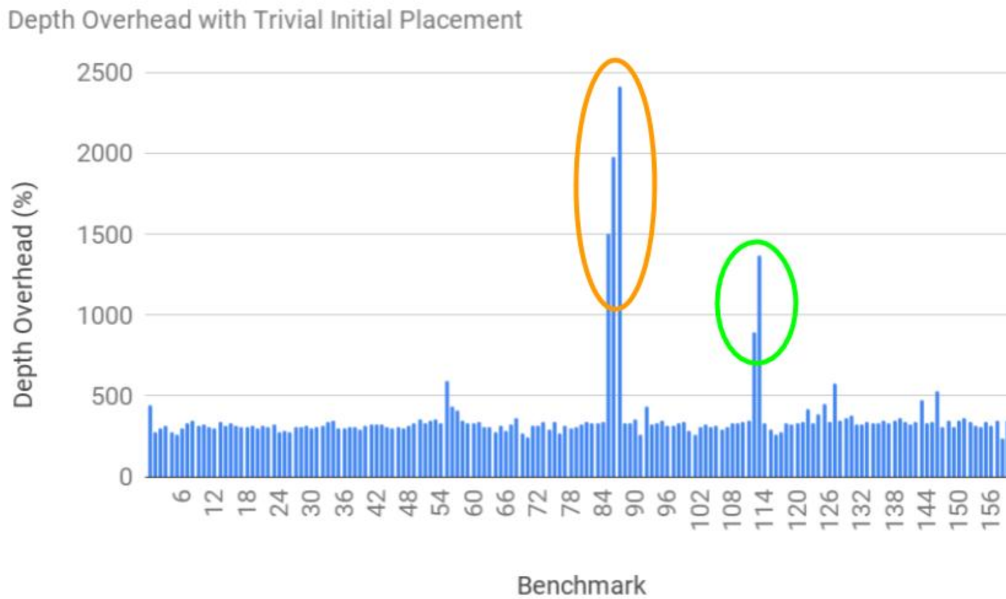


Figure 6.2: This column graph shows the percentage overhead of depth added by the mapper in all the benchmarks

As explained in chapter 4, in our implementation of the compiler, whenever there is a $\sqrt{SWAP}$ gate (or a SWAP) a division of kernels is made. This means that the code is more sequential and the depth is increased. And the waiting instruction is only a consequence of the crossbar constraints. In section 6.3, we will see how we can try to solve this waiting.

Similarly to the previous experiment, there are some benchmarks which stand out from the rest. For example, the benchmarks marked with an orange circle corresponds to the *ising_model* algorithms, which, as mentioned before, they have a high percentage of single qubit gates. However, it is important to highlight that these gates are *different*. Due to this fact, it will produce a high number of extra single-qubit gates that can not be easily parallelized using the semi-global rotation. For that reason, the overhead of the *ising_model* algorithms is higher than the average.

In addition, the algorithms *qft_10* and *qft_16*, marked with a green circle, have a fully connected graph based on the CNOT gates (where each node is a virtual qubit and each edge is a CNOT gate). Each physical qubit has at most 4 connections with other physical qubits in the crossbar topology. Due to this fact, a high number of SWAPs must be added to execute this algorithm. And, since each SWAP divides the kernel, it adds more depth to the overall algorithm.

6.2.2. Mapping Results with Initial Placement

As we have explained in chapter 4, the trivial initial placement is far from optimal, since the interactions between virtual qubits can be arbitrary. So it seems logical to study how to *initially map* the virtual qubits onto the physical qubits based on the interactions of two-qubit gates.

For these experiments, we have used the initial placement algorithm implemented in OpenQL [2]. As already mentioned, the problem of finding the optimal initial placement is NP-complete. So to find the optimal solution this implementation uses mixed-integer linear programming. Naturally, this implementation can take a large amount of time to find the solution, so it will only be practical with a small number of qubits. In fact, for the next

experiments, we will only be able to run the initial placement implementation with 10 qubits or less. This means using at most a crossbar of size 5x5.

6.2.2.1. Gate Overhead

The following experiment measures the gates overhead of the mapping process using the optimal initial placement. The algorithms with more than 10 qubits are discarded.

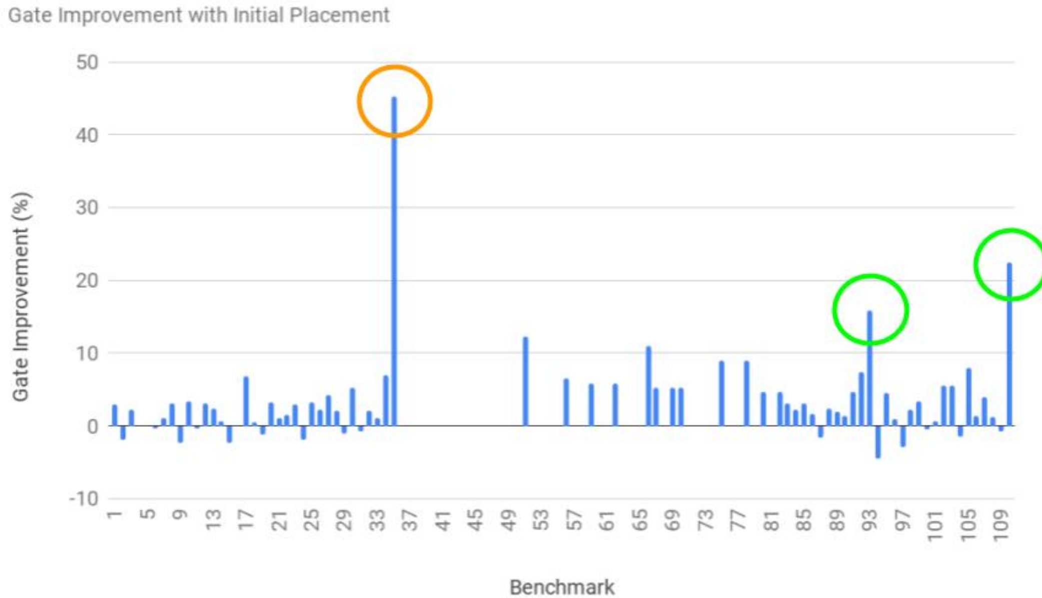


Figure 6.3: This column graph shows the improvement of the depth overhead compared to not using the initial placement

In Figure 6.3 shows the improvement (in terms of gates overhead) of enabling the initial placement in the crossbar compiler. The average improvement is around 9.11%. Although it might seem as a good improvement, the benchmarks that have been used have a low amount of qubits. Therefore, we can not rely on these results to draw conclusions. However, as in previous experiments, some benchmarks stand out from the rest which are worth explaining.

The benchmark marked with an orange circle is the *benstein_vazirani_1b_1* algorithm. It uses only 2 qubits, 5 single-qubit gates, and 1 two-qubit gate, so just by placing correctly both of the qubits, the need to route the qubits is gone. Therefore, the number of gates used in the algorithm drops significantly.

The benchmarks marked with a green circle are the *cucarroMultiplier_1b* and *xor5_254* algorithms. Both of these algorithms show a high improvement (15% and 22%, respectively) for the same reason. The common aspect of these algorithms is that their CNOT graph follows a similar distribution to a power law. In other words, the majority of the CNOTs used have a few physical qubits in common. Figure 6.4b shows an example of a graph following a power law distribution, where a few nodes (2, 5 and 7) are adjacent to the majority of the edges. On the other hand, there are algorithms, such as *qft_16*, that have a fully connected graph. Figure 6.4a shows an example of this, where each node is connected to every other node.

It is important to highlight this because we are comparing the trivial initial placement (*one-to-one*) with an optimal solution. In the case of the *cucarroMultiplier_1b* and *xor5_254*, it is easier to have a high improvement because the number of physical qubits needed to improve it is lower. In other words, just correctly placing the physical qubits with the most connections will highly reduce the number of SWAPs needed. On the other hand, in the case of the *qft_16*, it is easier for a trivial initial placement to return a mapping which requires a

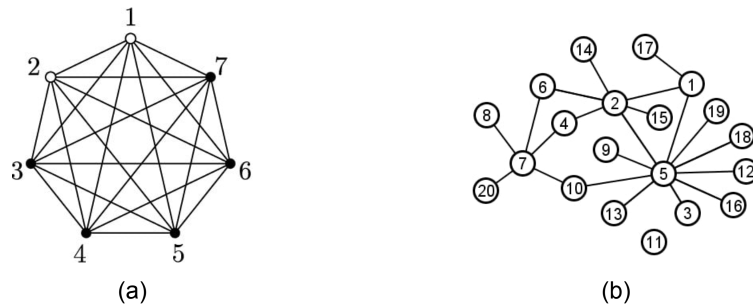


Figure 6.4: **(a)** An example of a fully connected graph. **(b)** An example of a graph that follows a power law distribution

similar amount of SWAPs to the optimal, since all the physical qubits will need to interact with each other at some point. Although this explanation clarifies the improvement seen in the results, it does not take into account multiple interactions between two pair of physical qubits. Additional experiments would have to be done to analyze these cases.

Moreover, there are some benchmarks which return a negative improvement after using the optimal initial placement. This is due to the randomness factor inside the mapper implementation. In this experiment, we used the mapper called *base*, which takes randomly a chain of SWAPs among all of the possible shortest paths between two physical qubits. For example, if we want to make qubit A and qubit B interact, we can move qubit A near qubit B, or move qubit B near qubit A, or move both so that they can meet in the middle. The place where these two qubits meet is random and it will affect the path of the next routing. Thus, it is not guaranteed to find the optimal path or even the same one that we used in the previous experiment with the trivial initial placement.

6.2.2.2. Depth Overhead

This subsection describes the analysis of the depth improvement of the previous experiment 6.2.2.1.

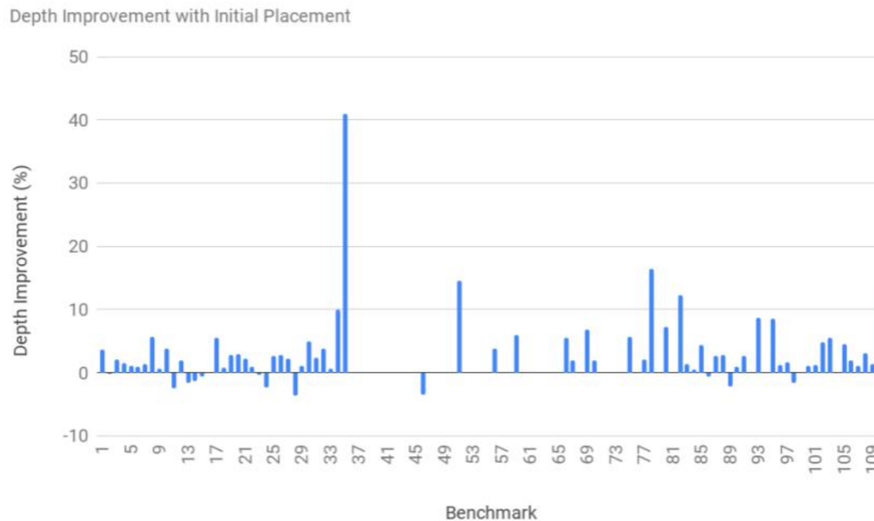


Figure 6.5: This column graph shows the percentage overhead of depth added by the mapper using the initial placement

Figure 6.5 shows the improvement made compared to the trivial initial placement. The average improvement is around 2.9%. This result validates the improvement of the gate overhead from the previous result. Since most of the resulting cQASM code is sequential

(due to the constraints) and each SWAP is composed of 4 gates with depth 2, then, it seems logically to see at least a reduction of 2 compared with the result in section 6.2.2.1. Despite that fact, there is not much insights that we can obtain from this experiment. The negative improvement is due to the same reason as the previous experiment (the randomness in the mapping process) and, the same benchmarks stand out from the rest because a better initial placement means less gates (swap-based shuttles) and less depth.

6.3. Comparison of Different Mappers

After analyzing the mapping overhead and the initial placement, we will now compare the different mapper implementations. We have already discussed in Chapter 4 about the possible strategies of routing the qubits through the crossbar. The mapper called *base* is the one we have been using in the previous experiments. The following experiments will compare the rest of strategies called: *minextend* and *minextendrc*.

Firstly, we are going to test the mapper called *minextend*. Essentially, this mapper minimizes the depth by choosing the meeting point between the two physical qubits that are going to interact. Whereas the *base* mapper randomly chooses a meeting point.

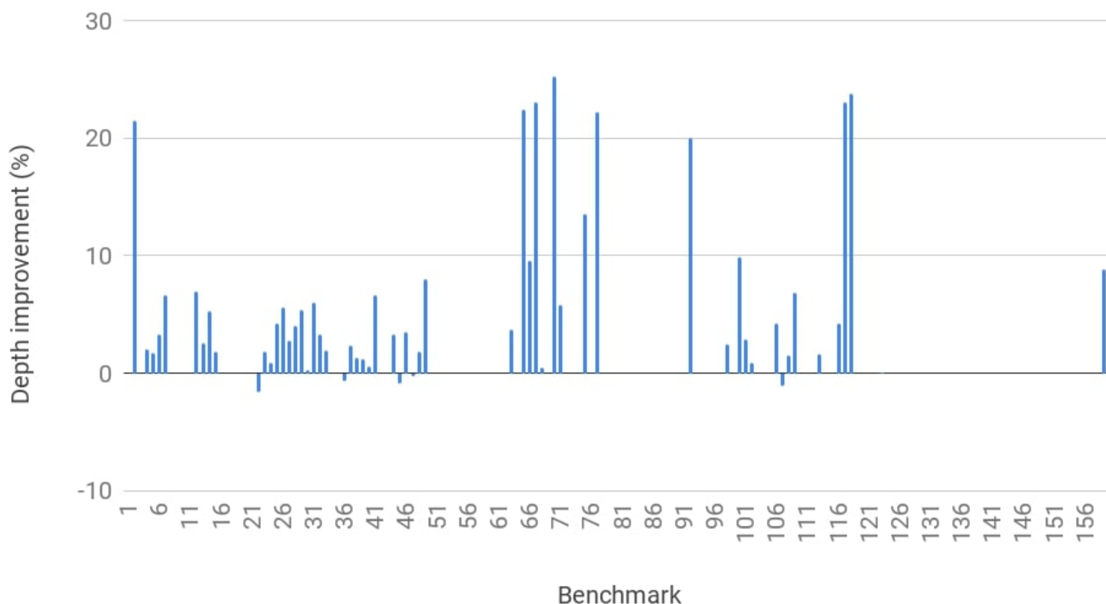


Figure 6.6: This column graph shows the percentage improvement in depth of the *minextend* mapper compared to the *base* mapper

Figure 6.6 shows the comparison between the *base* mapper and the *minextend* mapper. The average improvement of the *minextend* mapper is 2.3% over the *base* mapper. Since the improvement is based on the reduction of SWAPs, it also reduces the depth. In addition, even though we are choosing the less extensive set of SWAPs, there is still a random selecting process when two or more paths have the same extension. This means that the mapper does not always select the optimal path, in terms of depth. For this reason, Figure 6.6 shows some negative improvement in some cases where the trivial mapper did a better routing.

Finally, the *minextend* mapper does make a significant improvement over the trivial *base* mapper for most of the benchmarks. The implementation of these mappers route the physical qubits based on SWAPs and, in Chapter 4, we transformed the crossbar topology into one

based on SWAPs to easily use these mappers, among other reasons. In other words, a routing based on shuttling would have need a different mapper implementation and the current one could have not been able to make such improvements, since the shuttles requires a different strategy, as explain in Chapter 4.

In this experiment, we will compare the *minextendrc* mapper to the already improved *minextend* mapper. The *minextendrc* mapper is based on *minextend*, so not only tries to minimize the extension, but it also takes into account the crossbar constraints to insert SWAPs. Since this is just another layer of optimization, it does not interfere with the optimization based on the extension. So, in theory, we only expect the *minextend* mapper to produce an equal or better overhead.

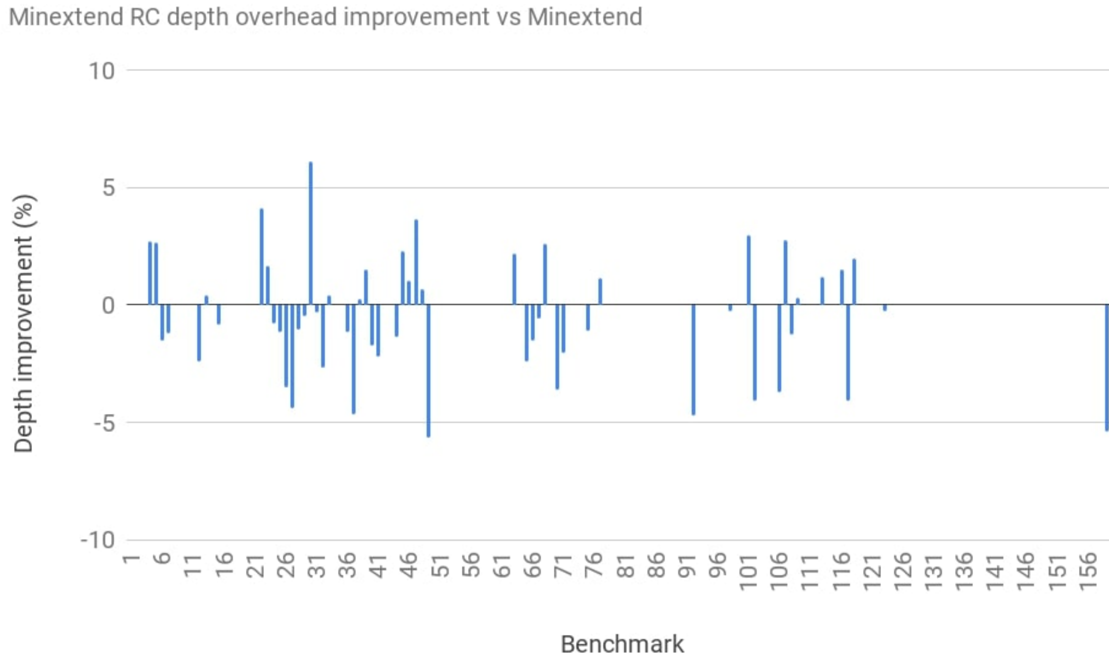


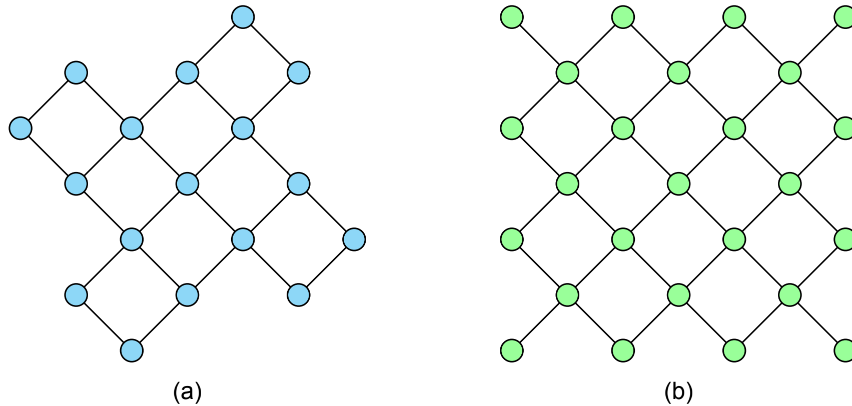
Figure 6.7: This column graph shows the percentage improvement in depth of the *minextendrc* mapper compared to the *minextend* mapper

Figure 6.7 shows the comparison between the *minextendrc* mapper and the *minextend* mapper. Based on these results, there is no consistent improvement through all the benchmarks. In fact, although this mapper makes an improvement in the Surface-17 architecture [20], it is not useful in the crossbar compiler due to: the division of kernels and swap-like shuttles. The *minextendrc* mapper only adds SWAPs, it does not decompose them into shuttles. And since the SWAP gate is not supported in the crossbar architecture, the additional SWAP gate will not produce the conflict of constraints necessary to make the improvement. On top of this, all the SWAPs added will be forced to a sequential execution due to the decisions made in Chapter 4. This means that it is not useful to consider the constraints in this version of the compiler. Hence, the improvement of the *minextendrc* mapper is just random. This can be seen in the results, where the majority of the previous results could not be improved and the rest of them produce a small positive or negative improvement.

6.4. Comparison with the Surface 17 chip

Until now we have been analyzing the overhead of the implemented crossbar compiler. We already have other prototypes of quantum computers working and their respective compil-

ers. For example, [45] compares their compiler with the Qiskit compiler [25] from IBM for the IBM QX2 and IBM QX3 architectures. Although it is not appropriate to compare the performance of two compilers for different platforms, because each architecture might have different constraints and support different gates, it seems reasonable to compare the parallelism of two architectures based on a common compiler. For example, there is a previous work that maps quantum algorithms to the Surface-17 chip [20] using the OpenQL compiler. Since its routing is implemented based on SWAPS and our routing uses swap-like shuttles, it seems reasonable to compare the difference between the two architectures. In addition, the topology of the Surface-17 is similar to the routing topology in the crossbar architecture; in fact, its routing topology graph is a subgraph of the crossbar routing topology. An example of this relation is shown in Figure 6.8. Note that although the configuration of the crossbar can be customized, the topology shown in Figure 6.8b is the routing topology chosen for our compiler to implement the swap-like shuttles while using the idle configuration.



(a) (b)

Figure 6.8: Routing topology of **(a)** the Surface-17 chip and **(b)** the crossbar architecture of size 7x7. The nodes represent the physical qubits and the edges represent the possible SWAP interactions between them

As mentioned in Chapter 2, the majority of the compilers built for quantum architectures only take into account the connectivity problem. However, the other main source of overhead is the physical constraints of the architecture. Since we are still in a early stage, regarding quantum compilers, it seems reasonable to tackle only the connectivity problem. Nonetheless, this experiment will show that the physical constraints (beside the connectivity) has a high impact in the mapping overhead. On top of this, this analysis also provides some insights into the parallelism of the crossbar architecture.

Regarding the experiment, it does not seem reasonable to measure the gate overhead since the decomposition of gates and routing gates are not the same. For example, to execute a swap between two qubits, the Surface-17 would perform 3 CNOT gates, while the crossbar architecture would perform 4 shuttles. That will meant a big difference in the number of gates, even though both of them had only done a swap. Moreover, since both architectures use the same initial placement, have a similar topology and use the same way of routing qubits, then the only difference that we can see in the depth overhead is due to the constraints. A quantum chip which has more flexible constraints will be able to schedule the routing gates easily with the rest of gates; hence, producing less depth overhead. Otherwise, if the constraints are less flexible, then the mapping gates will be more difficult to parallelize with the rest of gates; hence, it will produce a higher depth overhead. So it seems reasonable to compare the percentage of depth overhead between these two architectures and see which one has more flexible constraints.

Regarding the setup, both of the compilers used for these architectures are based on *OpenQL* [2], so both of them use the same initial placement and the same routing strategy, called *minextend*. Also, similar to the previous experiments, we have used the minimal crossbar size to allocate the qubits.

Table 6.1 shows the depth overhead of the resulting circuit after mapping each benchmark to the Surface-17 chip and the crossbar architecture. There is clearly a difference in depth overhead between the two architecture. In 5 of the 6 benchmarks, the Surface-17 chip has a lower overhead than the crossbar architecture. The benchmark *xor5_254* is the only one where the crossbar has a lower overhead, but the difference is not bigger than in the rest of benchmarks. Evidently, this overhead is due to the swap of qubits. The reason why it is higher in the crossbar is due to the “division of kernels” implemented in the compiler. As previously explained, this division of kernels is a way to handle the constraints of the crossbar. For this reason the swaps can not be executed in parallel, which increases rapidly the depth. So we can see that the constraints of the crossbar architecture are less flexible than the constraints of the Surface-17. Also note that the results of this experiment are limited by the benchmarks obtained from the results in [20]. Although the number of benchmarks is low, we can still see the difference in the overhead due to the constraints.

	Depth overhead (%)	
Benchmark	**Surface-17**	**Crossbar**
xor5_254	260	210
ham3_102	46.3	172.58
cuccaroAdder_1b	38.8	480
alu_v0_27	38.9	147
rd32_v0_66	59.1	176
miller_11	48.6	183

Table 6.1: Comparison of the depth overhead between the Surface-17 chip [20] and the crossbar architecture

6.5. Mapping Overhead Analysis

In the previous sections, we have discussed the mapping overhead for each benchmark and we have seen that some quantum algorithms return a higher or lower overhead than the rest. Knowing what kind of quantum algorithm will produce a higher or lower depth overhead is useful for future applications. This can give some insights into what quantum algorithm can be easily executed into the crossbar architecture with less overhead. In this section, we will analyze some characteristics of the quantum algorithms that have shown low depth overhead in section 6.2.

6.5.1. Characteristics

Firstly, it is important to highlight that, during the process of building our crossbar compiler, we have made some decisions to simplify the process of mapping (e.g. avoiding deadlocks or only shuttling to one direction to execute the phase shift gates). This means that the compiler built for this thesis is thought to be far from optimal and complete. For example, in Chapter 4 we have explained a simplification that divides the kernels of a QASM program. This might be a good countermeasure to deal with the deadlock problem, but it has clearly the disadvantage of making the resulting code less parallel. Some possible improvements to this problem will be discussed in Chapter 7.

From the way our compiler is built, we have already mentioned that the routing of physical qubits will increase the overhead and reduce the parallelism, thus we need to avoid the swap-like shuttles. Based on the findings of Section 6.2, other possible characteristics that might influence the parallelism are:

- *Percentage of single-qubit gates*: we have seen that the crossbar is able to execute the same gate in multiple qubits at the same time, using the semi-global qubit rotation. So

if a higher percentage of single-qubit gates can be parallelized, the depth overhead will be reduced.

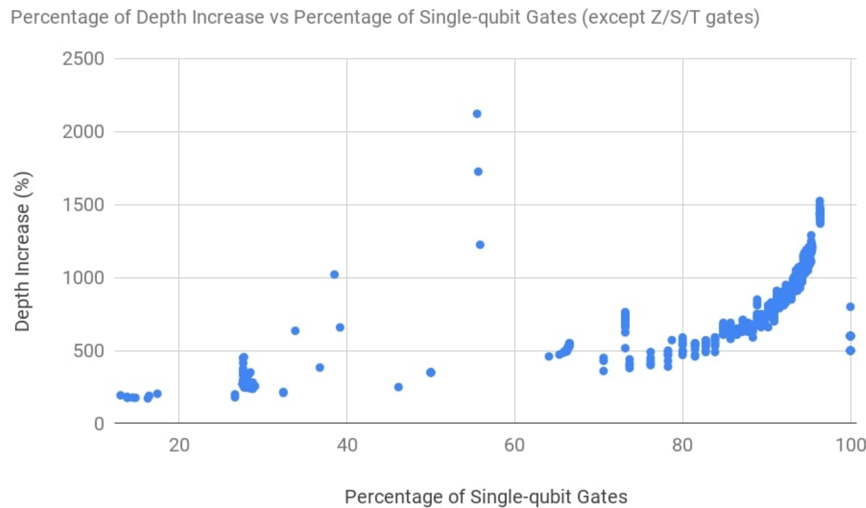
- *Percentage of Z, S and T Gates*: these gates are executed using the shuttle operation, therefore they have the same potential to leverage the parallelism of the architecture as the coherent shuttles in [13].
- *Percentage of two-qubit gates*: although additional shuttles may be added to route the physical qubits, we have seen (in the *cuccaroAdder* benchmark) that it is possible to parallelize a fraction of CNOT gates through the semi-global scheme.

Although these characteristics might be enough to show if a quantum algorithm produces low overhead in the crossbar architecture, there are other characteristics, mentioned in Section 6.2, that might also influence on the depth overhead, such as the pattern of gates or the interactions between the virtual qubits. In order to assess these last characteristics is necessary to come up with a way to quantify and compare them, which is beyond the scope of this thesis. However, in Section 6.6, we will show how some examples of these cases behave.

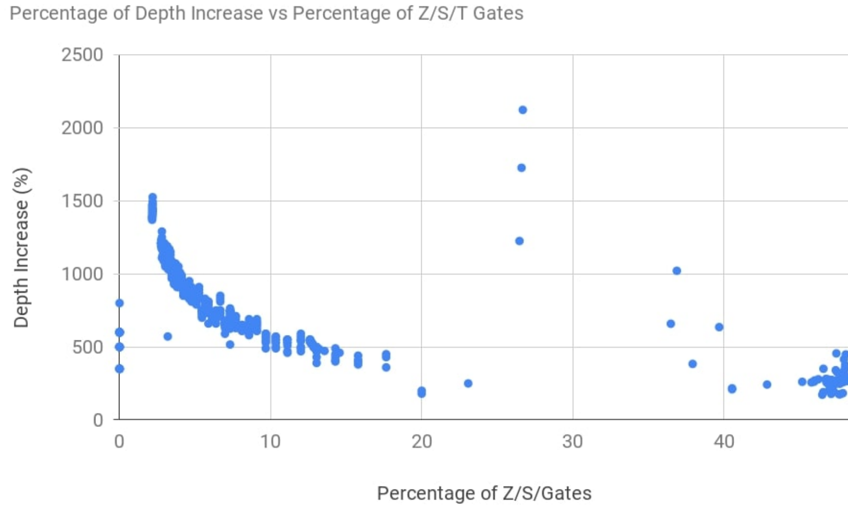
Note we have ignored other characteristics, such as the percentage of SWAPs. In this case, there is no reason to analyze its behaviour since, due to the "division of kernels", adding a SWAP will always increase the depth of the resulting algorithm.

6.5.2. Experiments

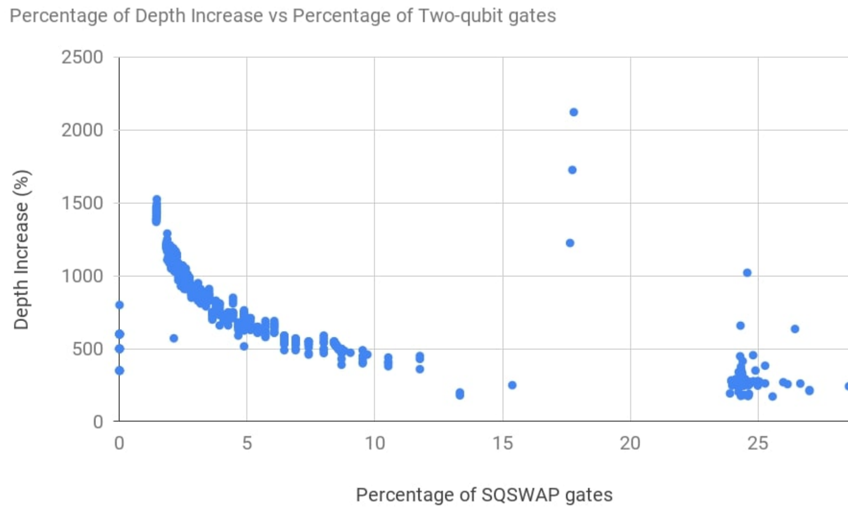
The next step in this section is to map into the crossbar architecture all set of benchmarks and see if any of the listed characteristics are relevant. The following dispersion charts in Figure 6.9 show how the percentage of depth overhead behaves when the value of these selected characteristics is increased.



(a)



(b)



(c)

Figure 6.9: Dispersion graphs that show how the overhead depth behaves with the increase of **(a)** single-qubit gates, **(b)** Z, S & T gates and **(c)** two-qubit gates (SQSWAP gates)

First of all, it is important to highlight that different algorithms will show a different behaviour. As we have seen in the previous sections, this might be, for example, due to a high number of SWAPs or a high number of different single-qubit gates. This leads to different patterns in the charts shown in Figure 6.9. However, to see the behaviour of each kind of quantum algorithm we need a set of them with different number of qubits. In other words, we need to analyze the overhead of each family of algorithms. Unfortunately, the set of benchmarks that we have used in these experiments does not have enough quantum algorithms per family so, in most cases, it is not possible to see the behaviour of a particular quantum algorithm. Therefore, we will only focus on the families of algorithms that have enough of them to detect a pattern and get some insights. In summary, the purpose of the charts in Figure 6.9 is to see if any of the proposed characteristics have a positive impact in the overhead of all or some algorithms.

To begin with, from the charts in Figure 6.9b and Figure 6.9c, we can see, at the bottom left of each one, a common group of benchmarks belonging to the *Bernstein Vazirani* algorithm, which forms a negative correlation (and a positive correlation in Figure 6.9a). This algorithm, built upon Deutsch and Jozsa [5], is used to determine the mathematical function of the quantum oracle function. To do this, it uses a high number of H gates in parallel and only a few CNOT gates, which corresponds to the oracle function. In terms of overhead, this means that the number of H gates increases with the number of qubits while the number of CNOT gates stays constant. In general, since the H gates can be easily executed in parallel using the semi-global rotation scheme, the resulting depth of this algorithm is relatively small. The problem is that as the number of qubits increases, the crossbar size does too. Thus, the number of SWAPs necessary to execute the CNOT gates is higher. In summary, although, in Figure 6.9a, it appears that having more single-qubit gates produces a higher depth increase, it is, in fact, the SWAPs that produce that overhead increase.

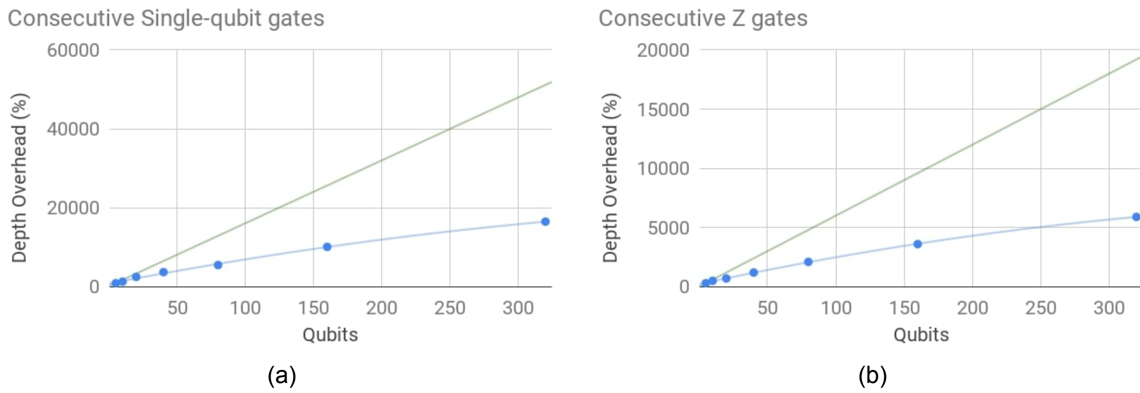
Furthermore, since the decomposition of the CNOT gates produce S gates and the number of Z, S and T gates in our set of algorithms is low, the graph from Figure 6.9b is very similar to the chart in Figure 6.9c. Unfortunately, this means that we can not see the expected improvement of parallel Z, S and T gates. For this reason we will make some artificial benchmarks in the next section to show how the depth behaves in these types of algorithms.

Moreover, the rest of algorithm families, that have at least 10 different algorithms, show a similar trend as the *Bernstein Vazirani* algorithm due to the same reasons explained above; except where none of the proposed characteristics change when the algorithms increases the number of qubits. An example of this exception is the *ising_model* algorithm. Although it grows in the number of single-qubit gates, the percentage of such gates does not change. This neutral correlation can be seen in the bottom left of Figure 6.9a and, bottom right of Figure 6.9b and Figure 6.9c. In summary, although the experimental results from Figure 6.9 does not show any new insights into the crossbar architecture, we can confirm that the two-qubit gates is the main bottleneck in this version of the compiler.

6.6. Scalability

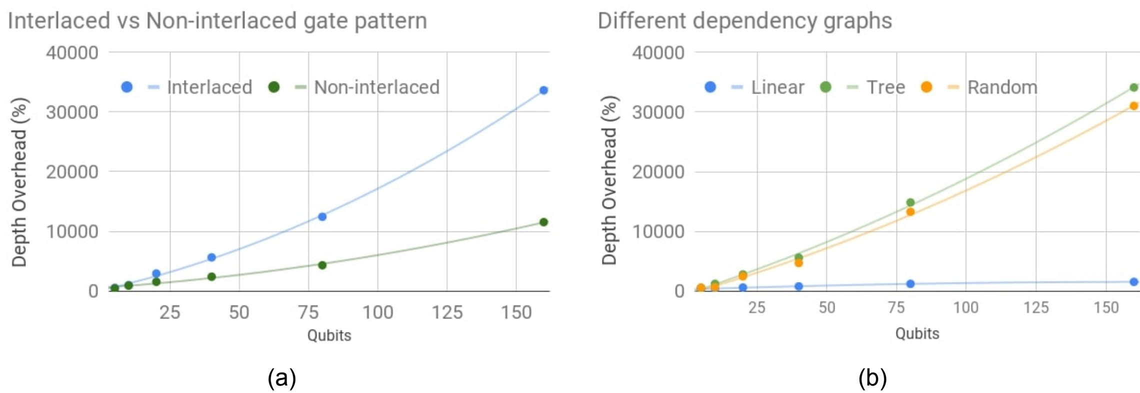
From the start of this thesis, we have been making decisions to try to exploit the possible parallelism inside the crossbar architecture. Our goal is not only to take into account the physical constraints but also to optimize the resulting circuit depth. Therefore, it is important to analyze the parallelism provided by the architecture. To do so we will study how the parallelism behaves as we increase the circuit size. We will use as a metric the percentage of depth overhead.

After going through all the experiments and gaining some insights into what kind of algorithms can be executed in the crossbar architecture, it seems reasonable to create artificial benchmarks that prove the observations taken in the previous sections. We have already mentioned that, in order to provide relevant results, it is important to focus on the characteristics of the crossbar and not on the limitations of the compiler. However, since this is the first version of the crossbar compiler, some improvements can still be made and the limitations of the implementation of the compiler might standout in the next experiments. For example, the “division of kernels” used to avoid the deadlock will influence in the results. So we must take this into consideration when making observations.



(a) (b)

Figure 6.10: Dispersion graphs that show how the overhead depth behaves with the increase of qubits in different kind of algorithms. **(a)** Based on consecutive single-qubit gates, **(b)** Based on consecutive Z gates. For comparison, the green straight line represents a linear scalability based on the initial depth overhead (with 5 qubits).



(a) (b)

Figure 6.11: **(a)** Different behaviour in scalability between an algorithm with interlaced CNOT gates and other algorithm with no interlaced CNOT gates. **(b)** Dispersion graph that shows how different CNOT graphs affect the scalability of the algorithm.

To begin with, we have seen in Chapter 3 and Section 6.5.2 that the semi-global rotation scheme is a characteristic of the crossbar architecture that allows to parallelize a high amount of single-qubit gates. Figure 6.10a shows the behaviour of the depth overhead of an algorithm based on the *same* consecutive single-qubit gate. This example shows that, even by taking the crossbar constraints into account, the semi-global rotation successfully takes advantage of the crossbar parallelism. In addition, the example in Figure 6.10b uses a similar algorithm except it uses Z gates. This algorithm can be easily parallelized by using the shuttle scheme explained in Chapter 3 for the Z, S and T gates. Note that, in both examples, the depth of the algorithm is 1 but the compiler returns a bigger overhead due to the constraints.

Moreover, in the previous experiments we have seen that although the *ising_model* algorithm has a high percentage of single-qubit gates, it has two-qubit gates interlaced every 2 or 3 single-qubit gates. Since the current version of the compiler produces a division of kernels in each two-qubit gate, the single-qubit gates can not be scheduled in parallel, producing more overhead. An example of this pattern is shown in Figure 6.12b. To test this behaviour in the crossbar architecture, we have made two quantum algorithms. The first quantum algorithm has consecutive single-qubit gates and then consecutive two-qubit gates. The second quantum algorithm has the same gates but interlacing two-qubit gates with the single-qubit gates. A short example of these algorithms is shown in Figure 6.12.

Both of the algorithms shown in 6.12 scale linearly with the number of qubits. This means that for every new qubit, a single-qubit gate and a two-qubit gate are added. The experiment

```
1 y q[0]
2 y q[1]
3 y q[2]
4 y q[3]
5 cnot q[0], q[1]
6 cnot q[1], q[2]
7 cnot q[2], q[3]
8 cnot q[3], q[4]
```

(a)

```
1 y q[0]
2 cnot q[0], q[1]
3 y q[1]
4 cnot q[1], q[2]
5 y q[2]
6 cnot q[2], q[3]
7 y q[3]
8 cnot q[3], q[4]
```

(b)

Figure 6.12: **(a)** Shows the cQASM code of a non-interlaced pattern. **(b)** Shows the cQASM code of a interlaced pattern.

results of running these algorithms in the compiler is shown in 6.11a. As expected, the non-interlaced version (green line) has a better curve. In fact, it is lower than half of the depth overhead of the interlaced version. Logically, these quantum algorithms are not equivalent. However, in this experiment we have shown that the pattern of gates in the dependency graph will make an impact in the scalability behaviour.

On top of this, there is another reason why a consecutive chain of CNOT gates can be parallelized with smaller overhead. There are some benchmarks, like *qft_16* and *rd84_142*, which have a relative high percentage of CNOT gates (□46% for *qft_16* and □44% for *rd84_142*) and do not show a high depth overhead. One of the reasons for this is the decomposition of the CNOTs. As shown in Figure 3.13, the first phase of the CNOT, composed of single qubit gates, can be parallelized by using the semi-global rotation scheme.

For example, Figure 6.13a shows two CNOT gates decomposed into the gates supported by the crossbar architecture. After compiling the cQASM code, taking into account the constraints, the Y90 gates can be executed in parallel with other Y90 gates, as seen in Figure 6.13b. Depending on the constraints, it is even possible to execute some of the S gates in parallel. Logically, if the CNOTs from the example needed any additional SWAPs, then this optimization might not be possible. But in a consecutive chain of CNOTs with no SWAPs, the compiler can take advantage of this small optimization.

Finally, it is worth pointing out the effect of the two-qubit gates in the compilation process. Logically, more two-qubit gates will produce more depth overhead. However, we have not seen the propotion of this increase. To check this we have made three different algorithms that are only composed of two-qubit gates and have the *same* number of gates. The first algorithm forms a line dependency graph with the two-qubit gates, the second forms a tree dependency graph and, the third forms a random dependency graph. Note that we have not compiled quantum algorithms with more than 300 qubits, otherwise the compiler will run out of memory. The results from Figure 6.11b shows three things. Firstly, the compiler implementation has a high impact in the resulting cQASM program, since the division of kernels produce a high amount of depth overhead. Secondly, in terms of scalability, the percentage of two-qubit gates (in this case 100%) can be as impactful as the connectivity they form. Thirdly, it seems that a linear dependency of two-qubit gates is preferred over a tree dependency graph (which shows a worse scalability than random).

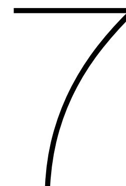
```
1  # cnot q[0], q[1]
2  y90 q[1]
3  sqswap q[0], q[1]
4  s q[0]
5  sdag q[1]
6  sqswap q[0], q[1]
7  my90 q[1]
8
9  # cnot q[2], q[3]
10 y90 q[3]
11 sqswap q[2], q[3]
12 s q[2]
13 sdag q[3]
14 sqswap q[2], q[3]
15 my90 q[3]
```

(a)

```
1  {y90 q[1] | y90 q[3]}
2  {shuttle_right q[1] | shuttle_right q[3]}
3  {my90 q[4] | my90 q[4]}
4  {shuttle_left q[1] | shuttle_left q[3]}
5
6  shuttle_left q[1]
7  sqswap q[0], q[1]
8  shuttle_right q[1]
9
10 shuttle_left q[3]
11 sqswap q[2], q[3]
12 shuttle_right q[3]
13
14 {s_shuttle_right q[0] | sdag_shuttle_left q[1]
15     | s_shuttle_right q[2]
16     | sdag_shuttle_left q[3]}
17
18 shuttle_left q[1]
19 sqswap q[0], q[1]
20 shuttle_right q[1]
21
22 shuttle_left q[3]
23 sqswap q[2], q[3]
24 shuttle_right q[3]
25
26 {my90 q[1] | my90 q[3]}
27 {shuttle_right q[1] | shuttle_right q[3]}
28 {y90 q[5] | y90 q[5]}
29 {shuttle_left q[1] | shuttle_left q[3]}
```

(b)

Figure 6.13: **(a)** cQASM code of two decomposed CNOTs. **(b)** The compiled cQASM code of **(a)**. Note that *wait* instructions have been removed to increase the readability of the example.



Conclusions and future work

7.1. Conclusions

In this thesis we have designed and implemented a mapper pass in OpenQL for the Si spin crossbar architecture. To do this, we have modelled the crossbar layout and the constraints of each operation. We have also defined the crossbar configurations that must be avoided, called the undecidable configurations. For the initial placement, we have used the trivial placement and an ILP algorithm implemented in OpenQL to find the optimal solution. For the routing, we have designed a new routing topology to easily maintain the idle configuration after each operation. To keep the operations compatible with the new topology, a new layer was added to the compiler, called “mapping decomposition”. A deadlock can happen when routing qubits in the crossbar architecture, so we have proposed three approaches for dealing with this problem in the scheduling process: avoid the deadlock, a backtrack method and suffering a side effect. For this first version of the compiler, we have decided to avoid the deadlock at the expense of producing more overhead.

Regarding the simulation framework, we have developed a verification program to check the output of the compiler. This program checks not only parsing errors of the compiled cQASM, but also the conflicts between the constraints of the crossbar architecture and the undecidable configurations. In addition, this program will be useful to debug future versions of the compiler, for example, to test the routing strategies by using the visualisation tool. Finally, since the crossbar architecture provides some flexibility in the execution of its operations, as explained in Chapter 3, we have made some simplifications by adding some restrictions to the parameters of the operations.

Regarding the results, we have been able to analyze the mapping overhead in terms of gates and depth of the crossbar architecture. These experiments had lead to some insights into what kind of algorithms will produce a higher or lower mapping overhead. For example, we have seen that depending on the type of graph produced by the CNOT gates, the algorithm for the initial placement will return a better improvement. In addition, we have seen that the pattern of interlaced CNOT gates stops the scheduler from grouping single-qubit gates with the semi-global rotation scheme, producing a higher depth overhead in the final circuit. Also, we have shown that depending on the dependency graph of the quantum algorithm, the depth overhead will scale differently. And the worst case occurs when the dependency graph forms a tree. Moreover, we have shown that the depth overhead of consecutive single-qubit gates and phase shift gates scales sublinearly with the number of qubits. As a final note, since we have shown that the characteristics of quantum algorithms influence in the behaviour of the depth overhead, we can conclude that, for real applications, these characteristics must be taken into account.

7.2. Future work

During the design and implementation of the crossbar compiler, we have mentioned some additional lines of work that are worth looking into, such as the following:

Initial placement: The implementation of the initial placement is done in a different process than the routing algorithm. Another approach is to merge these two stages into one algorithm to improve the performance and reduce the number of SWAPs introduced.

Division of kernels: Evidently, the most restrictive aspect of the mapping design is the division of kernels. We can improve this by using the rest of approaches mentioned in Chapter 4: backtracking and suffering a side effect.

Line by line operation: In [23], R. Li et al. explained that some imperfections in the manufacturing of the crossbar quantum chip might be difficult to avoid. These imperfections means that for some operations, such as shuttling, the barriers might need to be raised at different times depending on the sites of the crossbar that are using it. To solve this problem, they propose a line-by-line shuttling. This approach needs to be implemented and analyzed in the compiler.

Arbitrary operations: The semi-global rotation allows to execute any single-qubit gate rotation and depending on the rotation it will take shorter or longer. However, it is convenient to use cycles in the compiler in order to measure the circuit depth. We need to investigate how we could measure the time.

Ancilla qubits: In the experiments we have left out the analysis of the ancilla qubits. Future work will analyse the position of the ancilla qubits in the crossbar and how they are selected to performed the measurement.

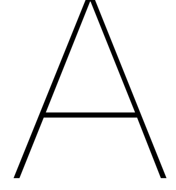
More characteristics of the algorithms: In this thesis we have shown that some characteristics like the pattern of gates in an algorithm can influence the mapping overhead. We proposed looking into more characteristics that make a quantum algorithm produce a low mapping overhead in the crossbar architecture.

CNOT interactions: In the thesis, we have also highlighted the influence of the graphs formed by CNOT gates on the improvement of the initial placement. We think it is worth studying the depth overhead based on the topology of these graphs. For example, comparing graph characteristics, such as the size of the cliques.

Quantify the characteristics: Finally, some of the characteristics shown in the experiments were not quantified. For example, we measured the overhead of the pattern of gates based on two examples. We think that is a necessary to come up with a metric to quantify it, for example the percentage of interlaced CNOT gates. Without these metrics is difficult to predict the mapping overhead of a particular algorithm.

The compiler built in this thesis targets only a particular quantum chip. However, there is already research in creating a more general compiler by creating a model based on time planning [43]. Although these compilers allow to model the constraints of more architectures, there are problems when encoding these constraints due to the limitations of the language used, such as *Planning Domain Definition Language* (PDDL). We think that encoding the

model and constraints of the architecture into languages, like C++, could be more flexible and easier when targeting different quantum processor architectures.



Gate decomposition

To be able to run any benchmark in the crossbar architecture, the gates must be decomposed into the support set of gates. In this case our set of gates was: one-qubit gates and the $\sqrt{SWAP}$ gate; although we give preference to the single-qubit phase rotations because they require less time to execute. In this appendix we show how we have derived to the decompositions shown in section 3.5. We will start with the decomposition of the CPHASE since it is the basic block from which the rest of gates are decomposed.

A.1. CPHASE Decomposition

We can check that the CPHASE gate can be expressed based on $\sqrt{SWAP}$ gates and phase shift gates, as shown in Equation A.1.

$$\begin{aligned} CPHASE &= \sqrt{SWAP} \times (S^{\dagger} \otimes S) \times \sqrt{SWAP} \otimes (I \otimes Z) \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2}(1+i) & \frac{1}{2}(1+i) & 0 \\ 0 & \frac{1}{2}(1-i) & \frac{1}{2}(1+i) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2}(1+i) & \frac{1}{2}(1+i) & 0 \\ 0 & \frac{1}{2}(1-i) & \frac{1}{2}(1+i) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &\times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \end{aligned} \tag{A.1}$$

A.2. CNOT Decomposition

In this section, we will show how we have derive the decomposition of the CNOT based on the previous CPHASE decomposition.

We can verify that $X = R_y(\frac{\pi}{2}) \times Z \times R_y(\frac{-\pi}{2})$ (as shown in Equation A.2).

$$X = \begin{pmatrix} \frac{1}{2}(1+i) & \frac{1}{2}(-1-i) \\ \frac{1}{2}(1+i) & \frac{1}{2}(1+i) \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} \frac{1}{2}(1-i) & \frac{1}{2}(1-i) \\ \frac{1}{2}(-1+i) & \frac{1}{2}(1-i) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{A.2}$$

Thus, we can express a CNOT gate as $(I \otimes R_y(\frac{-\pi}{2}) \times CPHASE \times (I \otimes R_y(\frac{\pi}{2}))$ (a visual representation is shown in Figure A.1).

Then we can decompose the CPHASE gate based on the decomposition of Section A.1.The result of such decomposition is shown in Figure A.1.

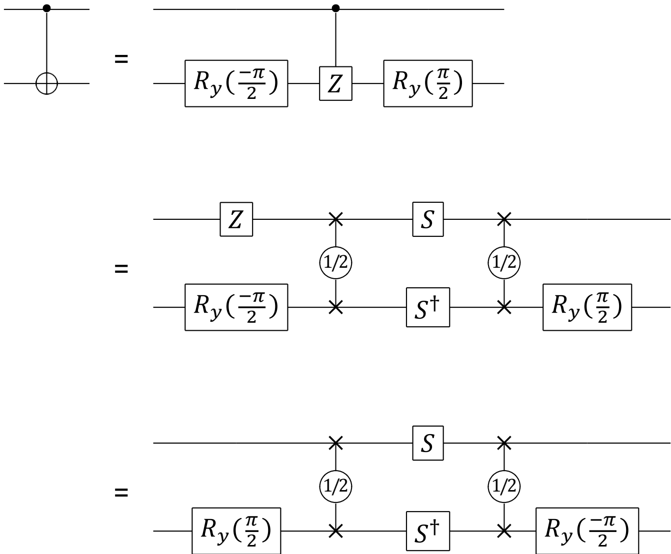


Figure A.1: A quantum circuit for executing the CNOT gate based on supported gates in the crossbar architecture.

B

Benchmarks

In this appendix, we show all the benchmarks used in chapter 6. For readability purpose, we have assigned a number (identifier) to each benchmark. The following table shows what identifier corresponds to what benchmark. It also shows the main characteristics of each benchmark used in previous chapters.

Table B.1: The benchmarks used in the experiments of chapter 6

Id	**Name**	**Qubits**	**Gates**	**Depth**
1	0410184_169	14	211	104
2	3_17_13	3	36	22
3	4_49_16	5	217	125
4	4gt10-v1_81	5	148	84
5	4gt11_82	5	27	20
6	4gt11_83	5	23	16
7	4gt11_84	5	18	11
8	4gt12-v0_86	6	251	135
9	4gt12-v0_87	6	247	131
10	4gt12-v0_88	6	194	108
11	4gt12-v1_89	6	228	130
12	4gt13_90	5	107	65
13	4gt13_91	5	103	61
14	4gt13_92	5	66	38
15	4gt13-v1_93	5	68	39
16	4gt4-v0_72	6	258	137
17	4gt4-v0_73	6	395	227
18	4gt4-v0_78	6	235	137
19	4gt4-v0_79	6	231	132
20	4gt4-v0_80	6	179	101
21	4gt4-v1_74	6	273	154
22	4gt5_75	5	83	47
23	4gt5_76	5	91	56
24	4gt5_77	5	131	74
25	4mod5-bdd_287	7	70	41
26	4mod5-v0_18	5	69	40
27	4mod5-v0_19	5	35	21
28	4mod5-v0_20	5	20	12
29	4mod5-v1_22	5	21	12

30	4mod5-v1_23	5	69	41
31	4mod5-v1_24	5	36	21
32	4mod7-v0_94	5	162	92
33	4mod7-v1_96	5	164	94
34	9symml_195	11	34881	19235
35	adr4_197	13	3439	1839
36	aj-e11_165	5	151	86
37	alu-bdd_288	7	84	48
38	alu-v0_26	5	84	49
39	alu-v0_27	5	36	21
40	alu-v1_28	5	37	22
41	alu-v1_29	5	37	22
42	alu-v2_30	6	504	285
43	alu-v2_31	5	451	255
44	alu-v2_32	5	163	92
45	alu-v2_33	5	37	22
46	alu-v3_34	5	52	30
47	alu-v3_35	5	37	22
48	alu-v4_36	5	115	66
49	alu-v4_37	5	37	22
50	C17_204	7	467	253
51	clip_206	14	33827	17879
52	cm152a_212	12	1221	684
53	cm42a_207	14	1776	940
54	cm82a_208	8	650	337
55	cm85a_209	14	11414	6374
56	cnt3-5_179	16	175	61
57	cnt3-5_180	16	485	209
58	co14_215	15	17936	8570
59	con1_216	9	954	508
60	cycle10_2_110	12	6050	3386
61	dc1_220	11	1914	1038
62	dc2_222	15	9462	5242
63	decod24-bdd_294	6	73	40
64	decod24-enable_126	6	338	190
65	decod24-v0_38	4	51	30
66	decod24-v1_41	5	85	50
67	decod24-v2_43	4	52	30
68	decod24-v3_45	5	150	84
69	dist_223	13	38046	19694
70	ex-1_166	3	19	12
71	ex1_226	6	7	5
72	ex2_227	7	631	355
73	ex3_229	6	403	226
74	f2_232	8	1206	668
75	graycode6_47	6	5	5
76	ham15_107	15	8763	4819
77	ham3_102	3	20	13
78	ham7_104	7	320	185
79	hwb4_49	5	233	134
80	hwb5_53	6	1336	758
81	hwb6_56	7	6723	3736
82	hwb7_59	8	24379	13437

83	hwb8_113	9	69380	38717
84	hwb9_119	10	207775	116199
85	inc_237	16	10619	5863
86	ising_model_10	10	200	70
87	ising_model_13	13	263	71
88	ising_model_16	16	326	71
89	life_238	11	22445	12511
90	majority_239	7	612	344
91	max46_240	10	27126	14257
92	miller_11	3	50	29
93	mini_alu_305	10	173	69
94	mini-alu_167	5	288	162
95	misex1_241	15	4813	2676
96	mlp4_245	16	18852	10328
97	mod10_171	5	244	139
98	mod10_176	5	178	101
99	mod5adder_127	6	555	302
100	mod5d1_63	5	22	13
101	mod5d2_64	5	53	32
102	mod5mils_65	5	35	21
103	mod8-10_177	6	440	251
104	mod8-10_178	6	342	193
105	one-two-three-v0_97	5	290	163
106	one-two-three-v0_98	5	146	82
107	one-two-three-v1_99	5	132	76
108	one-two-three-v2_100	5	69	40
109	one-two-three-v3_101	5	70	40
110	plus63mod4096_163	13	128744	72246
111	plus63mod8192_164	14	187112	105142
112	pm1_249	14	1776	940
113	qft_10	10	110	63
114	qft_16	16	272	105
115	radd_250	13	3213	1781
116	rd32_270	5	84	47
117	rd32-v0_66	4	34	20
118	rd32-v1_68	4	36	21
119	rd53_130	7	1043	569
120	rd53_131	7	469	261
121	rd53_133	7	580	327
122	rd53_135	7	296	159
123	rd53_138	8	132	56
124	rd53_251	8	1291	712
125	rd53_311	13	275	124
126	rd73_140	10	230	92
127	rd73_252	10	5321	2867
128	rd84_142	15	343	110
129	rd84_253	12	13658	7261
130	root_255	13	17159	8835
131	sao2_257	14	38577	19563
132	sf_274	6	781	436
133	sf_276	6	778	435
134	sqn_258	10	10223	5458
135	sqrt8_260	12	3009	1659

136	squar5_261	13	1993	1049
137	square_root_7	15	7630	3847
138	sym10_262	12	64283	35572
139	sym6_145	7	3888	2187
140	sym6_316	14	270	135
141	sym9_146	12	328	127
142	sym9_148	10	21504	12087
143	sym9_193	11	34881	19235
144	sys6-v0_111	10	215	75
145	urf1_149	9	184864	99585
146	urf1_278	9	54766	30955
147	urf2_152	8	80480	44100
148	urf2_277	8	20112	11390
149	urf3_155	10	423488	229365
150	urf3_279	10	125362	70702
151	urf4_187	11	512064	264330
152	urf5_158	9	164416	89145
153	urf5_280	9	49829	27822
154	urf6_160	15	171840	93645
155	wim_266	11	986	514
156	xor5_254	6	7	5
157	z4_268	11	3073	1644

Bibliography

[1] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32, 06 2012. doi: 10.1109/TCAD.2013.2244643.

[2] QCA Lab at TU Delft. OpenQL: Quantum Compiler. `https://github.com/QE-Lab/OpenQL`, 2019. [Online; accessed 2-August-2019].

[3] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995. doi: 10.1103/PhysRevA.52.3457. URL `https://link.aps.org/doi/10.1103/PhysRevA.52.3457`.

[4] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, C. Neill, P. O'Malley, P. Roushan, A. Vainsencher, J. Wenner, A. N. Korotkov, A. N. Cleland, and John M. Martinis. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature*, 508(7497):500–503, apr 2014. doi: 10.1038/nature13171. URL `https://doi.org/10.1038/nature13171`.

[5] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 11–20, New York, NY, USA, 1993. ACM. ISBN 0-89791-591-7. doi: 10.1145/167088.167097. URL `http://doi.acm.org/10.1145/167088.167097`.

[6] Sergey Bravyi, David Gosset, and Robert König. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, 2018. ISSN 0036-8075. doi: 10.1126/science.aar3106. URL `https://science.sciencemag.org/content/362/6412/308`.

[7] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the Qubit Routing Problem. In Wim van Dam and Laura Mancinska, editors, *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, volume 135 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-112-2. doi: 10.4230/LIPIcs.TQC.2019.5. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10397`.

[8] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A*, 86:032324, Sep 2012. doi: 10.1103/PhysRevA.86.032324. URL `https://link.aps.org/doi/10.1103/PhysRevA.86.032324`.

[9] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels. A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 323–330, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2906827. URL `http://doi.acm.org/10.1145/2903150.2906827`.

[10] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels. A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 323–330, New York, NY,

USA, 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2906827. URL http://doi.acm.org/10.1145/2903150.2906827.

[11] Google. Quantum - google ai, 2019. URL https://ai.google/research/teams/applied-science/quantum/.

[12] Wakaki Hattori and Shigeru Yamashita. Quantum circuit optimization by changing the gate order for 2d nearest neighbor architectures. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, pages 228–243, Cham, 2018. Springer International Publishing. ISBN 978-3-319-99498-7.

[13] Jonas Helsen, Mark Steudtner, Menno Veldhorst, and Stephanie Wehner. Quantum error correction in crossbar architectures. *Quantum Science and Technology*, 3(3):035005, 2018. URL http://stacks.iop.org/2058-9565/3/i=3/a=035005.

[14] Charles D. Hill, Eldad Peretz, Samuel J. Hile, Matthew G. House, Martin Fuechsle, Sven Rogge, Michelle Y. Simmons, and Lloyd C. L. Hollenberg. A surface code quantum computer in silicon. *Science Advances*, 1(9), 2015. doi: 10.1126/sciadv.1500707. URL https://advances.sciencemag.org/content/1/9/e1500707.

[15] Adam Holmes, Sonika Johri, Gian Giacomo Guerreschi, James S. Clarke, and A. Y. Matsuura. Impact of qubit connectivity on quantum algorithm performance, 2018.

[16] IBM. Ibm, ibm q experience backend information, 2018. URL https://github.com/QISKit/ibmqx-backend-information.

[17] 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.

[18] Will Knight. Ibm raises the bar with a 50-qubit quantum computer, 2017. URL https://www.technologyreview.com/s/609451/ibm-raises-the-bar-with-a-50-qubit-quantum-computer/.

[19] QCA Lab. libqasm: Library to parse cqasm v1.0 files. https://github.com/QE-Lab/libqasm, 2019.

[20] Lingling Lao, Daniel M. Manzano, Hans van Someren, Imran Ashraf, and Carmen G. Almudever. Mapping of quantum circuits onto nisq superconducting processors, 2019.

[21] Bjoern Lekitsch, Sebastian Weidt, Austin G Fowler, Klaus Mølmer, Simon J Devitt, Christof Wunderlich, and Winfried K Hensinger. Blueprint for a microwave trapped ion quantum computer. *Science advances*, 3(2):e1601540, February 2017. ISSN 2375-2548. doi: 10.1126/sciadv.1601540. URL http://europepmc.org/articles/PMC5287699.

[22] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. *ArXiv*, abs/1809.02573, 2018.

[23] Ruoyu Li, Luca Petit, David P. Franke, Juan Pablo Dehollain, Jonas Helsen, Mark Steudtner, Nicole K. Thomas, Zachary R. Yoscovits, Kanwal J. Singh, Stephanie Wehner, Lieven M. K. Vandersypen, James S. Clarke, and Menno Veldhorst. A crossbar network for silicon quantum dot qubits. *Science Advances*, 4(7), 2018. doi: 10.1126/sciadv.aar3960. URL http://advances.sciencemag.org/content/4/7/eaar3960.

[24] Chia-Chun Lin, Amlan Chakrabarti, and Niraj K. Jha. Qlib: Quantum module library. *J. Emerg. Technol. Comput. Syst.*, 11(1):7:1–7:20, October 2014. ISSN 1550-4832. doi: 10.1145/2629430. URL http://doi.acm.org/10.1145/2629430.

[25] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. Qiskit backend specifications for openqasm and openpulse experiments, 2018.

[26] N. David Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007. doi: 10.1017/CBO9780511813870.

[27] Microsoft. Quantum computing | micrsoft, 2019. URL `https://www.microsoft.com/en-us/quantum/`.

[28] A. R. Mills, D. M. Zajac, M. J. Gullans, F. J. Schupp, T. M. Hazard, and Jason R. Petta. Shuttling a single charge across a one-dimensional array of silicon quantum dots. *Nature Communications*, 10(1), 12 2019. ISSN 2041-1723. doi: https://doi.org/10.1038/s41467-019-08970-z.

[29] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 527–540, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322273. URL `http://doi.acm.org/10.1145/3307650.3322273`.

[30] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011. ISBN 1107002176, 9781107002173.

[31] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. ISSN 2521-327X. doi: 10.22331/q-2018-08-06-79. URL `https://doi.org/10.22331/q-2018-08-06-79`.

[32] Chad Rigetti. The rigetti 128-qubit chip and what it means for quantum, 2019. URL `https://medium.com/rigetti/the-rigetti-128-qubit-chip-and-what-it-means-for-quantum-df757d1b71ea`.

[33] Joschka Roffe. Quantum error correction: An introductory guide, 2019.

[34] Kamyar Saeedi, Stephanie Simmons, Jeff Z. Salvail, Phillip Dluhy, Helge Riemann, Nikolai V. Abrosimov, Peter Becker, Hans-Joachim Pohl, John J. L. Morton, and Mike L. W. Thewalt. Room-temperature quantum bit storage exceeding 39 minutes using ionized donors in silicon-28. *Science*, 342(6160):830–833, 2013. ISSN 0036-8075. doi: 10.1126/science.1239584. URL `https://science.sciencemag.org/content/342/6160/830`.

[35] Norbert Schuch and Jens Siewert. Natural two-qubit gate for quantum computation using the XY interaction. *Phys. Rev. A*, 67:032301, Mar 2003. doi: 10.1103/PhysRevA.67.032301. URL `https://link.aps.org/doi/10.1103/PhysRevA.67.032301`.

[36] A. Shafaei, M. Saeedi, and M. Pedram. Qubit placement to minimize communication overhead in 2d quantum architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 495–500, Jan 2014. doi: 10.1109/ASPDAC.2014.6742940.

[37] A. Shafaei, M. Saeedi, and M. Pedram. Qubit placement to minimize communication overhead in 2d quantum architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 495–500, Jan 2014. doi: 10.1109/ASPDAC.2014.6742940.

[38] Vivek V. Shende and Igor L. Markov. On the cnot-cost of toffoli gates. *Quantum Information and Computation*, 9, 03 2008.

[39] Marcos Yukio Siraichi, Vinicius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. Qubit Allocation. In *CGO 2018 - International Symposium on Code Generation and Optimization*, pages 1–12, Vienna, Austria, February 2018. doi: 10.1145/3168822. URL `https://hal.archives-ouvertes.fr/hal-01655951`.

[40] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture, 2016.

[41] Swamit S. Tannu and Moinuddin K. Qureshi. Not all qubits are created equal: A case for variability-aware policies for nisq-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 987–999, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304007. URL http://doi.acm.org/10.1145/3297858.3304007.

[42] Menno Veldhorst, H. G. J. Eenink, Chao He Yang, and Andrew Dzurak. Silicon cmos architecture for a spin-based quantum computer. In *Nature Communications*, 2017.

[43] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Compiling quantum circuits to realistic hardware architectures using temporal planners. *Quantum Science and Technology*, 3(2):025004, feb 2018. doi: 10.1088/2058-9565/aaa331.

[44] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. Revlib: An online resource for reversible functions and reversible circuits. In *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, pages 220–225, May 2008. doi: 10.1109/ISMVL.2008.43.

[45] A. Zulehner, A. Paler, and R. Wille. Efficient mapping of quantum circuits to the ibm qx architectures. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1135–1138, March 2018. doi: 10.23919/DATE.2018.8342181.

[46] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures, 2017.