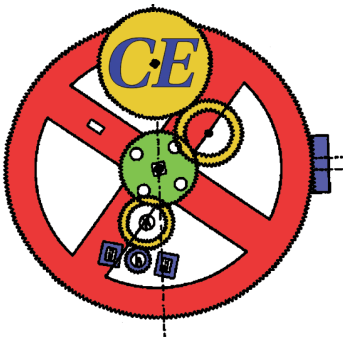


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

Pauli Frames for Quantum Computer Architectures

Leon Riesebos

CE-MS-2016

Abstract

Quantum computers hold the promise to solve problems that are intractable to classical computers. Since qubits suffer from extremely short lifetime and unreliable operations, Quantum Error Correction (QEC) forms a vital part of a quantum computer to enable Fault-Tolerant quantum computing. The usage of a Pauli frame can relax the time constraints on QEC by keeping track of detected errors in classical logic. For the first time, in the background of a heterogeneous quantum computer architecture, we clarified the input/output and working principles of a Pauli frame, which can soon be mapped to a hardware implementation. We proposed the first functional quantum computer architecture simulation platform, QPDO, which can connect to different quantum simulators, such as QX Simulator or CHP, and is used to verify the logical operations of a Surface Code 17 (SC17) logical qubit. Finally, by using QPDO we found that a Pauli frame does not improve the Logical Error Rate (LER) of a SC17 logical qubit, which is opposite to previous understanding. By further reasoning, we also expect no improvement in LER by using a Pauli frame for surface codes with a larger distance. Nevertheless, the usage of a Pauli frame is still crucial for relaxing the timing constraints on QEC.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Pauli Frames for Quantum Computer Architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Leon Riesebos
born in Amsterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Pauli Frames for Quantum Computer Architectures

by Leon Riesebos

Abstract

Quantum computers hold the promise to solve problems that are intractable to classical computers. Since qubits suffer from extremely short lifetime and unreliable operations, Quantum Error Correction (QEC) forms a vital part of a quantum computer to enable Fault-Tolerant quantum computing. The usage of a Pauli frame can relax the time constraints on QEC by keeping track of detected errors in classical logic. For the first time, in the background of a heterogeneous quantum computer architecture, we clarified the input/output and working principles of a Pauli frame, which can soon be mapped to a hardware implementation. We proposed the first functional quantum computer architecture simulation platform, QPDO, which can connect to different quantum simulators, such as QX Simulator or CHP, and is used to verify the logical operations of a Surface Code 17 (SC17) logical qubit. Finally, by using QPDO we found that a Pauli frame does not improve the Logical Error Rate (LER) of a SC17 logical qubit, which is opposite to previous understanding. By further reasoning, we also expect no improvement in LER by using a Pauli frame for surface codes with a larger distance. Nevertheless, the usage of a Pauli frame is still crucial for relaxing the timing constraints on QEC.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016

Committee Members :

Advisor:	Koen Bertels, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Carmen G. Almudever, CE, TU Delft
Member:	Tim H. Taminiau, QuTech, TU Delft
Member:	Leo DiCarlo, QuTech, TU Delft
Member:	Xiang Fu, CE, TU Delft

To my parents, sister, and friends

Contents

List of Figures

List of Tables

List of Acronyms

ESM Error Syndrome Measurement

FT Fault-Tolerant

LER Logical Error Rate

LUT Look-Up Table

PEL Physical Execution Layer

PER Physical Error Rate

PF Pauli Frame

PFU Pauli Frame Unit

QASM Quantum Assembly

QCI Quantum-Classical Interface

QCU Quantum Control Unit

QEC Quantum Error Correction

QED Quantum Error Detection

QEX Quantum Execution

QISA Quantum Instruction Set Architecture

QPDO Quantum Platform Development framewOrk

SC17 Surface Code 17

SCC Surface Code Cycle

Acknowledgements

I would like to express my very great appreciation to my Thesis advisor Dr. Koen Bertels for providing me the opportunity to do an MSc Thesis on the interesting topic of quantum computing, for the valuable advice during my work, and for the great pizza/beer/wine evenings. Also, I would like to offer my special thanks to my daily supervisor Xiang Fu. His extraordinary motivation and support has been of great importance and has been very much appreciated.

I would like to thank my colleagues from the Quantum Computing Team of the Computer Engineering department for their support on my work. Especially I would like to mention Nader Khammassi for his support on the QX server, Savvas Varsamopoulos for providing me with the decoder software, and Dan Iorga for helping me with the flexible wrapper for CHP. It has been a pleasure to work with all of you as a team, and I hope we will continue to work together for a couple more years.

I wish to thank my parents and sister for their unconditional support and continuous encouragement throughout all the years of my study. This academic, as well as personal accomplishment, would not have been possible without them.

Finally, I would like to show my gratitude to the friends that have been standing by my side for the last years. I have always been able to count on them during the good times and the bad times.

Thank you.

Leon Riesebos
Delft, The Netherlands
June 23, 2016

1 Introduction

Quantum computing is an emerging technique that promises to solve problems in a reasonable time which are intractable by classical computers. For certain problems, quantum computers running specialized quantum algorithms can gain exponential speedup compared to classical computers running equivalent classical algorithms. The speedup is caused by exploiting quantum phenomena (i.e. superposition and entanglement) for computational purposes using qubits. Well-known applications of quantum algorithms are factoring large numbers using Shor's algorithm [1] and searching large data sets using Grover's search algorithm [2], but applications can also be found in the field of chemistry, optimization, and quantum simulation.

Superposition refers to the fact that qubits can reside in not only a single state but also a superposition of states. A classical bit has two exclusive states, 0 or 1, and can only be in one state at any point in time. A qubit however has two basis states, $|0\rangle$ and $|1\rangle$, and can be in a superposition of both states represented as a linear combination of $|0\rangle$ and $|1\rangle$: $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ are the probability amplitudes satisfying the normalization condition $|\alpha|^2 + |\beta|^2 = 1$, and $|\alpha|^2$ / $|\beta|^2$ represent the probability of getting the measurement result $+1/-1$ (resulting state $|0\rangle/|1\rangle$) respectively when measuring the qubit in the basis $\{|0\rangle, |1\rangle\}$. Note that the action of measuring the qubit will project the state of the qubit onto one of the measurement basis states which means that a quantum state cannot be measured directly without losing the information stored.

In classical computing, a system composed by n classical bits can only store and process one of the 2^n possible states at a time. However, in quantum computing multiple qubits can be combined, resulting in a new state that is a superposition of all 2^n possible states $|\psi\rangle = \alpha_0\,|0\ldots00\rangle + \alpha_1\,|0\ldots01\rangle + \ldots + \alpha_{2^n-1}\,|1\ldots11\rangle$, where $\alpha_i \in \mathbb{C},\ \sum |\alpha_i|^2 = 1$. Entanglement is a special case of such combination meaning that the combined qubit state cannot be decomposed into separate states. When applying a (quantum) operation on those combined qubits, the operation is applied on all 2^n possible states at the same time.

Various implementations of qubits and small quantum systems already exist, and all of them share one property: qubit states are very fragile. Qubits interact with the environment and information stored in the qubits tends to get corrupted, which is known as decoherence. Due to decoherence, qubits cannot reliably store information for enough time and quantum operations are error prone. For example, superconducting qubits may lose their information in tens of microseconds [3, 4]. Also, quantum operations are unreliable with error rates around 0.1% [5, 3, 4]. Quantum algorithms require qubits with long coherence time and operations with high fidelity to perform meaningful computations, making existing qubits unsuitable to use directly for computational purposes. For example, factoring a 2000-bit number using Shor's algorithm is estimated to require error rates below 4×10^{-13} [6, Appendix M] being far below current quantum operation

error rates.

To enable quantum computing using qubits with high error rates, Quantum Error Correction (QEC) was introduced [7]. QEC encodes a single quantum state in a logical qubit created from multiple physical qubits and can detect errors on physical qubits based on error syndromes. These error syndromes are created by repeatedly executing Error Syndrome Measurement (ESM) circuits on the qubits. The error syndromes are decoded using classical algorithms and make it possible to find and correct erroneous qubits in the system. By using QEC, we can create logical qubits that have lower error rates than the physical qubits they were made of, making it possible to satisfy the demands of quantum algorithms to have qubits with low error rates.

Besides from the benefits, QEC introduces overhead and new challenges. Execution of ESM circuits has to be performed in short time periods putting high demands on qubit gate and measurement times. Also, decoding of error syndromes should be executed in a very short period to prevent QEC procedures from stalling. The requirement of fast error decoding introduces high demands on the classical algorithms and computational devices.

The concept of Pauli frames was introduced [8] to loosen the timing constraints on ESM circuits and decoding algorithm execution time. A Pauli frame consists of a combination of classical memory and logic that can track the errors of qubits. By using a Pauli frame, qubit errors found by decoding error syndromes can be tracked in classical electronics, making it unnecessary to apply corrections on physical qubits. The Pauli frame can loosen the timing constraints on qubit measurement times and decoding algorithms, making it easier to implement fully functional QEC.

In this work, we investigate how to implement a Pauli frame, and we propose a Pauli frame implementation as part of a quantum computer architecture targeted for a Surface Code 17 (SC17) quantum chip. We perform functional simulations of quantum computer architectures with integrated QEC and verify the fault-tolerant operations of SC17 systems. We also check the control logic of our Pauli frame implementation by simulation. Finally, we study the effect of a Pauli frame on the error rate of a SC17 logical qubit. Quantum simulations are performed with the universal quantum simulator QX Simulator [9] and the stabilizer simulator CHP [10]. For simulating the quantum computer architecture and the Pauli frames, we used the QPDO software which is specifically developed for this work.

This report is organized as follows: Chapter 2 provides a background to introduce the reader to the concepts used throughout this report. Chapter 3 introduces the reader to the concept and working principles of Pauli frames. In this chapter, we will also discuss the benefits of a Pauli frame and propose an implementation as part of a quantum computer architecture. In Chapter 4, we introduce the simulations software that was used for our experiments which include a self-developed framework for functional simulation of quantum computer architectures: QPDO. Chapter 5 discusses the experiments we performed and the results of those experiments. These experiments include verification of the logical operations for a SC17 logical qubit, verification of the Pauli frame working principles, and experiments to observe the impact of a Pauli frame on the error rate of a SC17 logical qubit. Finally we conclude in Chapter 6.

2 Background

The fundamental elements of quantum computers are quantum bits, also known as qubits. Qubits can be represented as mathematical objects, but can also be realized in a physical system, just as classical bits. Bits and qubits are related to each other, but qubits extend the features of bits. The two extending features are superposition and entanglement.

2.1 Qubits

Where classic bits can only be in a 0 or 1 state at a certain point in time, qubits can be in a superposition of both. Two basis states are defined: $|0\rangle$ and $|1\rangle$. Qubits can be in a linear combination of both states and are therefor represented like: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ are complex probability amplitudes. The sum of all probabilities within a system should always be 1, therefor: $|\alpha|^2 + |\beta|^2 = 1$. The state of a qubit can be treated as a unit column vector in a two dimensional complex space $\mathbb{C}^2$. This vector lives in the so called Hilbert space $\mathcal{H}$. The states $|0\rangle$ and $|1\rangle$ are an orthonormal basis for $\mathcal{H}$, also known as the computational basis. By defining $|0\rangle = \begin{bmatrix}1 & 0\end{bmatrix}^T$ and $|1\rangle = \begin{bmatrix}0 & 1\end{bmatrix}^T$, we can introduce the vector notation of a qubit state as shown in Equation (2.1). This notation is known as the Dirac bra-ket notation.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha\begin{bmatrix}1\\0\end{bmatrix} + \beta\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\alpha\\\beta\end{bmatrix} \quad \text{where} \quad |\alpha|^2 + |\beta|^2 = 1 \tag{2.1}$$

The probabilistic nature of qubits presents themselves when being measured. When measuring a single qubit in the computational basis, there is a probability of $|\alpha|^2$ to measure $|0\rangle/+1$ and a probability of $|\beta|^2$ to measure $|1\rangle/-1$. Observing a qubit collapses its state to one of the possible measurement outcomes and destroys the information in the probability amplitudes. As a result, α and β are not directly observable.

For an n-qubit system, the global state is represented as a column vector with $2^n = N$ entries. For example, an $n = 2$ qubit system has $N = 4$ basis states and can, therefore, be fully described by four complex amplitudes. The state vector of a two-qubit system lives in a four-dimensional Hilbert space.

The state of a quantum system $|\psi\rangle$ has a global phase δ. Every state $|\psi\rangle$ could therefore be written like $e^{i\delta}|\psi\rangle$. Since the global phase has no observable influence on the measurement results, it is in general ignored.

The second feature of qubits, that extends the capabilities of classical bits, is entanglement. Qubits can be entangled with each other, which means that the state of the entire system cannot be represented as the production of individual qubit state anymore. An example of a non-entangled state is $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. This state can still be written as the tensor product of two individual states and is therefor

not entangled. The state $|\Phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled since it can not be written as two individual states.

2.2 Quantum gates

To manipulate a qubit state, we use quantum gates which can be expressed as matrices. Applying a gate can be modeled as a matrix-vector multiplication. A qubit state vector is always a unit vector and for that reason, matrices representing quantum gates are always unitary. For unitary matrices it holds that:

$$UU^\dagger \;=\; U^\dagger U \;=\; I \tag{2.2}$$

The fact that quantum gates are unitary matrices implies that quantum gates are always reversible.

2.2.1 Single-qubit gates

A single qubit gate can be represented as a 2×2 unitary matrix. The simplest single qubit gate is the Pauli-X gate. The Pauli-X gate transforms the state $\alpha|0\rangle + \beta|1\rangle$ to $\beta|0\rangle + \alpha|1\rangle$. The matrix representation of the Pauli-X gate is shown in Equation (2.3).

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.3}$$

A few other basic single qubit gates are the Pauli-Y, Pauli-Z, and Hadamard(H) gate. Especially the H gate is interesting since it maps a computational basis state to a superposition state. Their corresponding matrices are:

$$Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad , \quad Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad , \quad H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.4}$$

The Z gate is part of the Z-axis rotation-gate family. The general form of a Z-axis rotation-gate is shown in Equation (2.5). The rotation angle $\theta \in \mathbb{R}$ only has an influence on the $|1\rangle$ part of the qubit state. When $\theta = \pi$, the rotation-gate is equivalent to a Z gate. Other common rotation angles are $\theta = \frac{\pi}{2}$ and $\theta = \frac{\pi}{4}$ for the corresponding S and T gate. The matrices for the S and T gate are shown in Equation (2.6).

$$R_Z(\theta) \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \tag{2.5}$$

$$S \equiv R_Z\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad , \quad T \equiv R_Z\left(\frac{\pi}{4}\right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \tag{2.6}$$

2.2.2 Multi-qubit gates

Besides from single-qubit gates, there are also quantum gates applying on multiple qubits. Multi-qubit gates are essential for creating entanglement. A well known multi-qubit gate is the controlled NOT ($CNOT$) gate. $CNOT$ transfers the basis state $|a\rangle \otimes |b\rangle$ to

another basis state $|a\rangle \otimes |b \oplus a\rangle$, where $a, b \in \{0, 1\}$ and $\oplus$ is classical XOR operation. The $CNOT$ gate can also be interpreted as a controlled-X gate, which only applies an X gate to the target qubit if the control qubit is in the $|1\rangle$ state. The matrix representing the $CNOT$ gate looks as follows:

$$U_{CNOT} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.7}$$

There are more two-qubit gates and among them a simple type is that with a control qubit and a target qubit. A typical example is the controlled-Z gate, also known as the CZ gate. This gate performs a Z gate on the target qubit if the control qubit is in the $|1\rangle$ state. Three-qubit gates also exist and one commonly-used example is the $Toffoli$ gate, also known as the controlled-controlled NOT or $CCNOT$ gate. The $Toffoli$ gate works similar as the $CNOT$ gate but has two control qubits. Only if both control qubits are in the $|1\rangle$ state, the $Toffoli$ gate will apply a X operation to the target qubit.

2.3 Quantum math formalism

The mathematical model of quantum mechanics is mainly based on linear algebra. In this section, we introduce mathematical concepts which are used in the remainder of this report.

2.3.1 Group

As described in [11, Appendix 2], a group is defined as a 2-tuple $(E, \bullet)$ where E is a set and $\bullet$ is is the so-called group law. Groups are in general referred as E without the group law, but it must be defined. A subgroup F of E, noted as $F \subseteq E$, is a subset of E using the same group law as E. For E to be qualified as a group, the following requirements must hold:

1. $\forall a, b \in E$ it holds that $a \bullet b \in E$.
2. $\forall a, b, c, \in E$ it holds that $(a \bullet b) \bullet c = a \bullet (b \bullet c)$.
3. $\exists e \in E$ where $\forall a \in E$ it holds that $e \bullet a = a \bullet e = a$. Element e is the so called identity element.
4. $\forall a \in E$ there $\exists b \in E$ such that $a \bullet b = b \bullet a = e$.

To describe a group we can use group generators. The generators of a group S is a set $G_s \subseteq S$ such that each $s_i \in S$ can be expressed as a finite combination of $g_i \in G_s$ using the group law of S. The set generated by G_s is $\langle G_s \rangle$ and G_s contains the generators of $\langle G_s \rangle$. By describing a group by their generators, we have a compact notation that covers all the elements in a group.

A group can also have a normalizer. Assume we have a group $(T, \bullet)$, then the normalizer of a group $S \subseteq T$ is the group $N_S \subseteq T$ where $\forall n_i \in N_S$ and $\forall s_i \in S$ there

$\exists s_j \in S$ where it holds that $n_i \bullet s_i \bullet n_i^{-1} = s_j$. This means that the normalizer of S maps elements of S to elements of S.

2.3.2 Gate properties

Since all gates are defined by unitary matrices, we can derive useful mathematical properties. As mentioned earlier (see Section 2.2) all quantum gates have an inverse. The Pauli gates (X, Y, and Z) are not only unitary but also Hermitian. For Hermitian matrices Equation (2.8) holds. Now we can derive Equation (2.9) using both the unitary and Hermitian properties of the Pauli gates.

$$A = A^\dagger \tag{2.8}$$

$$XX^\dagger = XX = YY^\dagger = YY = ZZ^\dagger = ZZ = I \tag{2.9}$$

The Pauli gates also have other interesting properties. The X and Z gate anti-commute and the Y gate can be decomposed into an X, Z, and iI component as shown in Equation (2.10) and (2.11). The iI component trivially commutes with every matrix since it is a scalar matrix.

$$XZ = -ZX \tag{2.10}$$

$$Y = iXZ \tag{2.11}$$

The Hadamard gate is just like the Pauli gates a Hermitian matrix, as shown in Equation (2.12), and has some interesting relations with the Pauli gates. Those relations are shown in Equation (2.13) and (2.14).

$$HH^\dagger = HH = I \tag{2.12}$$

$$HX = ZH \tag{2.13}$$

$$HZ = XH \tag{2.14}$$

2.3.3 Gate groups

Quantum gates can be divided into groups as described in [12]. The infinite group $U(2^n)$ is the group of unitary transformations on n qubits with (matrix) multiplication as group law. The most basic subset of gates are the Pauli gates, which are part of the Pauli group. The Pauli group on n qubits is defined as $P_n \subset U(2^n)$. The Pauli group is finite and can be described by its generators G_{P_n} as shown in Equation (2.15) as taken from [12].

$$G_{P_n} = \langle iI, X_1, Z_1, \ldots, X_n, Z_n \rangle \tag{2.15}$$

Another group of gates are the Clifford gates. The Clifford group for n qubits $C_n \subset U(2^n)$ is defined in [12] as the normalizer for P_n as shown in Equation (2.16). We can conclude that gates of the Clifford group C_n map Pauli gates of the group P_n to Pauli gates of the same group P_n. The generators for C_2 are defined in [12] and can be found in Equation (2.17).

$$C_n = \{U \in U(2^n) | \forall P \in P_n, \exists P', UPU^\dagger = P'\} \tag{2.16}$$

$$G_{C_2} = \langle H_1, S_1, H_2, S_2, CNOT_{1,2} \rangle \tag{2.17}$$

The last group of interest is the group of non-Clifford gates. This infinite group consists of all the gates that are not in the Clifford group and is defined as $U_{NC}(2^n) = U(2^n) \setminus (P_n \cup C_n)$ for n qubits. Concrete examples of non-Clifford gates discussed in Section 2.2 are the T gate and the $Toffoli$ gate.

2.4 Universal quantum computation

For realizing universal quantum computation, there is a minimal set of requirements which includes a universal quantum gate set (as proposed in [11]). For universal quantum computation, the following conditions should be met:

1. Adequate amount of classical computing resources for controlling the quantum process.
2. Adequate amount of qubits to have a state space large enough to perform the computation.
3. Ability to initialize any qubit in the system to a computational basis state.
4. Ability to perform a universal set of quantum gates on any subset of qubits in the system.
5. Ability to perform measurements in the computational basis.

At this moment, we want to point out the definition of the universal set of quantum gates. Universal gates are a well-known concept in Boolean logic. For binary logic systems, having $NAND$ or NOR gates is enough to implement any arbitrary Boolean function. In quantum computing, a set of quantum gates is universal if any unitary operation can be approximated to arbitrary accuracy by a quantum circuit involving only gates in the set. An example of a universal set of quantum gates is the set $H, T, CNOT$. Any universal gate set should at least contain a multi-qubit gate and a non-Clifford gate.

2.5 Quantum error correction

Up to this point, we have described qubits as mathematical objects. Just as classical bits, qubits need to be physically realized to use them for actual calculations. There are many different ways to construct physical qubits, and all realizations have one factor in common: physical qubits suffer from decoherence. A qubit loses its state in a short period which makes it hard to store a quantum state for a long time. Also, the physical execution of initialization, gates and measurement operations is not perfect and can introduce errors.

For example, superconducting qubits may lose their information in tens of microseconds [3, 4] and quantum operations are unreliable with error rates around 0.1% [5, 3, 4]. Quantum algorithms require qubits with long coherence time and operations with high fidelity to perform meaningful computations, making existing qubits unsuitable to use directly for computational purposes. For example, factoring a 2000-bit number using Shor's algorithm is estimated to require error rates below 4×10^{-13} [6, Appendix M]

being far below current quantum operation error rates. To still be able to do meaningful computations using qubits with high error rates, Quantum Error Correction (QEC) was introduced [7].

A quantum state can be encoded redundantly to protect it from noise using QEC codes. These encoding schemes entangle multiple (physical) qubits to form a logical qubit that can store a single qubit state. By using QEC it is possible to create logical qubits that have lower error rates than their underlying physical qubits. These error rates are also referred to as the Logical Error Rate (LER) and Physical Error Rate (PER). In the next section, we will discuss a QEC scheme which is known as the surface code.

2.5.1 Surface code

The surface code is a topological QEC code [12] and is derived from Kitaev's toric code [13]. The surface code is a promising QEC code and attracts a lot of theoretical and experimental research [6, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22]. There are several different methods [6, 14, 23] to encode logical qubits using the surface code. The method studied in this thesis is to encode a logical qubit in a sheet or patch, also known as planar surface code, and will from now on be referred to as surface code. The surface code is interesting mainly because of its good tolerance to errors and convenient two-dimensional layout with only nearest-neighbor qubit interactions, which contributes in a positive way to the feasibility to manufacture such a system. The two-dimensional layout of the surface code can exist in various shapes and sizes. Data qubits, which hold the encoded state of the logical qubit, are aligned in a grid with ancilla qubits between them. Figure 2.1 displays an example using nine data qubits (in blue) and eight ancilla qubits (in red and green). The red/green ancilla qubits are used to check X/Z parity between neighboring data qubits. The red and green connection lines show how ancilla qubits interact to the data qubits. Since the surface code checks both X and Z parity, all types of errors can be detected. A surface code using 17 qubits, which is also known as Surface Code 17 (SC17) or a ninja star, can detect up to one error and is described in [19] by the stabilizers shown in Table 2.1. The additional stabilizers that are shown in Table 2.2 can be used to describe various logical states of a ninja star. More information about stabilizer formalism can be found in [11].

The example in Figure 2.1 shows a surface code with 17 qubits, but the surface can be extended by repeating the two-dimensional pattern. By increasing the size of the lattice, the distance d increases where the distance is defined by [6] as the minimum number of gates required to inflict a logical operation (more about logical operations in Section 2.6.1). The SC17 has a distance $d = 3$. The effect of a different distance d can be explained by the threshold p_{th} of a QEC code. The threshold p_{th} is defined by [6] as the PER p where for $p < p_{th}$, the LER P_L decreases exponentially with d, while for $p > p_{th}$, the LER P_L increases with d. If we work in the regime where the PER $p < p_{th}$, a larger distance d will increase the error tolerance of a QEC code and decrease the LER. For the surface code, the threshold is defined by [6] as $p_{th} = 5.7 \times 10^{-3}$. A QEC with a specified distance d also has a pseudo-threshold p_{pth}, which is defined by [19] as the PER p where for $p < p_{pth}$, the LER $P_L < p$, while for $p > p_{pth}$, $P_L > p$. For a certain QEC and a given distance d this means that if we work in the regime where

PER $p < p_{th}$, QEC yields logical qubits that have lower error rates than the underlying physical qubits.

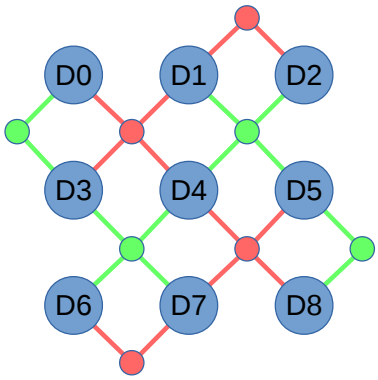


Figure 2.1: The layout of a surface code using 17 qubits, also known as the ninja star.

X Stabilizers	Z Stabilizers
$X_0X_1X_3X_4$	Z_0Z_3
X_1X_2	$Z_1Z_2Z_4Z_5$
$X_4X_5X_7X_8$	$Z_3Z_4Z_6Z_7$
X_6X_7	Z_5Z_8

Table 2.1: Stabilizers of the Surface Code 17.

Stabilizer	Logical state
$Z_0Z_4Z_8$	$\|0\rangle_L$
$-Z_0Z_4Z_8$	$\|1\rangle_L$
$X_2X_4X_6$	$\|+\rangle_L$
$-X_2X_4X_6$	$\|-\rangle_L$

Table 2.2: Additional stabilizers to describe the SC17 logical states.

The surface code can detect errors by executing an error correction circuit, also known as an Error Syndrome Measurement (ESM), which is performed during a Surface Code Cycle (SCC). When executing an ESM, all red and green ancilla qubits perform a particular circuit interacting only with their neighboring data qubits. These circuits are shown in Figure 2.2 and 2.3 and are also known as stabilizer circuits. The ancilla qubits interact with their specific neighboring data qubits at a predefined timing. The order of interacting with the data qubits is crucial to ensure correct functionality of the ESM. Neighboring data qubits of an ancilla qubit can only be accessed in a Z or S pattern. Figure 2.2 shows the S pattern while Figure 2.3 demonstrates the Z pattern. It is possible to use a single pattern or different patterns for red and green ancilla qubits. As shown in [19] it is preferred to use different patterns for the red and green ancilla

qubits to prevent error insertion in the logical state of the ninja star due to errors on ancilla qubits.

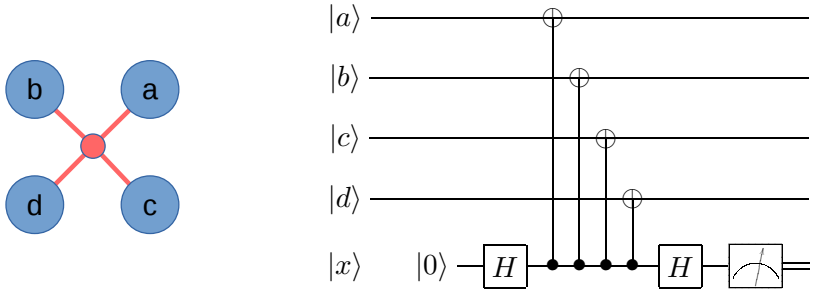


Figure 2.2: Left the orientation and naming of the qubits, right the circuit for the X parity checks.

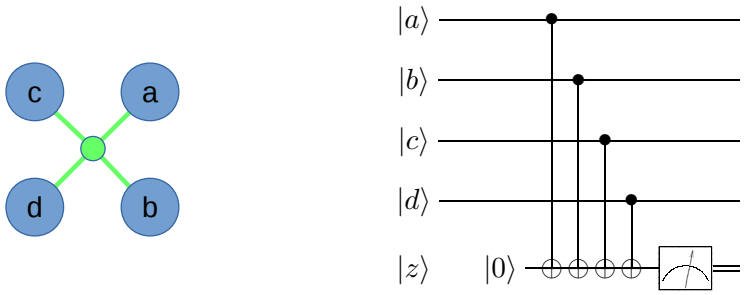


Figure 2.3: Left the orientation and naming of the qubits, right the circuit for the Z parity checks.

The execution of an ESM starts with initializing all ancilla qubits to $|0\rangle$ and applying a Hadamard gate to the red ancilla qubits. After that, all ancilla qubits perform their four $CNOT$ operations at the same time where the timing and order of the $CNOT$ operations are crucial for correct results. After the $CNOT$ operations, a Hadamard gate is applied on the red ancilla qubits before all ancilla qubits are measured. The measurement results of all ancilla qubits together form an error syndrome which can be decoded (using a decoder) to find the most likely error happened on the data qubits. The error found in the data qubits can then be corrected by applying a correction gate on the corresponding data qubit. When repeatedly executing the ESM procedure including error syndrome decoding and error correction, a logical state encoded in a surface code can be protected from errors for a longer period.

2.6 Fault-tolerant quantum computing

Encoding a quantum state using a QEC code can protect it against a certain level of noise, but more effort is required to realize a full Fault-Tolerant (FT) quantum system. Let us first quote a definition of fault-tolerance from [11, p. 476] to make the concept clear:

> "We define the fault-tolerance of a procedure to be the property that if only one component in the procedure fails then the failure causes at most one error in each encoded block of qubits output from the procedure."

The idea of FT quantum computing does not only require QEC but also requires us to perform operations on logical qubits without decoding the protected quantum state. Therefore, we need to have FT logical operations that operate directly on logical qubits and prevent us from decoding an encoded state at any time. To make the concept of FT quantum computing universal, both QEC as well as all required logical operations for universal quantum computing (see Section 2.4) need to be performed in a FT way. Required FT operations include initialization of logical qubits to a computational basis state, measurement of logical qubits in the computational basis, and a universal set of quantum gates.

2.6.1 Surface code 17

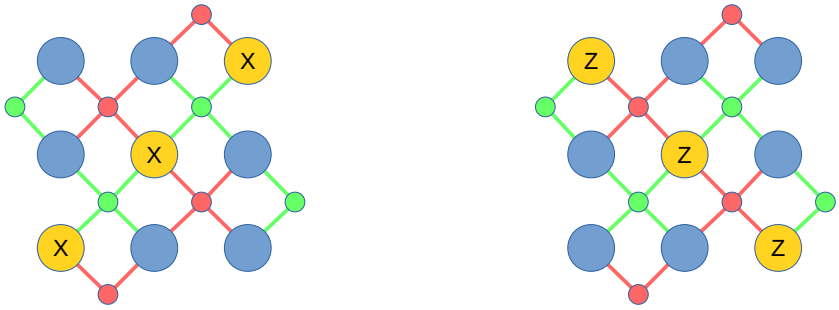
To operate the SC17 fully FT, both QEC and logical operations have to be implemented in a FT way. To make the ESM and error detection FT, we use no FT ESM and rely on complex decoders that can filter inconsistencies in the error syndromes. Such decoding algorithms use d error syndromes to be able to detect errors. An example algorithm used to decode error syndromes is the Blossom algorithm [24, 25]. SC17 supports a limited set of FT logical operations. In the next paragraphs, we will discuss a set of FT logical operations and the timing challenges for the FT operation of a ninja star.

Initialization of a ninja star to the logical $|0\rangle$ state is done by executing multiple rounds of ESM and rely on the decoder to fix initialization errors. The number of error syndromes required for the decoder to work correctly is equal to the distance of the logical qubit where for a SC17 logical qubit $d = 3$. The $|0\rangle_L$ state was described in [19] by the stabilizers shown in Table 2.1 and the stabilizer $Z_0Z_4Z_8$. The following procedure can be used to initialize a ninja star to the $|0\rangle$ state:

1. Reset all data qubits to the $|0\rangle$ state.
2. Execute d rounds of ESM.
3. Run regular decoding algorithms to fix initialization errors.

Logical X and Z operations are implemented by executing a chain of X or Z operations on the data qubits that reach from one boundary to the other boundary. For a ninja star (as shown in Figure 2.1), the logical X gate is performed by executing X gates on data qubit $D2$, $D4$, and $D6$. The logical Z gate is performed by executing Z gates on data qubit $D0$, $D4$, and $D8$. Figure 2.4 shows graphically how the X_L and Z_L operations are performed.

The logical Hadamard gate for SC17 is implemented transversely, which means that a Hadamard gate is executed on all data qubits. After a logical Hadamard operation, the green ancilla qubits become red ancilla qubits and vice versa. The switched ancilla functions can be interpreted as a 90 degrees rotation of the lattice. Therefore, the X_L and Z_L operations also rotate with 90 degrees. Despite from the lattice rotation, the



(a) X_L operation for a ninja star. (b) Z_L operation for a ninja star.

Figure 2.4: Logical Pauli operations for a ninja star.

addressing of the qubits does not change. Figure (2.5) shows a graphical representation of the rotation and the modified logical Pauli gates.

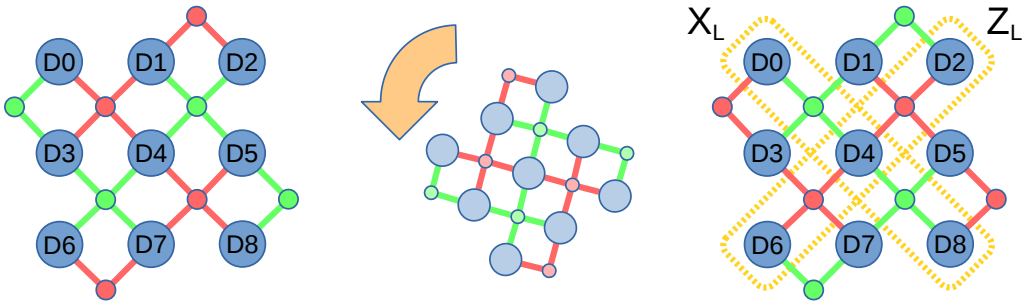


Figure 2.5: Rotation of a ninja star.

For SC17, the logical $CNOT$ gate can be implemented by transversely applying $CNOT$ gates between the data qubits of two logical qubits. The transversal $CNOT_L$ operation between two logical qubits is dependent on the rotation of the lattices. A ninja star can be in a normal orientation or a rotated orientation due to a H_L operation. If two ninja stars A and B share the same orientation, the transversal $CNOT_L$ is executed between data qubits (A_{Dn}, B_{Dn}) where n ranges from 0 to 8. In case the ninja stars are in different orientations, the transversal $CNOT_L$ is executed in a rotated fashion. The $CNOT$ operations will be executed between the following pairs of data qubits of ninja star A and B: $\{(A_{D0}, B_{D6}), (A_{D1}, B_{D3}), (A_{D2}, B_{D0}), (A_{D3}, B_{D7}), (A_{D4}, B_{D4}), (A_{D5}, B_{D1}), (A_{D6}, B_{D8}), (A_{D7}, B_{D5}), (A_{D8}, B_{D2})\}$.

The logical CZ gate for planar surface code is executed transversally, just like the $CNOT_L$. The only difference is the way to deal with different rotational orientations of ninja stars. If the orientation of two ninja stars A and B is different, the transversal CZ_L gate is executed between data qubits (A_{Dn}, B_{Dn}) where n ranges from 0 to 8. In case ninja star A and B share the same orientation, the transversal CZ_L is executed in a rotated fashion as explained for the $CNOT_L$.

Logical measurement of a ninja star in the Z_L basis can be performed by measuring all data qubits (transversal measurement) in the Z basis. The product of the data qubit measurement outcomes (± 1) will yield the logical qubit measurement result. The

procedure for FT logical measurement can be summarized in the following steps:

1. Measure all data qubits in the computational basis. Retrieve their measurement outcomes.
2. Run several ESM rounds, but only for the Z ancillas. Running the partial ESM enables us to detect possible X errors that happened during the measurement procedure.
3. Calculate the product of the measurement outcomes (± 1) of all data qubits. The result, which can be seen as a parity check, represents the logical measurement outcome.

Table 2.3 recaps in one table how FT logical operations are implemented for SC17.

Logical operation	Implementation
X_L	Chain
Z_L	Chain
H_L	Transversal
$CNOT_L$	Transversal
CZ_L	Transversal
Reset to $\|0\rangle_L$	Transversal
M_{Z_L}	Transversal

Table 2.3: List of how logical operations are implemented for Surface Code 17.

ESM rounds can repeatedly be executed to correct and maintain a ninja star state for a longer period. We can execute one or more (1..*) rounds of ESM to collect enough error syndromes for the used decoder. The decoder can generate a set of corrections which can then be applied on the data qubits. After applying the corrections, we can apply a logical operation before we execute the next rounds of ESM. By repeating this execution scheme of 1 or more rounds of ESM, decoding, applying corrections, and logical operations we can perform FT computations with logical qubits while protecting the logical state from errors. A schematic view of the execution scheme is shown in Figure 2.6.

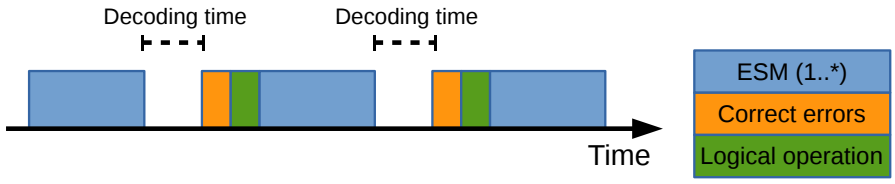


Figure 2.6: Execution scheme of a ninja star logical qubit.

3 Pauli frames

In the previous chapter, we have seen that qubits suffer from decoherence. Execution of fewer gates on qubits can result in reduced operation times, which reduces the probability of errors. A technique that reduces the amount of gates that have to be applied on qubits is called Pauli frames. This chapter explains more about the technique and the benefits of Pauli frames.

3.1 Working principles

The basic idea of Pauli frames is to track gates from the Pauli group in classical electronics instead of applying them on qubits. In that case, every qubit in the system has a Pauli record that tracks the Pauli gates for that specific qubit. The Pauli records of all qubits in a quantum system make a Pauli frame. This idea was first proposed in [8] but has also been discussed in [26, 27, 12, 28, 17]. Three essential elements form the basis of the working principles of Pauli frames:

1. The first element is the effect of qubit initialization on a Pauli record. If a qubit is initialized to $|0\rangle$, its corresponding Pauli record is reset to an empty state (i.e. nothing was tracked yet). All tracked Pauli operators from the past are erased, and the system is in a known state.

2. The second element is the effect of the Pauli records on qubit measurements in the Z basis. Assume m_q is the measurement result of qubit q which was measured in the Z basis and R_q is the Pauli record of qubit q. All tracked Pauli operators can be described by the generators of the Pauli group G_{P_n} as defined in equation (2.15). We can conclude that R_q can be expanded to a series of X, Z, and iI gates. Both Z and iI gates have no effect on the measurement result (see equation (3.1)). Only the X gates in R_q have an effect on the measurement result. An odd amount of X gates in R_q will invert measurement result m_q to $-m_q$ (see equation (3.2)) while an even amount of X gates in R_q will have no effect on m_q.

$$m_q \cdot Z \equiv m_q \cdot iI \equiv m_q \tag{3.1}$$

$$m_q \cdot X \equiv -m_q \tag{3.2}$$

3. The last element is based on the gate groups mentioned in Section 2.3.3. Let us recall the group of Clifford operators on n qubits C_n and the group of Pauli operators on n qubits P_n. For those groups it holds that C_n is the normalizer of P_n, as defined in Section 2.3.1. This means that every Clifford gate $c_i \in C_n$ maps a Pauli gate $p_i \in P_n$ to a Pauli gate $p_j \in P_n$. In short, for $\forall n$ it holds that for

$\forall c_i \in C_n$ and $\forall p_i, p_j \in P_n$, $c_i p_i = p_j c_i$. Since it is know that all elements in C_n have an inverse, it also holds that $c_i p_i c_i^{-1} = p_j$. This means that the application of a Clifford gate c_i on qubit q will map the corresponding Pauli record R_q to a new valid Pauli record R'_q.

Up to now, we assumed that all Pauli gates have to be tracked. Fortunately, this is not the case since Pauli gates have certain beneficial properties and global phase has no effect on the measurement results. All Pauli gates tracked in Pauli record R_q can be decomposed into a series of Pauli generators as described in equation (2.15). Since global phase has no effect on measurement results, only the X and Z generators have to be tracked in the Pauli records. We define R'_q as the Pauli record which only contains the X and Z elements of R_q. The Pauli generators X and Z anti-commute (i.e. $XZ = -ZX$). Reordering the elements of R'_q can only introduce new global phase elements which again can be dropped. In this case, we can order R'_q in a way where all X gates and Z gates are combined. Since X and Z gates are Hermitian, every even sequence of those gates will cancel to I as described in equation (2.9). By canceling out as much as possible X and Z gates we can compress R'_q to R''_q where R''_q contains a maximum of one X gate and one Z gate. We can conclude that any Pauli record R_q can be expressed by one out of four compressed records $R''_q \in \{I, X, Z, XZ\}$.

Up to now, we described how Pauli frames can handle qubit initialization, qubit measurement, Pauli gates and Clifford gates. In Section 2.4, we defined the requirements for universal quantum computation. All requirements mentioned are met, except for requirement 4: The ability to perform a universal set of quantum gates. Pauli frames are not compatible with non-Clifford gates, and therefore, the mechanism can not support a universal set of quantum gates. Non-Clifford gates can still be applied by flushing Pauli record R_q of qubit q before applying the non-Clifford gate on qubit q which means that all pending Pauli gates in record R_q are executed on qubit q to empty the record. Now the non-Clifford gate can be applied on qubit q. After the non-Clifford gate, Pauli gates can be tracked again. By using this flushing technique, the Pauli frame technique can be utilized as an architectural component of a universal quantum computer.

3.2 System specification

Every physical qubit in a quantum system will be assigned a piece of classical memory to store the Pauli record of that qubit. The Pauli records of all qubits in a single quantum system together make a Pauli frame. Since every Pauli record $R_q \in \{I, X, Z, XZ\}$, every Pauli record can be a 2-bit memory. A system with n qubits will need $2n$ bits of memory for the Pauli frame. The Pauli frame together with required Pauli frame mapping logic will be called a Pauli Frame Unit (PFU). A Pauli frame can be seen as an abstract layer on top of the qubits that exists in the classical domain as shown in Figure 3.1. In this figure, *Operations'* does not have to be equal to *Operations*.

Different operations are handled in a variety of ways by the Pauli arbiter and the PFU. The operations can be divided into five categories: Initialization (in the computational basis), Measurement (in the computational basis), Pauli gates, Clifford gates, and non-Clifford gates. Table 3.1 presents how every category is processed.

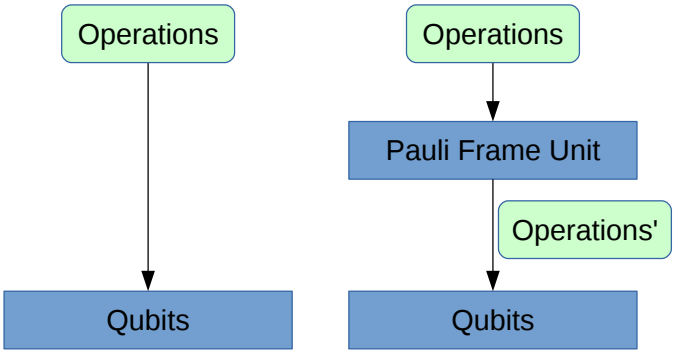


Figure 3.1: Representation of the location of the Pauli frame unit.

Operations	Execution steps
Initialization to $\|0\rangle$	1. Initialize the target qubit to $\|0\rangle$. 2. Set the corresponding Pauli record to I.
Measurement	1. Measure target qubit. 2. Modify measurement result based on Pauli record.
Pauli gates	1. Map Pauli record (no interaction with qubit).
Clifford gates	1. Map Pauli record(s). 2. Apply Clifford gate on target qubit(s).
Non-Clifford gates	1. Flush Pauli record(s). a. Apply gates in Pauli record(s) on target qubit(s). b. Reset Pauli record(s) to I. 2. Apply non-Clifford gate on target qubit(s).

Table 3.1: Pauli frame execution steps for different operations.

From Table 3.1, we can see that measurement results are modified based on the current state of the target qubit Pauli record. If the Pauli record of the measured qubit contains an X operator (i.e. $R_q \in \{X, XZ\}$), the measurement result is inverted as defined in equation (3.2). In any other case, the measurement remains in its original state. These modifications to the measurement results are shown in Table 3.2.

Pauli record R_q	Modified measurement result
I	m_q
X	$-m_q$
Z	m_q
XZ	$-m_q$

Table 3.2: The modifications for measurement results of qubit q with Pauli record R_q.

Pauli gates modify the Pauli record of the target qubit and do not have to be executed

on the actual qubits. The mapping of Pauli records by Pauli gates is shown in Table 3.3 where R_q is the Pauli record of qubit q and R'_q is the new Pauli record for qubit q. This table shows the mappings for the generators of the Pauli group where operators that only influence global phase are excluded. Mappings of other gates in the Pauli group can easily be derived from Table 3.3 by expanding the Pauli gate to its set of generators.

Input record R_q	Executed gate	Output record R'_q
I	X	X
	Z	Z
X	X	I
	Z	XZ
Z	X	XZ
	Z	I
XZ	X	Z
	Z	X

Table 3.3: The mappings for Pauli record R_q for the Pauli generators.

Clifford gates map Pauli records of their target qubits to new Pauli records and also have to be executed on the actual qubits. The mappings of Pauli records by single qubit Clifford gates is shown in Table 3.4 where R_q is the Pauli record of target qubit q and R'_q is the mapped Pauli record. This table only covers the generators for the single qubit Clifford gates, but can easily be extended to all single qubit Clifford gates by combining multiple mapping rules.

Input record R_q	Executed gate	Output record R'_q
I	H	I
	S	I
X	H	Z
	S	XZ
Z	H	X
	S	Z
XZ	H	XZ
	S	X

Table 3.4: The mappings for Pauli record R_q the for single qubit Clifford generators.

Two-qubit Clifford gates map the Pauli records of both the control and the target qubit. The Pauli record of the target qubit has an influence on the Pauli record of the control qubit and vice versa because Pauli gates can propagate to other qubits through multi-qubit gates. Besides from mapping the Pauli record, the Clifford gate also has to be applied on the actual qubits. Table 3.5 shows the mapping tables of the $CNOT$ gate,

the only two-qubit Clifford gate in the Clifford generator set. In this table, R_c and R_t are the Pauli records for the control and target qubit.

Input records		Output record	
Control R_c	Target R_t	Control R'_c	Target R'_t
I	I	I	I
	X	I	X
	Z	Z	Z
	XZ	Z	XZ
X	I	X	X
	X	X	I
	Z	XZ	XZ
	XZ	XZ	Z
Z	I	Z	I
	X	Z	X
	Z	I	Z
	XZ	I	XZ
XZ	I	XZ	X
	X	XZ	I
	Z	X	XZ
	XZ	X	Z

Table 3.5: The mappings for the $CNOT$ gate on Pauli records R_c and R_t for the control and target qubits.

3.3 Applications and benefits

The Pauli frame technique can be applied in several ways and has multiple benefits which make them attractive to use in real world applications. The most interesting application is probably to use a Pauli frame for physical qubits in combination with Quantum Error Correction (QEC) (see Section 2.5). A system structure with a QEC and a Pauli frame is shown in Figure 3.2. In such a structure, correction gates from QEC, and logical Pauli gates can be handled by the Pauli frame resulting in fewer gates being executed on the physical qubits. We compiled a few example quantum programs provided with the ScaffCC compiler [29] and found that the resulting circuits contain up to 7% Pauli gates. Since Pauli gates are tracked in classical electronics only, Pauli gates are processed faster and with a 100% fidelity.

Using a Pauli frame as shown in Figure 3.2 has a second major benefit. Correction gates for detected errors are always Pauli gates, which means that all of them can be stored in the Pauli frame. As a result, the QEC system does not have to wait for the decoder to generate and apply a set of corrections before we can execute a logical gate and execute the next Error Syndrome Measurement (ESM) circuit (as shown in Figure

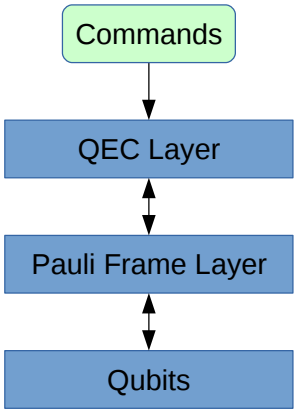
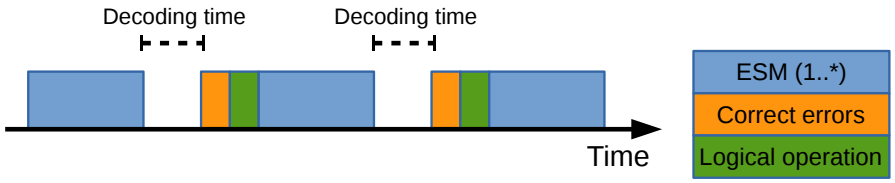
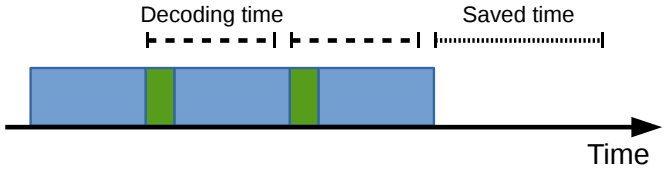


Figure 3.2: A layered system with physical qubits, a Pauli frame unit and a Quantum Error Correction layer.

3.3a). By eliminating this dependency, we can create a new schedule for the ESM rounds and logical gates which is shown in Figure 3.3b. The new schedule effectively removes the time reserved for applying corrections and the waiting time for the decoder, resulting in a more time-efficient schedule. As a consequence, we can perform the same number of logical operations and ESM rounds in less time, effectively reducing the error probability per logical operation. On top of that, the new schedule also loosens the timing constraint on the decoding process as well as the timing constraint on the execution speed of the ESM circuit.



(a) Schedule without Pauli frame.



(b) New schedule with Pauli frame.

Figure 3.3: Schedules of steps in the Quantum Error Correction process with and without Pauli frame.

3.4 Pauli frame example

In this section, we will discuss a few examples to give a more visual representation of the Pauli frame behavior. In this example case, we assume we have 17 qubits in a Surface Code 17 (SC17) setup, also known as the ninja star and shown in Figure 2.1. The ninja star has 9 data qubits and 8 ancilla qubits. The system layout is presented in Figure 3.2 where the Pauli frame unit resides between the qubits and the QEC layer. For this example, we assume that we only have a Pauli frame for the 9 data qubits. We represent the data qubits as blue circles labeled from $D0$ to $D8$. The ancilla qubits will not be shown and are considered out of scope for this illustrative example. The Pauli records are represented as yellow squares labeled from $D0$ to $D8$ where the labels correspond to the labels of their qubit. The schematic system overview is shown in Figure 3.4.

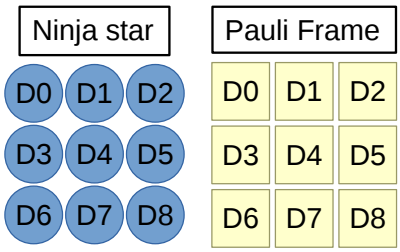


Figure 3.4: The schematic overview of the data qubits of the ninja star and their corresponding Pauli records.

The first step is to initialize the ninja star to a logical $|0\rangle$ state. All data qubits are reset and entangled, and the Pauli records of all data qubits are reset to the I state. For this example, we assume that no errors occurred during initialization. This step is illustrated in Figure 3.5. The yellow boxes of the Pauli records now show the state of the records and the empty blue circles indicate that the qubits are in a correct state. All data qubits are now entangled and represent the logical $|0\rangle$ state.

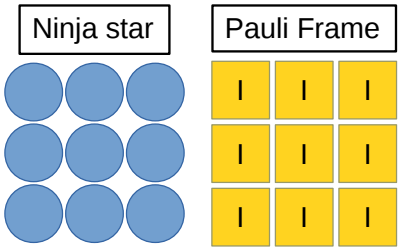


Figure 3.5: All data qubits and their corresponding Pauli records are reset to be able to initialize the ninja star.

We continue to apply quantum error correction on the ninja star. Assume that two errors are detected, an X error on data qubit $D2$ and a Z error on qubit $D4$. These detected errors are corrected, and the Pauli frame processes the corrections. The Pauli records of data qubits $D2$ and $D4$ are mapped to their corresponding values. The data qubits stay in an erroneous state while the Pauli frame tracks the errors. Figure 3.6

shows the new state of the Pauli frame while the current detected errors on the ninja star are indicated with red circles.

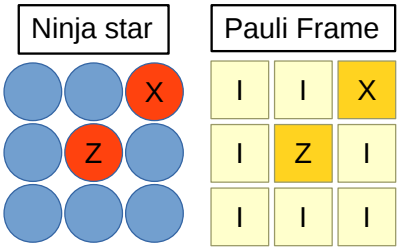


Figure 3.6: Two errors are detected and processed by the Pauli frame.

Now assume that both an X and a Z error is detected on data qubit $D4$, which will be processed by the Pauli frame. The Pauli record of data qubit $D4$ already contained a tracked X error. The combined XZ error maps the Pauli record to Z as shown in Table 3.3. The two X errors have canceled each other out (up to a global phase which can be ignored), and therefore, only the Z error has to be tracked. Figure 3.7 shows the detection event and the mapped Pauli record of $D4$.

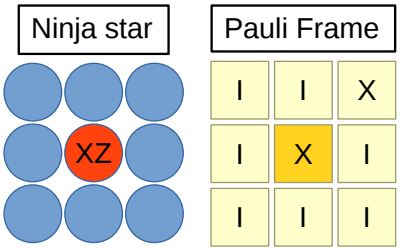


Figure 3.7: A double error on $D4$ is detected and processed by the Pauli frame.

Let us assume a logical Hadamard gate is applied to the ninja star. Recall that the logical Hadamard gate is implemented by applying a Hadamard gate on all data qubits as noted in Table 2.3. The Hadamard gate is a Clifford gate and therefore maps the Pauli records, but is still applied to the data qubits. Using the mappings in Table 3.4 we can see that the two X entries in the Pauli frame will be mapped to Z entries. Figure 3.8 shows the new Pauli frame state after applying the logical Hadamard gate.

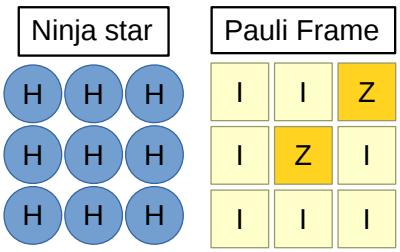


Figure 3.8: A logical Hadamard gate is applied an the Pauli records are mapped.

Finally, the logical qubit is measured, which means that all data qubit are measured,

and the results are combined to retrieve the logical measurement result. The Pauli frame first processes the measurement results of the data qubits. In this example, we only have Pauli records which are in the state I or Z. Both of them have no influence on the measurement results as we can see in Table 3.2. Therefore, all measurement results can be further processed to obtain the logical measurement result. If any Pauli record were in the state X or XZ the corresponding measurement results would have to be inverted before further processing.

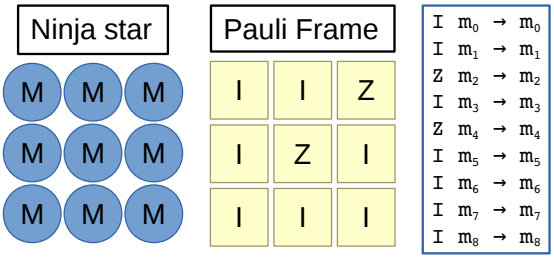


Figure 3.9: All data qubits are measured and measurement results are mapped by their corresponding Pauli record.

3.5 Implementation

Up to now, multiple papers [8, 26, 27, 12, 28, 17] have covered the topic of Pauli frames, and other papers [30, 31, 32, 33] have discussed various architectures for quantum software and hardware, but no practical implementations of Pauli frames for future quantum computers have been proposed so far. In this section, we would like to cover the implementation of a Pauli frame in a quantum computer architecture.

In [34] we propose a heterogeneous quantum computer architecture which supports all operations of a ninja star implemented with transmon qubits. Figure 3.10 shows a simplified version of the proposed architecture which mainly focuses on the Quantum Control Unit (QCU). The QCU decodes the instructions belonging to the Quantum Instruction Set Architecture (QISA) and performs the required quantum operations, feedback control, and QEC. The QCU can also communicate with the host CPU where classical computations are performed, and quantum instructions are fetched. The QCU outputs a sequence of physical operations to the Physical Execution Layer (PEL) to control the physical qubits. The PEL takes charge of all technology-dependent control making the QCU technology-independent. The QCU defined in this paper can support not only transmon qubit but can also support other technologies.

The PEL converts the physical operations to a set of waveforms that represent elementary quantum gates supported by the underlying technology. After conversion, the PEL manages the timing of the waveform outputs. These waveforms are fed to the Quantum-Classical Interface (QCI) that routes the waveforms to the correct qubits on a quantum chip. Regarding physical measurements, the PEL provides a readout pulse to the QCI and the QCI returns the measurement signal. The PEL processes the measurement signal and discriminates the measurement results. These physical measurement

results are then returned to the QCU for further processing. In the next sections, we will focus on the QCU and briefly discuss the main parts of this module. For more details about the proposed architecture, we refer to the original paper [34]. After this brief overview, we will focus on the PFU.

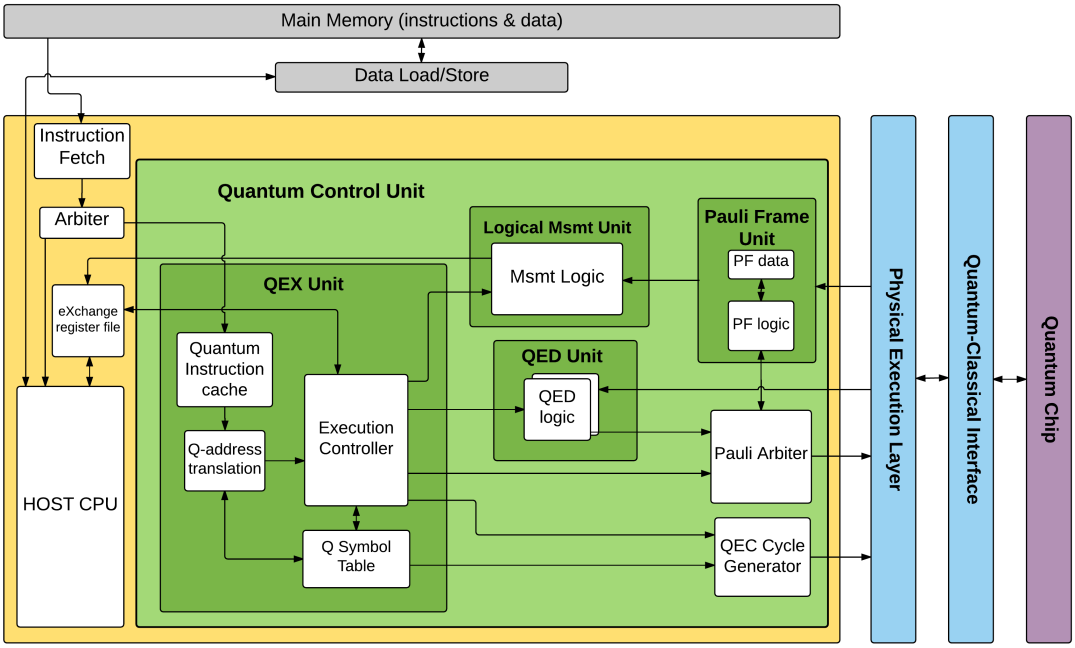


Figure 3.10: Simplified quantum computer architecture for a ninja star.

3.5.1 Quantum control unit

We assume that a binary is loaded into memory, and the instruction fetch unit fetches the instructions. Based on the opcode of the instruction, the arbiter sends the instruction either to the host CPU or the QCU. In the remainder of the text, we focus on the architectural support for the execution of quantum instructions and not on the execution of instructions on the classical CPU. In an earlier stage, a compiler maps a quantum circuit using logical and virtual qubit addresses for the logical and physical qubits respectively. Instructions from the Quantum Instruction Cache are first address-translated by the Q-Address Translation module which means that compiler-generated, virtual qubit addresses are translated into physical ones. The translation procedure is based on the information contained in the Q Symbol Table which provides the overview of the exact physical location of the logical qubits and contains information on what logical qubits are still alive.

The Execution Controller can be seen as the brain of the QCU. The Execution Controller decodes the various instructions that are fetched:

- **Physical gate/measurement/reset**: The Execution Controller sends these instructions to the Pauli Arbiter for further processing.

- **Update Q Symbol Table**: Based on a series of instructions such as the ones performing a logical Hadamard gate or a logical measurement, the Q Symbol Table needs to be updated. Also for deallocating qubits such an update is required.

- **QEC slot**. The Execution Controller sends this instruction to the QEC Cycle Generator, which is triggered to generate an ESM circuit.

As far as error correction is concerned, the necessary ESM instructions for the entire qubit plane are added at run-time by the QEC Cycle Generator, based on the information stored in the Q Symbol Table. The responsibility of the Quantum Error Detection (QED) Unit is to detect errors based on ESM results. The decoder will use decoding algorithms such as the Blossom algorithm [24, 25]. QED only starts to work when d rounds of error syndromes are collected, where d equals the distance of the surface code.

The function of the Logic Measurement Unit is to combine the data qubit measurement results into a logical measurement result for a logical qubit. Once the Execution Controller receives a logical measurement instruction on a specified logical qubit, it notifies the Logic Measurement Unit to wait for measurement results to arrive from the PEL.

3.5.2 Pauli frame unit

The Pauli Frame Unit (PFU) consists of a Pauli frame (PF data) and Pauli frame mapping logic (PF logic), and works closely together with the Pauli arbiter. Figure 3.11 shows a detailed schematic of the components in the PFU. The Pauli frame contains a two-bit Pauli record for every physical qubit in the system as mentioned in Section 3.2. For a single SC17 logical qubit this would be $2 \cdot 17 = 34$ bit of memory. The mapping of the Pauli records based on the applied physical operations are managed by the PF logic module which holds all the required mapping tables. Finally, the Pauli arbiter decides which operations in the stream are forwarded to the Physical Execution Layer (PEL) and which ones not. The proposed PFU design complies with the system described in Section 3.2 and supports all operation types mentioned in Table 3.1. In the next paragraphs, we will explain how the different type of operations from Table 3.1 are handled by the PFU and the Pauli arbiter.

If the Pauli arbiter receives a reset operation (step 1), the operation will be forwarded to both the PFU and the PEL (step 2). While the PEL further processes the reset operation, the PFU also processes the reset operation using the PF logic module. The Pauli record of the target qubit will be set to I regardless of its current state (step 3). This sequence of steps is schematically shown in Figure 3.12a.

In case the Pauli arbiter receives a measurement operation (step 1), the Pauli arbiter will forward the operation to the PEL without taking any further action (step 2). The PEL and other parts of the system perform the measurement operation, and the measurement results are returned to the QCU (step 3). This measurement result is picked up by the PFU which maps the measurement result based on the Pauli record of the target qubit (step 4) using the PF logic module. Finally, the mapped measurement result is forwarded to other parts of the QCU (step 5). The measurement procedure is schematically shown in Figure 3.12b.

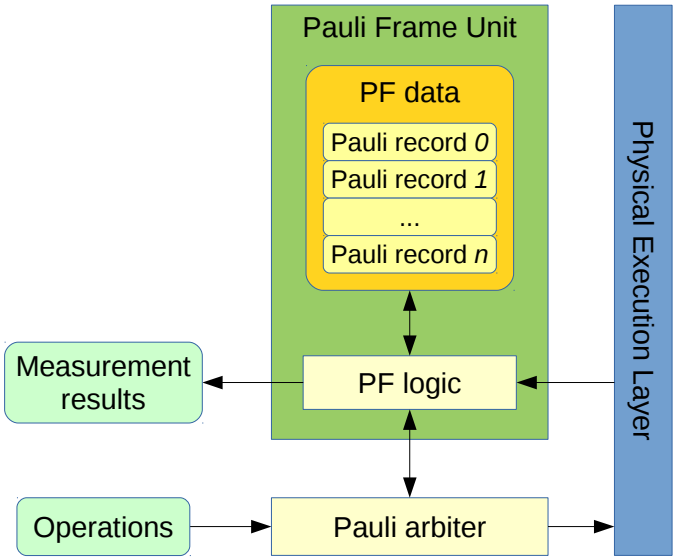
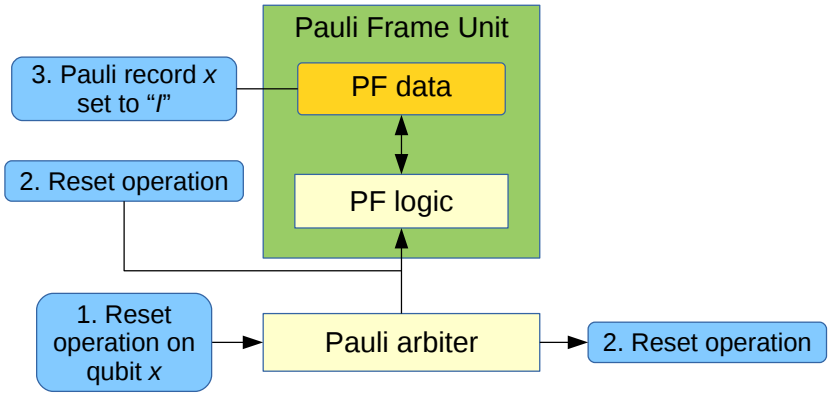


Figure 3.11: Detailed schematic of the Pauli Frame Unit.

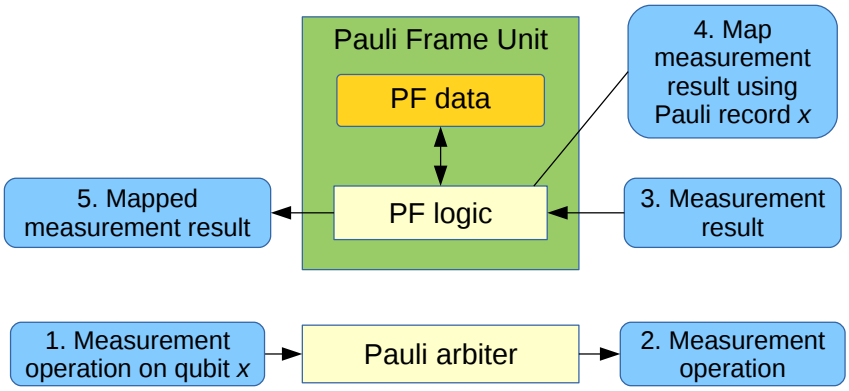
Pauli operations are handled in the most efficient way. When the Pauli arbiter receives a Pauli gate (step 1), the Pauli arbiter will only forward the gate to the PFU (step 2). The PEL is not required when executing Pauli operations. The PF logic module of the PFU will map the Pauli record of the target qubit (step 3), and the execution of the Pauli gate is finished. This procedure is also shown in Figure 3.12c.

If the Pauli arbiter receives a Clifford gate (step 1), the Pauli arbiter will forward the gate to the PEL and the PFU (step 2). While the PEL handles further execution of the Clifford gate, the PF logic module of the PFU updates the Pauli record of the target qubit(s) (step 3). Figure 3.12d shows a graphical representation of this procedure.

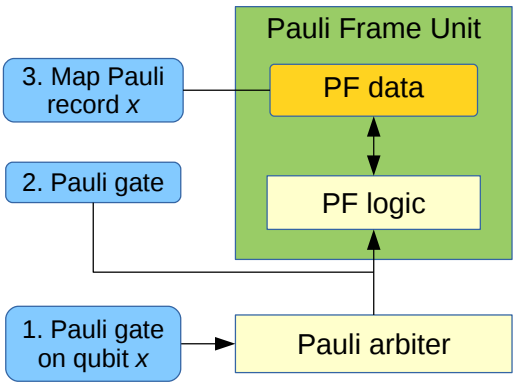
In case the Pauli arbiter receives a non-Clifford gate (step 1), the Pauli arbiter stalls the stream of operations and requests the PFU to flush the Pauli record of the target qubit (step 2). The PF logic module of the PFU returns the Pauli gate(s) currently present in the Pauli record of the target qubit and resets the Pauli record to I (step 3 and 4). Finally the Pauli arbiter forwards the flushed Pauli gates to the PEL and appends the initial non-Clifford gate (step 5 and 6). The Pauli arbiter will continue processing the stream of operations. Figure 3.12e shows the steps of handling a non-Clifford operation in a graphical way.



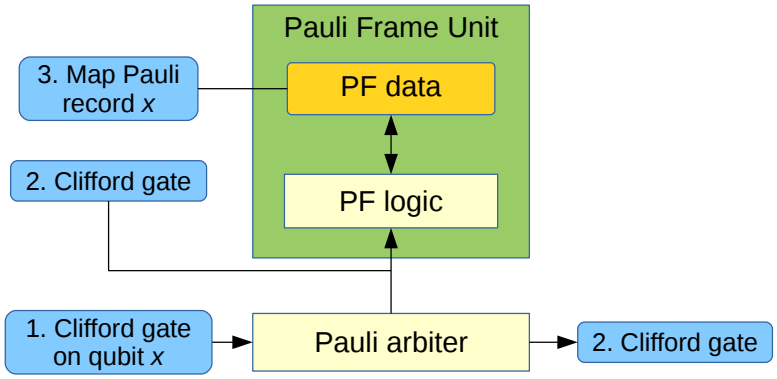
(a) Reset operation.



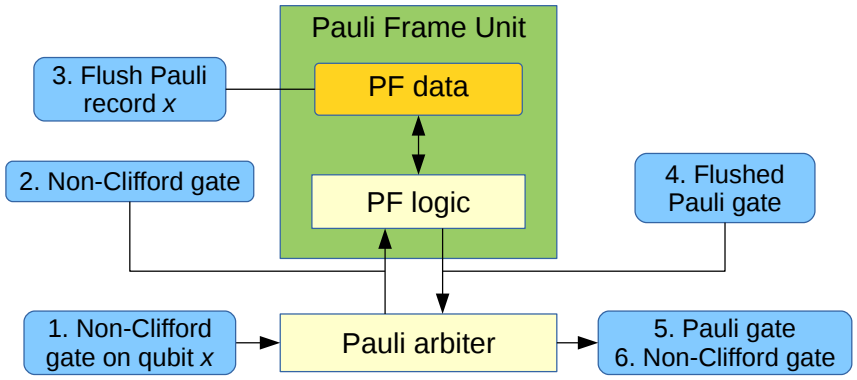
(b) Measurement operation.



(c) Pauli gate.



(d) Clifford gate.



(e) Non-Clifford gate.

Figure 3.12: Schematics how the Pauli arbiter and Pauli Frame Unit process different operations.

4 Simulation platform

To gain more knowledge about quantum computer architectures and Pauli frames we require a simulation platform. This simulation platform should fit into the big picture of a full quantum computer development environment that connects from quantum algorithms down to the physical operations on qubits of a quantum chip.

A quantum computer will always consist of both quantum and conventional computing components because of the following two reasons: the quantum algorithms and consequently the quantum applications that will be executed, include both classical as well as quantum parts and will thus be executed by their respective computing blocks [11]. The second reason is that, as is explained in Section 2.5, a quantum computer requires very close monitoring and, if necessary, correction by classical logic. In [34] we present a high-level view of the quantum system stack consisting of multiple layers which are shown in Figure 4.1. The top layers represent the algorithms for which specific language constructs and compilers need to be developed such that the algorithms can exploit the underlying quantum hardware. Here the qubits are defined as logical qubits. Figure 4.2 depicts the compiler infrastructure consisting of a conventional host compiler and a quantum accelerator compiler. The former compiles the classical logic and the latter will produce the quantum circuits. The quantum compiler will perform quantum gate decomposition, reversible circuit design, and circuit mapping but also translates the logical quantum operations to a series of physical operations. As represented by the third dimension of Figure 4.1, the logical-to-physical quantum instruction translation is driven by choices regarding the Quantum Error Correction (QEC) code for the logical qubits.

The next layer is the Quantum Instruction Set Architecture (QISA) which is the dividing line between hardware and software. The algorithm designer and programmer are offered a logical instruction set with the possibility to opt for certain encoding schemes, thus exposing the relevant error correction functionality. As stated above, the compiler will translate logical instructions into the physical instructions that belong to the QISA and for which architectural support is provided. Examples of physical instructions in the QISA are initialization, measurements, and quantum gates such as Hadamard and $CNOT$.

The Quantum Execution (QEX) block will execute the quantum instructions that are generated by the compiler infrastructure. It will also provide the necessary hardware support such as the insertion of quantum error correction circuits or the use of Pauli Frames for error tracking. These operations are finally sent to the Quantum-Classical Interface (QCI), which will apply the proper electrical signals to the quantum chip. Note that the QCI is responsible for all the conversions between the analog qubit plane and the digital layers in the system stack. The QEC layer, is in charge of the error detection and correction. It will receive the error syndrome data from the QCI which it will process

to identify possible errors. Then, it will make the required corrections by updating the Pauli frame or by sending the appropriate corrective operations when required. The Quantum Control Unit (QCU) proposed in Section 3.5 matches the QEX/QEC layer.

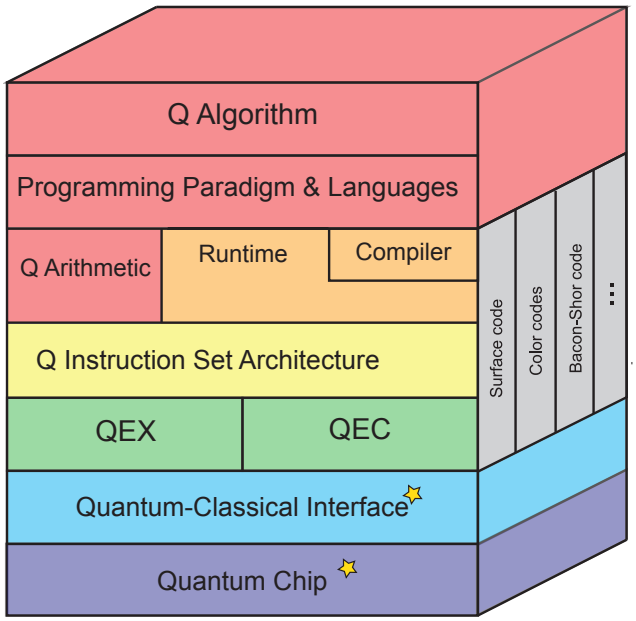


Figure 4.1: The global picture of a quantum computing environment.

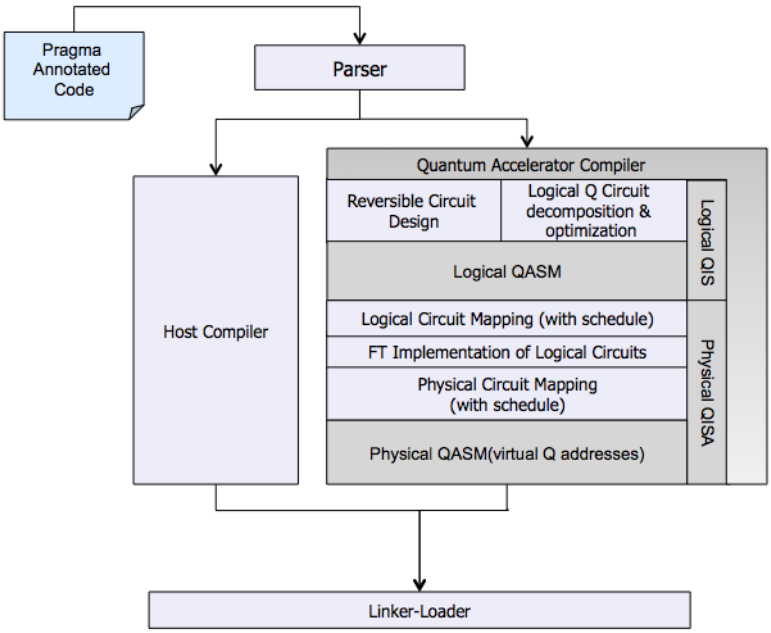


Figure 4.2: Compiler infrastructure

The main interest of this thesis lies in the QEX/QEC layer shown in Figure 4.1. The QEX/QEC layer is where the quantum computer architecture resides and where all quantum instructions are processed. The quantum algorithm and compiler levels are out of the scope of this thesis and are therefore not further discussed. This chapter presents the simulation software platform for the quantum layer and the QEX/QEC layer. The simulation platform consists of three software tools of which two are quantum simulators, and one is a self-developed software package for functional testing of quantum computer architectures (QPDO). These three software tools will be discussed in the remainder of this chapter.

4.1 Quantum simulators

Quantum behavior can be simulated on classical computers using software. There are different ways to simulate qubits, and every type of simulation has its advantages and disadvantages. In this section, we will discuss two quantum simulators which are used for this thesis: QX Simulator and CHP.

4.1.1 QX Simulator

The QX Simulator [9] is a universal quantum simulator developed by the Computer Engineering lab of the Delft University of Technology. Universal quantum simulators store the complex amplitudes of the quantum state vector. Quantum gates are represented as matrices, and matrix-vector multiplications are used to apply gates. A universal quantum simulator can simulate any quantum gate by using its matrix representation, but there are also limitations. The memory usage of universal quantum simulators grows exponentially with the number of qubits simulated. Due to high memory usage, universal quantum simulators are only able to simulate up to 30 to 45 qubits depending on the available hardware resources.

There are two options to interface with the QX Simulator. One of them is by using Quantum Assembly (QASM) files. These files describe quantum circuits as a list of instructions and are limited to quantum instructions only. The QX Simulator can read the QASM file and execute the described circuit. The second way of interfacing with the QX Simulator is by using sockets. The QX Simulator can start as a server and establish TCP connections. Any program can connect to the QX server and send QASM instructions to the QX server. The TCP connection provides a flexible and interactive interface to the capabilities of the QX Simulator that can be accessed from almost any development environment. The QASM syntax for both interfaces is identical.

4.1.2 CHP

The CHP quantum simulator [10] is a quantum simulator developed by Scott Aaronson and Daniel Gottesman. CHP is a stabilizer simulator and is therefore only able to process stabilizer circuits using $CNOT$, H, and S gates beside from qubit initialization and measurement. Stabilizer simulators store a quantum state by storing the stabilizers of that specific state, which is very memory efficient. The limitation of stabilizer simulators

is that stabilizer circuits do not possess quantum speed-up regarding computation power. Stabilizer simulators are still interesting to simulate large numbers of qubits and quantum error correction codes. CHP can read QASM like files and execute the circuit described in the file.

For this thesis, the quantum simulation capabilities of CHP have been exposed to Python by creating a Python wrapper [35]. The Python wrapper was created using the wrapper generation software SWIG which stands for 'Simplified Wrapper and Interface Generator'. CHP functionality that was exposed to Python includes the creation of a qubit register, qubit initialization, execution of quantum gates, and qubit measurements.

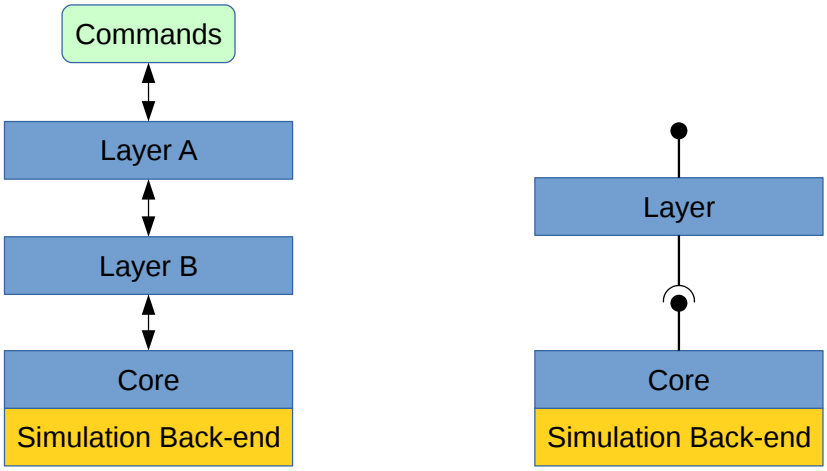
4.2 QPDO

For this thesis, we require a software tool that can simulate different modules of a quantum computer architecture in a flexible way. To enable fast development and testing, we allow functional simulation of the modules. The chosen programming language should be accessible to a large crowd, cross-platform and high-level to decrease development time. It is also preferable to be able to link the selected programming language easily against code written in other languages. Execution speed of the quantum computer architecture simulations is not a constraint since we allow functional simulation. Therefore, we do not require high-speed, low-level programming languages. The tool should be able to connect to different quantum simulators in a transparent way to allow simulations against different simulator back-ends. The first prototypes of such a tool were written in Java. We experimented with different structures and design methodologies using the Java software. Finally, we chose to develop a tool based on a layered structure programmed in Python 3. The name of the software package is Quantum Platform Development framewOrk, or in short QPDO.

4.2.1 Layered structure

QPDO provides shared interfaces and data structures to support layered systems. Interfaces are implemented using abstract base classes and shared data structures are expressed using classes. The layers are inspired by the OSI-model [36] and can be seen as a black box that processes a stream of commands. Every layer has its responsibilities and should be able to work without knowledge of lower or top layers. Commands are implemented using function calls, and data is transferred using shared memory. Layers can be stacked on each other in a flexible way to create a control stack. Such a control stack contains the modules of a quantum computer architecture and enables simulation of this architecture. The bottom layer of a control stack should always be a core. A core connects to a quantum simulator to allow simulation of the invoked commands. A control stack has one core and zero or more layers on top of it. The schematic view of a layered control stack is shown in Figure 4.3a.

The key feature of the layers used in QPDO is that they all support the same interface and that they share the same data structures. The shared *Core* interface consists of a set of functions that can control quantum simulators or devices on an abstract level. The *Core* interface has functions for creating and removing qubits, queuing quantum



(a) A schematic view of a layered control stack in QPDO.

(b) Interfaces for the core and the other layers.

Figure 4.3: Schematics for QPDO control stacks with layers.

circuits and executing queued quantum circuits. It is also possible to request the binary state or quantum state of the system. The functions of the *Core* interface are listed in Table 4.1. The lowest layer in a control stack, the core, supports the functions of the shared *Core* interface and simulates the quantum behavior using a simulation back-end. Layers on top of the core support the shared interface too and also require a lower layer that supports the same interface. A schematic view of a layer and core are shown in Figure 4.3b. By using this layered structure, it is possible to interchange layers easily which make it possible to assemble different control stacks and use different simulation back-ends.

Function	Short description
createqubit(size)	Allocate new qubits.
removequbit()	Remove existing qubits.
add(circuit)	Queue a quantum circuit.
execute()	Execute the queued quantum circuits.
getstate()	Retrieve the (binary) state of the qubits.
getquantumstate()	Retrieve the quantum state (if supported).

Table 4.1: The functions of the shared *Core* interface between layers in QPDO.

4.2.2 Shared data structures

The shared data structures provided by QPDO are used to transfer information between layers and to process data generated by the control stack. All shared data structures by default provide rich functionality to analyze and manipulate its contents. The most important shared data structures are the *Circuit*, the *State*, and the *QuantumState*. The

Circuit class represents a quantum circuit and contains a set of operations. Operations can be single- or multi-qubit gates, initialization, and measurement. Operations in circuits are grouped into time slots. In one time slot, every qubit can only be involved in one operation. Thereby we can see a time slot as a part of a circuit that can be executed in parallel whereby we assume that every operation takes the same amount of time to execute. A schematic view of a circuit with time slots and operations is shown in Figure 4.4.

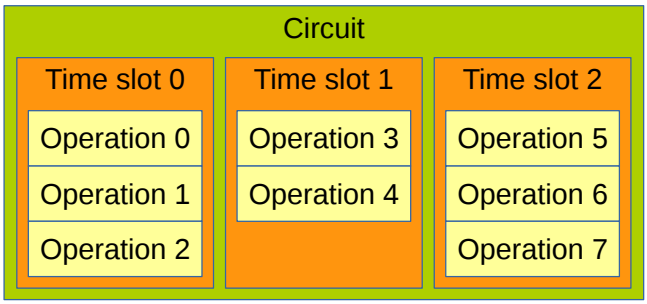


Figure 4.4: A schematic view of data structure for a circuit consisting of time slots with operations.

The *State* and *QuantumState* classes are shared data structures that describe the current state of the quantum system. The *State* class represents a set of values that correspond to the binary values of qubits. A measured qubit can either be in the *0* or *1* state. A qubit that is reset to $|0\rangle$ will also be in the *0* state. After applying a quantum gate on a qubit, the state will become x (i.e. unknown) until it is measured or reset. A *State* object holds a set of these binary values. A *QuantumState* represents the full state vector with complex amplitudes and corresponding states. The *QuantumState* can only be retrieved if a simulation back-end is used that supports outputting a quantum state.

4.2.3 Implemented layers

To use QPDO for our simulations, we implemented different layers. The first implemented layers are two core layers that implement the *Core* interface: *ChpCore* and *QxCore*. The *ChpCore* uses the Python wrapper for CHP to use it as a simulation back-end. The CHP simulator is not universal, but can simulate Clifford gates efficiently. Since QEC codes use Clifford gates only, those can be simulated using the *ChpCore*. The *QxCore* uses the universal QX Simulator as a back-end and can simulate both Clifford and non-Clifford gates. The *QxCore* maintains a TCP connection to the QX server. Commands are sent over TCP and results are retrieved over the same connection.

Two QEC layers have been implemented: The *SteaneLayer* and the *NinjastarLayer* for respectively the Steane code and planar surface code using 17 physical qubits per logical qubit. The QEC layers supports generic control of independent logical qubits including insertion of QEC routines, logical operation translation to a set of physical operations, and logical operation post-processing. As specified by QPDO, the QEC layers work in a transparent way and support the *Core* interface as shown in Table 4.1.

It is for example possible to concatenate QEC layers by adding multiple QEC layers to a control stack.

To be able to test QEC layers we need to be able to insert errors. Up to now, only one error layer is developed, which is based on the symmetric depolarizing error model [11, 19].

In the interest of this thesis, we developed a Pauli frame layer. This layer implements the functionality of a Pauli frame unit as presented in Section 3.5.2. The layer manages memory for a Pauli frame, maps the Pauli records according to mapping tables, maps measurement results based on stored Pauli records, and filters Pauli gates. Since the Pauli frame unit is implemented as a layer, it is possible to add multiple Pauli frame layers on different levels in a control stack.

Finally, we implemented some diagnostic layers that do not modify the command streams in a control stack but retrieves statistics about the command stream instead. An example of such a layer is the counter layer that counts the number of operations and commands that pass between two other layers. The information gained by diagnostic layers can give us useful insight into the execution flow in a control stack.

4.2.4 Test benches

The previous sections presented how control stacks can be created and configured using QPDO. Such a control stack can potentially perform functional simulations of a quantum computer architecture, but requires useful input for simulation. It is possible to write custom test benches in Python that provides input commands to a control stack and diagnoses the output. To speed up development of test benches, QPDO provides a test bench environment and simple ready-to-use test benches.

The test bench environment provides a set of base classes that implement generic control of a certain class of test benches. Generic control can contain for example looping of a test for a certain amount of times, handling test outcomes, and acting on test results. Programmers only have to write an initialization procedure for the test bench, a single test procedure, and a shutdown procedure for the test bench. The test bench environment makes it possible to develop quickly test benches that can test against various control stacks using the generic *Core* interface. An example of a setup with a control stack and a test bench is shown in Figure 4.5.

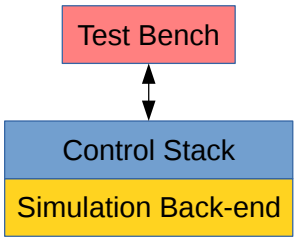


Figure 4.5: A schematic view of a control stack and a test bench.

QPDO provides a set of ready-to-use test benches that are derived from the test bench base classes. These simple test benches only need a control stack and maybe a couple of settings to run and can be used to verify the basic behavior of a control stack.

Two examples of such test benches are the *BellStateHistoTb* and the *GateSupportTb*. The *BellStateHistoTb* resets two qubits and creates a Bell state using a Hadamard and a $CNOT$ gate. The two qubits are measured, and the results are stored. At the end of the test a histogram of the measurement results are printed. The *GateSupportTb* runs a predetermined script that tests various gates that are available in QPDO on a control stack. It verifies if the control stack supports various gates and if the measurement result after the execution of the gates matches the expected outcome. Finally, the test bench prints a report that shows which gates are supported and executed correctly.

5 Experiments

In this chapter, we present the results of the experiments performed using the simulation tools presented in Chapter 4. Three different experiments were performed considering Surface Code 17 (SC17) and Pauli frames. The first experiment is a verification of the known logical operations of the SC17. The second experiment is a verification of the Pauli frame mechanism. With the last experiment, we investigate the impact of a Pauli frame on the error rates of a ninja star.

5.1 Ninja star logical operations verification

The first experiment is to verify the logical operations for a ninja star using QPDO. The logical operations we would like to verify are the initialization to $|0\rangle$, X gate, Z gate, H gate, $CNOT$ gate, CZ gate, and measurement in the computational basis as shown in Table 5.1.

Logical operation	Implementation
X_L	Chain
Z_L	Chain
H_L	Transversal
$CNOT_L$	Transversal
CZ_L	Transversal
Reset to $\|0\rangle_L$	Transversal
M_{Z_L}	Transversal

Table 5.1: List of how logical operations are performed for a ninja star.

To perform the experiments to verify the ninja star logical operations, we require specifying which run-time properties have to be tracked for every ninja star and how we can perform the logical operations in a correct way based on these properties. The next two sections will discuss what run-time properties need to be tracked for a ninja star and how these properties cooperate with the logical operations.

5.1.1 Run-time properties of a Ninja star

To operate a Ninja star we need to track different properties which can change after executing various logical operations. The first property to track is the rotation of the lattice. The *rotation* property of a ninja star can have two values, *normal* or *rotated*, where the *rotated* value indicates that the lattice is in a 90 degree rotated orientation. Tracking of the current rotation is essential for executing the correct Error Syndrome

Measurement (ESM) circuit and is also required for applying logical gates in the correct way.

The second property that is tracked is the dance mode. The dance mode indicates whenever a full ESM circuit has to be executed or if only Z ancilla qubits are active in the ESM circuit. The *dancemode* property can have two values: *all* or *z_only*. The value *all* indicates that the full ESM circuit will be executed, while the *z_only* value indicates that only Z ancilla qubits are active in the ESM circuit.

Finally, there is a *state* value that indicates what the current binary state is of the logical qubit. This value can be *0*, *1*, or x (i.e. unknown). Table 5.2 summarizes the list of properties for a ninja star, their possible values and the initial values at system start-up.

Ninja star property	Possible values	Initial value
rotation	*normal, rotated*	*normal*
dancemode	*all, z_only*	*z_only*
state	*0, 1, x*	x

Table 5.2: List of properties of a ninja star.

5.1.2 Logical operation conversion and property updating

Logical operations for ninja star logical qubits have to be converted to a set of physical operations and some classical post-processing. Conversion of the logical operations to a set of physical operations can be done at run-time based on the current properties of the involved logical qubits. Both the conversion of X_L and Z_L gates depend on the lattice rotation, while the conversion of transversal $CNOT_L$ and CZ_L gates rely on the rotations of both the target and the control ninja star. The details of logical operation conversions for a ninja star are described in section 2.5.1.

Some logical operations update the run-time properties of a ninja star and therefore their execution requires post-processing. The H_L gate inverts the current rotation of the lattice. The reset operation sets the current rotation of the ninja star to *normal* and the dance mode to *all*. Measurement requires the most post-processing. After measuring all the data qubits, the dance mode will be set to *z_only*. Only measuring Z ancilla qubits enables us to perform partial ESM rounds that can be used to detect X errors that happened during the measurement of the physical qubits. If any X errors are detected, the corresponding measurement results are corrected. Finally, the parity of the data qubit measurement results is calculated yielding the logical measurement result which is stored in the *state* property of the ninja star. The relations between the logical operations and the properties of the logical qubit(s) they apply on are shown in Table 5.3.

5.1.3 QPDO implementation

We used QPDO to verify the logical operations of the ninja star. To do so, we implemented a ninja star layer that accepts logical circuits as input and outputs all the

Logical operation	Property dependencies	Property influences
X_L	*rotation*	*state*
Z_L	*rotation*	*state*
H_L	-	*rotation, state*
$CNOT_L$	*rotations*	*state*
CZ_L	*rotations*	*state*
Reset to $\|0\rangle_L$	-	*rotation, dancemode, state*
M_{Z_L}	-	*dancemode, state*

Table 5.3: List of logical operations and their relation to properties of a ninja star.

circuits and commands required to perform both Quantum Error Correction (QEC) and converted logical operations. The output operations of the ninja star layer can then be executed on a simulation core or can be forwarded to another layer within the QPDO environment (see section 4.2.1). The ninja star layer can control multiple independent ninja stars and also supports two-qubit logical gates. Every ninja star can have a unique set of ancilla qubits, or one set of ancilla qubits can be shared over all ninja stars in the system to lower the amount of qubits that is required to be simulated.

The ninja star layer uses multiple classes to manage all the run-time properties, execution of QEC, and execution of logical operations as shown in Table 5.4. Run-time properties of every individual ninja star (see Table 5.2) are stored in a *NinjaStarQubit* object. Based on the current properties, the *NinjaStarQubit* object can generate the correct ESM circuit. Every ninja star also requires a decoder for error detection. For the verification of logical operations, we used a decoder based on two look-up tables. The two look-up table decoder independently decodes error syndromes of X and Z ancilla qubits using separate look-up tables and returns the union of all resulting corrections. Logical operations are converted to a set of physical operations using the *NinjaStarGate* class, which is an extension to the QPDO *Circuit* class. The *NinjaStarGate* object converts the logical operation to a circuit containing the corresponding physical operations based on the current properties of the targeting *NinjaStarQubit* objects. The *NinjaStarGate* object also contains post-processing procedures for the logical operations.

We used the test bench facilities of QPDO to verify the logical operations for a ninja star. For our test setup, we assembled a control stack with a *NinjaStarLayer* and a *QxCore*. A schematic view of the test setup is shown in Figure 5.1.

5.1.4 Simulation results

We first verified if ninja star initialization to logical state $|0\rangle$ is performed correctly. We apply a circuit on the test setup that only contains a *Reset* operation. The ninja star layer resets all the data qubits to $|0\rangle$ and performs a single round of ESM to initialize the ninja star. A ninja star does not always initialize correctly, and errors during initialization are corrected using the look-up table decoder. The initialization procedure is repeated for 100 iterations and the resulting quantum state always equals the state shown in Listing 5.1 where the rightmost bit represents the value of data qubit 0. The state in Listing 5.1

Class	Responsibilities
NinjaStarQubit	• Store run-time properties • Store physical qubit address table • Generate ESM circuits • Manage decoder
NinjaStarGate	• Convert logical operations to physical operations • Post-processing of logical operations
NinjaStarLayer	• Overall execution control of QEC and logical operations

Table 5.4: List of all classes used for the implementation of a ninja star layer in QPDO with their corresponding responsibilities.

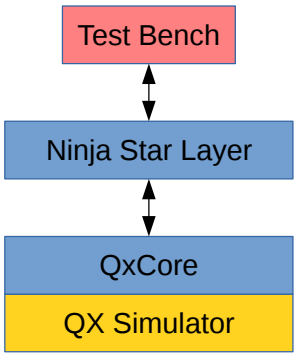


Figure 5.1: The test setup for simulations of the ninja star control software.

matches the logical $|0\rangle$ state as described in section 2.5.1. Hence, we can conclude that the initialization procedure of the ninja star to the logical state $|0\rangle$ is correct.

```
(0.25+0j)    |000000000>
(0.25+0j)    |000000110>
(0.25+0j)    |000011011>
(0.25+0j)    |000011101>
(0.25+0j)    |011000000>
(0.25+0j)    |011000110>
(0.25+0j)    |011011011>
(0.25+0j)    |011011101>
(0.25+0j)    |101101011>
(0.25+0j)    |101101101>
(0.25+0j)    |101110000>
(0.25+0j)    |101110110>
(0.25+0j)    |110101011>
(0.25+0j)    |110101101>
(0.25+0j)    |110110000>
(0.25+0j)    |110110110>
```

Listing 5.1: The quantum state of the nine data qubits of a ninja star after initialization.

We continue by testing the logical X and Z gate. We know that $Z_L |0\rangle_L = |0\rangle_L$

and $Z_L |1\rangle_L = -|1\rangle_L$. We test the X_L and Z_L gates by checking if they replicate this behavior. Simulation reproduces these results and therefore we can conclude that the X_L and Z_L gates are correct. The obtained nine qubit quantum state for the logical $|1\rangle$ state is shown in Listing 5.2.

```
(0.25+0j)    |001001001>
(0.25+0j)    |001001111>
(0.25+0j)    |001010010>
(0.25+0j)    |001010100>
(0.25+0j)    |010001001>
(0.25+0j)    |010001111>
(0.25+0j)    |010010010>
(0.25+0j)    |010010100>
(0.25+0j)    |100100010>
(0.25+0j)    |100100100>
(0.25+0j)    |100111001>
(0.25+0j)    |100111111>
(0.25+0j)    |111100010>
(0.25+0j)    |111100100>
(0.25+0j)    |111111001>
(0.25+0j)    |111111111>
```

Listing 5.2: The quantum state of the nine data qubits of a ninja star in the $|1\rangle_L$ state.

The next gate we tested was the logical Hadamard gate. To test the logical H gate, we initialize the ninja star to the $|0\rangle_L$ state and apply the H_L gate to obtain the state $H_L |0\rangle_L$ which should equal the state $|+\rangle_L$. We will test if the resulting state is indeed the $|+\rangle_L$ state to confirm that the H_L gate works correctly. We know that $X_L |+\rangle_L = |+\rangle_L$ and $Z_L |+\rangle_L = |-\rangle_L = H_L |1\rangle_L$. Simulations show that the state $H_L |0\rangle_L$ indeed reproduces the same behavior as the $|+\rangle_L$ state. Hence, we can conclude that the H_L gate is correct.

To test the logical $CNOT$ gate and the CZ gate, two ninja stars are required. We set the two ninja stars to the four possible computational basis states (i.e. $|0_1 0_0\rangle_L$, $|1_1 0_0\rangle_L$, $|0_1 1_0\rangle_L$, and $|1_1 1_0\rangle_L$) and apply a $CNOT_L$ gate with ninja star 0 as control qubit, and ninja star 1 as target qubit. The simulation results are then compared to the expected outcome states. As shown in Table 5.5, all four simulation results match the expected results. Hence, we can conclude that the $CNOT_L$ gate is correct. We performed the same test for the CZ_L gate and, as shown in Table 5.6, all four simulation results again match the expected outcome. We can conclude that both the $CNOT_L$ and the CZ_L are correct.

Initial state	Expected state after $CNOT_L$	Simulation result
$\|0_1 0_0\rangle_L$	$\|0_1 0_0\rangle_L$	$\|0_1 0_0\rangle_L$
$\|1_1 0_0\rangle_L$	$\|1_1 0_0\rangle_L$	$\|1_1 0_0\rangle_L$
$\|0_1 1_0\rangle_L$	$\|1_1 1_0\rangle_L$	$\|1_1 1_0\rangle_L$
$\|1_1 1_0\rangle_L$	$\|0_1 1_0\rangle_L$	$\|0_1 1_0\rangle_L$

Table 5.5: Logical initial state, expected state after applying a logical $CNOT$ gate (with qubit 0 as control and qubit 1 as target), and the state obtained by simulation.

Initial state	Expected state after CZ_L	Simulation result
$\|0_1 0_0\rangle_L$	$\|0_1 0_0\rangle_L$	$\|0_1 0_0\rangle_L$
$\|1_1 0_0\rangle_L$	$\|1_1 0_0\rangle_L$	$\|1_1 0_0\rangle_L$
$\|0_1 1_0\rangle_L$	$\|0_1 1_0\rangle_L$	$\|0_1 1_0\rangle_L$
$\|1_1 1_0\rangle_L$	$-\|1_1 1_0\rangle_L$	$-\|1_1 1_0\rangle_L$

Table 5.6: Logical initial state, expected state after applying a logical CZ gate, and the state obtained by simulation.

Finally, we want to verify the logical measurement. Logical measurement of a ninja star is performed by measuring all data qubits. The product of the nine measurement results (± 1) yields the logical measurement result. To verify the logical measurement procedure, we will take a closer look to the nine qubit quantum states of the $|0\rangle_L$ state (Listing 5.1) and the $|1\rangle_L$ state (Listing 5.2). The quantum states consist of binary states with corresponding amplitudes. When measuring all nine data qubits, one of the binary states in the quantum state will be the final measurement result. To verify if the measurement procedure is correct, we have to confirm the procedure against every possible measurement outcome. We can observe that for the $|0\rangle_L$ state, all binary states have an even parity (i.e. an even number of 1's) while for the $|1\rangle_L$ all binary states have an odd parity. Binary states with an even parity will yield a $+1$ logical measurement result while binary states with an odd parity will yield a -1 logical measurement result. We can conclude that measuring the $|0\rangle_L$ state will always return a logical $+1$ measurement result while measuring the $|1\rangle_L$ state will always return a logical -1 measurement result. Hence, the logical measurement procedure must be correct.

We want to mention that it is also possible to measure data qubits 0, 4, and 8 to perform a logical measurement. The parity of these three data qubits also yields the logical measurement result, but there is a difference compared to measuring and calculating the parity of all nine data qubits. The three-qubit logical measurement depends on the current rotation of the ninja star lattice while the nine-qubit logical measurement does not. In case the ninja star is in the rotated state, data qubits 2, 4, and 6 have to be measured instead. The simulations done for this thesis use the nine-qubit logical measurement for the ninja star.

5.2 Pauli frame verification

The second set of experiments were designed to verify the working mechanism of Pauli frames. The goal is to confirm that we can use a quantum system with a Pauli frame that works as described in Table 5.7 and still produce the same results as a quantum system without Pauli frame up to a certain unimportant global phase. We will first discuss the implementation of the Pauli frame before we discuss the test setup and the simulation results.

Operations	Execution steps
Initialization to $\|0\rangle$	1. Initialize the target qubit to $\|0\rangle$. 2. Set the corresponding Pauli record to I.
Measurement	1. Measure target qubit. 2. Modify measurement result based on Pauli record.
Pauli gates	1. Map Pauli record (no interaction with qubit).
Clifford gates	1. Map Pauli record(s). 2. Apply Clifford gate on target qubit(s).
Non-Clifford gates	1. Flush Pauli record(s). a. Apply gates in Pauli record(s) on target qubit(s). b. Reset Pauli record(s) to I. 2. Apply non-Clifford gate on target qubit(s).

Table 5.7: Pauli frame execution steps for different operations.

5.2.1 QPDO implementation

To verify the working mechanism of Pauli frames we created a Pauli frame layer using QPDO. We first implemented a Pauli frame class that manages the memory for the Pauli records and contains all the mapping tables required for the following set of gates: $\{I, X, Y, Z, H, S, CNOT, CZ, SWAP\}$. Gates that do not have a mapping table are treated as non-Clifford gates. Pauli records are implemented as an enumeration type that can be in one of the four states $\{I, X, Z, XZ\}$. The Pauli frame class also has specific procedures to handle initialization to $|0\rangle$ and measurement operations. The Pauli frame class takes a *Circuit* object as input and modifies the Pauli records based on the gates in the circuit and their corresponding mapping tables. The Pauli frame class returns a modified *Circuit* object where Pauli gates initially present in the input circuit are removed, and some Pauli gates are added in case Pauli records are required to be flushed. The Pauli frame class is also able to modify measurement results based on the current state of the Pauli records. Finally, there is an option to flush the complete Pauli frame.

To be able to add a Pauli frame to a QPDO control stack in a flexible way, we implemented a Pauli frame layer. This Pauli frame layer uses the Pauli frame class and implements the QPDO *Core* interface as presented in Table 4.1. An example of a control stack containing a Pauli frame layer is shown in Figure 5.2.

5.2.2 Random circuit simulation results

We used the test bench facilities of QPDO to verify the working mechanism of Pauli frames. First, we created a test bench based on random circuits to verify that the Pauli frame layer indeed does not change the final quantum state after executing a circuit. The random-circuit test bench creates a random circuit that contains the following gate types: $\{I, X, Y, Z, H, S, CNOT, CZ, SWAP, T, T^\dagger\}$. The circuit is then executed using

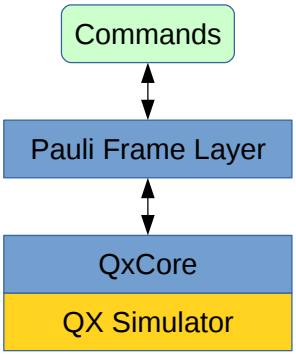


Figure 5.2: Schematic overview of a control stack with a Pauli frame layer and a *QxCore* layer.

the QX Simulator, and the resulting quantum state is stored. Now the same circuit is executed on a control stack consisting of a Pauli frame layer and a *QxCore* as shown in Figure 5.2. After execution of the circuit, the Pauli frame is flushed. The resulting quantum state is retrieved and compared to the earlier stored quantum state of the execution without Pauli frame layer. If the two quantum states match (up to a certain global phase) then we verified that the Pauli frame working mechanism is correct. A schematic view of the test setup is shown in Figure 5.3.

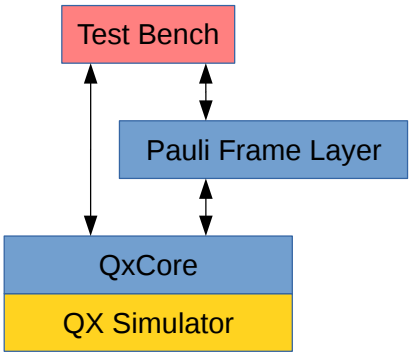


Figure 5.3: The test setup for the random-circuit test bench.

An example of a generated random circuit for five qubits with 20 gates is shown in Figure 5.4. This circuit was first executed without a Pauli frame and the resulting quantum state, which will be the reference quantum state, is shown in Listing 5.3. The qubits are reset, and the same random circuit is executed with a Pauli frame. The resulting quantum state before flushing the Pauli frame is shown in Listing 5.4. We can see that the quantum state before flushing the Pauli frame is different compared to the reference quantum state. The status of the Pauli frame layer is shown in Listing 5.5 and we can see that the Pauli frame tracked Pauli gates for multiple qubits. We flush the Pauli frame, and the final quantum state is shown in Listing 5.6. We can conclude that final quantum state equals the reference quantum state up to an unimportant global phase of -1.

```
                -0.5j      |10001>
                 0.5j      |10101>
 (0.353553+0.353553j)      |11001>
(-0.353553-0.353553j)      |11101>
```

Listing 5.3: The five qubit quantum state after executing the random circuit without Pauli frame.

```
                 (0.5+0j)      |00000>
                 (0.5+0j)      |00100>
(-0.353553+0.353553j)      |01000>
(-0.353553+0.353553j)      |01100>
```

Listing 5.4: The five qubit quantum state after executing the random circuit with Pauli frame before flushing.

```
Pauli frame layer with Pauli records:
        0    :    XZ
        1    :    I
        2    :    XZ
        3    :    I
        4    :    XZ
```

Listing 5.5: The status of the Pauli frame before flushing.

```
                 0.5j      |10001>
                -0.5j      |10101>
(-0.353553-0.353553j)      |11001>
(0.353553 +0.353553j)      |11101>
```

Listing 5.6: The five qubit quantum state after executing the random circuit with Pauli frame after flushing.

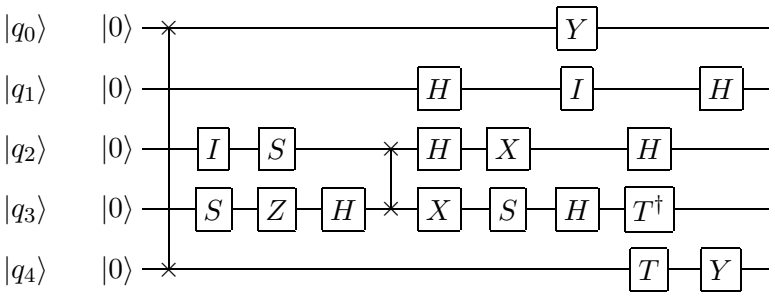


Figure 5.4: An example of a generated random circuit for 5 qubits with 20 gates.

The random circuit test bench was executed 100 times for a system with ten qubits using random circuits containing 1000 gates each. A graphical overview of the test setup is shown in Figure 5.3. The test bench reported that the simulations with Pauli frame yield the same results as the simulations with Pauli frame for all iterations. Thereby we verified that the Pauli frame working mechanism is correct and does not change the final quantum state after executing a circuit.

5.2.3 Ninja star measurement simulation results

The Pauli frame layer is also tested in combination with the ninja star layer to verify that measurements are handled correctly by the Pauli frame in case we do not flush the Pauli frame. We used a control stack consisting of a ninja star layer, a Pauli frame layer, and a *QxCore* layer. A schematic overview of the test setup is shown in Figure 5.5. To verify the measurement behavior of the control stack with a Pauli frame layer, we used a test bench that applies the circuit shown in Figure 5.6 to create the logical state $(|01\rangle + |10\rangle)/\sqrt{2}$ (which we will refer to as the odd Bell state) and measure the outcome. The measurement results of multiple iterations are collected and plotted in a histogram. The circuit to create the odd Bell state will be applied on the ninja star layer which expands all logical operations to physical operations and inserts ESM circuits. The explained test bench is relevant since the test procedure includes execution of Pauli gates and measurements on the same qubits, which will test the measurement procedures of the Pauli frame. None of the gates that will be executed due to the logical circuit or ESM circuits are non-Clifford, which means that the none of the Pauli records will be flushed during execution.

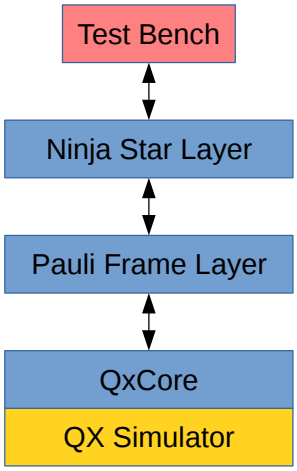


Figure 5.5: The test setup with a ninja star layer and a Pauli frame layer.

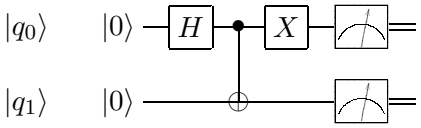
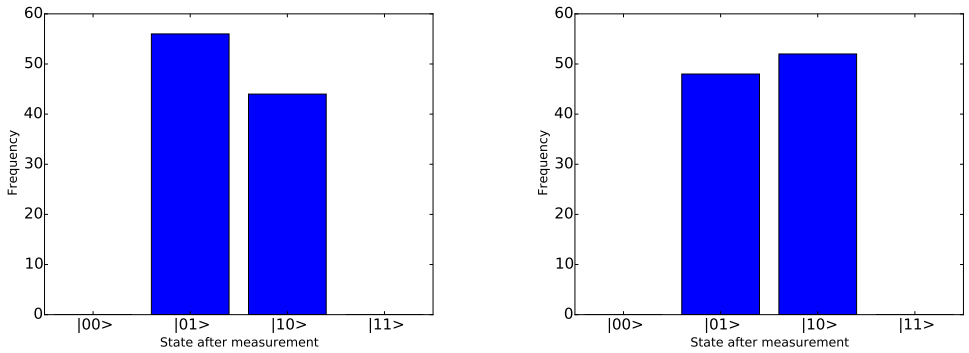


Figure 5.6: The circuit used to create an odd Bell state.

The expected outcome of the odd Bell state test bench would be a histogram with equal frequencies at the states $|01\rangle_L$ and $|10\rangle_L$. We performed the test bench with 100 iterations on the test setup shown in Figure 5.5 and on a similar test setup without Pauli frame. Both tests yielded equivalent results and the resulting histograms are shown in Figure 5.7a and 5.7b for respectively the control stack with and without Pauli frame. We can conclude that both resulting histograms match the expected outcome. Hence,

we can conclude that a system with a Pauli frame yields similar measurement results as a system without Pauli frame.



(a) Result with Pauli frame.

(b) Result without Pauli frame.

Figure 5.7: The resulting histograms of the odd Bell state test bench with and without Pauli frame.

5.3 Ninja star logical error rates

The last experiment is designed to investigate the effect of the Pauli frame mechanism on QEC. To do so, we calculate the Logical Error Rate (LER) P_L of an idling Surface Code 17 (SC17) logical qubit for different Physical Error Rate (PER) values p. The LER P_L is defined as the probability of a logical error happening within a single window [19] where a window is defined as the time to execute one or more rounds of ESM and a set of required corrections after decoding the obtained error syndromes.

For a given p, P_L can be calculated by simulation. In the simulation, a ninja star, constructed from 17 physical qubits, is initialized into the $|0\rangle_L$ or $|+\rangle_L$ state before we repeatedly perform ESM rounds without executing any logical operation. After every window, we check if there are any observable errors on data qubits, and if not, we check if a logical error occurred. We repeat this procedure and count the number of windows executed R and the number of logical errors detected m until m reaches a predefined maximum value. The LER P_L corresponding to a given p can be written as:

$$P_L|_p = \frac{m}{R} \tag{5.1}$$

Listing 5.7 contains a piece of pseudo code based on Python that summarizes the simulation procedure.

```
# Initializing used variables
window_count = 0
logical_error_count = 0

# Initialize the logical qubit
initialize_logical_qubit()

# Perform the experiment
while logical_error_count < MAX_LOGICAL_ERROR:
    execute_window()
    window_count += 1

    if no_observable_errors():
        if logical_error_happened():
            logical_error_count += 1

# Calculate the logical error rate
logical_error_rate = logical_error_count / window_count
```

Listing 5.7: Pseudo code summary of calculating the P_L of a ninja star.

5.3.1 Test setup

We used QPDO to perform the LER experiments. We chose CHP as the simulation back-end to save simulation time since this experiment only executes Clifford gates. The symmetric depolarizing error model [11, 19] is used to insert errors and is implemented as a QPDO layer. The Pauli frame is also implemented as a layer as explained in Section 5.2. Besides of the main layers required for the simulation, three counter layers are added to the control stack to count the number of operations and time slots transferred between the layers. The full control stack used for the LER experiments is shown in Figure 5.8.

The error layer implements the symmetric depolarizing error model. In this model, the PER p is the probability of an error occurring while executing a single operation on a physical qubit [11, 19]. For every single-qubit operation, the probability of a Pauli error, X, Y or Z, is for all $p/3$. Measurement in the computational basis can only have an X error with the probability of p. Idling a physical qubit for a time slot is also treated as a physical operation, the identity gate, I. For two-qubit gates the probability to insert a specific combination of two single-qubit errors is $p/15$ for all errors in the set $(\{I, X, Y, Z\} \times \{I, X, Y, Z\}) \setminus \{(I, I)\}$.

Error syndromes are the measurement result of the ancilla qubits and represent the parity among data qubits. A decoder is required to decode error syndromes into errors that happen on data qubits. For the LER experiments, the decoder used is a rule-based Look-Up Table (LUT) decoder as presented in [19] and implemented by [37]. The rule-based LUT decoder is specifically designed for the ninja star and uses three rounds of ESM results to detect errors. Every window w uses one round of ESM results from the previous window $w - 1$ as shown in Figure 5.9.

Detecting if there is any observable error in the data qubits is done by performing an extra round of ESM and verifying if the measurement results are all $+1$. This round of ESM used for diagnostic purposes should be error-free and not affect any counters in the

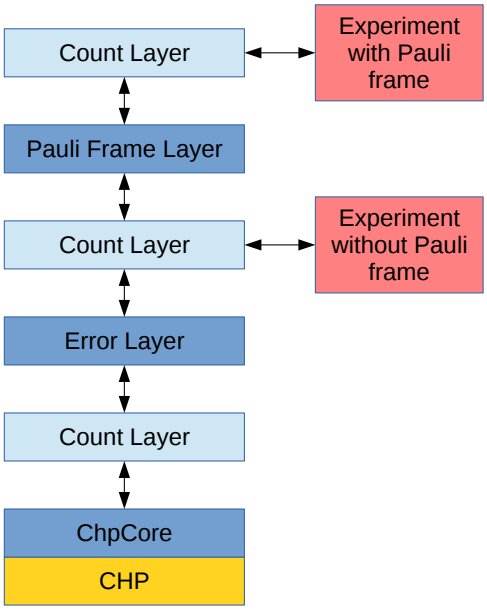


Figure 5.8: The test setup used for the Logical Error Rate experiments.

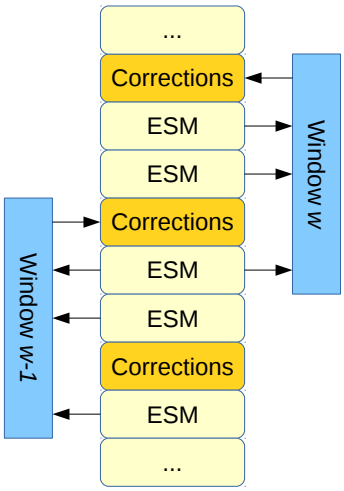
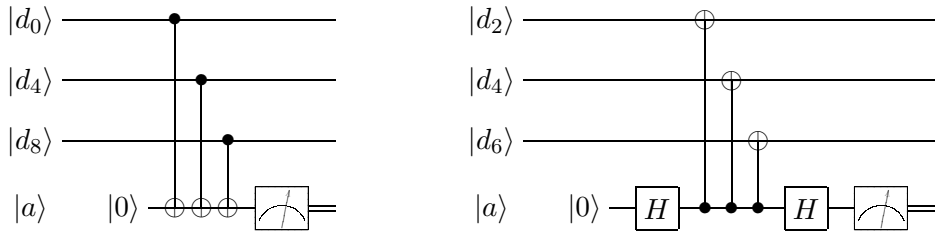


Figure 5.9: The scheme of Error Syndrome Measurement (ESM) results used for successive windows.

experiment. Therefore, apart from the normal execution mode, the control stack has a bypass mode which is only valid for system diagnostics circuits where the counter layers and error layers are bypassed.

Logical errors are detected using stabilizer circuits in the bypass mode. We can use the circuit shown in Figure 5.10a to verify if the ninja star is in the $|0\rangle_L/|1\rangle_L$ state which corresponds to the ancilla measurement result $+1/-1$. The circuit shown in Figure 5.10a is a $Z_0Z_4Z_8$ stabilizer measurement circuit and does not influence the logical $|0\rangle_L/|1\rangle_L$ state of the ninja star [11]. By detecting changes of logical state after successive windows,

we can detect X_L errors. For the same reason, Z_L errors can be detected without affecting the logical state using the stabilizer measurement circuit shown in Figure 5.10b with measurement result $+1/-1$ corresponding to the state $|+\rangle_L/|-\rangle_L$. More information about stabilizer circuits can be found in [6].



(a) Circuit for detecting X_L errors.

(b) Circuit for detecting Z_L errors.

Figure 5.10: The stabilizer circuits used to detect logical errors.

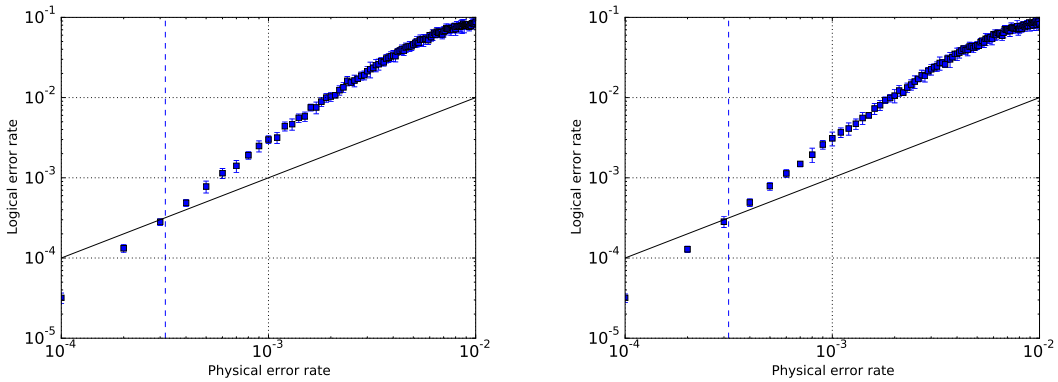
The ESM circuits for X and Z ancilla qubits, as described in Section 2.5.1, are performed in parallel which results in an ESM circuit containing a total of 48 gates divided over 8 time slots including preparation and measurements of ancilla qubits. Table 5.8 describes which time slot contains which gates.

Time slot	# Operations	Short description
1	4	Reset X ancilla qubits to $\|0\rangle$ state.
2	4	Reset Z ancilla qubits to $\|0\rangle$ state.
	4	Apply H gates on X ancilla qubits.
3-6	24	$CNOT$ gates between data qubits and ancilla qubits.
7	4	Apply H gates on X ancilla qubits.
8	8	Measure all ancilla qubits.

Table 5.8: Description of the Error Syndrome Measurement circuit used for the logical error rate experiment.

5.3.2 Results

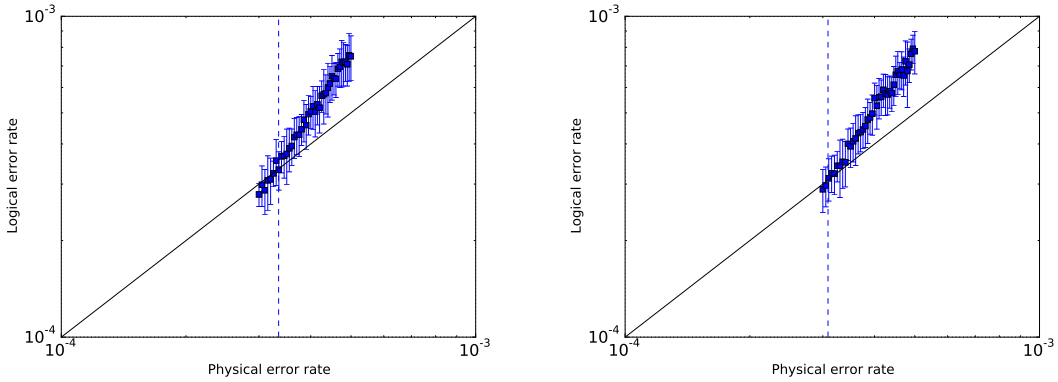
We performed separate LER experiments for X_L/Z_L errors, starting from the $|0\rangle_L/|+\rangle_L$ state, using the test setup shown in Figure 5.8 without Pauli frame. Both experiments were performed for PER values ranging from 1.0×10^{-4} to 1.00×10^{-2} with a step size of 1.0×10^{-4} where ten simulations are performed for every PER. Every simulation is terminated when 50 logical errors are detected. For every PER, we calculated the mean and standard deviation of the resulting LER of the ten individual simulations. The resulting graphs are shown in Figure 5.11 where the PER is represented on the horizontal axis and the resulting LER on the vertical axis. We also added the line $x = y$ and a vertical dashed line which indicates the intersection between the linear interpolated results and the line $x = y$.



(a) LER for logical X errors. (b) LER for logical Z errors.

Figure 5.11: Physical Error Rate versus Logical Error Rate for a Surface Code 17 logical qubit without Pauli frame.

From the graphs shown in Figure 5.11 we can conclude that the LER is similar for X_L and Z_L errors, which can be explained by the used error model. The symmetric depolarizing error model has equal probabilities for injecting X and Z errors on physical qubits resulting in an equal probability of causing a logical X or Z error. Both graphs intersect the line $x = y$ around $x \approx 3.0 \times 10^{-4}$ which is marked with a vertical dashed line and known as the pseudo-threshold (as explained in Section 2.5.1). We performed an other set of similar simulations for PER values around the pseudo-threshold ranging from 3.0×10^{-4} to 5.0×10^{-4} with a step size of 5×10^{-6} where 20 samples were taken for every PER. The results are shown in Figure 5.12.

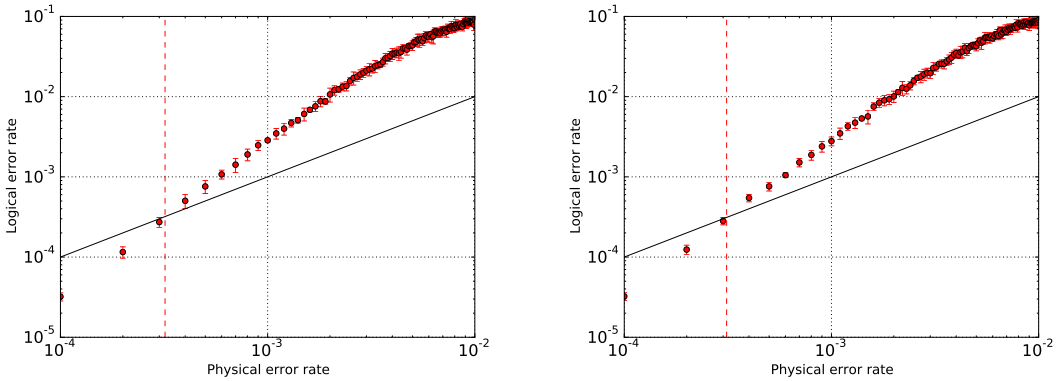


(a) LER for logical X errors. (b) LER for logical Z errors.

Figure 5.12: Physical Error Rate versus Logical Error Rate around the pseudo-threshold for a Surface Code 17 logical qubit without Pauli frame.

To observe the impact of a Pauli frame on the LER, we performed the same LER experiments as explained earlier, but then using the test setup shown in Figure 5.8 with a Pauli frame. The graphs with the resulting LER values are shown in Figure 5.13.

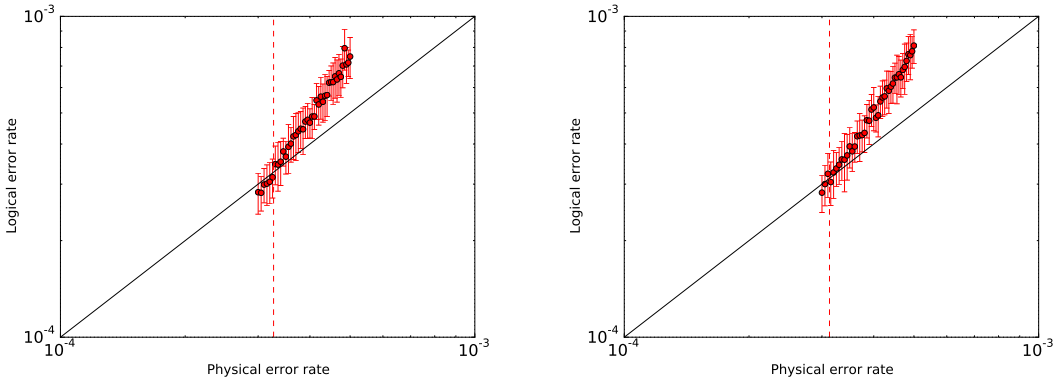
Again we observe that the LER is similar for X_L and Z_L errors, which was expected. The pseudo-threshold, which is marked with a vertical dashed line, can again be found around 3.0×10^{-4}. Again we performed extra simulations for PER values around the pseudo-threshold ranging from 3.0×10^{-4} to 5.0×10^{-4} of which the results are shown in Figure 5.14.



(a) LER for logical X errors.

(b) LER for logical Z errors.

Figure 5.13: Physical Error Rate versus Logical Error Rate for a Surface Code 17 logical qubit with Pauli frame.



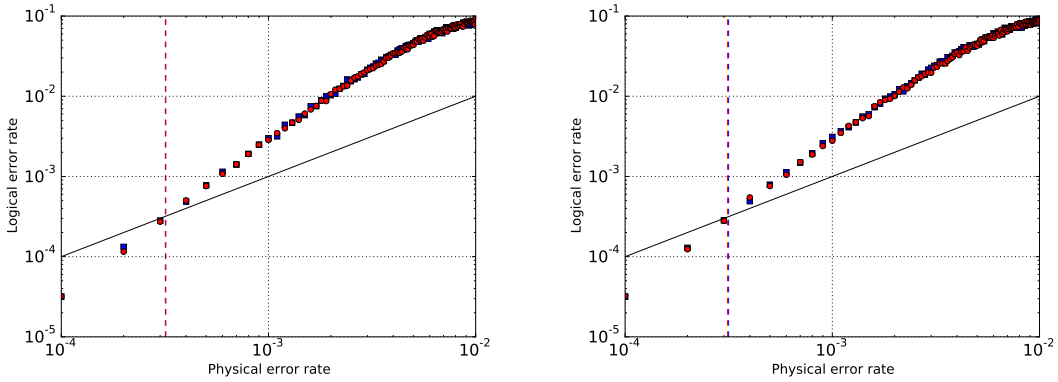
(a) LER for logical X errors.

(b) LER for logical Z errors.

Figure 5.14: Physical Error Rate versus Logical Error Rate around the pseudo-threshold for a Surface Code 17 logical qubit with Pauli frame.

We combined the results of the LER experiments with and without Pauli frame in one figure to be able to compare them. The results of the experiments performed over the range 1.0×10^{-4} to 1.00×10^{-2} are shown in Figure 5.16 while the results of the simulations around the pseudo-threshold of 3.0×10^{-4} are shown in Figure 5.16. Again the results with Pauli frame are indicated with red circles while the results without Pauli frame are indicated by blue squares. By visual inspection, we can conclude that the results of the LER experiments with Pauli frame look very similar to the results

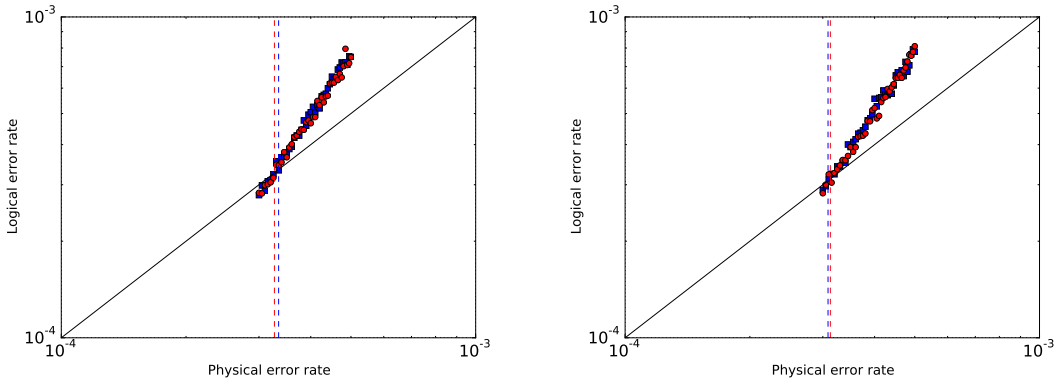
of the experiments without Pauli frame. Our observed results are also very similar to the results shown in [19] where they assume the presence of a Pauli frame during their simulations.



(a) LER for logical X errors.

(b) LER for logical Z errors.

Figure 5.15: Physical Error Rate versus Logical Error Rate for a Surface Code 17 logical qubit with (red circles) and without (blue squares) Pauli frame.



(a) LER for logical X errors.

(b) LER for logical Z errors.

Figure 5.16: Physical Error Rate versus Logical Error Rate for a Surface Code 17 logical qubit with (red circles) and without (blue squares) Pauli frame around the pseudo-threshold.

To take a closer look at the differences between the results with and without Pauli frame we calculated the absolute difference in LER δ_{P_L} for every PER p where δ_{P_L} is defined as:

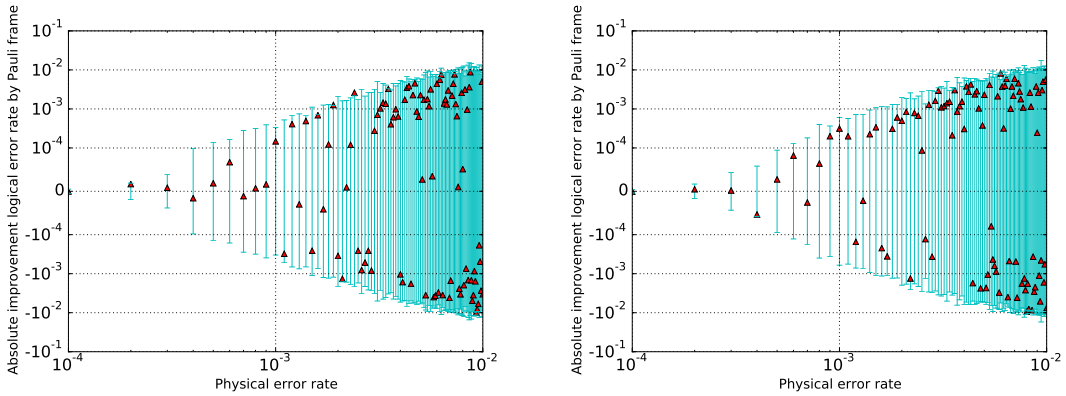
$$\delta_{P_L}|_p \quad = \quad P_{L_{\text{without PF}}} - P_{L_{\text{with PF}}} \tag{5.2}$$

Figure 5.17 shows the absolute LER difference δ_{P_L} (red triangles) for every p on a symmetric logarithmic scale where the vertical axis has a logarithmic scale except between $\pm 10^{-4}$ where the scale is linear. The δ_{P_L} for the results of the simulations around the

pseudo-threshold are shown in Figure 5.18 with a symmetric logarithmic scale on the vertical axis that is linear between $\pm 10^{-5}$. In the plots for δ_{P_L}, the maximum of the standard deviations of P_L with and without Pauli frame $\sigma_{\max}$ was added around the horizontal axis for every p where for a given p, $\sigma_{\max}$ is defined as:

$$\sigma_{\max}|_p \quad = \quad \max\left(\sigma(P_{L_{\text{with PF}}}) \ , \ \sigma(P_{L_{\text{without PF}}}) \right) \tag{5.3}$$

From Figure 5.17 and 5.18 we can observe that there is no consistent positive or negative difference in LER P_L for the simulations with and without Pauli frame. Also for nearly all p, δ_{P_L} can be found within the standard deviation regions $\pm\sigma_{\max}$. We can conclude that the Pauli frame has no measurable effect on the LER of a SC17 logical qubit.



(a) Difference for logical X errors.

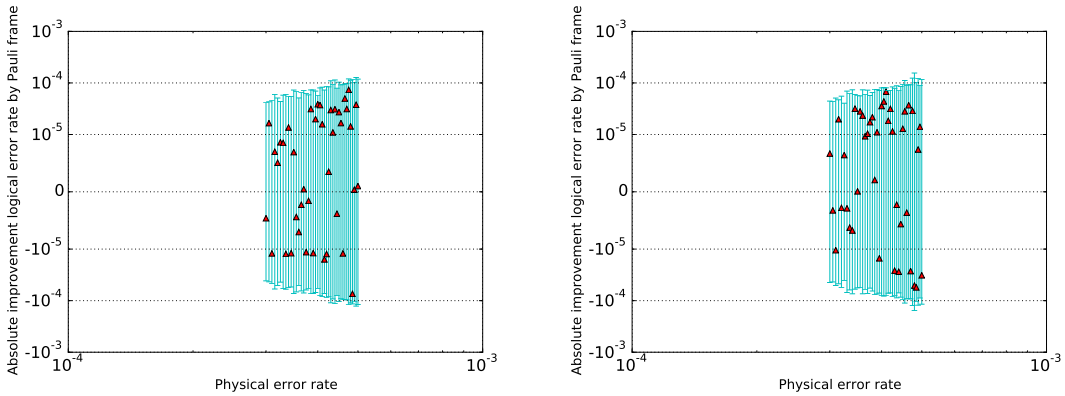
(b) Difference for logical Z errors.

Figure 5.17: The absolute Logical Error Rate difference between the experiments with and without Pauli frame (red triangles) plotted together with the standard deviations of the LER results (vertical bars).

We might wonder why the maximum standard deviations of the LER plotted in Figure 5.17 and 5.18 increase for higher PER values p. We know from Equation (5.1) that the LER P_L depends on the number of detected logical errors m and the number of counted windows R. The simulations we perform are always terminated when $m = 50$, which means that P_L can only be influenced by R. For a given p we would like to know the coefficient of variation (also known as the relative standard deviation) c of R which is defined as:

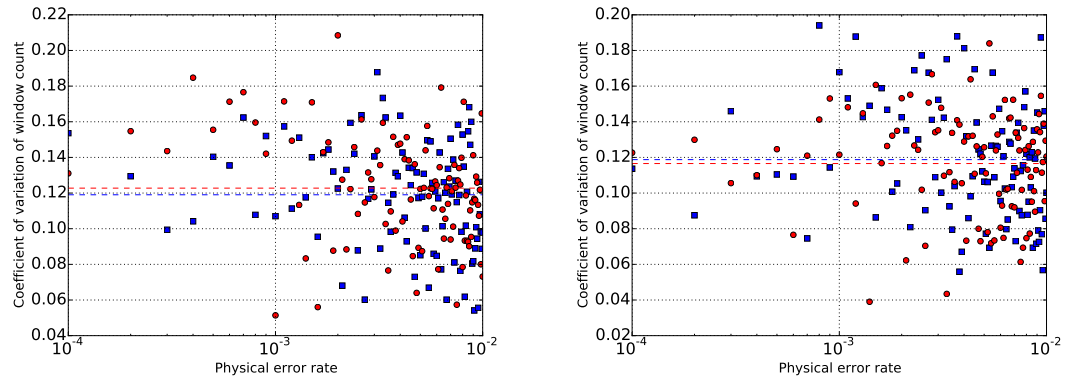
$$c_R|_p \quad = \quad \frac{\sigma_R}{\mu_R} \tag{5.4}$$

We calculated the c_R for every p and the results for the simulations with and without Pauli frame are shown in Figure 5.19 and 5.20 in red and blue. We can observe that for simulations with and without Pauli frame the average of c_R lies around 13% for all p which are marked with horizontal dashed lines. We can conclude that for every p, R has a relative standard deviation around 13%, which will result in a proportional coefficient of variation for P_L. Since P_L is larger for larger p, the absolute standard deviation for P_L will be greater for higher values of p. Hence, we observe larger absolute standard deviations for larger p in Figure 5.17 and 5.18.



(a) Difference for logical X errors. (b) Difference for logical Z errors.

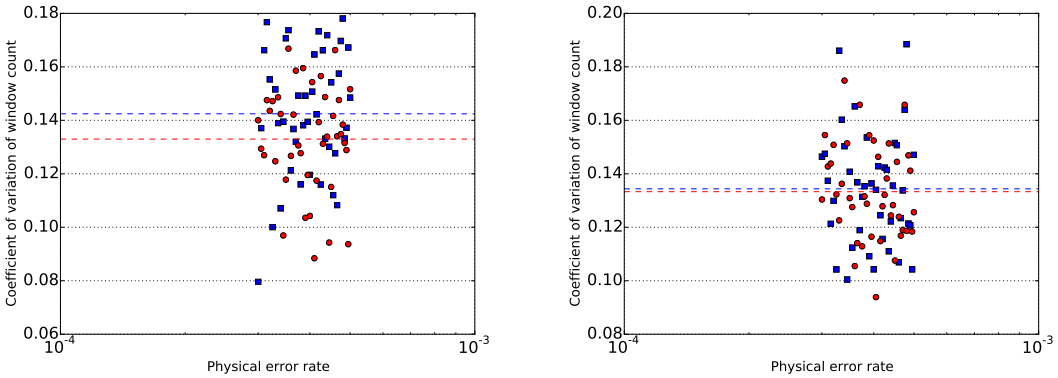
Figure 5.18: The absolute Logical Error Rate difference between the experiments with and without Pauli frame (red triangles) plotted together with the standard deviations of the LER results (vertical bars) around the pseudo-threshold.



(a) During LER simulations for X_L errors. (b) During LER simulations for Z_L errors.

Figure 5.19: The coefficient of variation of the number of counted windows with (red circles) and without (blue squares) Pauli frame.

We also performed statistical analysis on the LER samples retrieved by individual simulations. For every PER p we performed a t-test which can indicate if two data sets are significantly different from each other by calculating a ρ-value. A consistent ρ-value of 0.05 or lower would indicate that the difference between two sets of data is statistically significant. If not, the null hypothesis holds which indicates that the difference between two sets of data is not statistically significant. For a given p, we used one data set with LER values obtained by simulations without Pauli while the other data set contained the LER values obtained by simulations with Pauli frame. As mentioned earlier, simulations with and without Pauli frame were repeated 10 times for every p which means that the two data sets that we compare both have a sample size of 10. The t-test can be performed on matched pairs (paired t-test) or independent data sets (unpaired or independent t-

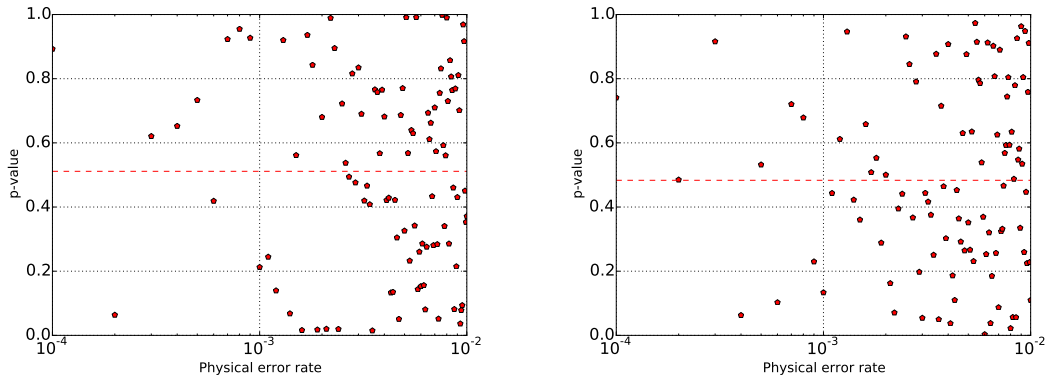


(a) During LER simulations for X_L errors.

(b) During LER simulations for Z_L errors.

Figure 5.20: The coefficient of variation of the number of counted windows with (red circles) and without (blue squares) Pauli frame around the pseudo-threshold.

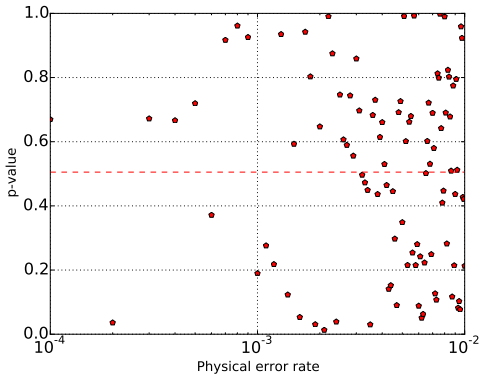
test). The LER samples obtained by our simulations are not paired for simulations with and without Pauli frame which would indicate the usage of an independent t-test, but all simulations do use the same simulation software which could imply that we are testing the same object and suggest the usage of a paired t-test. For those reasons, we performed both the paired as well as the independent t-test. The resulting ρ-values of the independent t-test for every p for simulations considering X_L and Z_L errors are shown in Figure 5.21 and the results for the same data using the paired t-test are shown in Figure 5.22. Figures 5.23 and 5.24 present the results of the individual and paired t-test performed on the data obtained by simulations around the pseudo-threshold where every data set has a sample size of 20. All graphs also contain a horizontal dashed line to indicate the mean of the ρ-values in the graph.



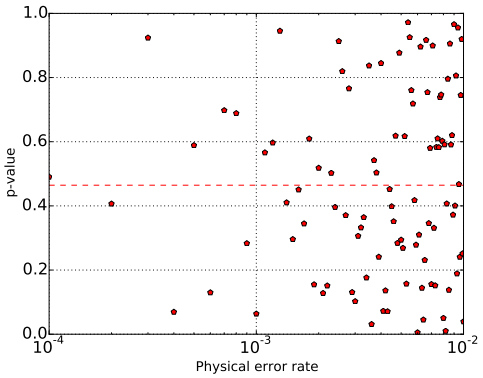
(a) ρ-values for data obtained by simulations for X_L errors.

(b) ρ-values for data obtained by simulations for Z_L errors.

Figure 5.21: The resulting ρ-values from the independent t-test performed on the data sets obtained with and without Pauli frame for different Physical Error Rate values.

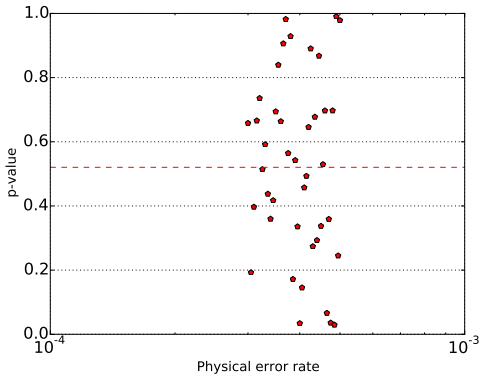


(a) ρ-values for data obtained by simulations for X_L errors.

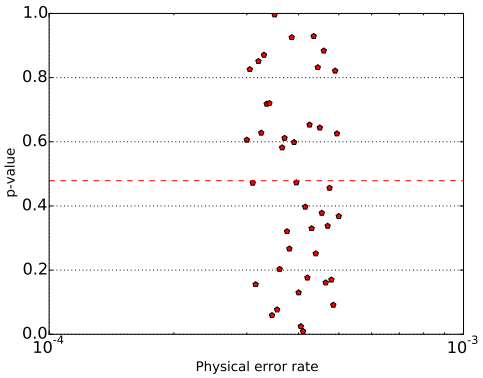


(b) ρ-values for data obtained by simulations for Z_L errors.

Figure 5.22: The resulting ρ-values from the paired t-test performed on the data sets obtained with and without Pauli frame for different Physical Error Rate values.



(a) ρ-values for data obtained by simulations for X_L errors.

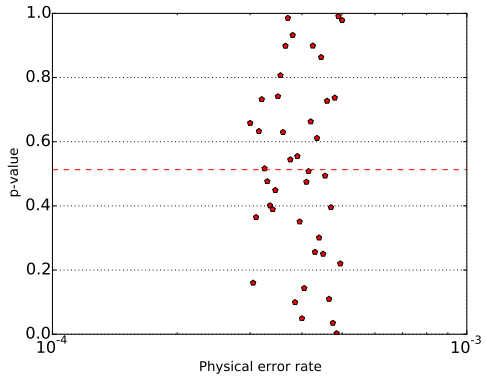


(b) ρ-values for data obtained by simulations for Z_L errors.

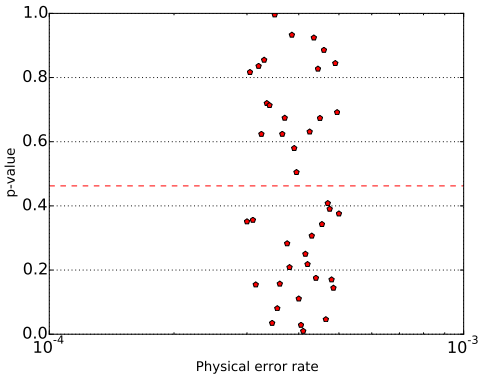
Figure 5.23: The resulting ρ-values from the independent t-test performed on the data sets obtained with and without Pauli frame for different Physical Error Rate values.

From the results of the t-tests shown in Figure 5.21, 5.22, 5.23, and 5.24 we can observe that the ρ-values vary a lot for different values of p and that the mean of all ρ-values lies around 0.5. Although we see a few ρ-values lower than 0.05 we do not observe such values consistently over many values of p. By conventional criteria, the difference between the data sets obtained with and without Pauli frame is considered to be not statistically significant. Hence, we can conclude that the Pauli frame has no statistically significant effect on the LER of a SC17 logical qubit.

To investigate why the usage of a Pauli frame does not significantly influence the LER we have taken a look at the gate and time slot counts that we collected during the simulations. We were especially interested in the number of Pauli gates that were filtered



(a) ρ-values for data obtained by simulations for X_L errors.



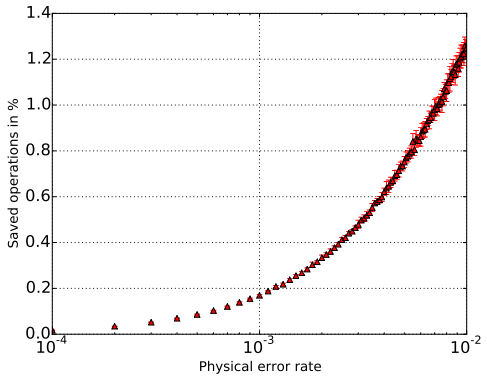
(b) ρ-values for data obtained by simulations for Z_L errors.

Figure 5.24: The resulting ρ-values from the paired t-test performed on the data sets obtained with and without Pauli frame for different Physical Error Rate values around the pseudo-threshold.

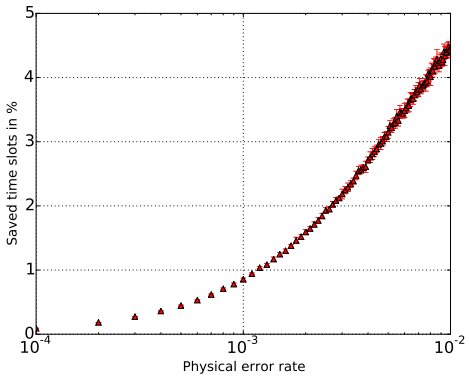
by the Pauli frame and the number of time slots that were removed as a result of that. An ESM circuit as shown in Table 5.8 does not contain any Pauli gate, so the only gates that can be filtered are Pauli gates that were inserted correct errors. A window consists of two rounds of ESM and one possible time slot for corrections. Every ESM round has 8 time slots resulting in a total of $2 \cdot 8 = 16$ time slots per window. As a result, a single window can have 16 or 17 time slots depending on if there are corrections to be applied or not. The only time slot that could potentially be filtered by the Pauli frame is the time slot containing the correction gates. If corrections are applied at the end of every window, $1/17 \approx 0.06$ or around 6% of the time slots could be filtered by the Pauli frame. As a result, the Pauli frame can never filter more than 6% of the time slots during our simulations. For all simulations regarding logical X errors, we calculated the percentage of gates and time slots that were filtered by the Pauli frame. We calculated the mean and standard deviation of these results for every PER p and the resulting graphs are shown in Figure 5.25. We also calculated the percentage of filtered gates and time slots for the simulations regarding logical X errors around the pseudo-threshold. Those results are shown in Figure 5.26.

From the graphs shown in Figure 5.25 and 5.26 we can conclude that the Pauli frame does filter gates and time slots, but the percentage of gates and time slots that are filtered is low, especially for PER values below the pseudo-threshold. These results can be explained by the distance d. For the SC17, distance $d = 3$ which means that the SC17 is able to correct $(d-1)/2 = 1$ error per ESM round. As a result, the pseudo-threshold of the SC17 is found at a very low PER. Due to the low PER and low distance of the SC17, there is a very low number of corrections to be applied over time, which results in a low number of gates and time slots being filtered by the Pauli frame.

The low proportion of gates and time slots that are filtered by the Pauli frame can also explain why we do not see any improvements in the LER by using a Pauli frame.

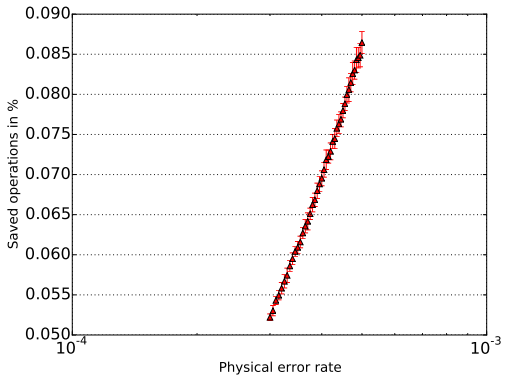


(a) Percentage of saved gates.

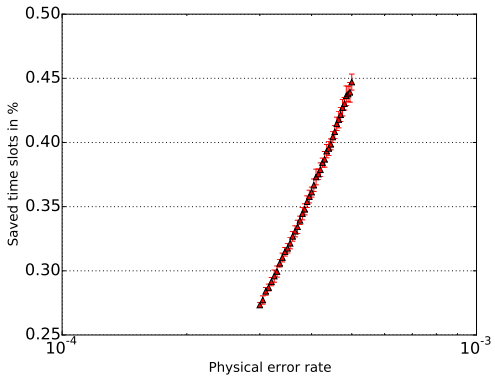


(b) Percentage of saved time slots.

Figure 5.25: The percentage of gates and time slots saved by the Pauli frame during Logical Error Rate simulations for X errors.



(a) Percentage of saved gates.



(b) Percentage of saved time slots.

Figure 5.26: The percentage of gates and time slots saved by the Pauli frame during Logical Error Rate simulations for X errors around the pseudo-threshold.

Improvements in the LER due to a Pauli frame can only be observed if a significant proportion of time slots is filtered. Since only a very low percentage of time slots is filtered at PER values below the pseudo-threshold, no improvement in the LER can be observed. For PER values above the pseudo-threshold the percentage of filtered time slots increases quickly, but still, we do not observe any improvement in LER. At PER values above the pseudo-threshold, the SC17 suffers from more errors than it can correct. Logical errors occur often, and a Pauli frame can not prevent that. Hence, we can not observe improvements in the LER at PER values above the pseudo-threshold.

We might wonder if we can expect improved LER values for a surface code with a larger distance $d > 3$ using a Pauli frame. Therefore, we need to look further into the theoretical benefit a Pauli frame can gain for the LER. The following reasoning is not meant as an analytical proof but rather a quantitative reasoning about the relative

impact of both the Pauli frame and the code distance on the LER.

Given a PER p, we could approximate the LER P_L based on the distance d, and the number of time slots required by a single window ts_{window}. We know that ts_{window} equals the sum of the time slots required by $d-1$ rounds of ESM ts_{rounds} and the time slot for corrections $ts_{\text{corrections}}$. In our simulations, the number of time slots in a single round of ESM $ts_{\text{ESM}} = 8$ as shown in Table 5.8. For the surface code, P_L is negatively correlated to the code distance d. Given the same PER and surface code lattice with a certain distance, more time slots required by a window would lead to more physical errors occurring per window, which could further increase the LER. Hence, we can assume that the LER is positively correlated to the number of time slots required by a window. Given a PER p, it would be a reasonable assumption to estimate the LER P_L by Equation (5.5):

$$P_L|_p \quad \propto \quad \frac{ts_{\text{window}}}{d} \tag{5.5}$$

where

$$ts_{\text{window}} = ts_{\text{rounds}} + ts_{\text{corrections}} \tag{5.6}$$

$$ts_{\text{rounds}} = (d-1)ts_{\text{ESM}} \tag{5.7}$$

$$ts_{\text{ESM}} = 8 \tag{5.8}$$

$$ts_{\text{corrections}} = \begin{cases} 1 & \text{if there are errors to be corrected} \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

We can improve the LER P_L by reducing ts_{window} using a Pauli frame. The Pauli frame can track all detected errors and as a result the value of $ts_{\text{corrections}}$ is reduced to $ts_{\text{corrections}} = 0$. Given a ts_{ESM}, let us calculate the upper-bound for the relative improvement in P_L that we can obtain by using a Pauli frame. Based on Equation (5.5) we will approximate P_L with/without Pauli frame as shown in Equation (5.10)/(5.11) where C is a constant:

$$C \cdot \frac{ts_{\text{window}}}{d}\bigg|_{\text{PF}} \quad \text{where} \quad ts_{\text{corrections}} = 0 \quad \text{(with Pauli frame)} \tag{5.10}$$

$$C \cdot \frac{ts_{\text{window}}}{d}\bigg|_{\text{wo PF}} \quad \text{where} \quad ts_{\text{corrections}} = 1 \quad \text{(without Pauli frame)} \tag{5.11}$$

Now we can define the upper-bound for the relative improvement in P_L as:

$$\begin{aligned} B_{\text{relative}} &\propto \frac{C \cdot \frac{ts_{\text{window}}}{d}\Big|_{\text{wo PF}} - C \cdot \frac{ts_{\text{window}}}{d}\Big|_{\text{PF}}}{C \cdot \frac{ts_{\text{window}}}{d}\Big|_{\text{wo PF}}} \\ &= \frac{ts_{\text{window}}|_{\text{wo PF}} - ts_{\text{window}}|_{\text{PF}}}{ts_{\text{window}}|_{\text{wo PF}}} \\ &= \frac{1}{ts_{\text{window}}|_{\text{without PF}}} \\ &= \frac{1}{(d-1)ts_{\text{ESM}} + 1} \end{aligned} \tag{5.12}$$

From Equation (5.12), we can conclude that the upper-bound for the relative improvement in LER that can be obtained by using a Pauli Frame converges to 0 for a large distance d and for a large ts_{ESM}. We plotted Equation (5.12) for $ts_{\text{ESM}} = 8$ in Figure 5.27 and we can observe that the upper-bound on the relative improvement quickly decreases to values below 3%. Hence, we do not expect an improved LER that for a larger distance d by using a Pauli frame.

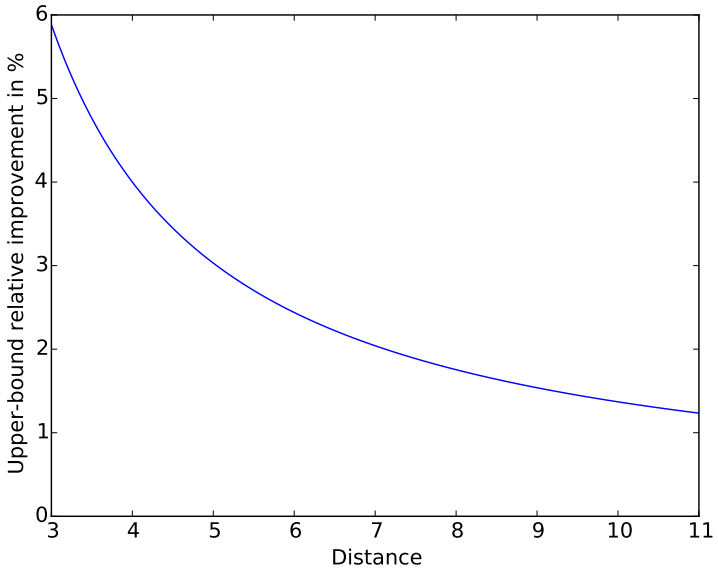


Figure 5.27: The upper-bound on the relative improvement in Logical Error Rate that can be obtained by using a Pauli frame for $ts_{\text{ESM}} = 8$.

6 Conclusion

We have analyzed the working principles of Pauli frames in detail and created an implementation using the QPDO software. Our simulations show that when executing the same circuits, a quantum system with a Pauli frame does yield the same measurement results as a quantum system without a Pauli frame. We have also shown that by tracking Pauli operations instead of executing them, we are still able to restore the quantum state to the state it would have been without the usage of a Pauli frame. For those reasons it is possible to add a Pauli frame to a quantum system while still yielding the same results, even when the usage of a Pauli frame causes fewer gates being applied on its corresponding qubits.

We have also verified the logical operations of the Surface Code 17 (SC17) by simulation. The verified logical operations include preparation to $|0\rangle_L$, X_L, Z_L, H_L, transversal $CNOT_L$, transversal CZ_L, and nine-qubit measurement in the Z_L basis. In our simulations, we preferred to use the nine-qubit measurement over the three-qubit measurement since lattice rotation does not influence the former.

We have shown that the usage of a Pauli frame does not cause an observable improvement on the Logical Error Rate (LER) of a SC17, even when the Pauli frame effectively filters all gates required to correct errors. By doing further analysis, we concluded that a surface code with a larger distance might also not benefit from an improved LER by the usage of a Pauli frame. Nevertheless, by using a Pauli frame we can still benefit from relaxed timing constraints for error decoding and Error Syndrome Measurement (ESM) circuit execution time.

Functional simulations have shown that our proposed design for a quantum computer architecture targeting a SC17 quantum chip can maintain a logical qubit. Our architecture is capable of performing Quantum Error Correction (QEC) and can schedule converted logical operations for execution.

Finally, we can conclude that the QPDO software is very flexible and offers powerful simulation capabilities. The combination of a layered structure combined with different simulation back-ends has proven to be very useful, especially in conjunction with automated test benches. All simulations required for this report, and more, have been performed with success using QPDO.

Future work

The work in this thesis could be extended in different directions.

We verified a set of logical operations for a SC17 logical qubit, but it would be interesting to extend the set of operations. In [14] a procedure for state injection was proposed that would make it possible to extend the set of logical gates for the SC17. It might also be interesting to look at different techniques to encode logical qubits using

the surface code (e.g. defect-based logical qubits as discussed in [6]) and verify the corresponding logical operations.

The experiments considering the effect of a Pauli frame on the LER of a SC17 can also be extended. It would be interesting to repeat these experiment using a larger distance surface code to verify our expectations that for a larger distance surface code, there will be no benefit in LER by using a Pauli frame. Such simulations would in the first place require error syndrome decoders that are suitable for larger surface codes. Future simulations could also benefit from more realistic error models.

Finally, we would like to take a deeper look into our proposed quantum computer architecture [34]. Interesting work could be done by performing more accurate simulations of the quantum computer architecture up to a level where we perform clock-cycle accurate emulation.

Bibliography

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pp. 124–134, IEEE, 1994.

[2] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Physical review letters*, vol. 79, no. 2, p. 325, 1997.

[3] D. Riste, S. Poletto, M. Z. Huang, *et al.*, "Detecting bit-flip errors in a logical qubit using stabilizer measurements," *Nat Commun*, vol. 6, 04 2015.

[4] A. Córcoles *et al.*, "Demonstration of a quantum error detection code using a square lattice of four super-conducting qubits," *Nature Comm.*, vol. 6, 2015.

[5] J. Kelly *et al.*, "State preservation by repetitive error detection in a superconducting quantum circuit," *Nature*, vol. 519, no. 7541, pp. 66–69, 2015.

[6] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, p. 032324, 2012.

[7] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Physical review A*, vol. 52, no. 4, p. R2493, 1995.

[8] E. Knill, "Quantum computing with realistically noisy devices," *Nature*, vol. 434, no. 7029, pp. 39–44, 2005.

[9] N. Khammassi, "QX simulator." `http://www.xpu-project.net/qx/download.html`, 2016. Universal quantum simulator developed by the Computer Engineering department of Delft University of Technology.

[10] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, no. 5, p. 052328, 2004.

[11] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[12] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, vol. 87, no. 2, p. 307, 2015.

[13] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, no. 1, pp. 2–30, 2003.

[14] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123011, 2012.

[15] A. G. Fowler, A. M. Stephens, and P. Groszkowski, "High-threshold universal quantum computation on the surface code," *Physical Review A*, vol. 80, no. 5, p. 052312, 2009.

[16] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, 2002.

[17] N. C. Jones, R. Van Meter, A. G. Fowler, P. L. McMahon, J. Kim, T. D. Ladd, and Y. Yamamoto, "Layered architecture for quantum computing," *Physical Review X*, vol. 2, no. 3, p. 031007, 2012.

[18] S. J. Devitt, W. J. Munro, and K. Nemoto, "Quantum error correction for beginners," *Reports on Progress in Physics*, vol. 76, no. 7, p. 076001, 2013.

[19] Y. Tomita and K. M. Svore, "Low-distance surface codes under realistic quantum noise," *Physical Review A*, vol. 90, no. 6, p. 062320, 2014.

[20] Y.-C. Zheng and T. A. Brun, "Fault-tolerant holonomic quantum computation in surface codes," *Physical Review A*, vol. 91, no. 2, p. 022302, 2015.

[21] M. Takita, A. Córcoles, E. Magesan, B. Abdo, M. Brink, A. Cross, J. M. Chow, and J. M. Gambetta, "Demonstration of weight-four parity measurements in the surface code architecture," *arXiv preprint arXiv:1605.01351*, 2016.

[22] J. Kelly, R. Barends, A. Fowler, A. Megrant, E. Jeffrey, T. White, D. Sank, J. Mutus, B. Campbell, Y. Chen, *et al.*, "State preservation by repetitive error detection in a superconducting quantum circuit," *Nature*, vol. 519, no. 7541, pp. 66–69, 2015.

[23] M. B. Hastings and A. Geller, "Reduced space-time and time costs using dislocation codes and arbitrary ancillas," *arXiv preprint arXiv:1408.3379*, 2014.

[24] J. Edmonds, "Maximum matching and a polyhedron with 0, l-vertices," *J. Res. Nat. Bur. Standards B*, vol. 69, no. 1965, pp. 125–130, 1965.

[25] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of mathematics*, vol. 17, no. 3, pp. 449–467, 1965.

[26] E. Knill, "Scalable quantum computing in the presence of large detected-error rates," *Physical Review A*, vol. 71, no. 4, p. 042322, 2005.

[27] D. P. DiVincenzo and P. Aliferis, "Effective fault-tolerant quantum computation with slow measurements," *Physical review letters*, vol. 98, no. 2, p. 020501, 2007.

[28] P. Aliferis and J. Preskill, "Fault-tolerant quantum computation against biased noise," *Physical Review A*, vol. 78, no. 5, p. 052331, 2008.

[29] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: A framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, p. 1, ACM, 2014.

[30] S. Balensiefer, L. Kregor-Stickles, and M. Oskin, "An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures," in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 186–196, IEEE Computer Society, 2005.

[31] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *Computer*, no. 1, pp. 74–83, 2006.

[32] D. Wecker and K. M. Svore, "Liqui—¿: A software design architecture and domain-specific language for quantum computing," *arXiv preprint arXiv:1402.4467*, 2014.

[33] K. Svore, A. Cross, A. Aho, I. Chuang, and I. Markov, "Toward a software architecture for quantum computing design tools," in *Proceedings of the 2nd International Workshop on Quantum Programming Languages (QPL)*, pp. 145–162, 2004.

[34] X. Fu, L. Riesebos, L. Lao, C. 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*, pp. 323–330, ACM, 2016.

[35] D. Iorga, "Python wrapper for CHP," 2016. Wrapper to enable quantum simulations with CHP using function calls from Python.

[36] H. Zimmermann, "OSI reference model–the iso model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

[37] S. Varsamopoulos, "Rule-based lookup table decoder," 2016. Python implementation of the rule-based lookup table decoder.