Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Development of a Vendor Independent Quantum Computing Transpiler

Thomas Wangler

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Marie Salm, M.Sc. Benjamin Weder, M.Sc.
Commenced:	May 4, 2020
Completed:	November 4, 2020

Abstract

In 2019 a quantum computer took 200 seconds for a task for which a state-of-the-art classical supercomputer would need roughly 10,000 years. Quantum supremacy was shown and the development of quantum technology proceeds quickly. However, several aspects limit the implementation and execution of quantum circuits. The vendors of quantum computers provide their own proprietary software development kit (SDK) to execute quantum circuits on their quantum processing units (QPUs). Furthermore, the computers differ regarding various properties like the native gate set and the qubit connectivity. Therefore, quantum circuits cannot be executed on arbitrary QPUs, but there exists a tight coupling between a QPU and a quantum algorithm implementation. To decouple the development of a circuit from the QPU, this thesis proposes a quantum circuit analysis and transpilation framework that integrates quantum SDKs and enables the comparison and analysis of quantum circuits, as well as the export of executable circuits in the respective assembly language of QPUs from different vendors. Currently existing QPUs, also called Noisy Intermediate-Scale Quantum (NISQ) machines, have a very limited number of qubits and are prone to failure. Thus, the integration of different QPUs enhances the possibilities of quantum circuit developers and avoids the SDK lock-in. Additionally, a graphical user interface is developed to support the user in the whole process of importing, visualizing, editing, and simulating a quantum circuit, as well as, choosing a suitable QPU to execute the circuit.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AI artificial intelligence. 31

CNOT controlled-NOT. 21

CSD Cosine-Sine Decomposition. 58

CZ controlled-phase. 22

DAG directed acyclic graph. 46

IBM QX IBM quantum experience. 33

NISQ Noisy Intermediate-Scale Quantum. 3

QDK quantum development kit. 37

QPU quantum processing unit. 3

qubit quantum bit. 17

quilc Quil compiler. 36

SDK software development kit. 3

1 Introduction

Quantum computers can solve several computational tasks efficiently for which no classical efficient solution is known [NC02]. Shor's algorithm, e.g., is a quantum algorithm that solves integer factorization in polynomial time which has a significant impact on the security of cryptography schemes [Sho99]. Several vendors like IBM and Rigetti provide *quantum processing units (QPUs)* that are used to execute implementations of quantum algorithms. There exist various quantum computing models like one-way, adiabatic, and gate-based [AVK+08; NC02; RB01]. The QPUs of IBM and Rigetti are gate-based models. However, the properties of gate-based QPUs differ in several aspects like the number of qubits, the qubit connectivity, the native gate set, the decoherence time, and the error rate. The native gate set refers to the gates which can be directly implemented on a specific QPU. The qubit connectivity describes the connections between the qubits. Not all qubits on a QPU are directly connected. Therefore, quantum operations that are applied to multiple qubits cannot be executed on arbitrary qubits. Quantum algorithm implementations can be described by quantum circuits. A quantum circuit consists of gates and measurements that are applied to qubits [NC02]. A quantum circuit is, therefore, a gate-based representation of a quantum algorithm [SBB+20]. The mapping of a quantum circuit to an executable quantum circuit for a specific QPU is the task of a transpiler. In this context, executable means that the quantum circuit consists merely of native gates of the specified QPU and the qubit connectivity is taken into account.

Despite the differences in the hardware, most vendors also provide their own software development kit (SDK) to implement a quantum circuit specifically for the chosen hardware [LaR19]. The usage of proprietary SDKs leads to a tight coupling between a QPU and a quantum algorithm implementation. As a consequence, the developers have to decide which SDK to use before the development of the algorithm implementation which also limits the set of QPU which can be used to execute the quantum circuit. Deciding which SDK should be used for a specific quantum circuit, therefore, requires immense mathematical knowledge of the algorithm and, additionally, technical knowledge of the QPU and the SDK [SBB+20].

This thesis proposes a quantum circuit analysis and transpilation framework that allows the execution of SDK specific quantum algorithm implementations on QPUs from IBM and Rigetti to reduce the SDK lock-in. Furthermore, the QPUs are analyzed regarding the number of additional native gates that are required for executing the given algorithm implementation. Various metrics like the maximum number of two-qubit gates or the maximum number of hardware pulses executed on a specific qubit in the transpiled quantum circuit are taken into consideration. The quantum SDKs and languages Qiskit, pyQuil, OpenQASM, and Quil are supported for the import, the export, and the transpilation of quantum circuits. The conversion functionality is based on a plugin model to allow the integration of further SDKs and languages. Besides the implementation of the framework, this thesis contains an analysis of current QPUs from IBM and Rigetti regarding their native gate sets and the qubit connectivity, as well as, a comparison of different quantum SDKs with respect to their standard gate sets and their conversion functionalities.

The thesis is structured in the following way: Firstly, in Chapter 2 the fundamental concepts of quantum computing and the motivation for the development of a quantum circuit analysis and transpilation framework are discussed. Chapter 3 introduces related work regarding the decomposition of quantum gates, mapping algorithms to solve the qubit connectivity constraint, and quantum transpilers in general. Afterwards, in Chapter 4 current quantum SDKs are analyzed. This includes the comparison of currently available QPUs from IBM and Rigetti regarding their native gate sets and the qubit connectivity. In Chapter 5, the architecture of the developed framework is explained including the internal data model and the language and SDK conversion. Chapter 6 discusses the implementation of the framework. Afterwards, in Chapter 7 the advantages and limitations of the developed system are discussed. Chapter 8 concludes the thesis.

2 Concepts and Motivation

This chapter gives a brief introduction to the fundamental concepts of quantum computing. Afterwards, the concept of a quantum transpiler that integrates quantum SDKs and QPUs from different vendors is motivated.

2.1 Fundamental Concepts of Quantum Computing

In the following, the concepts of quantum computing that are important for the development of a quantum computing transpiler are explained. Firstly, the concept of quantum bits (qubits) in Section 2.1.1 and the operators that manipulate qubits in Section 2.1.2 are explained. This knowledge is then applied to quantum circuits and the concept of quantum algorithms in Section 2.1.4. Furthermore, the limitations of NISQ devices and the requirements for a quantum transpiler following from these limitations are introduced.

2.1.1 Qubits

In classical computing the *bit* is the fundamental unit to store information. The analogous unit in quantum computation is the *quantum bit (qubit)* [NC02]. The states of a qubit are $|0\rangle$ and $|1\rangle$ which correspond to the states 0 and 1 in the classical computation. However, contrary to classical bits that are in either state 0 or state 1, a qubit can be in any state that is a linear combination of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.1}$$

where α and β are complex numbers [NC02]. Therefore, the state of a qubit can be described by a two-dimensional complex vector in a Hilbert space [NC02].

Measurement of a Qubit

To measure a qubit the measuring device must contain two preferred states whose corresponding vectors $\{|u\rangle, |u^{\perp}\rangle\}$ form an orthonormal basis [Sca12]. Applying the measurement operation transforms the state to one of the basis vectors $|u\rangle$ or $|u^{\perp}\rangle$ [Sca12], e.g. measuring the qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ in the computational basis ($\{|0\rangle, |1\rangle\}$) leads with probability $|\alpha|^2$ to 0 and with probability $|\beta|^2$ to 1 [NC02]. Therefore, the equation $|\alpha|^2 + |\beta|^2 = 1$ must hold. The property that the state of a qubit can be $|0\rangle$ and $|1\rangle$ at the same time is also called *superposition*. However, measuring the state leads to the collapsing of the superposition to the measured state [NC02].

Single Qubits

There are infinitely many states of a single qubit which makes an intuitive understanding of a qubit state rather difficult [Sca12]. The *Bloch sphere* is a three-dimensional sphere that allows the geometric representation of a state of a qubit [NC02]. The following equation shows how the two-dimensional complex space that characterizes the state can be described by a three-dimensional sphere.

Equation (2.1) can be rewritten by applying $|\alpha|^2 + |\beta|^2 = 1$ as

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle\right) \equiv \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle \tag{2.2}$$

with θ, ϕ and $\gamma \in \mathbb{R}$. θ and ϕ describe a point on the Bloch sphere (see Figure 2.1) [NC02]. Global factors do not influence the outcome of the measurement of a qubit, hence the factor $e^{i\gamma}$ can be omitted.

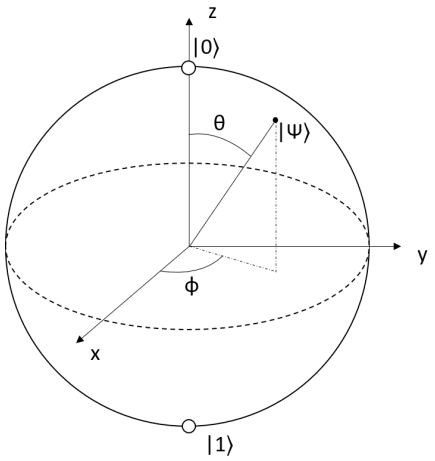


Figure 2.1: Bloch sphere representation

Important single-qubit states besides the computational basis are the following states: $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$, $|-\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$, $|i\rangle = \frac{|0\rangle+i\cdot|1\rangle}{\sqrt{2}}$, $|-i\rangle = \frac{|0\rangle-i\cdot|1\rangle}{\sqrt{2}}$ [Sca12]. These are often referred to and, they have, therefore, special labels. Furthermore, they are in superposition. The basis $\{|+\rangle, |-\rangle\}$ is called Hadamard basis [Sca12].

Qubit Register

Previously single qubits were discussed. However, to achieve performance improvements with quantum computing compared to classical computing multiple qubits are needed. The state vector that describes two qubits in relation to the computational basis is

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{bmatrix}\alpha_{00}\\ \alpha_{01}\\ \alpha_{10}\\ \alpha_{11}\end{bmatrix} \tag{2.3}$$

where $\alpha_{ij} \in \mathbb{C}$ [NC02]. The probabilities for the computational basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ must sum up to 1: $\sum_{x\in\{0,1\}^2} |\alpha_x|^2 = 1$. A quantum register is the generalization to n qubits. A quantum register of size n can represent 2^n values concurrently because the state vector is in a 2^n dimensional Hilbert space that is constructed as the tensor product of the Hilbert spaces of the single qubits [NC02; Sca12]. Therefore, the quantum computer operating on these qubits can manipulate 2^n values concurrently. This is also called *quantum parallelism* [NC02]. In contrast, classical computing uses the direct sum instead of the tensor product to describe the state space for which it holds true that $dim(V \oplus W) = dim(V) + dim(W)$ [Sca12].

A fundamental concept of quantum computing is *quantum entanglement* [Sca12]. The state of a quantum register is entangled if the state cannot be defined as the tensor product of the qubits. In contrast to this, a state is called separable if it can be defined as the tensor product of the qubits. An entangled quantum state can only be seen as the whole register, but not as the single qubits, i.e. entangled qubits cannot be individually represented in the Bloch sphere, because the state cannot be separated to the single qubits [NC02]. An interesting consequence of quantum entanglement can be seen at the *Bell states*:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.4}$$

$$|\phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \tag{2.5}$$

$$|\psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \tag{2.6}$$

$$|\psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \tag{2.7}$$

The Bell states are entangled [Sca12]. If the first qubit is measured, the whole state collapses. E.g. if the first qubit of $|\phi^+\rangle$ is measured (see Section 2.1.1) with the outcome 0, the measurement of the second qubit must also yield 0, and the post-measurement state would become $|\phi'\rangle = |00\rangle$ [NC02]. The measurement outcomes are correlated [NC02].

2.1.2 Operators

Quantum transformation is a mapping from the state space of the quantum system to itself [Sca12]. This definition excludes measurements.

Unitary Operator

In the gate-based approach of quantum computing transformations are implemented with unitary operations that represent quantum gates [SBB+20; Sch16]. An operator U on a Hilbert space $\mathbb{H}$ is called unitary [Sch16] if

$$\langle U\psi|U\phi\rangle = \langle\psi|\phi\rangle \quad \forall\, |\psi\rangle, |\phi\rangle \in \mathbb{H} \tag{2.8}$$

Unitary operators are linear, length-preserving, volume-preserving, isometric, and their adjoint operator is their inverse [NC02; Sch16]. Moreover, for all eigenvalues $\lambda \in \mathbb{C}$ of U it holds true that $|\lambda| = 1$ and, i.e., $\lambda = e^{it}$ where $t \in \mathbb{R}$ [NC02]. Additionally, the product U_1U_2 of two unitary operators U_1, U_2 is unitary and the tensor product $U_1 \otimes U_2$ is a unitary transformation if U_1 and U_2 are unitary operators [Sca12].

Unitary operators are a special case of isometries. Isometries are distance-preserving transformations between two Hilbert spaces [ICK+16].

Single-qubit Gates

A quantum gate manipulating a single qubit can be described as a 2x2 matrix [NC02]. In contrast to classical computing where the only non-trivial single bit gate is the *NOT* gate, there do exist infinitely many single-qubit gates in quantum computing. If a gate U should be executed on a single qubit of a quantum register it means that the operator $I \otimes ... \otimes I \otimes U \otimes I \otimes ... \otimes I$ is applied to the quantum register where I describes a single-qubit identity gate. [Sca12]. Alternatively, several single quantum gates can be applied in parallel if there is no intersection between the qubit sets they are operating on.

Important gates are the Pauli gates [NC02]:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}; \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.9}$$

The names of the gates describe around which axis in the Bloch sphere (see Section 2.1.1) the gate rotates. X is the negation and acts on $|0\rangle$ and $|1\rangle$ like the classical NOT operation [Sca12]. Z changes the relative phase of a qubit in superposition in the standard basis and Y is a combination of the negation and a phase change. Sometimes Y is also defined as $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ [Sca12]. The two definitions are equivalent up to a global phase factor of $-i$.

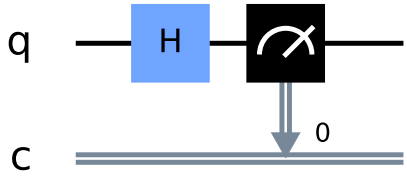
Furthermore, the following gates are used frequently [NC02]:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}; \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}; \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix} \tag{2.10}$$

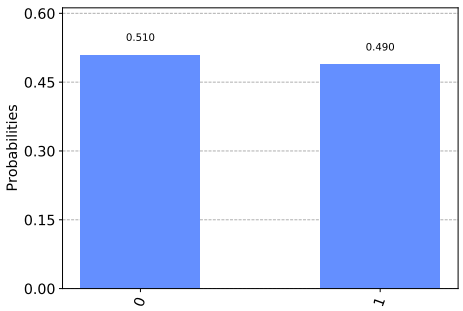
The phase gate (S) is T^2. T is also called $\pi/8$ gate due to historical reasons [NC02]. The Hadamard gate (H) is equal to $(X + Z)/\sqrt{2}$ and can be used to transform a qubit with state $|0\rangle$ or $|1\rangle$ to an uniform superposition [Sca12]:

$$H(|0\rangle) = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \tag{2.11}$$

The computational basis states $|0\rangle$ and $|1\rangle$ are both measured with probability $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$. Figure 2.2 shows a quantum circuit with one Hadamard gate and measurement operation that measures in the computational basis. The graph shows the measurement results which are obtained by executing the circuit several times.



(a) Quantum circuit with a Hadamard gate and a measurement operation



(b) Results of the repeated execution of the quantum circuit

Figure 2.2: Quantum circuit with a single Hadamard gate to achieve superposition

The exponentiation of the Pauli gates creates the *rotation operators* $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$ that are used to rotate around an arbitrary angular [NC02]:

$$R_x(\theta) \equiv e^{-i\theta X/2} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}X = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{2.12}$$

$$R_y(\theta) \equiv e^{-i\theta Y/2} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}Y = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{2.13}$$

$$R_z(\theta) \equiv e^{-i\theta Z/2} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}Z = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{2.14}$$

The observation that every single-qubit state can be represented on the Bloch sphere and the rotation gates can be used to rotate around an arbitrary angle shows that every single-qubit gate can be implemented with a combination of the three rotation gates [NC02]. $U(\theta, \phi, \lambda)$ is an elementary single-qubit operation allowing to rotate around arbitrary angles and each axis of the Bloch sphere [ZPW18]. The Euler decomposition allows implementing this operation as the sequence $R_Z(\phi)R_Y(\theta)R_Z(\lambda)$ consisting of only R_Z and R_Y rotations [ZPW18]. Other decompositions are possible as well.

Multi-qubit Gates

In quantum computing, the typical multi-qubit gate is the controlled-NOT (CNOT) gate [NC02]. This gate has two inputs (see Figure 2.3). One of the inputs is considered as the *control* (the top line in the sample circuit) and one the *target* qubit (the bottom line in the sample circuit) [NC02]. The control qubit decides if the target qubit is flipped, i.e. if the control qubit is $|1\rangle$, the target qubit is flipped. In the computational basis the matrix representation is

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.15}$$

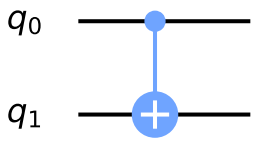


Figure 2.3: CNOT gate applied to two qubits

assuming that the qubits are in the sequence $|control, target\rangle$ [NC02]. The matrix is unitary, otherwise, CNOT would not be a valid quantum computing gate, and the matrix is its own inverse [Sca12]. Another important consequence is that the CNOT gate can change the entanglement of two qubits, e.g. the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$ is separable, but the state

$$\begin{aligned} CNOT(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle) &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ &= |\phi^+\rangle \end{aligned} \tag{2.16}$$

is entangled [Sca12]. Similarly, the gate can transform an entangled state to a separable state because the operation is its own inverse.

Instead of applying the NOT operation if the control qubit is set to $|1\rangle$, one could also apply an arbitrary unitary single-qubit gate U. The 4×4 matrix representation [Sca12] of U^C, the controlled version of U, in the computational basis is

$$U^C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \multicolumn{2}{c}{\multirow{2}{*}{U}} \\ 0 & 0 & & \end{bmatrix} \tag{2.17}$$

One common controlled gate beside the CNOT gate is the *controlled-phase (CZ)* gate.

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{2.18}$$

Furthermore, the *SWAP* gate plays an important role in this context. This gate swaps the states of two qubits [Sca12]:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.19}$$

An important three qubit gate is the *Toffoli* operator that has two control and one target qubit [NC02]:

$$Toffoli = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.20}$$

If both control qubits are set to $|1\rangle$, the target qubit is flipped. Therefore, the classical NAND gate can be simulated with the Toffoli gate. In this case, only the target gate is used for further computation. This shows that a quantum computer can also execute arbitrary classical circuits [NC02].

Universal Operators

In classical computing the NAND gate is a *universal* gate, i.e., any computable function can be achieved by a sequence of NAND gates [NC02]. However, this gate is irreversible, i.e. it is not possible to determine the inputs from the output. In quantum computing, every gate must be invertible because the inverse of a unitary operation is also unitary (see Section 2.1.2). Like the NAND gate that is universal regarding classical computing, there also exist universal gates for quantum computing. In quantum computing, a set of gates is universal if any unitary gate can be approximated to arbitrary accuracy by a sequence of these gates [NC02].

Unitary matrices that act on two-or-fewer qubits can be used to construct unitary matrices acting on arbitrary dimensions [NC02]. Additionally, single-qubit gates combined with, e.g., the CNOT gate can be used to implement an arbitrary two-qubit unitary operation. Combined these results show that any gate can be implemented with CNOT and single-qubit gates [NC02]. To achieve a single-qubit rotation around an arbitrary angle a combination of Hadamard, phase, and $\pi/8$ gates can be used [NC02]. Dawson and Nielsen [DN05] show that they can approximate a unitary operation U consisting of m CNOT or single-qubit operators with $O(m \cdot \log^c(m/e))$ gates from the set $\{H, S, T, CNOT\}$ where $\epsilon > 0$ is the accuracy and c is a constant.

2.1.3 Equivalence of States and Gates

Two states ψ and χ are equivalent up to a global phase if $|\psi\rangle = e^{i\theta}|\chi\rangle$ for a real number θ and state vectors $|\psi\rangle$ and $|\chi\rangle$ [NC02]. Interestingly, such states cannot be distinguished by measurement. The global phase factor, in this case, is $e^{i\theta}$. In contrast to this, they differ by a relative phase if for their amplitudes a and b it holds true that $a = e^{i\theta}b$ [NC02]. As opposed to the global phase the relative phase is a basis-dependent concept and can, therefore, lead to different measurement results. Global phase and relative phase are commonly used terms regarding quantum computing and have several implications on the development of a quantum transpiler.

Quantum states that are the same up to a global phase factor cannot be distinguished by measurement. Therefore, two gates that manipulate a qubit ψ in a way that it is mapped to two equivalent states are exchangeable with respect to the measuring of the quantum circuit. In the following, this is exemplarily shown with the gates RZ and $U1$.

$$U1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} \equiv e^{-i\frac{\lambda}{2}} \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} = \begin{pmatrix} e^{-i\frac{\lambda}{2}} & 0 \\ 0 & e^{i\frac{\lambda}{2}} \end{pmatrix} = RZ(\lambda) \tag{2.21}$$

Equation (2.21) shows the equivalence of the matrices up to a global phase factor of $e^{-i\frac{\lambda}{2}}$. However, there are applications of gates that are equivalent up to their global phase, where it is important to distinguish between the gates, especially in the context of a quantum SDK that allows the conversion between various quantum circuit representations. The following example shows the necessity of distinguishing equivalent gates. Several SDKs offer the possibility to implement a controlled version of the standard gates, e.g. Qiskit and Forest. The controlled versions *CRZ* and *CU1* of the equivalent gates RZ and U1 are not equivalent and do have a relative phase difference (see Equation (2.22) where the first qubit acts as the control qubit).

$$\begin{aligned} CU1(\lambda) &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\lambda} \end{pmatrix} \equiv e^{-i\frac{\lambda}{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\lambda} \end{pmatrix} \\ &= \begin{pmatrix} e^{-i\frac{\lambda}{2}} & 0 & 0 & 0 \\ 0 & e^{-i\frac{\lambda}{2}} & 0 & 0 \\ 0 & 0 & e^{-i\frac{\lambda}{2}} & 0 \\ 0 & 0 & 0 & e^{i\frac{\lambda}{2}} \end{pmatrix} \neq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{-i\frac{\lambda}{2}} & 0 \\ 0 & 0 & 0 & e^{i\frac{\lambda}{2}} \end{pmatrix} = CRZ(\lambda) \end{aligned} \tag{2.22}$$

2.1.4 Quantum Circuits and Algorithms

In the following, the properties of a quantum circuit are explained. Afterwards, two prominent quantum algorithms are discussed that are based on an oracle function. These algorithms make use of the introduced quantum concepts like superposition and quantum entanglement to achieve superior results compared to classical computing.

Quantum Circuit

The sequence of gates and measurements acting on specific qubits is a quantum circuit and can be graphically represented [Sca12]. A quantum circuit is a gate-based representation of a quantum algorithm implementation [SBB+20]. The lines in the graphical representation correspond to so-called *wires* which carry signals to different points in the quantum circuit [NC02; ZPW18]. Concretely, the wires correspond in most cases to the qubits. However, the graphical representation can be misleading in some cases, because it does not follow the same rules as the representation of classical circuits. One of the main differences is that the effect that a quantum circuit has on a qubit cannot be determined by only analyzing the line corresponding to the qubit, but the result must be explicitly calculated [Sca12].

Gates that are applied concurrently in a quantum circuit form so-called layers or levels [ZPW18]. Therefore, each layer only consists of gates that act on a distinct set of qubits [ZPW18]. The number of layers of a quantum circuit is equal to the depth of the circuit. Computing the layers can be done greedily by simply moving all gates in the circuit as far to the left as possible without changing the sequence of gates that act on the same qubit [ZPW18]. Another important property of a quantum circuit is the width describing the number of qubits that are manipulated by the quantum circuit [LB20].

Quantum circuits can be described by quantum assembly languages [CBSG17]. These assembly languages are used to provide instructions to the QPUs. They describe which operations should be performed on which qubits [LaR19]. Furthermore, measurement operations can be applied to the quantum circuits that write the measurement outcome into classical bits.

Deutsch's Algorithm

Deutsch's algorithm was the first algorithm to show the possibility of quantum computation to outperform classical computation [Sca12]. The problem that is solved by the algorithm is to determine whether a boolean function $f : Z_2 \rightarrow Z_2$ is a constant function [DJ92; Sca12]. While any classical algorithm requires at least two calls to the function, one function call is sufficient for Deutsch's quantum algorithm [Sca12]. The idea is that a qubit is first placed in superposition and, afterwards, the function f is called as an *oracle function* on the prepared qubits [Sca12]. An oracle function is a black box function implemented as a unitary operator that operates on a qubit register $|x\rangle$ and a single-qubit $|q\rangle$ [NC02]:

$$|x\rangle|q\rangle \rightarrow |x\rangle|q \oplus f(x)\rangle \tag{2.23}$$

where $\oplus$ is the addition modulo 2. If $f(x) = 1$, the oracle qubit $|q\rangle$ is flipped. Otherwise, it is unchanged [NC02]. Based on this, it can be easily checked whether $f(x) = 1$ by preparing $|q\rangle = 0$ and measuring the qubit after the oracle function was executed. Deutsch's Algorithm uses the oracle function as follows:

$$\begin{aligned} U_f\,(|+\rangle|-\rangle) &= U_f\,(\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)) \\ &= \frac{1}{2}(|0\rangle(|0 \oplus f(0)\rangle - |1 \oplus f(|1\rangle) + |1\rangle(|0 \oplus f(0)\rangle - |1 \oplus f(|1\rangle)) \\ &= \frac{1}{2}\sum_{x=0}^{1}|x\rangle(|0 \oplus f(x)\rangle - |1 \oplus f(|x\rangle)) \\ &= \frac{1}{\sqrt{2}}\sum_{x=0}^{1}(-1)^{f\,(x)}|x\rangle|-\rangle \end{aligned} \tag{2.24}$$

If f is constant, the state is $|+\rangle|-\rangle$ up to a global phase while if f is not constant, the state is $|-\rangle|-\rangle$ [Sca12]. Afterwards, a Hadamard gate is applied to the first qubit and the measurement of this qubit can then determine, with certainty, if the function is constant or not [Sca12]. The measurement obtains $|0\rangle$ or $|1\rangle$ respectively. Interestingly, even though the algorithm makes use of quantum processes like quantum entanglement and superposition, the execution of the algorithm is deterministic.

Grover Algorithm

The Grover algorithm is used to search in an unsorted database for a particular entry [Gro96]. The problem statement is that only one entry in an unsorted database, which contains N records, satisfies a specific property and the corresponding record should be identified. A classical algorithm trivially needs $O(N)$ steps, because on average the algorithm needs to analyze a large fraction of the elements [Gro96]. However, the Grover algorithm based on quantum computation only needs $O(\sqrt{(N)})$ steps to find the record. Grover [Gro96] even shows that the approach is within a constant factor of the fastest algorithm that is based on quantum mechanics and solves this specific problem. Compared to other problem statements where quantum computing enables an exponential speedup, the Grover algorithm only offers a quadratic speedup [NC02]. Nevertheless, the algorithm is of great interest because it can be used in a wide range of practical applications such as searching a database or calculating the mean and median of statistical distributions [Gro98; NC02].

Grover works by realizing a superposition such that every entry of the database is encoded in the superposition of the qubits [Gro96]. Afterwards, an oracle function is used to identify the record fulfilling the specific property [Sca12]. The oracle function negates the amplitude of this element. However, this could not be determined by measurement. Therefore, all amplitudes are reflected at the midpoint which amplifies the amplitude of the identified record [Sca12]. This process is iterated until the difference in the amplitudes is high enough such that the marked entry is measured with a high probability.

2.1.5 NISQ

The currently existing QPUs are rather limited in their functionality considering the low number of qubits and the high noise in the quantum gates. Because of these limitations, they are also called Noisy Intermediate-Scale Quantum (NISQ) devices [Pre18]. The noise limits the practicability of QPUs and negatively impacts what QPUs can achieve in the near future. However, these QPUs consisting of 50-100 qubits can be used to explore several useful applications and provide decisive advantages compared to classical machines [Pre18]. On top of this, they are a significant step towards more powerful quantum computers in the future. In this section, the physical limitations of currently existing QPUs are introduced. Furthermore, the tasks of a transpiler are outlined and quantum simulators are introduced.

Native Gates

QPUs have a set of native gates. These are elementary operations that can be physically executed on the QPU. The gates used in a quantum circuit must be decomposed to a subroutine of native gates [ZPW18]. For this purpose, the native gate set must be universal. Decomposing gates will - in most cases - considerably increase the number of operations in the circuit and, therefore, the depth of the circuit [LB20].

In the context of native gates, the *virtual Z gate* is an important concept with a major effect on the depth of quantum circuits, the error rate, as well as, the comparability of different QPU vendors. This approach considers the physical properties of a quantum computer. Physically, X and Y gates are computed by modulating the coupling of the states $|0\rangle$ and $|1\rangle$ [MWS+17]. The modulation

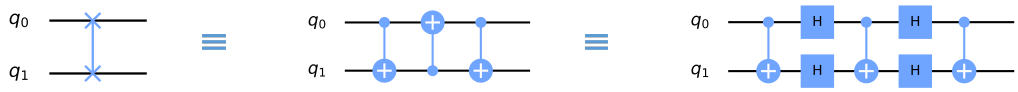


Figure 2.4: SWAP implementation with CNOT gates [ZPW18]

drive can be described by the formula $\Omega(t)\cos(w_D t - \gamma)$ where $\Omega(t)$ is the drive strength of the rotation around the axes, w_D is the frequency, and γ the phase of the drive [MWS+17]. Depending on the drive phase γ the qubit state is either rotated around the X-axis or the Y-axis. These rotations are implemented by microwave pulses. The phase difference between these axes in the Bloch sphere is $\pi/2$. In contrast to this, Z gates correspond to a modification in the relative phase of the states $|0\rangle$ and $|1\rangle$ [MWS+17]. Z-Gates can be physically implemented by detuning the frequency with respect to the drive field or by combining X and Y gates. Another way to implement it is by adding a phase offset to the drive for all following X and Y gates in the control software of the QPU [MWS+17]. This results in a rotation of the axes concerning the qubit state. Implementing a Z gate in this way is also called a virtual Z gate and leads to a perfect gate. The gate has a gate time of zero, nearly no error, because the classical control hardware is self-calibrated, and can even be used to compensate for some errors in the X and Y gates [MWS+17]. Because of the implementation in the control software by adding phase offsets, the execution of a virtual Z gate is called *frame change*.

The virtual Z gate is only applicable to quantum computers that are based on *superconducting* qubits. Qubits can be physically realized by different technologies with the superconducting qubits being based on electronic circuits that are described by condensate wave functions [GZB+04]. The QPUs from IBM and Rigetti are based on this technology [PAFL18].

Qubit Connectivity

In quantum computing, especially in the context of a quantum transpiler, the differentiation between *physical* and *logical* qubits is important. Physical qubits refer to the qubits of a QPU which are limited by the physical properties of the QPU [ZPW18]. In contrast to this, logical qubits refer to the qubits of a quantum circuit. The QPUs differ in the connectivity of their physical qubits. Multiple qubit operations can only be executed on specific physical qubits [ZPW18], i.e. a quantum gate cannot be applied to arbitrary qubits, but only to qubits that are adjacent on the chip. The topology is described by coupling maps and differs from QPU to QPU. Furthermore, the direction - which qubit must be the control and which the target qubit - can be limited by the coupling map [ZPW18]. This has consequences for the CZ gate (see Section 2.1.2) which is, e.g., used by the QPUs of Rigetti[1]. The CZ gate is symmetric and the order of the input qubits is irrelevant. As a result of this, the direction constraint does not effect this gate.

In usual settings, it is not possible to determine one mapping that satisfies all topology constraints throughout the whole circuit [ZPW18]. Therefore, the solution satisfying the constraints must be constructed in a way that the mapping from logical to physical qubits changes over time, i.e. for each layer of the quantum circuit a valid mapping is computed and *SWAP* gates are inserted into the

[1] https://pyquil-docs.rigetti.com/en/stable/basics.html

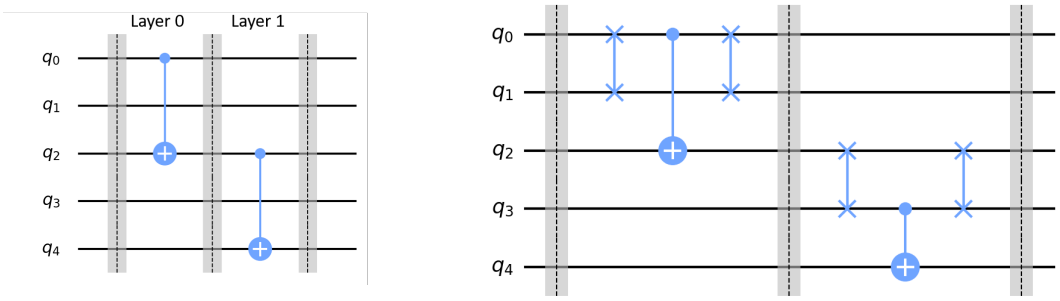


Figure 2.5: Circuit with SWAP gates to satisfy the qubit connectivity

quantum circuit to change the mapping from logical to physical qubits for each layer. This rewiring process is also called topology mapping. The topology mapping should be as optimal as possible because each rewiring increases the depth of the circuit by several gates, e.g. the implementation of the SWAP operation with a CNOT and single-qubit gates requires three gates if the connection between the gates is bilateral and otherwise seven gates (see Figure 2.4) [ZPW18].

Figure 2.5 shows a circuit with two layers. If we assume that each qubit is only connected to the next qubit on the QPU that should execute the quantum circuit, we cannot directly execute the given circuit. To make the quantum circuit executable, SWAP gates could be inserted to change the mapping of the logical to the physical qubits, e.g. in the first layer the logical qubit q_0 could be mapped to the physical qubit q_1 by the addition of SWAP gates and afterwards the CNOT could be applied to the physical qubits q_1 and q_2 which denote the logical qubits q_0 and q_2. By the addition of the SWAP gates also the number of qubits used in the quantum circuit, i.e. the width of the circuit, increases. Alternatively, in this simple example, a suitable initial layout could have been selected that satisfies the qubit connectivity constraints for both CNOT gates without adding additional SWAP gates.

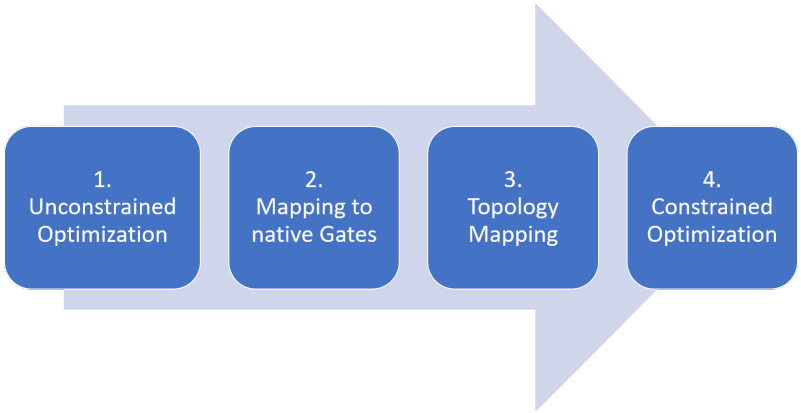


Figure 2.6: Transpilation process

Quantum Transpiler

The computation of an equivalent quantum circuit that satisfies the hardware constraints is done by a transpiler. Figure 2.6 shows the general transpilation steps that are applied to the quantum circuit to make it executable on a specific QPU[CBSG17]. The first and the last step are only optimizing steps, i.e. they are not required for the satisfaction of the hardware constraints:

1. Firstly, the quantum circuit is optimized without any constraints and gates can be combined arbitrarily as long as the semantics of the quantum circuit do not change. The depth of a circuit has a high impact on the error rate on NISQ machines. Therefore, the optimization steps are important for the practicability of the quantum circuit.
2. Afterwards, the gates that are used in the quantum circuit are mapped to the native gates of the desired architecture.
3. For each layer (see Section 2.1.4) of the quantum circuit the qubits must be rewired such that it is ensured that all gates of the layer that affect multiple qubits are feasible regarding the qubit connectivity of the qubits that the gates operates on. The initial layout should be chosen such that the rewiring needed for the subsequent layers is as minimal as possible.
4. Lastly, the computed quantum circuit is optimized again. In contrast to the first point, this optimization step must consider the hardware constraints, i.e., the applied operations cannot be swapped around arbitrarily and no non-native gates may be added because otherwise the qubit connectivity or the gate constraints would be violated. The main optimization possibility in this step is the single-qubit optimization where consecutive single-qubit gates are composited under certain circumstances [CBSG17].

Finding the global optimum of arbitrary quantum circuits is QMA-hard [JWB03; NRS+18]. QMA is the quantum analogue to the probabilistic version of NP [KR03]. Therefore, the techniques used in the transpiler apply heuristics to reduce the number of gates and the depth of the quantum circuit. It is important to note that the mapping to native gates must be done before applying the topology mapping. This is essential because during the native gate mapping new gates can be appended to the quantum circuit that operates on qubits that are not necessarily connected on the QPU.

Simulator

Besides the execution on QPUs, quantum circuits can also be simulated on classical hardware [NMI02]. Quantum simulators can be used to examine the properties of quantum circuits. This is especially useful when implementing and testing quantum algorithm implementations because the execution of quantum circuits on QPUs can be expensive. Some simulator implementations even support noise models[2]. However, as a result of quantum computers having an exponential performance advantage solving specific problems [AL99], running practical quantum circuits with more than 30 qubits on a simulator is infeasible [NMI02].

[2] https://qiskit.org/documentation/tutorials/simulators/2_device_noise_simulation.html

2.2 Problem Statement

Several vendors like IBM and Rigetti provide QPUs that are used to execute implementations of quantum algorithms. Even though the QPUs of IBM and Rigetti follow the same quantum computing model, the gate-based model [NC02], the architectures of the QPUs differ in several aspects like the number of qubits, the qubit connectivity, the gates that are natively supported, as well as, the decoherence and error rates. The susceptibility to errors, the limitations of the number of qubits of a QPU, and the significant increase of the depth of a quantum circuit in the transpilation process make the choice of a QPU difficult. Circuits with a high depth are either highly error-prone or the number of qubits must be very limited which limits the performance advantages compared to classical computing [LB20]. The issue with the depth of a quantum circuit is that depending on which QPU is used, the depth varies significantly. This results from the transpilation process where the gates used in the algorithm implementation are unrolled to the native gates of the QPU and *SWAP* gates are added to the implementation to respect the qubit connectivity of the chip. The problem is that the number of qubits of currently existing quantum computers in the NISQ era is rather small and the error rates are quite high [SBB+20].

Furthermore, many quantum SDKs only allow the execution of quantum circuits on a specific subset of QPUs [LaR19]. The usage of proprietary SDKs leads to a tight coupling between a QPU and a quantum algorithm implementation restricting the QPUs that can be chosen for the execution of the quantum circuit. This SDK lock-in further complicates the choice of a suitable QPU. As a consequence, the choice of an appropriate SDK requires knowledge regarding the mathematics of the implemented quantum algorithm, as well as, the technicalities of the used SDK and QPU [SBB+20]. This thesis aims at removing the SDK lock-in by introducing a quantum analysis and transpilation framework that integrates different SDKs and allows the transpilation to various QPUs from different vendors.

3 Related Work

Various papers are related to the transpilation process of quantum circuits. The topics of the papers can be divided into the mapping from physical to logical qubits, the decomposition of quantum gates to the native gate set of a QPU, the optimization of quantum circuits regarding different metrics, and the development of transpilers in general.

Salm et al. [SBB+20] introduce a NISQ Analyzer for automated analysis and selection of quantum algorithm implementations and hardware. For their first approach of the NISQ Analyzer, they use the vendor-specific transpilers to gain information about the depth and the width of considered implementations which highly depend on the chosen hardware. This thesis aims at removing the dependence on proprietary transpilers. The NISQ Analyzer is part of the project PlanQK[1] that is an open platform for quantum supported artificial intelligence (AI).

Zulehner et al. [ZPW18] describe an algorithm to map quantum circuits to the IBM QPUs regarding the qubit connectivity. For this purpose, SWAP operations are used to enable two-qubit gates on arbitrary logical qubits. For the computation of a mapping from logical to physical qubits which allow the execution of the specified gates, they generate a state-space that is composed of qubit permutations. Their algorithm can be extended to be applicable for other QPUs as well. Another approach regarding qubit connectivity is described by Nash et al. [NGM20]. Their approach is based on a reduction to a Steiner tree. However, they only consider quantum circuits consisting of the gate set $\{CNOT, R_Z\}$.

Several papers tackle the decomposition of common quantum gates to specific gate sets like the Clifford+T set $\{Hadamard, Phase, T, CNOT\}$ [AMMR13; BBC+95]. Matsumoto and Amano [MA08] define a normal form for one qubit quantum circuits over the basis of $\{Hadamard, Phase, T\}$. Every circuit defined over this basis can be transformed into their proposed normal form. The normal form is unique and T-depth optimal which means the number of required T gates is minimal. [MWS+17] propose the implementation of rotations about the Z-axis in the control software of the QPU for superconducting qubits. The main advantage of the virtual implementation of these gates is that rotations about arbitrary angles are possible [MWS+17]. Furthermore, the gate time has zero duration and they can be used to compensate for certain unitary errors. With only virtual Z gates and $R_X(\frac{\pi}{2})$ gates, any arbitrary rotation in the Bloch sphere can be implemented [MWS+17]. Bullock and Markov [BM03] introduce an algorithm to implement an arbitrary two-qubit gate with only CNOT and single-qubit gates. Their algorithm is based on the KAK decomposition which is the decomposition of a unitary matrix to two rotation matrices and a diagonal matrix [BM03]. Compared to this, the decomposition algorithm Iten et al. [ICK+16] support arbitrary isometries as input, including any unitary gate of arbitrary size. Like Bullock and Markov [BM03] the target set for their algorithm is a sequence of CNOT gates combined with single-qubit gates.

[1] https://planqk.de/

[SDC+20] describe their retargetable compiler $t|ket\rangle$. From various quantum frameworks, quantum circuits can be imported and executed on QPUs from several vendors. The focus of $t|ket\rangle$ are NISQ devices, i.e. they include the device errors of specific operations in their computations. $t|ket\rangle$ implements the mapping to physical qubits, as well as, the decomposition of quantum gates to native gate sets of specific QPUs. Additionally, they support optimizations regarding the number of two-qubit gates used in the transpiled circuit, e.g. by analyzing subcircuits with the KAK decomposition [SDC+20]. In contrast to their approach, the proposed quantum circuit analysis and transpilation framework focuses on the integration of different SDKs and the unrolling step to native gates. $t|ket\rangle$ only supports the import of quantum circuits with rudimentary operations from quantum SDKs like the Forest SDK.

Bergholm et al. [BIS+18] introduce a quantum SDK with a quantum transpiler and the support of several SDKs and QPUs. Their SDK PennyLane focuses on hybrid quantum-classical machine learning algorithms. They provide plugins for several quantum SDKs. Furthermore, they integrate machine learning libraries like TensorFlow and PyTorch into their classical computation parts. Contrary to their focus on the hybrid algorithms, this thesis focuses on the transpilation process of arbitrary quantum circuits.

4 Analysis of Quantum SDKs and QPUs

In this chapter, several quantum SDKs and QPUs are introduced. The QPUs from Rigetti and IBM are discussed regarding their native gate set and the qubit connectivity. Furthermore, the quantum SDKs Qiskit[1], Forest SDK[2], Cirq[3], Strawberry Fields[4], QDK[5], Pytket[6], Pennylane[7], PyZX[8], and Quantastica[9] are briefly explained and a selection of these SDKs is analyzed regarding their standard gate sets.

4.1 Current QPUs

Several vendors offer the execution of quantum circuits on their QPUs via Cloud services, e.g. IBM quantum experience (IBM QX)[10], Rigetti[11], AWS Bracket[12], and Azure Quantum[13] provide Cloud offerings related to quantum computing. In this thesis, the focus is on the QPUs from IBM and Rigetti because they are publicly available.

In this section, the QPUs of these vendors and their hardware constraints will be explained. The measurement is in both approaches done by a projection onto the Z-basis, i.e. the computational basis $|0\rangle$ or $|1\rangle$ is measured [CBSG17].

[1] https://qiskit.org/documentation/
[2] https://pyquil-docs.rigetti.com/en/stable/
[3] https://cirq.readthedocs.io/en/stable/
[4] https://strawberryfields.readthedocs.io/en/stable/
[5] https://docs.microsoft.com/en-us/quantum/
[6] https://cqcl.github.io/pytket/build/html/index.html
[7] https://pennylane.readthedocs.io/en/stable/introduction/pennylane.html
[8] https://pyzx.readthedocs.io/en/latest/
[9] https://quantastica.com/
[10] https://quantum-computing.ibm.com/
[11] https://www.rigetti.com/
[12] https://aws.amazon.com/en/braket/
[13] https://azure.microsoft.com/en-us/services/quantum/

4.1.1 IBM

IBM QX offers roughly 20 quantum systems that are accessible through IBM's Cloud offerings. Physically the QPUs are based on superconducting qubit technology. The following gates are natively implemented:

$$U_1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} (\equiv R_Z(\lambda) \text{ up to a global phase factor of } e^{-i\frac{\lambda}{2}}) \tag{4.1}$$

$$U_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i\lambda+i\phi} \end{bmatrix} (= R_z(\phi)R_y(\frac{\pi}{2})R_z(\lambda)) \tag{4.2}$$

$$U_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}\cos(\frac{\theta}{2}) \end{bmatrix} (= R_z(\phi)R_x(-\pi/2)R_z(\theta)R_x(\pi/2)R_z(\lambda)) \tag{4.3}$$

Furthermore, the CNOT gate between specific qubits depending on the particular quantum computer topology is natively supported.

U_1 is a rotation about the Z-axis, U_2 rotates about the X and Z axis and U_3 is a generic rotation with three Euler angles. U_1 and U_2 are specific cases of U_3 with $\theta = \phi = 0$ for U_1 and $\theta = \frac{\pi}{2}$ for U_2[14]. As a result, one might question the usefulness of U_1 and U_2 as basic gates. Compared to the U_3 gate, they have advantages with respect to their error rates and gate times. U_1 is implemented as a virtual Z gate (see Section 2.1.5) and is, therefore, just a frame change leading to a gate time of zero and an almost nonexistent error rate [MWS+17]. Based on this observation U_2 and U_3 can be implemented with physical gates combined with frame changes. U_2 consists of one frame change and one R_Y gate (see Equation (4.2)) while U_3 is constructed using two R_X gates and three frame changes (see Equation (4.3)).

The topology of the QPUs is different from QPU to QPU. Figure 4.1 exemplarily shows the coupling map of the Rochester and the Böblingen QPU.

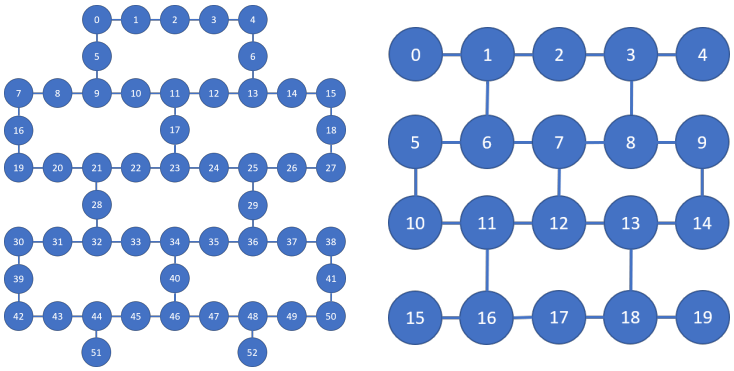


Figure 4.1: IBM QX Rochester and Böblingen coupling maps[15]

[14] https://qiskit.org/textbook/ch-states/single-qubit-gates.html
[15] https://www.ibm.com/blogs/research/2019/09/quantum-computation-center/

4.1.2 Rigetti

Rigetti offers four quantum systems that are based on superconducting quantum technology. *Aspen-8*, Rigetti's largest architecture, consists of 31 qubits. The native gates for Rigetti QPUs are

- $R_X(\theta)$ gate with angle $\theta = \pm\pi$ or $\pm\frac{\pi}{2}$
- $R_Z(\lambda)$ with an arbitrary angle
- CZ between adjacent qubits

R_Z is implemented as a virtual Z gate by a frame change in the control software just like the IBM QX gates U_1, U_2, and U_3. Similarly, to the construction of IBM's U_3 gate, the R_Z gate can be used in combination with R_X gates to enable rotations around all axes and arbitrary angles.

Figure 4.2 shows the qubit topology of the Aspen-8 architecture.

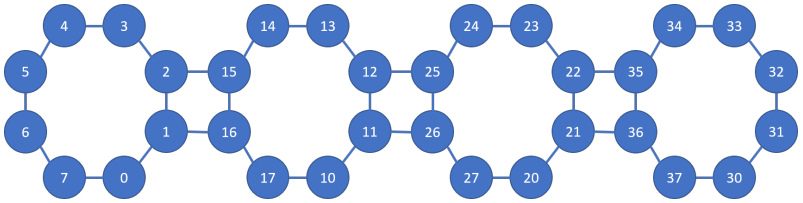


Figure 4.2: Rigetti Aspen-8 32 qubits coupling map[16]

4.2 Quantum SDKs

Quantum circuits can be programmed in several quantum SDKs. The core functionality of most of the SDKs is the possibility to create quantum circuits and transpile them for a specific QPU such that the implementation can be executed on the QPU. Furthermore, the optimization and the execution of quantum circuits on simulators or QPUs are supported by many SDKs.

4.2.1 Proprietary SDKs

In the following, quantum SDKs are introduced that are designed to operate independently from other SDKs. Most of them support the QPU of a specific vendor.

[16] https://docs.rigetti.com/en/welcome-to-quantum-cloud-services

Algorithm 4.1 OpenQASM description of a quantum circuit

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[1];
h q[0];
cx q[0],q[1];
measure q[1] -> c[0];
```

Qiskit

Qiskit (Quantum Information Software Kit) is a quantum SDK founded by IBM. The used instruction language is *OpenQASM*[17]. The syntax is based on Assembler and C [CBSG17]. OpenQASM is supported by many quantum computing-related tools like the quantum processing toolkit *staq* [AG20], the quantum circuit optimizer *PyZX* [KW19], or the NISQ compiler *t|ket⟩* [SDC+20]. Algorithm 4.1 shows a quantum circuit implemented in OpenQASM. Firstly, the version is specified and the imports are declared. Afterwards, quantum and classical registers are defined. The first qubit is put into superposition by applying a Hadamard gate. Lastly, a CNOT gate with the first qubit as control and the second as target qubit is applied followed by a measurement of the second qubit into the first classical bit.

The main version of Qiskit uses Python for the implementation, manipulation, and execution of quantum circuits. Qiskit provides functionality to compose, optimize, visualize and transpile quantum circuits for IBM QX QPUs. Qiskit, e.g., offers the possibility to convert a quantum circuit that is constructed by arbitrary operations to an instruction that can be used as a gate in another circuit. This allows the definition of *subcircuits* that can be applied to specific qubits or classical bits in another circuit.

Forest SDK

The *Forest SDK* is the quantum SDK developed by Rigetti. Forest uses the instruction language *Quil*[18]. Algorithm 4.2 shows an example of a quantum circuit implemented in Quil. The semantic of the implementation is the same as in Algorithm 4.1. Forest contains the python library *pyQuil*[19] that is used for writing and running quantum circuits on Rigetti's QPUs [CBSG17]. The Quil compiler (quilc) can be used to transpile Quil code for the execution on Rigetti's QPUs.

[17] https://github.com/Qiskit/openqasm/blob/master/spec/qasm2.rst
[18] https://github.com/rigetti/quil/blob/master/spec/Quil.md
[19] http://docs.rigetti.com/en/stable/

Algorithm 4.2 Quil description of a quantum circuit

```
DECLARE ro BIT[1]
H 0
CNOT 0 1
MEASURE 1 ro[0]
```

Cirq

Cirq is the SDK developed by Google. The QPUs of Google are not publicly available. Therefore, Cirq does not offer execution on real quantum computers, yet. However, the implementations can be run on simulators [Sin19]. Cirq offers methods to write, manipulate, and optimize quantum circuits.

Strawberry Fields

Strawberry Fields is a quantum SDK for photonic quantum computers developed by Xanadu. The key difference between photonic quantum computers to the quantum computers discussed so far is that the states reside in an infinite-dimensional Hilbert space, i.e. the quantum operators are continuous [KIQ+19]. As a result, the model of photonic quantum computers is called continuous-variable model [KIQ+19]. The basic element of the CV model is a *Qumode* which is defined as $|\psi\rangle = \int dx\ \psi(x)\ |x\rangle$ compared to the qubit $|\phi\rangle = \phi_0|0\rangle + \phi_1|1\rangle$ (see 2.1.1). Therefore, the gates and measurements that are used also differ from the concepts that we have seen so far (see Section 2.1.2) [KIQ+19]. As a result of the different computation models, a conversion between Strawberry Fields and the other introduced SDKs is not possible and the properties of the SDK do not have to be considered. The language used in Strawberry Fields is *Blackbird*[20].

QDK

The *quantum development kit (QDK)* is Microsoft's SDK. The primary feature is the high-level quantum-focused programming language Q# that is derived from C#. Additionally, the QDK consists of a code optimizer, simulators, and quantum algorithm libraries. Microsoft aims to use *topological qubits* in its quantum computers [LaR19]. Topological qubits use *anyons* as their physical unit to store and manipulate quantum information [LP17]. These anyons are quasiparticles with properties that are advantageous regarding the errors in the quantum computing process. Currently, QPUs based on topological qubits do not exist. Therefore, the algorithm implementations can only be simulated. Nevertheless, it is important to consider this SDK because quantum computers consisting of topological qubits implement an error-correction at the hardware level which improves the practicability of quantum computation [LP17].

[20] https://strawberryfields.readthedocs.io/en/stable/tutorials/blackbird.html

Standard Gate Sets

The quantum SDKs support a fixed set of quantum operations. Table 4.1 shows a comparison of the standard gate sets of the SDKs Qiskit, Forest, and Cirq. In contrast to native gates, standard gates do not refer to the set of gates that can be physically run on a QPU but to the set of gates that is supported by a specific SDK. The comparison of Qiskit and Forest is substantial because they provide access to the QPUs of IBM and Rigetti. Additionally, Cirq is included here because it is one of the main SDKs available and might support Google's QPUs based on superconducting qubits in the future[21]. The table lists the name of the gate in brackets if the name of the gate in the respective SDK differs from the name specified in the first column. The entries that are marked with a * do not exist as separate gates but can be implemented with *gate modifiers*. These modifiers are constructs implemented in the SDKs that allow the adjustment of the standard gates, e.g. getting the controlled version of a gate or the inverse.

Gate	Qiskit	Forest	Cirq
I	✓	✓	✓
X	✓	✓	✓
Y	✓	✓	✓
Z	✓	✓	✓
H	✓	✓	✓
S	✓	✓	✓
T	✓	✓	✓
RX	✓	✓	✓
RY	✓	✓	✓
RZ	✓	✓	✓
U1	✓	✓(Phase)	✗
U2	✓	✗	✗
U3	✓	✗	✗
Sdg (S-adjoint)	✓	✓(RZ($-\pi/2$))	✓(RZ($-\pi/2$))
Tdg (T-adjoint)	✓	✓(RZ($-\pi/4$))	✓(RZ($-\pi/4$))
CNOT	✓(CX)	✓	✓
CZ	✓	✓	✓
CY	✓	✓*	✓*
CRX	✓	✓*	✓*
CRY	✓	✓*	✓*
CRZ	✓	✓*	✓*
Toffoli	✓(CCX)	✓(CCNot)	✓(CCX)
CCZ	✓*	✓*	✓
C3X	✓	✓*	✓*
C4X	✓	✓*	✓*
CU1	✓	✓(CPHASE)	✗

[21] https://research.google/teams/applied-science/quantum/

CU3	✓	✗	✗
CPhase00	✗	✓	✗
CPhase01	✗	✓	✗
CPhase10	✗	✓	✗
Swap	✓	✓	✓
CSwap	✓	✓	✓
iSwap	✓	✓	✓
PSwap (Parametric Swap)	✗	✓	✗
MSGate (Mølmer–Sørensen)	✓	✗	✗
RCCX (Simplified CCX)	✓	✗	✗
RC3X (Simplified C3X)	✓	✗	✗
RXX (Rotation about XX)	✓	✗	✗
RYY (Rotation about YY)	✓	✗	✗
RZZ (Rotation about ZZ)	✓	✗	✗
RZX (Rotation about ZX)	✓	✗	✗
DCX (Double CX)	✓	✗	✗
Custom Gates	✓(UnitaryGate)	✓(DefGate)	✓(MatrixGate)
Controlled Gates	✓	✓	✓

Table 4.1: Comparison of gates from different quantum SDKs

The concept of the custom gate allows the specification of user-defined gates in quantum circuits which is done by providing unitary matrices that define the semantic of the gate. In theory, this allows the definition of arbitrary gates and, therefore, the implementation of equivalent quantum circuits in the mentioned SDKs with the same number of gates as the original quantum circuit. However, the definition of gates as custom gates has disadvantages and should only be considered in the case that no equivalent gates exist. The main issue is that the SDKs provide functionality for their standard gates, e.g. specified decompositions of the gates, which is not possible for arbitrary custom gates. The decomposition of a custom gate must be computed for each custom gate in the circuit leading to performance issues. Furthermore, the computed decomposition is not optimal in most cases. Another disadvantage is the overhead needed for the specification of the custom gates which negatively impacts the comprehensibility of a quantum circuit.

Cirq further implements a concept called *PowGates*. For each of their standard gates, they also offer the PowGate variant of the gate. These gates are equivalent to the traditional gates with a global phase factor of $e^{\frac{i \cdot \pi \cdot t}{2}}$ which increases the number of standard gates. Additionally, some gates in Cirq allow the specification of a global shift factor to adjust the global phase of the gate.

Table 4.1 shows that Qiskit supports the most standard gates of the compared SDKs. For the implementation of many standard Qiskit gates, the other SDKs must use gate modifiers or custom gates. Only a few gates that are supported by Cirq or Forest are not implemented in Qiskit. Furthermore, the table shows that equivalent concepts have different names in the SDKs. To compare the semantics of an operation it is necessary to make a comparison between the matrices that define the operations. Besides gates that are named differently but have the same semantic, it is also possible to have gates that have the same name but differ, e.g., with respect to the global phase of the gate.

Even though Qiskit, Cirq, and Forest use Python as the implementation language for quantum circuits, they are not interoperable. This is a result of the quantum circuit objects being inherently different, e.g. there are differences regarding the set of operations and the operation classes as shown in the previous table. Therefore, a converter is needed that enables the translation of a quantum circuit object of one of the SDKs to the other SDKs by replacing the operations used in the circuit with equivalent operations from the target SDK.

4.2.2 Cross-Platform SDKs

In contrast to the SDKs discussed so far, some SDKs support the integration of other quantum SDKs. These SDKs aim at being interoperable with different SDKs, quantum assembly languages, or QPUs.

Pytket

Pytket is a python library to interact with the quantum transpiler *t|ket>* [SDC+20]. Additionally, t|ket> supports various optimizing strategies for quantum circuits including features that consider quantum errors in the transpilation process [SDC+20]. Pytket can interoperate with various quantum frameworks, as well as, QPUs and quantum simulators from several vendors. For importing circuits Pytket supports the quantum assembly languages OpenQASM, Quil and Quipper, and the SDKs Qiskit, Cirq, PyZX, and pyQuil. However, there are several limitations like a high number of unsupported gates regarding many of the integrated languages and SDKs.

PennyLane

PennyLane is a quantum SDK that supports backends from several vendors. The focus of PennyLane is hybrid quantum-classical machine learning computation [BIS+18]. In addition to the qubit model, the continuous-variable paradigm is supported. However, there is no possibility to translate between these models due to structural differences of these quantum approaches. Therefore, a quantum circuit must be designed for the paradigm which is used for the execution. PennyLane supports the integration of popular machine learning libraries like TensorFlow[22] or PyTorch[23] [BIS+18]. Quantum circuits can interface with these libraries to simplify the combination of machine learning and quantum computing. Hybrid algorithms including CPUs, GPUs, and QPUs are possible. A disadvantage of PennyLane is that quantum circuits cannot be exported in quantum assembly languages like OpenQASM, but must be executed in the SDK.

[22] https://www.tensorflow.org/
[23] https://pytorch.org/

PyZX

PyZX uses the *ZX-calculus* to reason about quantum theory and to optimize circuits [KW19]. The ZX-calculus is an equational theory that is based on string-diagram-based representations of quantum circuits called *ZX-diagrams* that are lower-level than quantum instruction languages like OpenQASM [DKPV20]. The advantage of this technique is that they are more flexible than circuits because they can be deformed arbitrarily [DKPV20]. Technically, a ZX-diagram has *wires* and *spiders*. Special cases of wires are the input and the output wires. ZX-diagrams can be easily composed by connecting output to input wires [DKPV20]. The spiders describe the operations by representing linear maps. These maps have arbitrary many input and output wires [DKPV20]. Spiders are either Z or X spiders which is the reason for the name of this technique. Transformation rules and simplifying strategies can be used to convert arbitrary ZX-diagrams to ZX-diagrams that fulfill specific properties. Afterwards, the ZX-diagrams can be converted back to optimized quantum circuits in OpenQASM.

Quantastica

Quantastica offers a quantum tool suite written in JavaScript. Quantastica is a partner of Rigetti. Their core tool is the Quantum Programming Studio[24], a web-based quantum circuit editor that supports the generation and simulation of circuit, as well as, the execution on QPUs directly from the browser. The module QConvert supports rich converting functionalities including the import of OpenQASM and Quil. As an output, additionally, Qiskit, pyQuil, Cirq, Q#, Quest, and Qirk are possible options. Another feature is the drag and drop quantum circuit creation and editing. As a result of the implementation in JavaScript, Quantastica cannot directly interoperate with the SDKs that were presented so far but can only import and export instruction sequences as a text file. In contrast to this, e.g. PennyLane and Pytket support the direct import of a quantum circuit object specified in Qiskit or pyQuil.

[24] https://quantum-circuit.com/home

5 Quantum Circuit Analysis and Transpilation Framework

To decouple quantum circuits from the QPU vendors and enable the support of several SDKs, a quantum circuit analysis and transpilation framework[1] is implemented. In this chapter, the architecture of the framework is explained. Furthermore, the internal data model and the advantages of using this model are discussed. Afterwards, the architecture of the plugin-based SDK and language conversion is introduced.

5.1 Concept of the Framework

The implemented quantum circuit analysis and transpilation framework offers the functionality to import quantum circuits from various vendors and converts them to a canonical internal data model. The framework offers the functionality to transpile the implementation to several QPUs of Rigetti and IBM and analyze the properties of the transpiled circuit. Furthermore, the outcome circuit can be exported to the quantum instruction languages OpenQASM and Quil such that they are executable on the QPUs of Rigetti and IBM.

This approach decouples the quantum SDK from the quantum QPU. Additionally, the developers must not choose a QPU before implementing the quantum algorithm but can analyze the depth of a quantum circuit for several QPUs before deciding which QPU to use. Several aspects need to be considered to achieve the development of a transpiler that enables SDK specific quantum algorithm implementations to be executed on QPUs from different vendors. The implementation of the quantum circuit analysis and transpiler framework is developed with a plugin-based system to enable the integration of additional SDKs. The plugins are used to map between the quantum algorithm implementation data to the internal data model and back. Furthermore, a transpiler is implemented that maps a quantum circuit encoded in the internal data model to an equivalent quantum circuit that only consists of the native gates of the IBM or Rigetti architectures. Afterwards, the quantum circuit can be exported as the executable language of IBM and Rigetti QPUs.

Figure 5.1 shows an overview of the architecture of the framework. The input to the transpiler is a quantum circuit that can either be described in the quantum assembly languages OpenQASM or Quil, given as a quantum circuit object of a supported SDK or as a string containing the python instructions that form a quantum circuit of a supported SDK. The supported SDKs are Qiskit and Forest. However, the individual components and plugins are constructed in a modular manner such

[1] https://github.com/UST-QuAntiL/QuantumTranspiler

that new SDKs like Cirq can be easily integrated into the transpiler. Cirq is not supported in the first version of the framework because in contrast to Qiskit and Forest it does not support the execution of quantum circuits on QPUs.

The steps of the framework are the following:

1. **Import with conversion plugins:** A plugin converts the input circuit to an equivalent quantum circuit implemented in the internal data model. Besides having a circuit that is semantically the same, the goal is to keep as much of the original structure as possible. The internal data model must fulfill several properties such that the transpiler can efficiently operate on the quantum circuit.
2. **Transpilation:** The user can decide between the native gate set of Rigetti or IBM QPUs. Afterwards, the gates in the given quantum circuit are unrolled to the specified target gate set. Furthermore, the coupling map of the target QPU can either be specified manually or the architecture can be chosen out of several IBM and Rigetti QPUs. The transpilation framework computes a topology mapping that satisfies the specified coupling map.
3. **Export with conversion plugins:** The transpiled quantum circuit specified in the internal data model can be exported in different assembly languages and SDK formats by using the appropriate plugins.

The conversion to an intermediate data format has several advantages. With this approach, the number of plugins that are needed grows linearly with the number of languages that should be supported. Instead of requiring $2 \cdot \sum_{k=1}^{n-1} k = n(n-1)$ plugins, only $2n$ plugins are required to import and export n languages if an intermediate data format is used.

Besides the process depicted in Figure 5.1, a quantum circuit can also be converted from one data format - quantum assembly language or SDK circuit object - to another without unrolling the gates to a native gate set. In this case, the transpilation is omitted. The support of the conversion feature without unrolling seems to be solvable by just omitting the intermediate unrolling steps. However, this requires additional implementation effort with respect to the export of a specified format. Exporting an unrolled circuit only requires the support of a very limited set of operations, e.g. IBM QX and Rigetti support less than five native gates, while exporting an arbitrary circuit requires the handling of all standard gates of an SDK, as well as, the user-defined gates and other instructions. Instead of unrolling the gates to the native gates of a QPU and convert this limited set to the target assembly language, each gate used in the quantum circuit is directly converted to an equivalent gate in the target language. The number of gates used in the implementation of the quantum circuit in the import language should be equivalent to the implementation in the export language if the transpilation steps are omitted.

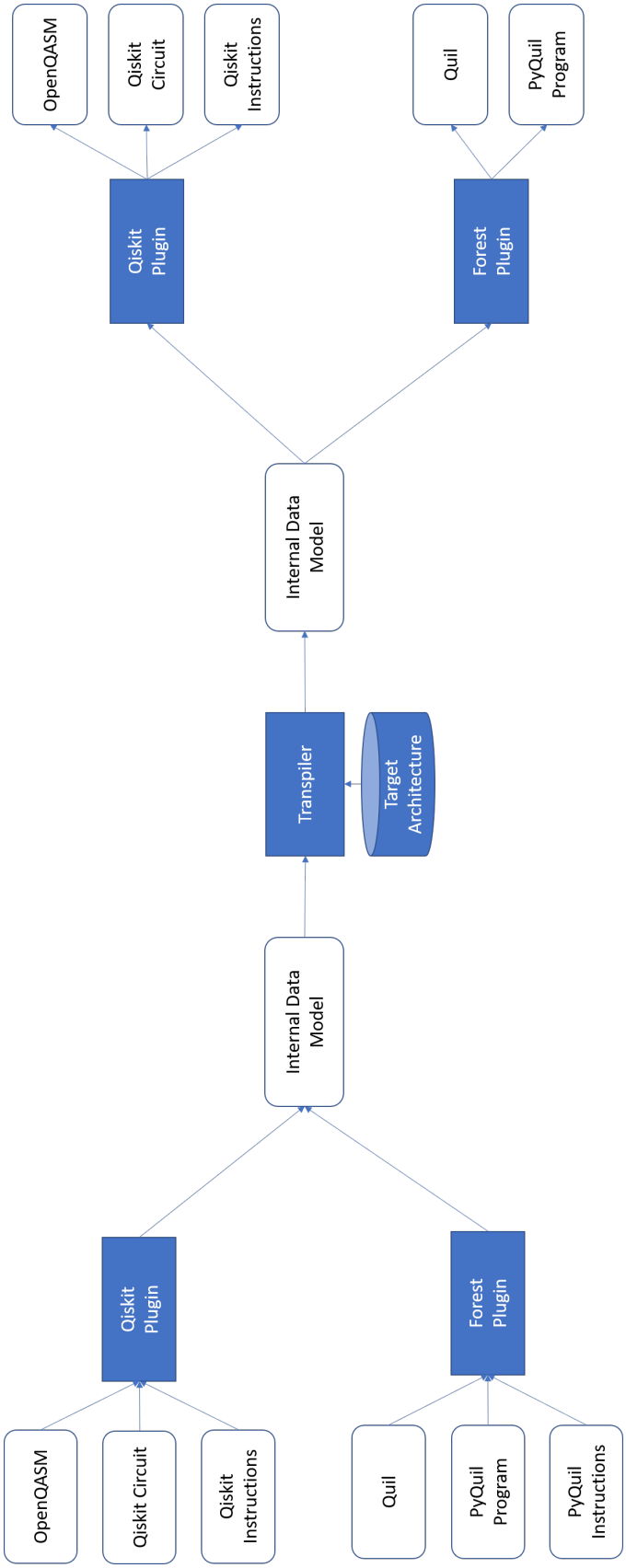


Figure 5.1: Overview of the architecture

5.2 Internal Data Model

An internal data model is useful to reduce the implementation overhead for the conversion plugins. Requirements for the data model are that it supports the concepts of various SDKs or implements equivalent alternatives, e.g. for special gates. This is necessary to enable the conversion of different formats without losing the structure of the quantum circuit, e.g., decomposing not supported gates to a sequence of gates that implements the same behavior increases the circuit depth and has, therefore, impact on the structure of the circuit. As shown in Section 4.2.1, Qiskit supports most of the standard gates from the other SDKs. Moreover, Qiskit supports custom gates, as well as, controlled gates. Therefore, gates that are not supported by default, can be represented as custom gates that implement the same functionality without decomposing the gate. Qiskit offers the *QuantumCircuit*[2] class which is their representation of a quantum circuit consisting of registers and quantum operations that are applied to these registers. The registers can include qubits or classical bits. Qiskit provides various utility methods for analyzing, manipulating, and composing their circuit objects which can be used by the analysis and transpilation framework.

For the actual operations on the quantum circuit in the transpilation process, a data model is needed that supports iterating over the gates with high performance and that is able to partition the instructions to operations that are executed on different nodes. This is important to build layers of the quantum circuit which are then used to solve the hardware constraints layer-wise to simplify the transpilation steps. Qiskit's circuit objects store the instructions that are applied to the circuit in an array. However, the circuit objects can be converted to a directed acyclic graph (DAG) and back. This conversion is implemented in Qiskit and does not lose information or substitute any of the operations because the Qiskit's DAG object and the circuit object support the same instruction set including the same gates. The conversion step between a DAG and a quantum circuit object, therefore, only changes the data structure of the quantum operations. The DAG is an appropriate representation to unroll, optimize, and compute a topology mapping for a quantum circuit because quantum circuits are executed in order and do not have cycles. Additionally, transpilation steps like unrolling or mapping from logical to physical qubits can be accomplished on a DAG with high performance. Furthermore, a DAG fulfills the mentioned requirement of building layers. Qiskit's DAG implementation is based on the graph library *retworkx*[3]. In contrast to the common Python graph library networkx[4], retworkx is implemented in Rust leading to performance advantages regarding the creation, manipulation, and iteration over the DAG. Using Qiskit's DAGCircuit implementation, additionally, has the advantage that the proposed framework can build upon the transpilation functionality that is already implemented and extend it to support the architectures of different vendors. Qiskit provides algorithm implementations to decompose custom gates that are defined by a matrix to a sequence of standard gates. These standard gates can then be mapped to the native gates of the selected QPU. Other quantum SDKs like the Forest SDK also use a DAG as their internal data model for compiling quantum circuits.

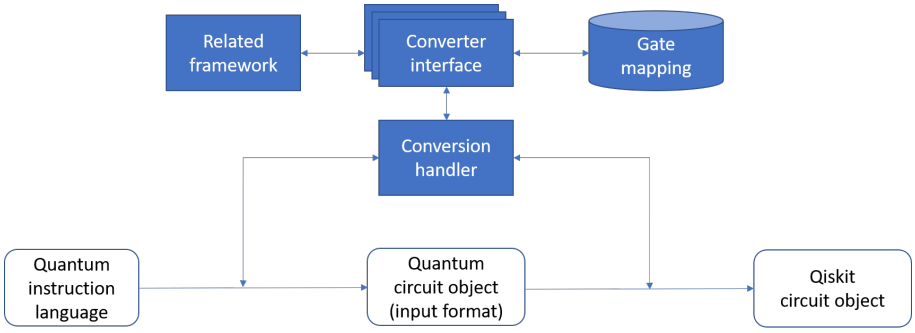


Figure 5.2: Plugin architecture

5.3 Language and SDK Conversion

Figure 5.2 shows the process of importing a quantum circuit. The export to a quantum assembly language is implemented analogously. The conversion functionality is implemented with the plugin-based architecture to support the integration of other SDKs independently from the main application. The core of the conversion tool is the *ConversionHandler*. This component handles all conversion steps like converting from a quantum instruction language to a quantum circuit object or from a quantum circuit object of a supported SDK to a Qiskit circuit object. The handler is initiated with a class that implements the *ConverterInterface*, e.g. the PyquilConverter. A plugin, therefore, has to implement the methods of the ConverterInterface.

Firstly, the quantum instruction language is parsed by the class that implements the ConverterInterface, e.g. in the case of the PyquilConverter this task is propagated to the Forest SDK which returns a Forest SDK quantum circuit object called *Program*. In the second step, the ConversionHandler calls the appropriate methods of the class implementing the ConverterInterface to iterate over the quantum circuit object and create an equivalent Qiskit circuit object. The main advantage of the architecture with the ConversionHandler is the export functionality, i.e. the conversion from a Qiskit circuit object to a circuit object of another SDK. This can be solved by iterating over the Qiskit circuit object and calling the respective methods of the class implementing the ConverterInterface. The iteration over the Qiskit circuit can be implemented completely generic without taking into consideration the respective SDK to which the circuit should be converted. Besides the quantum gates and the measurements, quantum circuits can have specific properties and functionalities depending on the SDK, e.g. Qiskit implements a *barrier* operation that instructs the transpiler to only optimize and rearrange each part of the quantum circuit separately where the parts are divided by barrier operations. If an SDK does not allow the specific functionality and no equivalent construct exists, an error is thrown. Alternatively, the method outputs a warning and returns without changing the quantum circuit object if the concept is not necessary for the circuit semantics, e.g. the barrier operator. In the next chapter, the methods of the ConverterInterface are explained in-depth.

[2]https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html
[3]https://github.com/Qiskit/retworkx
[4]https://networkx.github.io/documentation/stable/index.html

6 Implementation

The previous chapter discussed the architecture of the quantum circuit analysis and transpilation framework. In the following, the implementation details of the individual components are discussed. The programming language of the developed framework is Python because many quantum SDKs are implemented in Python (see Section 4.2) and this simplifies the interoperability of the tools regarding the import and export of quantum circuit objects from different SDKs.

6.1 Language and SDK Conversion Plugins

In Section 5.3, the architecture of the converter plugins is described. Figure 6.1 shows the class diagram of the classes related to the conversion functionality. The type *SDKCircuit* refers to the circuit class of the respective SDK, e.g. in the case of the PyquilConverter it refers to a *Program*. The ConversionHandler is instantiated with a ConverterInterface that is used to iterate over the circuit objects of the respective SDK when importing a circuit object and to create a circuit in the specific quantum assembly language when exporting a Qiskit circuit object.

A Plugin only has to implement the methods of the ConverterInterface to support the import and export of a quantum circuit object in the respective SDK. The ConverterInterface defines the following properties:

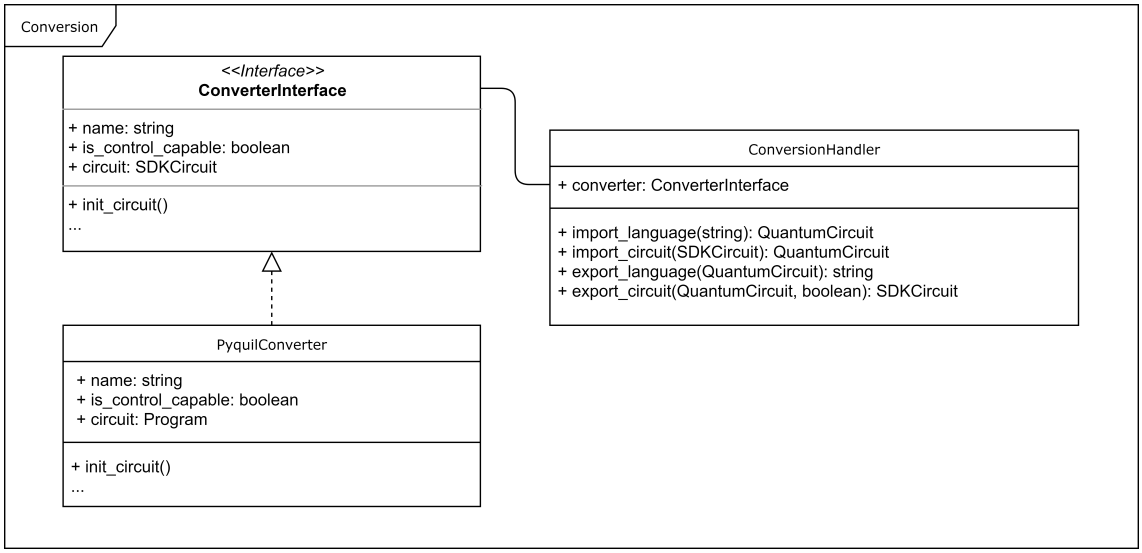


Figure 6.1: Plugin classes

- *name:* Describes the name of the SDK. This is important to retrieve the correct gate classes for the SDK from the gate mapping. The gate mapping defines equivalent gates of the supported SDKs.
- *is_control_capable:* Defines if the SDK supports a *control modifier* which is relevant for the gate conversion. A control modifier allows defining the controlled version of a standard gate as a new gate.
- *circuit:* Retrieves the generated circuit object of the respective SDK.

For the conversion from a language or an SDK circuit object to a Qiskit circuit object and from the SDK to the respective instruction language, the following methods are defined in the ConverterInterface:

- *import_circuit(circuit):* Generates the corresponding Qiskit circuit to the input circuit.
- *language_to_circuit(language):* Parses the instruction language and generates a circuit object of the respective circuit class. The implemented PyquilConverter that inherits from the ConverterInterface internally forwards this call to the pyQuil library. The main advantage of this is that it simplifies the integration of future versions of Quil because new concepts can be handled in the conversion step, rather than in the parsing step. In the conversion step, the new aspects of the SDK must be handled anyway to support the conversion between quantum circuit objects.
- *circuit_to_language(circuit):* Creates the instructions in the instruction language of the SDK that correspond to the given circuit. Like the *language_to_circuit* the PyquilConverter forwards this call to the pyQuil library.

The main advantage of this architecture is the conversion from a Qiskit circuit object to a circuit object of another SDK because this can be done by iterating over the Qiskit circuit object and calling the respective methods of the class implementing the ConverterInterface. A main benefit of the ConversionHandler is the completely generic implementation of the iteration over Qiskit circuit objects. For each instruction of the quantum circuit, the corresponding method of the respective plugin is called. In the following, these methods of the ConverterInterface are explained:

- *init_circuit():* Initializes an empty circuit in the respective circuit format.
- *create_qreg_mapping(qreg_mapping, qubit, index):* The *qreg_mapping* is a variable that maps a qubit from a Qiskit circuit to a qubit from a circuit in the respective SDK format. In the simplest form, this just maps indices. However, if a quantum circuit consists of several qubit registers, the qreg_mapping describes the mapping of qubits in the individual registers to qubits in the quantum circuit that is generated. The method includes the information that a qubit is mapped to a specific index in the qreg_mapping and returns the resulting mapping.
- *create_creg_mapping(cregs):* The *creg_mapping* is analogous to the qreg_mapping. However, this method is called with the whole list of classical registers of the Qiskit quantum circuit and the whole creg_mapping is created at once. The main reason for this is that some SDKs do not support several classical registers and this approach provides more flexibility to the individual converter class.

- *gate(gate, qubits, params, is_controlled, num_qubits_base_gate):* Adds a gate with the given parameters to the circuit that acts on the specified qubits. For SDKs that support controlled modifiers the gate is appended with a controlled modifier. The control qubits are the first $n - num\_qubits\_base\_gate$ qubits from the qubit list.
- *custom_gate(matrix, name, qubits, params):* Adds a custom gate that is defined by the parameter matrix to the circuit. The Forest SDK supports parameterized custom gates that are defined by parameterized matrices. Therefore, params is also a parameter for the custom_gate function call. However, if this functionality is not supported in the respective SDK and the method is called with a parameterized custom gate, the implementation must throw an error. Another interesting aspect regarding the conversion of custom gates is the sequence of qubits, e.g. the Forest SDK and Qiskit differently interpret the sequence of qubits with respect to custom gates.
- *parameter_conversion(parameter):* Converts a Qiskit parameter to the parameter of the respective SDK. Parameters are used to specify gates with variable angles.
- *parameter_expression_conversion(parameter_expression):* Converts a Qiskit parameter expression to a similar concept. The *ParameterExpression* class extends the *Parameter* class. The main difference between the classes is that the Parameter class additionally has a name. The distinction is necessary because each of the classes can be used as a parameter for Qiskit gates.
- *barrier(qubits):* Appends a barrier instruction to the circuit object.
- *measure(qubit, clbit):* Appends the measuring of the specified qubit to the given classical bit.
- *subcircuit(subcircuit, qubits, clbits):* The parameter *subcircuit* is a quantum circuit object of the respective SDK. The subcircuit is appended to the specified qubits and classical bits. Some SDKs natively support this operation. E.g. in Qiskit a circuit can be converted to an instruction that can be applied to a set of qubits and classical bits. In other SDKs, this concept must be implemented as additional functionality to the SDK.

When exporting a Qiskit circuit to another SDK the ConversionHandler first calls the methods for generating a qubit and classical bit mapping. With this information, the handler iterates over the instructions of the circuit and calls depending on the type of the instruction the respective methods. If the instruction is a gate, the handler pre-processes the gate first before calling the ConverterInterface method. Firstly, the parameters of the gate are iterated and for every parameter that is not fixed the *parameter_conversion* or *parameter_expression_conversion* method is called. Furthermore, the qubits to which the gate is applied are extracted. Afterwards, the corresponding gate class of the respective SDK is looked up in the gate mappings. If an equivalent gate exists, the *gate* method in the ConverterInterface is called. However, several exceptions need to be considered. Firstly, if no equivalent gate class exists, the gate is a controlled gate and the SDK supports the control modifier, the corresponding base gate is chosen instead and the *control_capable* parameter is set to true. Alternatively, gates can be defined either by replacement circuits or by a matrix in the gate mapping. In the case of a replacement circuit, the *subcircuit* method is called with the replacement circuit and the qubits to which the original gate was applied to. In the case of a definition by a matrix, the *custom_gate* method in the ConverterInterface is called with the matrix

Algorithm 6.1 Gate mapping example with the gate *CZ*

```
import qiskit.circuit.library.standard_gates as qiskit
import pyquil.gates as pyquil
...
gate_mapping["CZ"] = {
  "qiskit": {"g": qiskit.CZGate},
  "pyquil": {"g": pyquil.CZ},
  "matrix": np.array([
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,-1]
    ], dtype=complex)
}

gate_mapping["Sdg"] = {
  "qiskit": {"g": qiskit.SdgGate},
  "pyquil": {"r": pyquil_replacement.sdg_replacement},
  "matrix": qiskit.SdgGate.to_matrix()
}
...
```

and the name of the Qiskit gate class. This method is also called if the gate is already a unitary gate in which case the matrix and label are extracted from the gate before calling the *custom_gate* method.

Gate Mapping

The gate mapping is defined in a dictionary that maps from the name of the operation to the gate class of different SDKs (see Algorithm 6.1). This gate mapping dictionary is rearranged depending on which format the input circuit has, e.g. if the input circuit is a pyQuil program, a dictionary is generated in which the keys are the names of the pyQuil gates. For every SDK, a dictionary is given that contains either a gate definition or if no equivalent gate exists in the respective SDK, it contains a replacement circuit. The Sdg gate (S-adjoint), e.g., does not exist in pyQuil (see Section 4.2.1). In the gate mapping it is, therefore, described by a replacement circuit with equivalent functionality. In this case, the replacement circuit only consists of one R_Z gate with the angle $-\frac{\pi}{2}$ (see Algorithm 6.2). The replacements circuits are implemented as functions that return quantum circuits (denoted by an ”r” in the dictionary).

The advantage of storing the functions that return the replacement circuit in the gate mapping instead of storing the quantum circuit directly is that this approach allows the definition of parameters for the quantum circuit. Algorithm 6.3 shows the implementation of a parameterized replacement circuit. By storing the function in the gate mapping, the ConversionHandler has the possibility to generate a replacement circuit with arbitrary parameters without knowing the implementation details of the circuit, but only the parameters that the function expects. The sequence and the number of the function parameters are designed to be equal to the properties that the gate in the imported

Algorithm 6.2 Replacement circuit for the Sdg gate implemented by a function returning a quantum circuit

```
def sdg_replacement():
  p = Program()
  p += RZ(-np.pi/2, 0)
  return p
```

Algorithm 6.3 Replacement circuit with parameters (U3 gate)

```
from pyquil.quilatom import Parameter, quil_sin, quil_cos, quil_exp
from pyquil.quilbase import DefGate

def u3_replacement(theta: float, phi: float, lam: float):
  """ implemented with a custom gate """

  theta_param = Parameter('theta')
  phi_param = Parameter('phi')
  lam_param = Parameter('lam')
  matrix = np.array([
    [
      quil_cos(theta_param / 2),
      -quil_exp(1j * lam_param) * quil_sin(theta_param / 2)
    ],
    [
      quil_exp(1j * phi_param) * quil_sin(theta_param / 2),
      quil_exp(1j * (phi_param + lam_param)) * quil_cos(theta_param / 2)
    ]
  ])
  definition = DefGate('U3', matrix, [theta_param, phi_param, lam_param])
  U3 = definition.get_constructor()
  p = Program()
  p += definition
  p += U3(theta, phi, lam)(0)
  return p
```

circuit has. This leads to the benefit that the ConversionHandler can abstract from the individual gate and simply execute `replacement_circuit = replacement_function(*params)` where *params* are the parameters of the gate that is read from the input circuit. Furthermore, the corresponding unitary matrix is stored in the gate mapping for gates without parameters (see Algorithm 6.1).

It is important to respect the order of qubits for custom gates. Qiskit follows the little-endian convention, i.e. higher qubit indices are more significant than lower qubits[1]. In contrast to this, many textbooks and SDKs like the Forest SDK assume that lower qubit indices are more significant. Therefore, when applying a custom gate from Qiskit in Forest and the other way round the qubit order must be reversed.

6.1.1 Controlled Gates

Section 2.1.3 shows that the controlled versions CRZ and CU1 of the equivalent gates RZ and U1 are not equivalent. Interestingly, the implementation of the converter of Quantastica[2] contains a bug regarding this phenomenon. They exchange the pyQuil gate *CPHASE* which is the same as *CRZ* (see Section 4.2.1), with the Qiskit Gate *CU1*. Pennylane, e.g., does not support the common Qiskit/OpenQASM operation *CU1* at all.

Even in general it holds true that the controlled versions of two gates that are equivalent up to their global phase are not equivalent unless the global phase factor is 1. The following proof shows the general case of two equivalent two-qubit gates that differ by the global phase factor $e^{i\theta}$.

Proof
Suppose

$$e^{i\theta} \neq 1$$

for $\theta \in \mathbb{R}$. It follows directly that

$$e^{-i\theta} \neq 1$$

Furthermore, suppose that the gates $Gate_A$ and $Gate_B$ are equivalent up to the global phase factor $e^{i\theta}$

$$GATE_A = \begin{pmatrix} e^{i\theta}\alpha & e^{i\theta}\beta \\ e^{i\theta}\gamma & e^{i\theta}\delta \end{pmatrix} = e^{i\theta}\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \equiv \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = GATE_B$$

The following must be true such that the controlled versions of the gates are equivalent up to their global phase

$$ControlledGATE_A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{i\theta}\alpha & e^{i\theta}\beta \\ 0 & 0 & e^{i\theta}\gamma & e^{i\theta}\delta \end{pmatrix} = e^{i\theta}\begin{pmatrix} e^{-i\theta} & 0 & 0 & 0 \\ 0 & e^{-i\theta} & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & \gamma & \delta \end{pmatrix}$$
$$\equiv \begin{pmatrix} e^{-i\theta} & 0 & 0 & 0 \\ 0 & e^{-i\theta} & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & \gamma & \delta \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & \gamma & \delta \end{pmatrix} = ControlledGATE_B$$

As a result, the following must hold

$$e^{-i\theta} = 1$$

This contradicts the assumption. □

[1] https://qiskit.org/documentation/stubs/qiskit.circuit.library.CXGate.html
[2] https://github.com/quantastica/qconvert-js

The effect of this observation is that for the creation of an equivalent controlled version of a gate in a different SDK it is necessary to use the basic gate that is defined by the same matrix. It is not sufficient that the gates are equivalent up to their global phase. Therefore, in the conversion process, it must be distinguished between gates that are equivalent up to their global phase and gates that are equal.

6.2 Transpiler Implementation

The previous section discussed the integration of the quantum SDKs Qiskit and Forest SDK. In contrast to this, the transpile component deals with the support of the QPUs from Rigetti and IBM. In the following, several techniques are introduced to map standard gates, as well as, custom gates to the native gate set. For the mapping from physical to logical qubits algorithms from Qiskit are used. The topology mapping algorithms can be used to satisfy the coupling maps of QPUs from Rigetti and IBM. Therefore, this aspect will not be further discussed.

6.2.1 Mapping to Native Gates

The main constraint of a QPU is that every gate in the quantum circuit that should be executed must be a native gate of the QPU (see Section 2.1.5). To guarantee this property, the transpilation framework maps the gates that are used in a quantum circuit to the native gates of the specified QPU. The mapping is done differently depending on the type of gate that must be decomposed. For the standard gates, a hard-coded decomposition exists that transitively maps a gate to the specified native gate set. In the case of custom gates different algorithms to decompose the unitary matrix are used depending on the number of qubits of the gate.

Standard Gates

For each standard gate, a decomposition is defined such that the standard gate is either directly decomposed to the specified native gates or to other standard gates. If the gate is mapped to standard gates it is ensured that the mapping recursively reaches a decomposition only consisting of native gates. Supported native gate sets are the native gates of the IBM QX (see Section 4.1.1) and the Rigetti QPUs (see Section 4.1.2).

In this context, only decompositions should be considered that have a shorter path to the native gates. Otherwise, gates like *CZ* and *CNOT* could result in infinite loops because they can be substituted mutually (see Figure 6.2). One possibility to deal with the infinite loops would be to do a breadth-first search in the decomposition graph. However, besides being impractical this approach is only semi-decidable. If gate sets are specified that are not universal the algorithm is not guaranteed to halt in general.

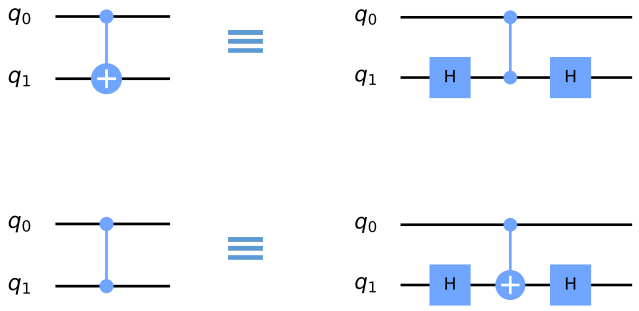


Figure 6.2: CNOT and CZ identities [GC11]

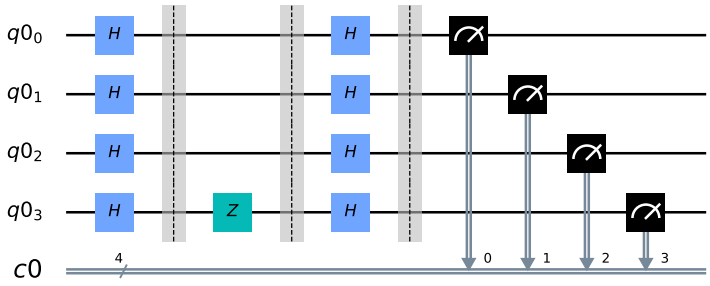


Figure 6.3: Bernstein-Vazirani without ID-Gates

Identity Gates

A special gate is the *Identity* gate. One reason to have identity gates in quantum circuits is the better clarity of the implementation. Removing them does not change the semantics of the circuit. As a result, one could assume that simply removing them is reasonable and reduces the number of gates used in the circuit. However, the following results from running quantum circuits on an IBM QX QPU indicate that Identity gates lead to slightly improved results regarding the error rates of the measurements compared to idling qubits.

Circuit Figure 6.4 and Figure 6.3 are Bernstein-Vazirani algorithm implementations. The Bernstein-Vazirani algorithm is used to learn a string encoded in a function. Bernstein and Vazirani [BV97] used it to prove the distinction between the complexity classes BQP and BPP.

The encoded string that is used in the algorithm implementation is *1000*. The Hadamard operation is applied to each qubit. Afterwards, depending on the input string either an Identity or a Z gate is executed on each qubit. In Figure 6.3 the Identity gates are omitted. Finally, another Hadamard gate, followed by the Measurement operation, is applied to each qubit. Important to note are the Barrier operations that embed the Identity gates. Without this distinction, the two Hadamard gates would be executed consecutively without idle time between them.

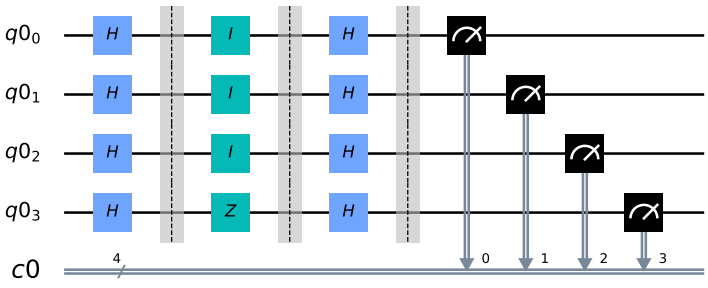
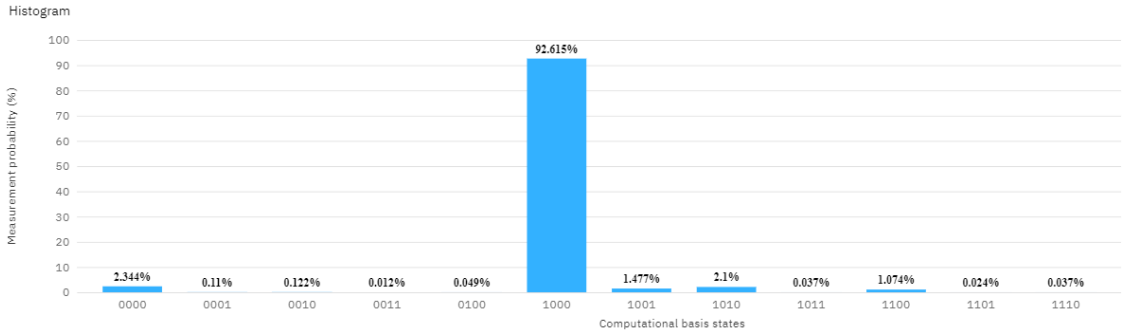
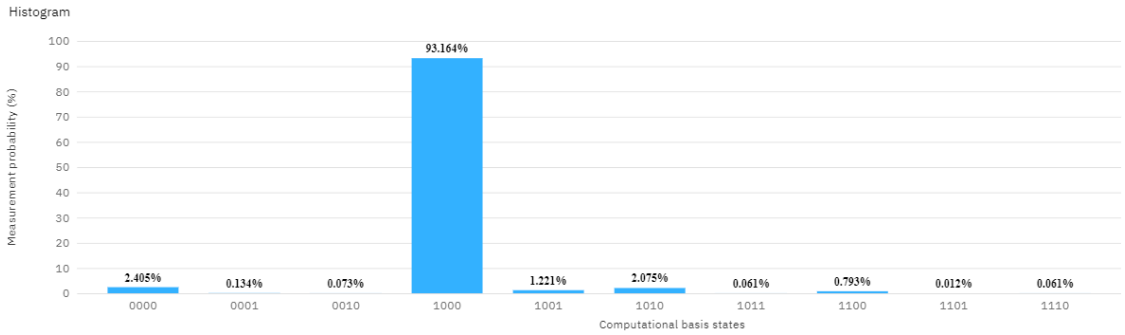


Figure 6.4: Bernstein-Vazirani with ID-Gates



(a) Without *Identity* gates.



(b) With *Identity* gates.

Figure 6.5: Results of the Bernstein-Vazirani algorithm with input string *1000* executed on IBM QX Ourense with 8192 shots

Results Figure 6.5 shows a quantum circuit with Identity gates that measures the correct result in roughly 93.2% of the runs while the implementation without these achieves this result in approximately 92.6%. The number of shots chosen for this experiment was 8192. The measuring of qubits leads to a collapse of the state (see Section 2.1.1). Therefore, the whole quantum circuit must be run several times if the measured qubit is in superposition to obtain a probability distribution of the outcomes. Repetitions of the experiment show similar results. The accuracy of the execution of the circuits that considers Identity gates is always around 0.5% higher than the accuracy of the implementations without Identity gates. The increase of accuracy is quite significant considering that the circuits consist of only one Identity gate for each qubit. Therefore, Identity gates should not be omitted in the transpilation process.

Custom Gates

In the case of custom gates, hard-coded decompositions are not possible, because as described in Section 2.1.2 there already exists an infinite number of single-qubit gates described by different unitary matrices. For the decomposition of custom gates, the functionality of Qiskit is used. If Rigetti's native gates are chosen as the target gate set, the gate sequence that is computed by the decomposition algorithm is further decomposed to this gate set. In the following, the used techniques are introduced.

Single-qubit Gates Single-qubit gates can be decomposed into Euler angle rotations[3]. The computed decomposition consists of the three Euler rotation angles θ, φ, λ, and the global phase parameter γ. The Euler rotation angles can be implemented with the gate U_3 up to a global phase:

$$GATE = e^{i\gamma} U_3(\theta, \varphi, \lambda) \tag{6.1}$$

Afterwards, the U_3 gate can be further decomposed to a sequence of *RZ* and *RX* gates as shown in Equation (4.3).

Multi-qubit Gates Qiskit handles user-defined quantum gates as isometry transformations[4]. For this purpose, they use the decomposition that is described by Iten et al. [ICK+16] to synthesize the operation to standard Qiskit gates. The optimization goal of this approach is to reduce the number of *CNOT* operations. The algorithm is based on the *Cosine-Sine Decomposition (CSD)*. The CSD describes that every unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$ can be decomposed as

$$U = \begin{bmatrix} A_0 & 0 \\ 0 & A_1 \end{bmatrix} \begin{bmatrix} C & -S \\ S & C \end{bmatrix} \begin{bmatrix} B_0 & 0 \\ 0 & B_1 \end{bmatrix} \tag{6.2}$$

where A_0, A_1, B_0, $B_1 \in \mathbb{C}^{2^{n-1} \times 2^{n-1}}$ and C, and S are real diagonal matrices satisfying $C^2 + S^2 = I$ [ICK+16; SBM06]. Therefore, the CSD allows the recursive decomposition of arbitrary quantum gates.

[3] https://qiskit.org/documentation/stubs/qiskit.quantum_info.OneQubitEulerDecomposer.html
[4] https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.isometry.html

6.3 Circuit Analysis

With the conversion and transpilation functionality, the proposed framework can integrate SDKs and QPUs from different vendors. The circuit analyzing component encapsulates this functionality and analyzes the transpiled quantum circuits for Rigetti and IBM QX QPUs to abstract from the algorithm implementation and provide properties of the quantum circuits to the user.

A substantial property of a quantum circuit is the depth. As described in Section 2.2 the feasibility of a quantum circuit is influenced by the depth and the width of the circuit. However, the width of the implementation is not depending on the transpilation process or the used QPU. The depth of a quantum circuit can easily be computed by finding the longest path in the DAG. The issue in the practice is that the depth of an unrolled circuit is not comparable between different vendors, e.g. the native gates that are supported by the control software of the QPU from Rigetti and Qiskit are on different abstraction levels. The single-qubit native gates of Rigetti and IBM need the following number of pulses: $U1$: 0, $U2$: 1, $U3$: 2, R_Z: 0, R_X: 1. Pulses are the physical operations that drive the rotations around the Bloch sphere for superconducting qubits (see Section 2.1.5) [MWS+17]. Counting the number of pulses on the longest path is the more appropriate metric compared to counting the native gates because the pulses describe the operations that are actually executed on the physical qubits.

An alternative to this approach is to only count the two-qubit gates. The reason for this is that the error rates of the two-qubit operations are significantly worse than the error rates of single-qubit gates on NISQ machines [AAB+19; SDC+20]. One reason for this is the virtual Z gate implementation for QPUs with superconducting qubits (see Section 2.1.5). This allows the implementation of arbitrary single-qubit gates with a maximum of two pulses and three virtual gates leading to a fast execution with a high-fidelity [SDC+20]. One effect of this is that quantum circuit optimizing approaches like the approaches that Sivarajah et al. [SDC+20] implement in t|ket> optimize primarily regarding the two-qubit gate count.

6.4 Frontend

Besides importing the developed framework as a Python module and calling the requested functionality directly, the framework can be used by interacting with a web app[5]. The web app offers the whole functionality of the transpiler in a user-friendly fashion. The web app is developed in the client-side framework Angular[6]. To allow users to write code in the browser with syntax highlighting, code completion features, and shortcuts, the frontend integrates the Monaco Editor[7].

The frontend offers functionality to import a circuit specified as a Quil or OpenQASM program, or alternatively as the Python instruction sequence generating a Qiskit circuit. Afterwards, the circuit can be edited either by changing the Qiskit instructions or by using a graphical circuit representation and drag and drop to change the sequence of gates or to add and remove gates from the circuit (see Figure 6.6). Changing either the instructions or the graphical representation dynamically updates the

[5] http://193.196.39.182:3000/
[6] https://angular.io/
[7] https://microsoft.github.io/monaco-editor/

other representation as well. The gates in the graphical representation can be hovered over to show data corresponding to the gate like the parameters. Furthermore, the line in the instruction sequence is marked in which the specific gate is declared. The quantum circuit can also be simulated and the counts of the measured qubits can be shown in a chart. The main benefit of the frontend is the unrolling step. In this step the various depth metrics (see Section 6.3) can be shown for the unrolled circuit for IBM and Rigetti QPUs. The unrolled circuit can then be exported in the executable format for the specific QPU (see Figure 6.7). Alternatively, the circuit can also be exported in other formats which is interesting for academic aspects regarding the analysis of the implementation related to different QPUs. Besides this usual utilization flow, the frontend offers the functionality to directly convert between an input and an output format without any intermediate steps.

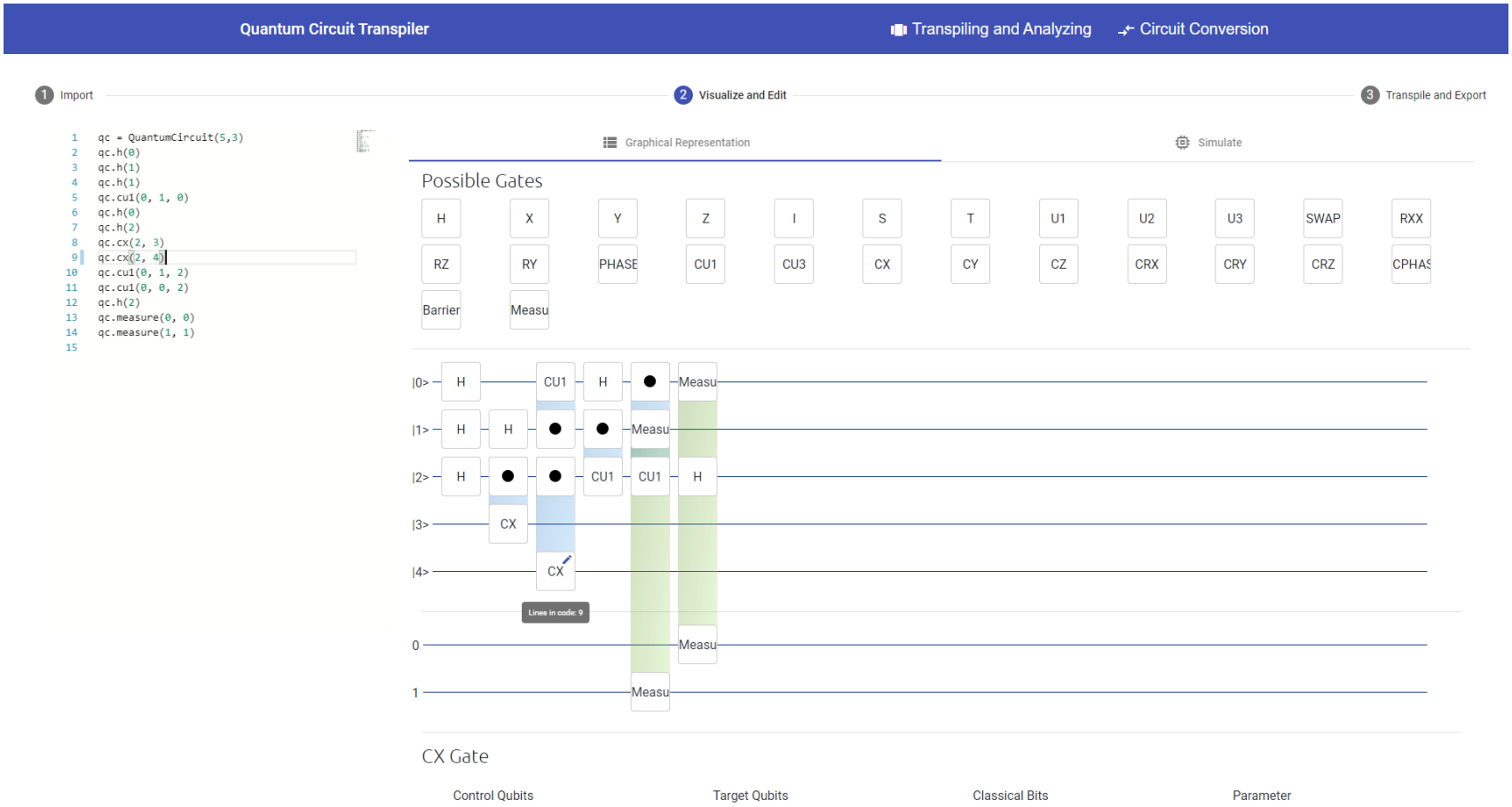


Figure 6.6: Visualizing and editing a quantum circuit in the frontend.

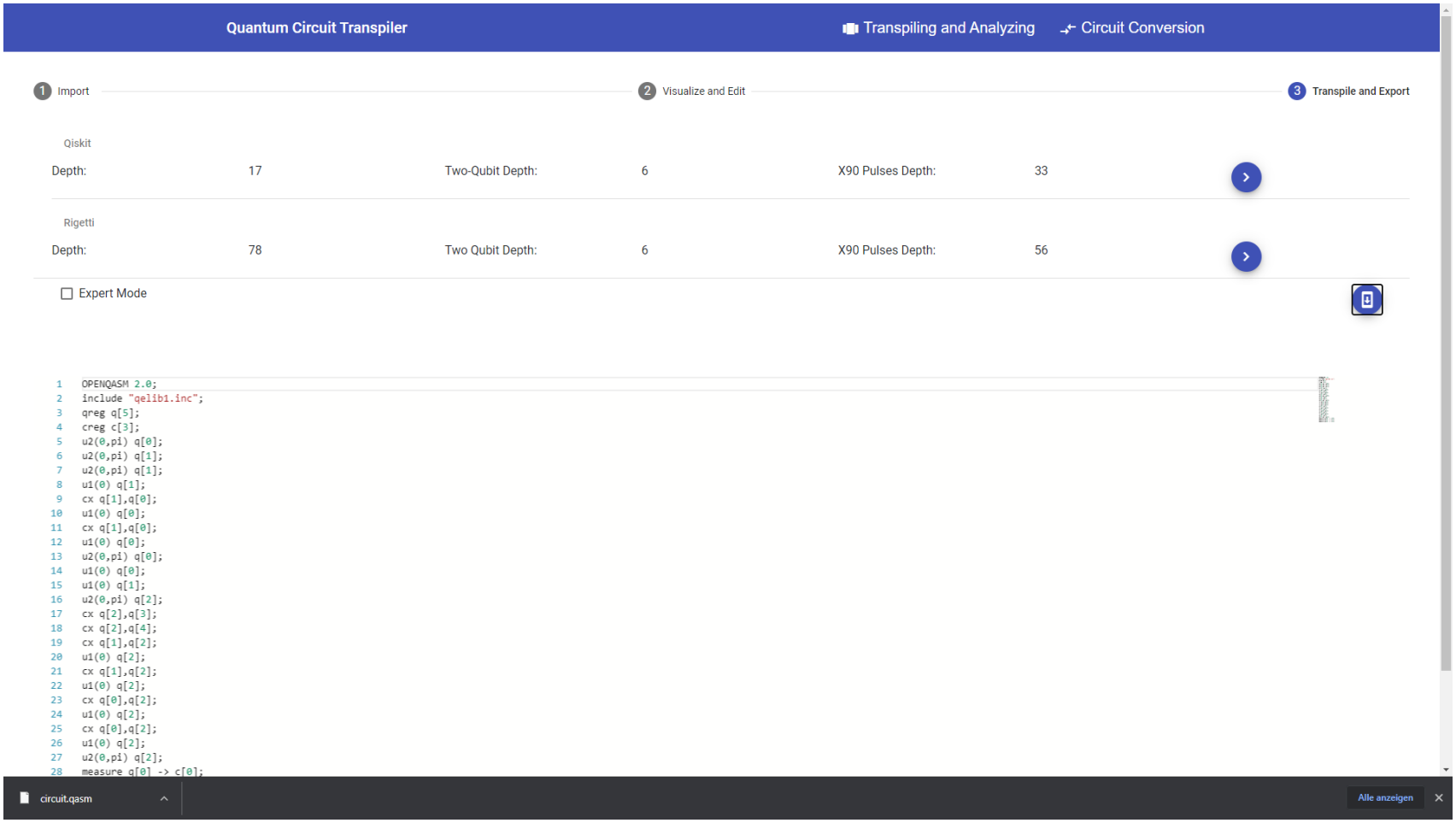


Figure 6.7: Export of a quantum circuit specified in OpenQASM

7 Discussion

In this chapter, the integrated platforms and languages are categorized into functionality classes. In this context, the conversion functionality of the developed framework is compared to Pytket, PennyLane, and Quantastica. Furthermore, the advantages, as well as, the limitations of the developed framework are discussed.

7.1 SDK and Language Classes

The SDKs and languages support different functionalities. In the following, the supported concepts are compared. Figure 7.1 shows the conversion classes. Conversions between all the supported platforms and languages are possible and do not change the semantics of the quantum circuit. However, some conversions do change the syntax of the implementation. A conversion in the direction of the arrows can be reverted and the input and output circuits are syntactically the same. The dotted arrows show conversions that lead to minor syntactical differences in some special cases. However, the depth of the quantum circuits does not change when converting a quantum circuit in the direction of a dotted arrow.

Qiskit supports the most standard gates (see Section 4.2.1). Furthermore, Qiskit supports many features like reverting the instructions of a quantum circuit, taking the adjoint of the whole circuit, defining a quantum circuit as a new gate, and applying the new gate to specific qubits and classical bits of another quantum circuit. The new gate that is defined by a quantum circuit can also be defined as a control gate. Qiskit, pyQuil, and Quil support the definition of custom gates. OpenQASM on the other hand supports more standard gates than pyQuil and Quil. An OpenQASM quantum circuit

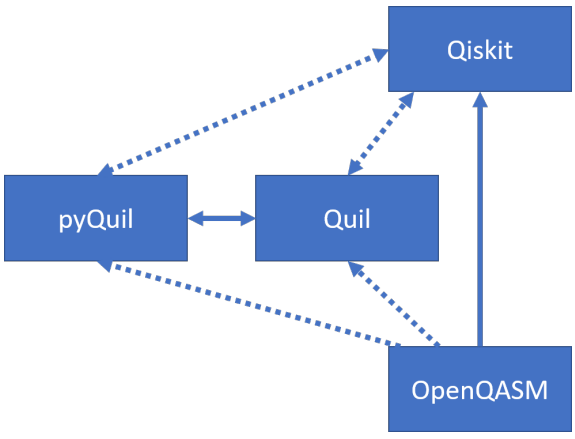


Figure 7.1: Conversion without syntactic loss between the supported SDKs and quantum assembly languages

that is converted to Qiskit and back does not lose any information and exactly the same sequence of instructions is returned. However, as a result of OpenQASM not supporting custom gates, the conversion to OpenQASM must decompose all gates that are not supported by OpenQASM. This decomposition cannot be reverted. Therefore, the conversion to OpenQASM affects the number of gates used in the quantum circuit and conversion to OpenQASM and back will lead to equivalent, but in some cases syntactical different quantum circuits.

Most pyQuil and Quil quantum circuits will stay syntactically the same when converting them to Qiskit and back. Minor differences can appear when using the gates *CCPHASE00*, *CCPHASE01*, and *CCPHASE10* because they will be converted to a Qiskit *UnitaryGate*. When converting the implementation back, the custom gate will be replaced with a *DefGate* which is the Quil equivalent of a custom gate. The conversion does not change any semantics of the quantum circuit nor does it affect the number of gates or the label of the gate. The only difference is that the converted gate is defined by its matrix as a custom gate instead of using the standard gate. Another concept that is supported in pyQuil/Quil, but not in Qiskit is the *PRAGMA* directive. This directive adds extra information to a quantum circuit that does not affect the semantics of the quantum circuit but changes the behavior of the transpiler [Com19; SCZ16]. The conversion from pyQuil/Quil to Qiskit and back can change, i.e., minor syntactical details. Also, the conversion from a Qiskit circuit to any of the other platforms can lose information, e.g. the possibility to interpret a quantum circuit as an individual gate in Qiskit does not have equivalent functionality in pyQuil or Quil.

7.1.1 Comparison of existing Conversion Tools

Several SDKs offer the conversion of quantum circuits between various programming languages and SDKs (see section 4.2). The SDKs support in general comparable operations, but the features slightly differ between the platforms, e.g., the measurement in PennyLane can only be done at the end of the circuit [BIS+18]. In the following, the features of the developed framework are compared to Pytket, PennyLane, and Quantastica. These frameworks are chosen for the comparison because unlike the other introduced platforms they offer the functionality to import and export quantum circuits from Qiskit and the Forest SDK.

Generally, PennyLane does not support the export of quantum circuits at all, but the direct execution on IBM's and Rigetti's QPUs, as well as, simulators. The code fragment that transfers the quantum circuit to a backend can be adapted to output the quantum circuit. However, only the unrolled circuit and, therefore, no gates other than native gates of the respective QPU can be exported in this way. Furthermore, PennyLane does not support the import of measurement instructions at all. Quantastica is implemented in JavaScript. As a result, quantum circuit objects of Qiskit and pyQuil cannot be imported, but only Quil and OpenQASM files. Furthermore, the Python quantum circuit objects cannot be exported. Quantastica tackles this issue by offering the possibility to export Python code that generates a quantum circuit object implementing the given circuit functionality.

One feature of importing Qiskit (see Table 7.1) that separates the proposed framework from the other approaches is that it offers the import of Python instructions for both Qiskit and pyQuil. This is especially relevant for the frontend (see Section 6.4). Another key feature is the support of *subcircuits*. Subcircuits describe the application of a quantum circuit to a specified set of qubits and classical bits in another quantum circuit.

Moreover, the support of standard gates is an important aspect. In contrast to Table 4.1 where the standard gates of Qiskit, Forest SDK, and Cirq are compared, this paragraph is about the support of the conversion tools for these standard gates. PennyLane, e.g., does not support the gate $CU1$ which is a common Qiskit gate. Quantastica, on the other hand, supports it but replaces it with the gate CRZ that is not equivalent and leads to wrong results. This results from the significant observation that the controlled versions of two gates that are equivalent up to their global phase are not equivalent (see Section 6.1.1). The bullet point parameterized circuits describe quantum circuits with variable parameters. One use case of parameterized circuits is the compiling of the parameterized circuit where the circuit acts like a template [Com19]. The circuit can be transpiled and optimized independently from the actual values. The template can then be used repeatedly with varying values which is a common setting in quantum computing [Com19].

Operation	Own Approach	Pytket	PennyLane	Quantastica
Standard gates supported	✓	✓	✗	✗
Python instructions as input	✓	✗	✗	✗
Subcircuits	✓	✗	✗	✗
Custom gates	✓	✓	✓	✗
Measurement	✓	✓	✗	✓
Parameterized circuits	✓	✓	✓	✗

Table 7.1: Importing Qiskit and OpenQASM

Interesting for the export to Qiskit (see Table 7.2) is that the developed transpilation framework supports the generation of Python code which is required for the frontend. As mentioned before, Quantastica offers similar functionality.

Operation	Own Approach	Pytket	PennyLane	Quantastica
Python instructions as output	✓	✗	✗	✓
Custom gates	✓	✓	✗	✓
Measurement	✓	✓	✗	✓

Table 7.2: Exporting Qiskit and OpenQASM

Regarding the import of pyQuil and Quil (see Table 7.3) various concepts are interesting. The variation in the support of the features between the different platforms is quite high. Pytket offers rather rudimentary support of pyQuil features, e.g. custom gates are not supported at all. Additionally, some of the standard gates from pyQuil are not supported either. Similar to the import of controlled Qiskit gates, Quantastica misinterprets some of the controlled pyQuil gates in the import process, e.g. the controlled phase gate *CPHASE*. Another feature that pyQuil offers are the gate modifiers *DAGGER*, *CONTROLLED*, and *FORKED* [Com19]. DAGGER constructs the adjoint of the specified gate which is equivalent to the inverse for unitary matrices [NC02]. The CONTROLLED modifier builds the controlled version of a gate and can be applied iteratively. Lastly, the FORKED modifier applies a parametric gate where the parameters are dependent on a qubit value [Com19]. None of the discussed conversion tools supports the FORKED modifier due to its complexity and the lacking of similar concepts in other platforms like Qiskit.

Operation	Own Approach	Pytket	PennyLane	Quantastica
Custom gates	✓	✗	✓	✗
All pyQuil gates supported	✓	✗	✓	✗
Measurement	✓	✓	✗	✓
Parameterized circuits	✓	✗	✓	✗
Gate modifier dagger and control	✓	✗	✓	✗
Gate modifier forked	✗	✗	✗	✗
Parameterized custom gates	✗	✗	✗	✗
Python instructions as input	✓	✗	✗	✓

Table 7.3: Importing pyQuil and Quil

One specialty with the export of pyQuil and Quil (see Table 7.4) is that Pytket cannot handle multiple quantum registers. Furthermore, the parameterized circuit cannot be exported to pyQuil by any of the available platforms. In contrast to the own approach, Quantastica offers the export of Python code that generates pyQuil circuits. However, this functionality is not needed for the developed transpiler, because the transpiler can directly output a pyQuil circuit object which can then be imported to other platforms. Additionally, the frontend does not depend on this functionality and is only dependent on the generation of Python code for Qiskit quantum circuits, because they are used in the processing pipeline.

Operation	Own Approach	Pytket	PennyLane	Quantastica
Multiple quantum register	✓	✗	✗	✓
Parameterized circuits	✓	✗	✗	✗
Python instructions as output	✗	✗	✗	✓

Table 7.4: Exporting pyQuil and Quil

7.2 Advantages

The proposed analysis and transpilation framework allows analyzing a quantum circuit that is given in either pyQuil, Quil, Qiskit, or OpenQASM. The analysis considers various QPUs from Rigetti and IBM and several metrics of the transpiled circuits. This allows the decoupling of SDKs and QPUs. The framework analyzes the transpiled circuits and allows choosing the most suitable QPU for execution without the need of rewriting the algorithm implementation. The frontend provides an editor with rich features to implement quantum circuits or alternatively quantum circuits can be implemented by using drag and drop in combination with a graphical representation of the quantum circuit. The correctness of the conversion and transpilation process was tested for several quantum circuits by using the transpilation functions of the respective SDK and comparing the simulation results with results from a quantum circuit that is imported and transpiled by the developed framework.

7.3 Limitations

Despite allowing the analysis of executing an arbitrary quantum circuit specified in either pyQuil, Quil, Qiskit, or OpenQASM the analysis and transpilation framework also has some limitations. One main limitation is that the framework does not optimize quantum circuits which limits the comparability of the transpiled circuits. The optimization step has significant effects on the performance or even the viability of the execution of quantum circuits because the depth and, therefore, the error rates are much smaller. However, constrained and unconstrained optimization can be included in the framework without adjusting the other functionalities.

Additionally, the frontend does not support the specification of coupling maps, but only the unrolling step of the transpilation process for IBM QX and Rigetti QPUs. Another issue with the topology mapping is that the state-of-the-art algorithms that compute a valid topology mapping are computationally expensive [ZPW18]. Executing a topology mapping algorithm for several QPUs is impractical for large quantum circuits like implementations of the Shor algorithm. An estimation algorithm that estimates the additional gates needed for specific QPUs based on the properties of an input quantum circuit could solve this issue. The advantage of this approach would be that the user could choose the QPU with the best characteristics for the given quantum circuit and only after deciding which QPU to use, the executable quantum circuit in the respective quantum assembly language is computed. The estimation could be done based on previous runs of topology mappings with quantum circuits with a similar gate count, a similar gate distribution, and a comparable number of qubits used for the specific QPU. This could enable high-performance evaluations of the circuit for a high number of QPUs with the limitation that the actual number of gates needed can slightly differ from the estimated count.

Another limitation is that some standard gates are first unrolled to a sequence of $U3$ and $CNOT$ gates which are then further unrolled to the Rigetti native gates if a Rigetti QPU is chosen. In these cases, better decompositions of the gate might exist. Direct decompositions of these gates to the Rigetti native gate set should be considered and implemented to achieve a more comparable unrolled quantum circuit if these gates are used. In this context, also a constrained optimization step could be useful to combine consecutive gates and improve the significance of the circuit analysis.

Furthermore, the current version of the framework only supports pyQuil, Quil, Qiskit, and OpenQASM. High-level languages like Silq[1] simplify the development of quantum algorithm implementations and will most likely gain momentum in the future. The integration of these languages into the framework is necessary to enable wider usability of the framework. The framework is designed to enable the further integration of quantum languages in a modular way (see Section 5.3).

[1] https://silq.ethz.ch/

8 Conclusion and Outlook

This thesis analyzes current quantum frameworks and QPUs with respect to their native gate sets and their qubit connectivity. An important finding in this context is that the native gates from IBM and Rigetti are defined on different abstraction levels. While IBM's native gate $U3$ is physically implemented with two pulses, all gates from Rigetti are either implemented by one pulse or only virtual in the control software of the chip. This impacts the analysis of executable circuits for different QPUs because the crucial metric regarding the feasibility of a circuit is not the number of gates listed in the instruction list, but the number of pulses or operations in general that are actually executed on a qubit. The comparison of different quantum frameworks shows that the capabilities of Qiskit represent a superset of the other frameworks for practical operations, e.g. regarding the defined standard gate set. Alternatively, gates that are not in the standard gate set can be substituted with custom gates. Therefore, the Qiskit DAGCircuit is chosen as the intermediate data format to manipulate a quantum circuit.

Based on the SDK and QPU analysis a quantum circuit analysis and transpilation framework is developed that supports the integration of quantum circuits from pyQuil, Quil, Qiskit, and OpenQASM. The importing and exporting functionality is compared to the quantum SDKs Pytket, Pennylane, and the converter developed by Quantastica. The proposed framework has advantages compared to these platforms concerning several aspects like the number of standard gates that are supported, the support of custom gates, as well as, the extensibility of the framework.

The framework aims at decoupling the quantum SDK from the hardware on which the implementation is executed. This removes the need of having technical knowledge of the existing frameworks and QPUs before implementing or choosing a quantum algorithm implementation. The proposed framework analyzes several QPUs from Rigetti and IBM in nearly real-time and allows the export of quantum circuits in the respective quantum assembly languages for the specified QPU. For this purpose, the gates used in the implementation are unrolled to the native gate set by using a specified gate mapping for the standard gates or the algorithm from Iten et al. [ICK+16] to decompose arbitrary custom gates. Furthermore, Qiskits' topology mapping algorithms are used to satisfy the qubit connectivity of a specified QPU. The interaction with the proposed framework can be either done by importing the framework as a Python module and call the methods directly or by using the developed frontend.

Outlook

The development in quantum computing proceeds quickly. In 2019 Arute et al. [AAB+19] showed quantum supremacy with a QPU with 53 qubits. While at the moment, the hardware of quantum computers is the restraining factor, the software gains momentum as well, similar to the history in classical computing. Various quantum SDKs are developed by different vendors and high-level

programming languages for quantum computing like Silq are proposed to simplify the programming of quantum algorithms while making the implementation more intuitive and robust. The algorithms that satisfy the qubit connectivity of QPUs and optimize quantum circuits for specific QPUs will improve. Therefore, the practicability of executing quantum circuits on the available NISQ machines will increase. Furthermore, improved algorithms handling the error correction of quantum circuits will enable the execution of more complex quantum algorithm implementations with a considerably higher - or even arbitrary high - depth.

Bibliography

[AAB+19] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510 (cit. on pp. 59, 69).

[AG20] M. Amy, V. Gheorghiu. "staq-A full-stack quantum processing toolkit". In: *Quantum Science and Technology* (2020) (cit. on p. 36).

[AL99] D. S. Abrams, S. Lloyd. "Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors". In: *Physical Review Letters* 83.24 (1999), p. 5162 (cit. on p. 29).

[AMMR13] M. Amy, D. Maslov, M. Mosca, M. Roetteler. "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.6 (2013), pp. 818–830 (cit. on p. 31).

[AVK+08] D. Aharonov, W. Van Dam, J. Kempe, Z. Landau, S. Lloyd, O. Regev. "Adiabatic quantum computation is equivalent to standard quantum computation". In: *SIAM review* 50.4 (2008), pp. 755–787 (cit. on p. 15).

[BBC+95] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, H. Weinfurter. "Elementary gates for quantum computation". In: *Physical review A* 52.5 (1995), p. 3457 (cit. on p. 31).

[BIS+18] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, C. Blank, K. McKiernan, N. Killoran. "Pennylane: Automatic differentiation of hybrid quantum-classical computations". In: *arXiv preprint arXiv:1811.04968* (2018) (cit. on pp. 32, 40, 64).

[BM03] S. S. Bullock, I. L. Markov. "An Arbitrary Twoqubit Computation In 23 Elementary Gates or Less". In: *Proceedings of the 40th Annual Design Automation Conference*. Association for Computing Machinery, 2003, pp. 324–329. ISBN: 1581136889. DOI: 10.1145/775832.775916 (cit. on p. 31).

[BV97] E. Bernstein, U. Vazirani. "Quantum complexity theory". In: *SIAM Journal on computing* 26.5 (1997), pp. 1411–1473 (cit. on p. 56).

[CBSG17] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta. "Open quantum assembly language". In: *arXiv preprint arXiv:1707.03429* (2017) (cit. on pp. 25, 29, 33, 36).

[Com19] R. Computing. "Pyquil documentation". In: *URL http://pyquil. readthedocs. io/en/latest* (2019) (cit. on pp. 64, 65).

[DJ92] D. Deutsch, R. Jozsa. "Rapid solution of problems by quantum computation". In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558 (cit. on p. 25).

[DKPV20] R. Duncan, A. Kissinger, S. Perdrix, J. Van De Wetering. "Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus". In: *Quantum* 4 (2020), p. 279 (cit. on p. 41).

[DN05] C. M. Dawson, M. A. Nielsen. "The solovay-kitaev algorithm". In: *arXiv preprint quant-ph/0505030* (2005) (cit. on p. 23).

[GC11] J. C. Garcia-Escartin, P. Chamorro-Posada. "Equivalent quantum circuits". In: *arXiv preprint arXiv:1110.2998* (2011) (cit. on p. 56).

[Gro96] L. K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219 (cit. on p. 26).

[Gro98] L. K. Grover. "A framework for fast quantum mechanical algorithms". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 53–62 (cit. on p. 26).

[GZB+04] Y. Guo, Y.-F. Zhang, X.-Y. Bao, T.-Z. Han, Z. Tang, L.-X. Zhang, W.-G. Zhu, E. G. Wang, Q. Niu, Z. Q. Qiu, et al. "Superconductivity modulated by quantum size effects". In: *Science* 306.5703 (2004), pp. 1915–1917 (cit. on p. 27).

[ICK+16] R. Iten, R. Colbeck, I. Kukuljan, J. Home, M. Christandl. "Quantum circuits for isometries". In: *Physical Review A* 93.3 (Mar. 2016). ISSN: 2469-9934. DOI: 10.1103/physreva.93.032318 (cit. on pp. 20, 31, 58, 69).

[JWB03] D. Janzing, P. Wocjan, T. Beth. "Identity check is QMA-complete". In: *arXiv preprint quant-ph/0305050* (2003) (cit. on p. 29).

[KIQ+19] N. Killoran, J. Izaac, N. Quesada, V. Bergholm, M. Amy, C. Weedbrook. "Strawberry fields: A software platform for photonic quantum computing". In: *Quantum* 3 (2019), p. 129 (cit. on p. 37).

[KR03] J. Kempe, O. Regev. "3-local Hamiltonian is QMA-complete". In: *arXiv preprint quant-ph/0302079* (2003) (cit. on p. 29).

[KW19] A. Kissinger, J. van de Wetering. "Pyzx: Large scale automated diagrammatic reasoning". In: *arXiv preprint arXiv:1904.04735* (2019) (cit. on pp. 36, 41).

[LaR19] R. LaRose. "Overview and comparison of gate level quantum software platforms". In: *Quantum* 3 (2019), p. 130 (cit. on pp. 15, 25, 30, 37).

[LB20] F. Leymann, J. Barzen. "The bitter truth about gate-based quantum algorithms in the NISQ era". In: *Quantum Science and Technology* 5.4 (Sept. 2020), p. 44007. DOI: 10.1088/2058-9565/abae7d (cit. on pp. 25, 26, 30).

[LP17] V. Lahtinen, J. K. Pachos. "A short introduction to topological quantum computation". In: *SciPost Physics* 3.3 (2017) (cit. on p. 37).

[MA08] K. Matsumoto, K. Amano. "Representation of Quantum Circuits with Clifford and pi/8 Gates". In: *arXiv preprint arXiv:0806.3834* (2008) (cit. on p. 31).

[MWS+17] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, J. M. Gambetta. "Efficient Z gates for quantum computing". In: *Physical Review A* 96.2 (2017), p. 22330 (cit. on pp. 26, 27, 31, 34, 59).

[NC02] M. A. Nielsen, I. Chuang. *Quantum computation and quantum information*. 2002 (cit. on pp. 15, 17–26, 30, 65).

[NGM20] B. Nash, V. Gheorghiu, M. Mosca. “Quantum circuit optimizations for NISQ architectures”. In: *Quantum Science and Technology* 5.2 (2020), p. 25010 (cit. on p. 31).

[NMI02] J. Niwa, K. Matsumoto, H. Imai. “General-purpose parallel simulator for quantum computing”. In: *International Conference on Unconventional Methods of Computation*. Springer. 2002, pp. 230–251 (cit. on p. 29).

[NRS+18] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, D. Maslov. “Automated optimization of large quantum circuits with continuous parameters”. In: *npj Quantum Information* 4.1 (2018), pp. 1–12 (cit. on p. 29).

[PAFL18] B. Pokharel, N. Anand, B. Fortman, D. A. Lidar. “Demonstration of fidelity improvement using dynamical decoupling with superconducting qubits”. In: *Physical review letters* 121.22 (2018), p. 220502 (cit. on p. 27).

[Pre18] J. Preskill. “Quantum Computing in the NISQ era and beyond”. In: *Quantum* 2 (2018), p. 79 (cit. on p. 26).

[RB01] R. Raussendorf, H. J. Briegel. “A one-way quantum computer”. In: *Physical Review Letters* 86.22 (2001), p. 5188 (cit. on p. 15).

[SBB+20] M. Salm, J. Barzen, U. Breitenbücher, F. Leymann, B. Weder, et al. “The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms”. In: *Communications in Computer and Information Science (CCIS)* (2020) (cit. on pp. 15, 19, 24, 30, 31).

[SBM06] V. V. Shende, S. S. Bullock, I. L. Markov. “Synthesis of quantum-logic circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.6 (2006), pp. 1000–1010 (cit. on p. 58).

[Sca12] V. Scarani. “Quantum Computing: A Gentle Introduction”. In: *Physics Today* 65.2 (Jan. 2012), p. 53. DOI: 10.1063/PT.3.1442 (cit. on pp. 17–20, 22, 24–26).

[Sch16] W. Scherer. *Mathematik Der Quanteninformatik*. Springer, 2016 (cit. on pp. 19, 20).

[SCZ16] R. S. Smith, M. J. Curtis, W. J. Zeng. “A practical quantum instruction set architecture”. In: *arXiv preprint arXiv:1608.03355* (2016) (cit. on p. 64).

[SDC+20] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, R. Duncan. “t| ket>: A retargetable compiler for NISQ devices”. In: *Quantum Science and Technology* (2020) (cit. on pp. 32, 36, 40, 59).

[Sho99] P. W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM review* 41.2 (1999), pp. 303–332 (cit. on p. 15).

[Sin19] K. Singer. “Entwicklung einer generischen Architektur für Quanten-Services in der Cloud”. PhD thesis. 2019 (cit. on p. 37).

[ZPW18] A. Zulehner, A. Paler, R. Wille. “An efficient methodology for mapping quantum circuits to the IBM QX architectures”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2018), pp. 1226–1236 (cit. on pp. 21, 24–28, 31, 67).

All links were last followed on October 29, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature