A Flexible FPGA-based Control Platform for Superconducting Multi-Qubit Experiments

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

von der KIT-Fakultät für Physik des

Karlsruher Instituts für Technologie (KIT)

angenommene

Dissertation

von

M. Sc. Richard Gebauer

Tag der mündlichen Prüfung:	17. Dezember 2021
Referent:	Prof. Dr. Marc Weber
Korreferent:	Prof. Dr. Martin Weides
Betreuer:	PD Dr. Oliver Sander

Contents

1 Introduction **1**

2 Fundamentals **3**
- 2.1 Quantum Computing 3
  - 2.1.1 Quantum Bits 4
  - 2.1.2 Quantum Gates 6
  - 2.1.3 Quantum Registers 7
  - 2.1.4 Quantum Algorithms 9
  - 2.1.5 Quantum Error Correction 13
- 2.2 Superconducting Quantum Bits 14
  - 2.2.1 Superconductivity 14
  - 2.2.2 Two-Level Quantum Systems 16
  - 2.2.3 Rabi Cycles 18
  - 2.2.4 Dispersive Readout 20
  - 2.2.5 Transition Frequency Tuning 22
  - 2.2.6 Example Implementation 22
  - 2.2.7 Multi-Qubit Interaction 23
  - 2.2.8 Quantum Software Tools 25
- 2.3 Typical Experiment Setup 27
- 2.4 FPGAs and Heterogeneous Systems 29
  - 2.4.1 Field-Programmable Gate Array 30
  - 2.4.2 Multi-Processor System-on-Chip 32
  - 2.4.3 Radio Frequency System-on-Chip 34
- 2.5 Summary 35

3 Platform Architecture **37**
- 3.1 Functional Requirements 37
- 3.2 QiController Concept 40
- 3.3 Analog RF Frontend 46
- 3.4 Programmable Logic 49
  - 3.4.1 Digital Unit Cell 49
  - 3.4.2 Wishbone Bus Infrastructure 50

3.4.3 Sequencer 52
3.4.4 Signal Generator 54
3.4.5 Signal Recorder 56
3.4.6 Data Storage 60
3.4.7 Digital Trigger Output 61
3.4.8 Cell Coordinator 62
3.4.9 Cell Multiplexer 63
3.4.10 Platform Information and Management Core 64
3.5 Software Stack 64
3.5.1 ServiceHub 65
3.5.2 Taskrunner 67
3.5.3 Boot and Initialization 70
3.5.4 System Reliability 70
3.6 Scalability 71
3.6.1 Single Platform 71
3.6.2 Multiple Platforms 72
3.7 Characterization Results 76
3.7.1 Resource Utilization 76
3.7.2 Analog Signal Properties 77
3.7.3 Digital Signal Properties 83
3.8 Summary 86

4 Platform Interface and Control 89
4.1 Philosophy 89
4.2 Python Client qiclib 90
4.3 Typical Use Cases 92
4.3.1 Setup and Initialization 92
4.3.2 Pseudo-VNA Measurement 95
4.3.3 Single-Qubit Characterizations 96
4.3.4 Single-Shot Resonator Response Plots 100
4.3.5 Multi-Qubit Characterizations 101
4.4 Experiment Description Language QiCode 103
4.4.1 Concept 104
4.4.2 Available Commands 108
4.4.3 Compilation 112
4.4.4 Execution 115
4.4.5 Custom Data Processing 117
4.4.6 Current Limitations 118
4.5 Qkit Integration 119
4.6 Qiskit Integration 121
4.7 Summary 123

5 Experimental Applications 125
5.1 Fast Feedback Control for State Preparation 125
5.1.1 Motivation and Physics Background 125
5.1.2 Experiment Setup 126
5.1.3 QiController Setup 126
5.1.4 Results and Discussion 127
5.2 Quantum Jumps Measurement 129
5.2.1 Motivation and Physics Background 129
5.2.2 Experiment Setup 130
5.2.3 QiController Setup 130
5.2.4 Results and Discussion 132
5.3 Detecting and Changing the Speed of Light 134
5.3.1 Motivation and Physics Background 134
5.3.2 Experiment Setup 136
5.3.3 QiController Setup 137
5.3.4 Results and Discussion 139
5.4 Simultaneous, Time-Resolved Multi-Qubit Characterizations 139
5.4.1 Motivation and Physics Background 139
5.4.2 Experiment Setup 141
5.4.3 QiController Setup 143
5.4.4 Results and Discussion 147
5.5 Summary 152

6 Conclusion 155

Bibliography 157

List of Publications 171

List of Abbreviations 173

Appendix 177
A Signal Processing 177
A.1 Quadrature Frequency Mixing 177
A.2 Digital Down Conversion 178
A.3 Frequency-Division Multiplexing 179
B Programmable Logic 180
B.1 Sequencer Commands 180
B.2 Signal Generator Timing 181
B.3 Measurement-based Feedback Timing 182
C Taskrunner 182
C.1 Interface to the Taskrunner Firmware 182

C.2 Interleaved Experiment Task for Multiple Qubits 184
C.3 Quantum Jumps Collection Task for Multiple Qubits 187
D Characterization 189
D.1 Phase Noise of Analog Output Signal 189

Acknowledgments 191

1 Introduction

Quantum computers promise to solve specific computational problems much faster than any conventional computer [1]. Their elementary building blocks are two-level systems called quantum bits (qubits) [2]. Like classical bits, they are comprised of two fundamental states, denoted as $|0\rangle$ and $|1\rangle$. Two important effects distinguish qubits from their classical counterparts: superposition and entanglement. The former describes the ability of qubits to be in both fundamental states at the same time. The latter is defined between multiple qubits and means that the states of these no longer have to be independent of each other.

Superposition and entanglement lead to the enormous gain in computing performance for certain problems [3]. Potential applications of quantum computers are manifold, including integer factorization [4], quantum encryption [5], [6], database search [7], simulations of large and complex systems [8], [9], drug research and discovery [10], [11], material science [12], as well as general optimization problems [13] like quantum machine learning [14].

In the last 30 years, different physical realizations of qubits have been proposed. Well-known examples are ion traps [15], [16], individual photons [17]–[19], semiconductor quantum dots [20]–[22], and superconducting circuits [23]–[26]. Recently, especially the latter gained additional momentum due to the increasing interest and research activities of major information technology companies like Google, IBM, and Microsoft [27]. In March 2018, Google presented a superconducting quantum processor with 72 qubits [28]. One and a half years later, they successfully demonstrated quantum advantage [29] on 53 qubits by outperforming a classical supercomputer when sampling the result of a pseudo-random quantum circuit.

Eventually, many more qubits are necessary for a universal quantum computer [30], as they are subject to noise of the environment. Together with intrinsic imperfections, this distorts the qubit states leading to errors [31], [32]. To perform successful calculations, these must be accounted for using quantum error correction. In contrast to classical correction schemes, hundreds to thousands of physical qubits are needed to form a protected, so-called logical qubit [33]. A universal quantum computer will thus need thousands of physical qubits for a few logical ones [30].

Scaling of multi-qubit systems is currently one of the challenges on the path to quantum computation and it is still unclear how the architecture of a future quantum processor will look like. Research towards a universal quantum computer has to consider the full hardware and software stack. An essential component is the interface between quantum algorithm and quantum processor which requires sophisticated control electronics [29], [34].

For superconducting qubits, such electronics needs to cover the generation of microwave signals with frequencies of up to 10 GHz [35], [36], as well as data acquisition and processing of similar signals to read out the state of the qubits [37], [38]. With arbitrary pulse shaping necessary and pulse durations between a few and hundreds of nanosecond [39], [40], this also implies tight timing requirements. Some quantum processors furthermore depend on nanosecond-accurate flux pulses generated by currents with microampere strength [41], [42], or DC bias currents, spanning a broad range from microampere to multiple amperes for some special applications [43], [44]. Advanced experiment and control schemes, as well as online data processing and reduction are necessary to cope with the high input data rates of tens of gigabyte per second and to reach sufficient statistics in reasonable time [45]–[47]. Custom electronics based on a field-programmable gate array (FPGA) can be utilized to meet the high processing demands [48]–[50].

In this thesis, such a control electronics for the interface between quantum and classical processing domains, called QiController, is designed, implemented, and tested. Major objectives for its development have been applicability to different types of superconducting qubits, as well as usability, scalability, and flexibility. It is targeted at advanced research and near-term applications with quantum processors. It also explores concepts to control future quantum processors. To leverage its full capabilities, the QiController is complemented by a high-level programming language, called QiCode.

The QiController is extensively tested with various superconducting single- and multi-qubit chips. The experiments range from single-qubit characterization measurements, over continuous observations of quantum jumps and time-resolved multi-qubit characterizations, to high-precision online state preparations using a fast feedback scheme. Some of these experiments are infeasible or even impossible to implement with commercial laboratory equipment. They demonstrate the capabilities of the QiController, as well as its flexibility, scalability, and usability.

2 Fundamentals

Building electronics for quantum computing in general and superconducting quantum bits (qubits) in particular requires a solid understanding of the physical principles regarding these qubits and belonging measurement concepts. This chapter first introduces the basic principles of quantum computing before elaborating in more detail on its realization using superconducting materials. Then, field-programmable gate arrays (FPGAs) and heterogeneous systems are introduced.

2.1 Quantum Computing

Quantum computers operate fundamentally differently than their classical counterparts. Problems that seem to be impractical to solve on a classical computer might be no challenge at all for a quantum computer [1]. Its power essentially emerges from the laws of quantum mechanics, especially the phenomena of superposition and entanglement [3]. However, quantum computers will most likely not replace classical computers as general-purpose machines. Rather, they can act as complementing accelerators in a heterogeneous computing system for special problems that can be efficiently solved on a quantum computer [51], [52].

Harnessing the quantum mechanical nature of these systems, one area of application relates to the simulation of systems that behave according to quantum mechanical laws [8]. This includes atoms as well as molecules and can, for example, be used for material science [12]. But it also includes more complex structures like proteins and their folding mechanisms which are very complex and resource-intensive to simulate classically [10]. Molecule and protein simulations both offer extensive opportunities in the field of drug research and discovery [11]. Other applications can be formulated as optimization problems which can be efficiently solved on a quantum computer [13]. This includes the area of quantum machine learning [14]. It also covers the simulation of large and complex systems, like solving the Navier-Stokes nonlinear differential equations for a viscous fluid [9]. Yet other applications center around the Grover algorithm to search efficiently through an unstructured database [7] or quantum encryption [5], [6] and Shor's algorithm to perform efficient integer factorization [4]. The latter can have a significant impact when universal quantum computers are available and powerful enough to break the security of modern cryptosystems like RSA [53].

However, right now, these applications are still in the future and it is not well-established when a universal quantum computer with sufficient qubit count will be available. Yet, a few basic conditions that any experimental setup needs to fulfill in order to build a universal quantum computer can already be formulated. These are called the DiVincenzo criteria [54] and require a setup that:

1. is a scalable physical system with well-characterized qubits,
2. has the ability to initialize the qubits into a well-defined state,
3. consists of qubits with long coherence times in comparison to gate operations,
4. supports a universal set of quantum gates, and
5. provides means to reliably measure the qubit states.

The following sections will therefore elaborate on the definition of a qubit, means to control and measure their state, and how they are scaled up in a quantum computing architecture.

2.1.1 Quantum Bits

Classical computers process information in binary form. The unit of information are bits which can only be in one of the two elementary states `0` and `1`, corresponding to two distinct voltage levels V_{GND} and V_{CC}, respectively. Likewise, their quantum computing counterpart, the qubits, have two fundamental states, labeled $|0\rangle$ and $|1\rangle$ [2]. In most cases, these are eigenstates of the Hamilton operator $\hat{H}$ of the system with energy eigenvalues E_0 and E_1.

As qubits behave according to quantum mechanical laws, they can also be in a linear superposition of these two eigenstates:

$$|q\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \quad \text{with } \alpha, \beta \in \mathbb{C},\ |\alpha|^2 + |\beta|^2 = 1 \ . \tag{2.1}$$

α and β are complex-valued coefficients which are only limited by the normalization requirement

$$\langle q|q\rangle := \int \Psi_q^*(x)\Psi_q(x)\,\mathrm{d}x = 1 \ . \tag{2.2}$$

Therefore, the superposition state $|q\rangle$ can be rewritten as follows:

$$|q\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{\mathrm{i}\varphi}\sin\frac{\theta}{2}\,|1\rangle \ . \tag{2.3}$$

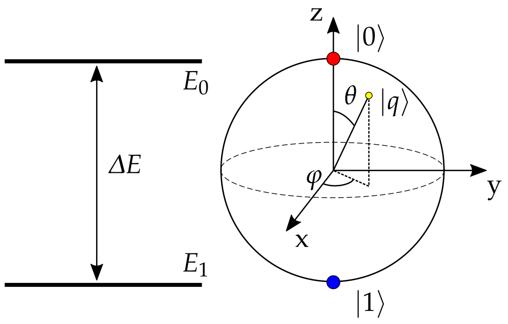


Figure 2.1: Schematic figure of the Bloch sphere, illustrating the possible states of a qubit. Next to it, the two energy levels E_0 and E_1 of the qubit corresponding to states $|0\rangle$ and $|1\rangle$ are shown. [43]

This representation emphasizes that there is a two-dimensional manifold of possible qubit states. These can be depicted as points on a spherical surface as shown in Figure 2.1. This representation is called Bloch sphere and states are depicted by a Bloch vector pointing from the origin onto the sphere's surface [55]. In this representation, the polar angle θ describes the probabilities of the states and the azimuth angle φ expresses the relative phase between them. On the Bloch sphere, θ defines the state component along the z axis and φ describes the component in the x-y plane. Accordingly, state $|0\rangle$ is located at the north pole of the sphere, and $|1\rangle$ on the south pole [56]. Consequently, operations on the qubit are in $\mathrm{SU}(2)$, i.e. rotations around the surface of the sphere.

In a physical system, the qubit is inevitably coupled to the environment. This coupling is necessary to interact with the qubit and manipulate and read out its state. It also leads to noise and decoherence of the qubit state [31]. To characterize these effects and thereby the quality of the qubit, two types of time constants are important: First, the energy relaxation time T_1 which describes the decay time of the higher energy state $|1\rangle$ back to the ground state $|0\rangle$. And second, the decoherence time T_2 which defines the timescale on which the state of the qubit is lost. Besides the energy relaxation, this also includes when the phase φ between both states becomes blurred. Both times are related via a third constant, T_φ, called the pure dephasing time. They can be linked when expressed as decay rates [32]:

$$\frac{1}{T_2} = \frac{1}{2T_1} + \frac{1}{T_\varphi} \quad . \tag{2.4}$$

2.1.2 Quantum Gates

Quantum gates are linear operations used to manipulate the state of a qubit. To be able to perform all possible operations, one needs to define a universal set of gates [57]. Single-qubit gates are rotations on the surface of the Bloch sphere, i.e. can be represented as operations in SU(2) [58]. One possible set of single-qubit gates are rotations around the different basis axes. These are expressed as $R_{x/y/z}(\theta)$ where $x/y/z$ denotes the axis around which the rotation is performed and θ the rotation angle. Special cases of such rotations are the Pauli gates $X/Y/Z$ which are rotations around the respective axis with $\theta = \pi$. The X operation can be compared to a classical not operation, as it flips the qubit state from $|0\rangle$ to $|1\rangle$ and vice versa. In mathematical representation, gates acting on a state are interpreted the same way as matrices being multiplied with a vector. This means that $AB|\psi\rangle$ says that gate B is applied onto the qubit state $|\psi\rangle$ and, afterwards, gate A is applied.

Probably the most popular single-qubit gate is the Hadamard gate H [59]. It is defined by its behavior on the two fundamental states $|0\rangle$ and $|1\rangle$:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad \text{and} \quad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{2.5}$$

creating a superposition state from the two basis states. It can be decomposed to rotations around the Bloch sphere as $H = X\sqrt{Y} = R_x(\pi)R_y(\frac{\pi}{2})$. This is a 90° rotation around the Y axis, followed by a 180° rotation around the X axis. An interesting property of the Hadamard gate is that it is an involution, i.e. $H^2|\psi\rangle = |\psi\rangle$ for all qubit states $|\psi\rangle$ of the computational subspace.

Besides the one qubit gates, a two-qubit gate is necessary to define a universal set of gates [57]. One candidate for such a two-qubit gate is the CNOT or CX gate (conditional not gate) [59]. It acts on two qubits where the first qubit is referred to as control qubit, and the second qubit as target qubit. When the control qubit is in state $|0\rangle$ it leaves the target qubit unchanged. If the control qubit is in state $|1\rangle$, an X operation is performed on the target qubit, thereby flipping its state:

$$\begin{aligned} \text{CNOT}|00\rangle &= |00\rangle \quad , \quad \text{CNOT}|01\rangle = |01\rangle \quad , \\ \text{CNOT}|10\rangle &= |11\rangle \quad , \quad \text{CNOT}|11\rangle = |10\rangle \quad . \end{aligned} \tag{2.6}$$

In this notation, the single qubit states have been summarized, i.e. $|01\rangle \equiv |0\rangle|1\rangle$. A different two-qubit gate is the CZ or CPhase gate. It behaves exactly the same as the CNOT gate but instead of a conditional X rotation, it performs a conditional Z rotation on the target qubit.

Yet another popular two-qubit gate is the iSWAP gate. Contrary to the conditional gates, the iSWAP is symmetric. It swaps the qubit states between two qubits and

Table 2.1: State of the art parameters for different qubit realizations. Values show the highest reported values for qubit lifetime T_2 and two-qubit gate fidelity. [62]–[67]

Qubit type	Lifetime T_2	Gate fidelity	Gate time	Gates per T_2
Superconducting	50 − 100 µs	99.7 %	10 − 50 ns	1000 − 10 000
Trapped ions	1 − 1000 s	99.9 %	3 − 100 µs	10 000 − 10^8
Photons	150 µs	98 %	1 ns	150 000

adds a phase of i to the amplitudes of the $|01\rangle$ and $|10\rangle$ contributions [60]. Its action on the basis states of the two qubits is therefore defined as:

$$\begin{aligned} \text{iSWAP}\,|00\rangle &= |00\rangle \quad , \quad \text{iSWAP}\,|01\rangle = \mathrm{i}\,|10\rangle \quad , \\ \text{iSWAP}\,|10\rangle &= \mathrm{i}\,|01\rangle \quad , \quad \text{iSWAP}\,|11\rangle = |11\rangle \quad . \end{aligned} \tag{2.7}$$

Physically, the iSWAP gate is realized by a resonant exchange interaction $X \otimes X + Y \otimes Y$ between the two qubits [61]. By cutting the interaction time in half, one can furthermore obtain a $\sqrt{\text{iSWAP}}$ gate which is suitable as two-qubit gate for a universal set of gates.

What kind of gates are available depends on the physical implementations of the qubits and the coupling between them. When one elementary two-qubit gate is available, other gates can be constructed as a composition of this two-qubit gate and additional single-qubit gates, similar to classical logic.

Each gate will also take a finite duration to execute. When assessing the quality of a qubit or a quantum processor, it is important to determine how many gate operations can be successfully performed. This has to take into account both the real-world fidelity of a gate implementation, as well as its duration when compared to the decoherence time T_2 of the qubits. Table 2.1 summarizes state of the art parameters for the three most popular qubit realizations to illustrate the current status and scale of these values.

2.1.3 Quantum Registers

Combining multiple qubits together leads to a quantum register. Similar to a classical register with N bits, a quantum register is a collection of N qubits that can hold values ranging from 0 to $2^N - 1$. The single qubit states of a register are grouped together to shorten the necessary notation, e.g.:

$$|0\rangle\,|0\rangle\,|1\rangle\,|1\rangle\,|0\rangle\,|1\rangle\,|1\rangle \equiv |0011011\rangle \equiv |27\rangle \quad . \tag{2.8}$$

As qubits can be in a superposition, a quantum register $(|q_{N-1}\rangle, \ldots, |q_0\rangle)$ can hold all 2^N possible values at the same time. If, for example, all qubits are in an equal superposition of both states, i.e. $|q_n\rangle = H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, one obtains:

$$\begin{aligned} |X\rangle = \frac{1}{2^{N/2}} \prod_{n=0}^{N-1} (|0\rangle + |1\rangle) =& \frac{1}{2^{N/2}} (|0\ldots00\rangle + |0\ldots01\rangle + |0\ldots10\rangle + \ldots) \\ =& \frac{1}{2^{N/2}} \sum_{m=0}^{2^N-1} |m\rangle \quad . \end{aligned} \tag{2.9}$$

This is an equal superposition of all classically possible register states [55].

If one uses this superposition state of the register to calculate the outcome of a classical function, one will thus calculate all possible outcomes at the same time. This can be further leveraged using quantum entanglement. When entangled, the qubit states within the register are no longer independent. An arbitrary, classical function of the form

$$f : \{0, 1, \ldots, 2^N - 1\} \ni m \mapsto f(m) \in \{0, 1, \ldots, 2^N - 1\} \tag{2.10}$$

can then be fully encoded using a register of $2N$ qubits in the following way:

$$|X\rangle_{\text{initial}} = \frac{1}{2^{N/2}} \sum_{m=0}^{2^N-1} |m\rangle |m\rangle \quad \xrightarrow{\mathbb{1}\otimes f} \quad |X\rangle = \frac{1}{2^{N/2}} \sum_{m=0}^{2^N-1} |m\rangle |f(m)\rangle \quad . \tag{2.11}$$

In this operation, the register $|X\rangle$ is initially prepared to represent the same number m twice in the first N and the second N qubits, i.e. $|m\rangle |m\rangle$. That means that the first N qubits of the register are now entangled with the second N qubits. Both parts will always hold the same number when read out, but all possible numbers with equal probability 2^{-N}. Then, the function f is applied once to the second N qubits in the register. The first part of the register stays untouched, still holding the functions input values. Now, the first N qubits encode the possible input values $|m\rangle$ and the second N qubits are entangled to represent the according function results $|f(m)\rangle$. With this mechanism, the whole function f is now encoded inside a $2N$-sized quantum register.

In a classical computation, the function would have been executed for each possible input value, i.e. 2^N times. Therefore, by preparing the quantum register in a superposition of all possible input values an exponential speedup is gained as the function only needs to be executed once. Additionally, while this takes only $2N$ bits in a quantum register, a classical computer would need $N2^N$ bits to store the same function in a lookup table. Therefore, also exponentially less space was needed to store the same amount of information. For algorithms that require all possible function values to be evaluated, this can have a major impact. This is especially the case if a global property of the function should be tested, like in the Deutsch-Jozsa algorithm which is introduced in the next section.

However, this example can also show a bottleneck of quantum computing. If the superimposed state of the register is read out, only one of the possible values will be returned. Moreover, the measurement projects the register to this result and the information of the whole function is lost. Generally speaking, while an N qubit quantum register can hold a large amount of information, a readout of such a register can only extract N classical bits of information. Therefore, it is essential that the register is prepared in a state where it contains the relevant information before it is read out. But measurements being projective is only one challenge for quantum algorithm engineers. Another challenge is the no-cloning theorem which states that it is not possible to copy an arbitrary qubit state to another qubit [68]. Therefore, classical algorithms are mostly not suited to be implemented the same way in a quantum computer.

2.1.4 Quantum Algorithms

This yields for a new class of algorithms, specifically tailored to the capabilities and limitations of quantum computers. One of these is the Deutsch-Jozsa algorithm [69]. It is used to determine a global property of an unknown function $f : \{0,1\}^N \rightarrow \{0,1\}$. For the Deutsch-Jozsa algorithm, the task is to find out if the function is either constant, i.e. has the same output for every input, or balanced, i.e. returns `0` for half of the input values and `1` for the other half. It is guaranteed that f fulfills exactly one of these two global properties. While this problem is of little practical relevance, it was the first quantum algorithm that was proven to solve a specific problem exponentially faster than any classical algorithm can [3].

In a classical computer, solving this problem would require to calculate the outcome of up to one more than half of the 2^N individual input values to be sure that it is really constant. Thus, it will need to execute the function up to $2^{N-1} + 1$ times. The quantum algorithm, on the other hand, only has to perform the calculation once due to a possible superposition of all input values. We thereby gain an exponential speedup in time complexity for this particular problem. Furthermore, the quantum computer can perform the whole algorithm with just $N + 1$ qubits.

The working principle of the quantum algorithm is simple: Instead of checking every single result one by one, the resulting quantum register with the superposition of all results is further transformed. At the end, all qubits of the register will be in state $|0\rangle$ if the function is constant. Otherwise, some or all of the qubits will be in

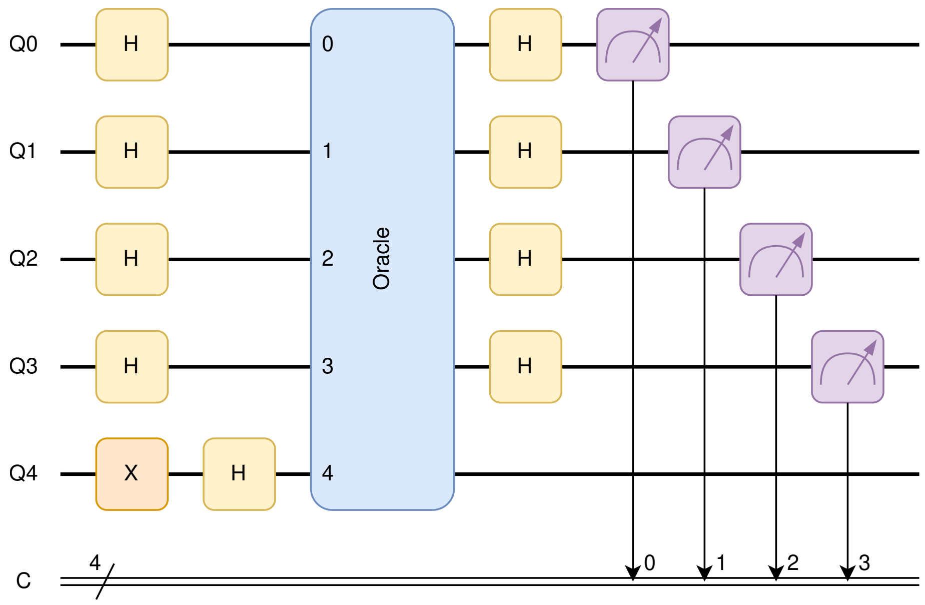


Figure 2.2: Circuit diagram of the Deutsch-Jozsa algorithm acting on $N + 1 = 5$ qubits Q0 to Q4. All qubits are initialized in the ground state $|0\rangle$ at the beginning. Then, different gate operations are applied, including Hadamard (H) and NOT (X) gates. The results of the final measurements (purple measurement blocks) are stored as bits of a classical register C. [70]

state $|1\rangle$. Then, the function is balanced. To determine which outcome is the case, the qubit states are measured once and thereby projected to a classical outcome. The algorithm is designed in a way, that the resulting state in the qubits is pure in both cases. That means that the projection of the measurement will not change the state in the quantum register. Thereby, one measurement is sufficient to determine this global property of the function. It should be noted that, while one learns about this property, one does not get any information about the single function output values [55], as opposed to the classical algorithm.

Algorithms acting on a quantum register are often depicted as circuits consisting of qubits (horizontal lines) and gates (boxes). The time sequence of the gates is the order along the horizontal axis. The circuit diagram of the Deutsch-Jozsa algorithm is shown in Figure 2.2 for $N = 4$ data qubits and one ancilla qubit (Q4). Ancilla qubits are auxiliary qubits to assist during calculations. The data qubits are initialized in an equal superposition of all states by performing a Hadamard gate on each of them. The ancilla qubit is first prepared in the $|1\rangle$ state by performing a X gate, before also being prepared in a superposition by a Hadamard gate. The oracle is the function implemented in a specific way. Let $|x\rangle = |q_0q_1q_2q_3\rangle$ be the data qubits and $|y\rangle = |q_4\rangle$ the ancilla qubit, then the oracle performs the following operation:

$$|x\rangle\,|y\rangle \overset{\text{oracle}}{\longrightarrow} |x\rangle\,|y \oplus f(x)\rangle \quad . \tag{2.12}$$

The operation $\oplus$ denotes the addition with modulo 2. This entangles the data qubits with the ancilla qubit:

$$\frac{1}{\sqrt{2^4}}\sum_{m=0}^{2^4-1}|m\rangle\,\frac{|0\rangle-|1\rangle}{\sqrt{2}} \overset{\text{oracle}}{\longrightarrow} \frac{1}{\sqrt{2^4}}\sum_{m=0}^{2^4-1}|m\rangle\,\frac{|0\oplus f(m)\rangle-|1\oplus f(m)\rangle}{\sqrt{2}} \quad . \tag{2.13}$$

As an effect, it will add a negative global phase to all states $|m\rangle$ in the data qubits where $f(m) = 1$, but not for those with $f(m) = 0$:

$$\frac{1}{\sqrt{2^4}}\sum_{m=0}^{2^4-1}(-1)^{f(m)}\,|m\rangle\,\frac{|0\rangle-|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2^4}}\sum_{m=0}^{2^4-1}(-1)^{f(m)}\,|m\rangle\,|q_4\rangle \quad . \tag{2.14}$$

Afterwards, a second Hadamard operation is performed on the data qubits. If no negative phases have been collected, this will return all zeros again. The same holds true if a negative phase was collected for all states, as it then only is a global constant in the state, i.e. $(-1)^{\text{const.}} = \pm 1,\ \text{const.} \in \{0,1\}$ for all contributions:

$$\frac{\pm 1}{\sqrt{2^4}}\sum_{m=0}^{2^4-1}|m\rangle\,|q_4\rangle \overset{[H]^4\otimes\mathbb{1}}{\longrightarrow} \pm\left[\prod_{n=0}^{4-1}H\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right]|q_4\rangle = \pm\left[\prod_{n=0}^{4-1}|0\rangle\right]|q_4\rangle \quad . \tag{2.15}$$

In contrast, if negative phases have been collected in exactly half of the cases, the state will be orthogonal to the state before the oracle:

$$\langle x,y|x,y\oplus f(x)\rangle = \frac{1}{2^4}\sum_{n=0}^{2^4-1}\sum_{m=0}^{2^4-1}(-1)^{f(m)}\,\langle n|m\rangle\,\langle q_4|q_4\rangle = \frac{1}{2^4}\sum_{m=0}^{2^4-1}(-1)^{f(m)} = 0 \quad . \tag{2.16}$$

Thus, in the constant case, the oracle keeps the data qubits untouched with exception of a non-measurable, global phase. In the balanced case, the data qubits will be orthogonal to the initial state and thereby also to the resulting state of the constant case. After applying the Hadamard gates in this balanced case, one obtains a new state $|\Psi\rangle$ of the data qubits. In the constant case, one obtains $\pm\,|0000\rangle$ as was shown. As the states after the oracle are orthogonal for both cases, this also applies for the states after the Hadamard gates, i.e. $\langle 0000|\Psi\rangle = 0$. Thus, the probability to measure all zeros will vanish for the balanced case [71]:

$$P(|0000\rangle) = |\langle 0000|\Psi\rangle|^2 = 0 \quad . \tag{2.17}$$

To summarize, after the Deutsch-Jozsa algorithm, one measurement of the data qubits is enough to determine if the investigated function is balanced (all qubits are in $|0\rangle$) or not (at least one qubit is in $|1\rangle$). This is achieved by applying the information of the function using the oracle onto the ancilla qubit. However, the ancilla qubit itself is not further investigated after the oracle. Instead, the operations will lead to a so-called phase kickback within the data qubits which is depending on the properties of the function. This is a common technique used in many quantum algorithms [71].

Grover's algorithm [7] is another well-known quantum algorithm which performs quantum searching. It provides quadratic speed up over a classical computer when searching through a large set of unstructured data. The underlying principle is called amplitude amplification technique and can also be used as part of a variety of other algorithms [72]. In essence, an equal superposition of all database entries is created in a quantum register. Subsequently, however, only the amplitudes of the relevant entries are amplified while the contributions of all other entries are reduced.

But the most famous quantum algorithm is probably Shor's algorithm [4] due to its relevance in cryptography. For a long time, it was assumed that the factorization of a composite number of two large primes cannot be performed in polynomial time. Composite integers are the product of two smaller integers. The security of many public key cryptography systems rely on this assumption. One of them is the RSA cryptosystem [53]. It is widely used for secure data transmission. While the assumption may still hold for classical computers, Shor's algorithm provides the means to factorize composite integers in polynomial time on a quantum computer. In future, quantum computers might thus be able to break these cryptosystems.

Shor's algorithm consists of two parts, one part which is performed on a classical computer and a quantum calculation. The classical part reduces the number factoring problem to the problem of order-finding. The latter is equivalent to finding the period of a function and can be efficiently solved by a quantum computer using a quantum Fourier transform. The result is then again post-processed classically to find one of the prime factors. Shor's algorithm is therefore a good example of a quantum computer acting as complementing accelerator in a heterogeneous computing system. The algorithm is furthermore probabilistic as, with a small probability, it will not return a prime factor. This is due to the fact that the algorithm relies on randomly picking a number as initial input. By repeating the algorithm multiple times with different inputs, the probability to not obtain a result is exponentially suppressed [73].

2.1.5 Quantum Error Correction

In a classical computer, errors can happen in the form of bit flips, i.e. the value of a bit changing unintentionally between `0` and `1`. For qubits, the energy relaxation time T_1 would be the typical time scale on which such bit flip errors occur. But also other types of errors can occur due to the ability of arbitrary superpositions. Therefore, also the decoherence time T_2 is an important figure. For most qubit realizations, these times are only a few magnitudes larger than the typical time to perform a gate operation [1]. Therefore, errors happen regularly during the calculation with such a system and an error correction mechanism is necessary.

Classical error correction is mostly handled by redundantly copying the information and performing checksum calculations to determine if a bit has flipped in one of the copies [74]. In quantum computation, this is not possible. The qubits can stay in arbitrary superpositions of the two fundamental states, but reading out the state will destroy this superposition and return only one of the two alternatives. More generally speaking, it is even impossible to copy the state of a qubit, which is also formulated as the no-cloning theorem [68]. Thus, classical error correction schemes cannot be applied.

Instead, entanglement is used to create an error correction code. The information of one logical qubit will be encoded in the entangled state of multiple physical ones, e.g. with three qubits [75]:

$$|q\rangle_\mathrm{L} = \alpha\,|0\rangle_\mathrm{L} + \beta\,|1\rangle_\mathrm{L} = \alpha\,|000\rangle + \beta\,|111\rangle \quad . \tag{2.18}$$

Now, if a bit flip occurs on one of these qubits, one can detect this error using syndrome measurements. By just comparing if the physical qubits are the same, but not checking the actual state they are in, one will not destroy the superposition but only determine if an error has happened or not. By performing multiple of these syndrome measurements, one can directly infer the qubit that has flipped and perform an appropriate correction gate. Furthermore, a continuous error on the Bloch sphere is projected onto just two discrete outcomes, i.e. state flipped or not. The correction algorithm used for this simple case is the bit-flip-code [75]. As the name suggests, it can only correct bit-flip-like errors within the logical qubit. A more sophisticated code which can fully correct a single qubit is the Shor code, which requires nine physical qubits to create one logical one [76].

A more general class of error correction codes are so-called stabilizer codes [77], [78]. In this case, the data qubits are appended by additional ancilla qubits. Then, one generates a highly entangled, encoded state for the logical qubit which corrects for local noisy errors. Regular, repetitive measurements of the ancilla qubits will stabilize the encoded quantum information. When an error happens, it is detected by the ancilla qubits and can be tracked or even corrected. One example of such a code is the surface code [79], [80].

2.2 Superconducting Quantum Bits

Superconducting qubits are artificial atoms built from superconducting materials. For a better understanding, first some basic concepts of superconductivity are given, before the implementation of superconducting qubits is covered, as well as readout and control mechanisms.

2.2.1 Superconductivity

First observed by Kamerlingh Onnes in 1911 [81], superconducting materials completely drop their electrical resistance below a critical temperature T_C. Also, multiple other effects have been observed in superconductors which cannot be explained classically. This includes the Meissner effect where an external magnetic field is completely expelled from the interior of the superconductor or phenomenons like flux quantization [82].

It was only in 1957, that John Bardeen, Leon Cooper and John Robert Schrieffer (BCS) proposed a microscopic theory that was able to explain these behaviors [83]. According to this BCS theory, two electrons attract each other via the exchange of a virtual phonon, the quantum of lattice excitations in solids. With thermal fluctuations becoming weaker at lower temperature, two electrons can form a bound state, a so-called Cooper pair. These pairs do not obey the Pauli exclusion principle anymore and thus can all occupy the same ground state similar to a Bose-Einstein condensate. The binding energy of the two electrons also leads to an energy gap within the spectrum between this bound state and the state of free electrons. If scattering processes cannot transfer enough energy to break these Cooper pairs, scattering will be suppressed. Cooper pairs can then travel without friction through the superconducting material. As these consist of two electrons and are therefore charge carriers, a current can flow without any resistance but is limited by a critical current threshold. Above it, the Cooper pairs have enough kinetic energy to participate in scattering processes leading to non-vanishing electrical resistance [84].

In the common ground state of the Cooper pairs, the whole superconductor is described by a single, global wave function

$$\Psi(\boldsymbol{x}) = |\Psi(\boldsymbol{x})| \cdot \mathrm{e}^{\mathrm{i}\varphi(\boldsymbol{x})} \quad . \tag{2.19}$$

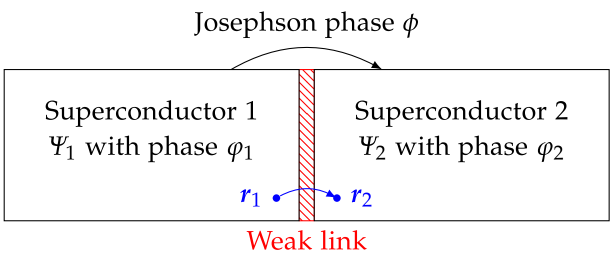


Figure 2.3: Schematic structure of a Josephson junction. [43]

This wave function can also be used to explain other effects observed in superconductors. One of these is the Josephson effect [85] which describes the behavior of two superconducting electrodes connected via a weak link called Josephson junction (see Figure 2.3). Each electrode has its own wave function Ψ_i due to the weak link, but both are related to each other via the Josephson phase

$$\phi = \varphi_2 - \varphi_1 - \frac{2\pi}{\Phi_0} \int_{r_1}^{r_2} \boldsymbol{A}\, \mathrm{d}\boldsymbol{r} \ . \tag{2.20}$$

Here, φ_i are the phases of the wave functions Ψ_i and $\Phi_0 = h/2\mathrm{e}$ is the superconducting flux quantum.

The Josephson phase is directly linked to observables via the Josephson equations [84] which describe the junction behavior:

$$U(t) = \frac{\Phi_0}{2\pi} \frac{\partial \phi}{\partial t} \tag{2.21}$$

$$I(t) = I_\mathrm{c} \sin(\phi(t)) \tag{2.22}$$

where I_c is the critical current through the junction. It is the highest possible current that can flow through the junction without causing any dissipation. As seen from the first equation, an applied voltage $U(t)$ across the junction will lead to a constant drift in the Josephson phase. Vice versa, a change in ϕ will also induce a voltage. By the second equation, the Josephson phase is directly linked to the current flowing through the junction $I(t)$. Following from these equations, one can calculate the inductance $L = U / \frac{\partial I}{\partial t}$ across the junction [31]:

$$L(\phi) = \frac{\Phi_0}{2\pi I_\mathrm{c} \cos(\phi)} \ . \tag{2.23}$$

As this inductance depends on the phase ϕ across the junction, it can be utilized as a non-linear element to design specific circuits and corresponding energy structures. One example are superconducting qubits.

2.2.2 Two-Level Quantum Systems

A fundamental concept of quantum mechanics is the discretization of energy. Systems like atoms, molecules, or the quantum harmonic oscillator all show discrete energy states in which they can be situated. For atoms, these can be different electron populations of the orbitals. For molecules, one can additionally observe discrete vibrational and rotational modes. A quantum harmonic oscillator shows equally spaced energy levels $E_n = \hbar\omega(n + \frac{1}{2})$ corresponding to different excitation modes.

To build a qubit, only two states $|0\rangle$ and $|1\rangle$ are needed to form the computational subspace: a ground state $|0\rangle$ with energy E_0 and an excited state $|1\rangle$ with energy $E_1 > E_0$. Spin 1/2 particles in a magnetic field, for example, fulfill this condition [2]. In general, also multi-level systems can be used when the different levels can be addressed independently. This requires that the anharmonicity $\alpha = \Delta E_{1,2} - \Delta E_{0,1}$ is large enough. One example would be Rydberg atoms where two distinguished energy states are picked from the spectrum to define the required computational subspace [86], [87]. In contrast, the harmonic oscillator is not suited for this, as $\alpha = 0$ and one cannot distinguish between the different energy levels.

Nevertheless, artificial two-level quantum systems are often based on a harmonic oscillator where a source of anharmonicity is added in order to reach the desired energy spectrum. For superconducting qubits, a simple implementation is a charge qubit [23]. It is constructed out of a classical LC harmonic oscillator where the inductance is replaced by a Josephson junction acting as non-linear element. The state of the charge qubit is determined by the number of Cooper pairs that have tunneled across the junction. It is therefore also quite sensitive to charge noise which limits its usability [88]. To alleviate this, different improved designs have been proposed, with the Transmon qubit currently being the most popular one.

Transmon qubits are charge qubits shunted by a large capacitance [24]. This leads to a suppression of the system's susceptibility to charge noise. The first Transmon qubit was introduced by Koch *et al.* A sketch of it is depicted in Figure 2.4. Fluctuations in the charge will not significantly influence the qubit properties anymore due to the large capacitance. The effective Hamiltonian $\hat{H}$ stays the same as for the charge qubit and is given by

$$\hat{H} = 4E_C(\hat{n} - n_g)^2 - E_J \cos\hat{\phi} \tag{2.24}$$

where $\hat{n}$ is the number operator of the Cooper pairs tunneling through the junction, n_g is the effective offset charge induced by an external gate voltage V_g and $\hat{\phi}$ is the operator of the Josephson phase which corresponds to the gauge-invariant phase difference between the two superconductors. The charging energy is defined as $E_C = (2\mathrm{e})^2/2C$ and the Josephson energy as $E_J = \Phi_0 I_c/2\pi$.

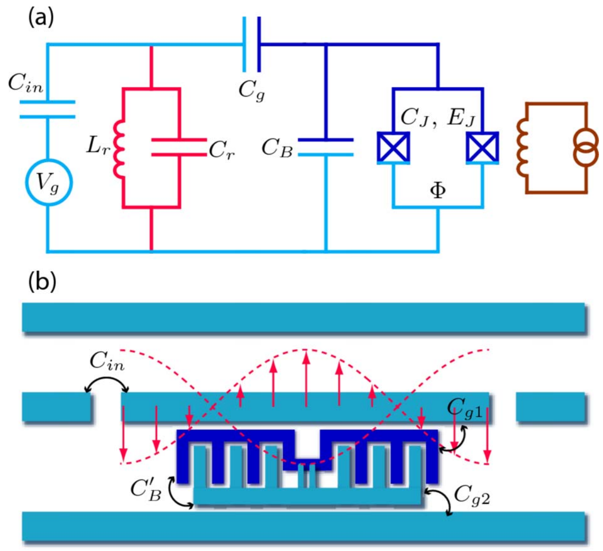


Figure 2.4: (a) Effective circuit of the original Transmon qubit by Koch *et al.* (b) Design schematics of the same Transmon qubit. The single Josephson junction is replaced by a superconducting quantum interference device (SQUID) so the qubit frequency can be tuned by an external flux. The coil and the belonging current source to create this flux are sketched in brown on the right side of the SQUID. The crossed boxes in the SQUID loop represent the Josephson junctions characterized by the intrinsic capacitance C_J and the Josephson energy $E_J \propto I_c$. Clearly visible is the large interdigital capacitor C_B that shunts the charge qubit. The qubit is capacitively coupled (C_g) to a transmission line resonator (L_r and C_r, sketched in red). The resonator is coupled to the transmission line capacitively via C_{in}. The system can be controlled by applying an external gate voltage V_g. [24]

In contrast to the charge qubit, for the Transmon qubit it holds $E_J \gg E_C$. With increasing $\sqrt{E_J/E_C}$, the sensitivity to charge noise, i.e. fluctuations in $\hat{n}$, exponentially decreases. At the same time, the anharmonicity $\alpha \approxeq -E_C$ also decreases. In order to obtain a useful qubit, the anharmonicity needs to remain sufficiently large, $\alpha/h \gtrsim 100\,\text{MHz}$. Otherwise the system cannot be approximated as a two-level system anymore and higher levels play a significant role in the dynamics of the structure. Thus, a trade-off needs to be found between anharmonicity and the suppression of charge noise [24].

Another type of improved qubit is the Fluxonium [25]. Instead of a large capacitance, the charge qubit is shunted by a large inductance. As both sides of the junction are now electrically connected, charge noise is also effectively suppressed. For the Fluxonium qubit, the inductance is realized by a series array of Josephson junctions, see Figure 2.5. Using this technique, high energy relaxation times T_1 of up to 1 ms have been achieved [89], [90]. Instead of fabricating a series of junctions, one can also use granular aluminum for the inductance [35]. It exhibits similar behavior due

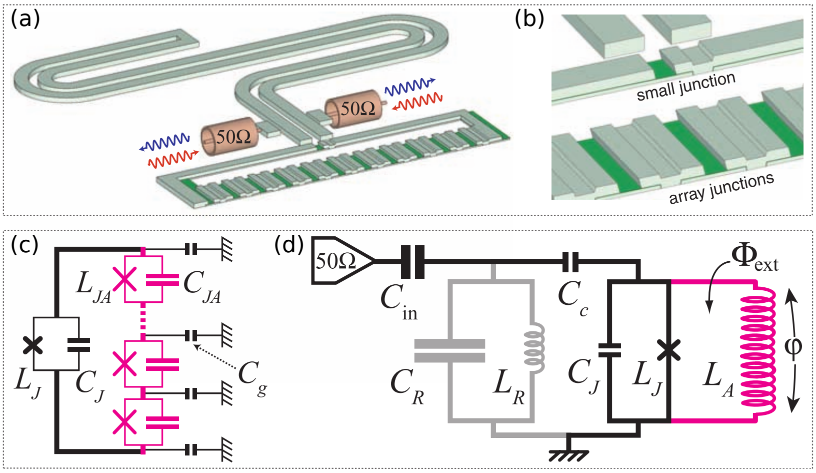


Figure 2.5: (a) Sketch of a Fluxonium qubit coupled to a readout resonator and a 50 Ω transmission line. (b) Structure of the series array of many Josephson junctions forming the large inductance. (c) Electric circuit representation of the qubit with the array of junctions. (d) Simplified circuit model of the Fluxonium qubit being a inductively-shunted charge qubit. The circuit also contains the readout circuitry (marked in gray). [25]

to the granular structure but is easier to fabricate. As the large inductance and the single Josephson junction form a loop, the qubit is tunable by an external magnetic flux Φ_{ext}. This also becomes obvious when looking at the Hamiltonian [25]

$$\hat{H} = 4E_C\hat{n}^2 + \frac{1}{2}E_L\hat{\varphi}^2 - E_J\cos\left(\hat{\varphi} - 2\pi\frac{\Phi_{\text{ext}}}{\Phi_0}\right) \quad . \tag{2.25}$$

On the downside, this dependency makes the qubit sensitive to flux noise. A mitigation strategy is to apply an external field to operate the qubit in the $\Phi_{\text{ext}}/\Phi_0 = \pm 0.5$ sweet spot. There, the spectrum is first-order insensitive to flux-noise. Furthermore, energy relaxation due to non-equilibrium quasi-particles tunneling through the Josephson junction is suppressed [91].

2.2.3 Rabi Cycles

The states of an atom can be probed and populated using light-matter interaction. To excite an atom, i.e. transferring an electron to a higher state, a photon with the energy difference of the old and the new state can be absorbed:

$$\omega_{01} = \frac{E_1 - E_0}{\hbar} \quad . \tag{2.26}$$

This scenario is well-studied in quantum optics. It is described by the Jaynes-Cummings model [92] where the Hamiltonian is split into three terms:

$$\hat{H} = \hbar\omega_{\mathrm{r}}\hat{a}^{\dagger}\hat{a} + \frac{\hbar}{2}\omega_{01}\hat{\sigma}_z + \hbar g\left(\hat{\sigma}^{+}\hat{a} + \hat{\sigma}^{-}\hat{a}^{\dagger}\right) \quad . \tag{2.27}$$

In this model, the atom is located inside a cavity. The cavity is a harmonic oscillator represented by the first term and its resonance frequency ω_{r}. Its creation and annihilation operators $\hat{a}^{\dagger}$ and $\hat{a}$, respectively, create and destroy a single photon in the cavity. Each photon contributes $\hbar\omega_{\mathrm{r}}$ energy to the system. The first term therefore expresses the total energy of the cavity, with $\hat{N}_{\mathrm{r}} = \hat{a}^{\dagger}\hat{a}$ being the number operator of photons in the cavity. The second term describes an atom consisting of two energy levels. It thus can be represented like a spin using Pauli operators. When projected, $\hat{\sigma}_z$ returns either $+1$ or -1. Thus, the energy difference of the two states is given by $\hbar\omega_{01} = E_1 - E_0$. Finally, the third term describes the interaction between atom and cavity with the constant g defining the coupling strength. The operators $\hat{\sigma}^{+/-}$ excite and de-excite the qubit. The term can thus be interpreted in the following way: The qubit transitions from $|0\rangle$ to $|1\rangle$ (corresponding to $\hat{\sigma}^{+}$) when a photon from the cavity is absorbed ($\hat{a}$). And it transitions back from $|1\rangle$ to $|0\rangle$ (corresponding to $\hat{\sigma}^{-}$) by emitting a photon back into the cavity ($\hat{a}^{\dagger}$).

This scheme also applies for artificial atoms like superconducting qubits. Typical resonance frequencies $f_{\mathrm{r}} = \omega_{\mathrm{r}}/2\pi$ and qubit transition frequencies $f_{01} = \omega_{01}/2\pi$ are located around a few gigahertz and thus in the microwave domain.

To alter the state of the qubit, a microwave pulse at drive frequency ω_{dr} and amplitude $A \propto \omega_{\mathrm{A}}$ can be applied. This will result in so-called Rabi cycles where the qubit is periodically excited and de-excited. When starting in the ground state $|0\rangle$, the probability to end up in the excited state $|1\rangle$ after a well-defined time t is given by:

$$P(|1\rangle) = \left(\frac{\omega_{\mathrm{A}}}{\Omega}\right)^2 \frac{1 - \cos(\Omega t)}{2} \tag{2.28}$$

with Rabi oscillation frequency $\Omega = \sqrt{(\omega_{\mathrm{dr}} - \omega_{01})^2 + \omega_{\mathrm{A}}^2}$ [93]. For the special case $\omega_{\mathrm{dr}} = \omega_{01}$, i.e. driving the qubit on resonance, this simplifies to:

$$P(|1\rangle) = \frac{1 - \cos(\omega_{\mathrm{A}} t)}{2} = \sin^2\left(\frac{\omega_{\mathrm{A}} t}{2}\right) \quad . \tag{2.29}$$

By varying amplitude and length of the microwave pulse, one can periodically drive oscillations between the qubit's $|0\rangle$ and $|1\rangle$ state population. This corresponds to rotations around the Bloch sphere, as introduced in Section 2.1.1. E.g. fixing a certain amplitude ω_{A} and choosing the time t such that $t = \pi/\omega_{\mathrm{A}}$, or vice versa, one obtains a π rotation.

Furthermore, the phase of the drive tone will determine the axis on the equator of the Bloch sphere around which the rotation takes place [94]. Typically, one arbitrarily chooses one phase which will correspond to rotations around the x axis, e.g. the phase of the first drive pulse. Then, a relative phase of $\pi/2$ with respect to this reference will lead to rotations around the y axis. A phase offset of π corresponds to counter clockwise rotations around the x axis, and so on. As the phase reference is only an arbitrary definition, one can also perform so-called virtual Z rotations [94]. That means that one intentionally changes the phase reference by a well-defined angle $-\eta$. This corresponds to a rotation of the reference frame by this angle around the z axis. As, obviously, the qubit state does not change by this redefinition, from the view of the adapted reference frame, the qubit has now itself rotated around the z axis by the angle $+\eta$. This can be used to perform error-free and instant rotations around the z axis. As this $R_z(\eta)$ rotation does not require physical interaction with the qubit, it is called a virtual gate.

2.2.4 Dispersive Readout

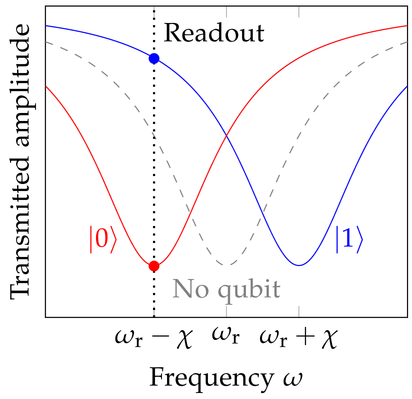
When a qubit is coupled to a harmonic oscillator, the state of the qubit will affect the properties of the oscillator. Transforming the Hamiltonian in Equation 2.27 using a unitary transformation and neglecting fast oscillating terms, one obtains an effective Hamiltonian which is diagonalized and time-independent. The approximation is a common element in quantum optics and called rotating wave approximation [95]. The diagonalized Hamiltonian can be written as

$$\hat{H}_{\text{eff}} \approx \frac{\hbar}{2}(\omega_{01} + \chi)\sigma_z + \hbar(\omega_\text{r} + \chi\hat{\sigma_z})\hat{a}^\dagger\hat{a} \tag{2.30}$$

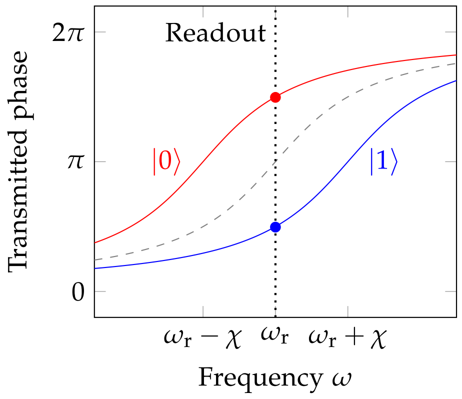
with dispersive shift $\chi = g^2/\Delta$ and detuning $\Delta = \omega_{01} - \omega_\text{r}$, $|\Delta| \gg g$ [37].

The second term represents the cavity. Due to the coupling, a shift in the resonance frequency $\omega_\text{r} \to \omega_\text{r} \pm \chi$ is observed. Its sign depends on the qubit state as given by $\hat{\sigma_z}$. Therefore, by observing the cavity's resonance frequency $f_\text{r} = \omega_\text{r}/2\pi$ one can infer the state of the qubit. This is also depicted in Figure 2.6.

In an experiment setup, one will most likely operate with fixed frequency pulses. Thus, the amplitude and phase response of such a pulse has to be evaluated. Depending on the resonator, the experiment setup, and the frequency used, the state can either be mostly encoded into an amplitude difference (when pulsing at frequency $\omega_\text{r} \pm \chi$) or a phase difference (using ω_r). In Figure 2.6, this is visualized as dotted black lines. As the dispersive shift χ is usually much smaller than depicted, it is typically beneficial to measure the phase. The first derivative of the amplitude vanishes around the resonance frequency which gives reduced sensitivity to small



(a) Amplitude response of the resonator.



(b) Phase response of the resonator.

Figure 2.6: Schematic picture showing the qubit chip response as a function of the frequency. Depending on the qubit state, the resonance frequency of the coupled resonator is shifted by plus (blue line, state $|1\rangle$) or minus (red line, state $|0\rangle$) the dispersive shift χ. The response of the bare resonator without coupled qubit is indicated as dashed gray line. (a) Amplitude of a signal transmitted through the cryostat with a Lorentzian-shaped dip at the resonance frequency. (b) Phase of the transmitted signal.

frequency changes. Contrary, the first derivative of the phase exhibits an extremum at this point which makes it better suited to observe the qubit state [37].

Performing measurements will project the qubit state to either state $|0\rangle$ or $|1\rangle$. Thus, one can retrieve two discrete phase values, φ_0 or φ_1, corresponding to these two qubit states. Depending on the measurement setup, the noise on the signal might exceed the difference of these values and averaging becomes a necessity. In this case, one cannot infer the qubit state for a single measurement outcome. By repeating the experiment and averaging, one obtains an approximation of the qubit population

$$P(|1\rangle) = \frac{\langle\varphi\rangle - \varphi_0}{\varphi_1 - \varphi_0} \quad . \tag{2.31}$$

If the qubit stays in the detected state after the measurement, one furthermore calls the operation a quantum non-demolition (QND) measurement [96]. This is a special type of detection scheme that minimizes the disturbance introduced by the measurement process. It is an important concept in quantum physics. The dispersive readout scheme is a QND measurement, as long as the power of the readout tone is sufficiently small to not cause adverse side effects [38].

2.2.5 Transition Frequency Tuning

Depending on the design of the superconducting qubits, their transition frequency f_{01} can either be fixed or influenced externally. Generally speaking, it is advantageous to have fixed-frequency qubits as otherwise additional noise channels can significantly increase qubit decoherence. However, depending on the experimental requirement and the realization of quantum gates, frequency tunability can sometimes be required, e.g. for an implementation of a two-qubit gate or to perform a physical Z rotation.

To be able to change the frequency, and equivalently the energy difference between $|0\rangle$ and $|1\rangle$, one can replace the Josephson junction by a split-pair geometry, i.e. two parallel ones. These then form a loop in the circuit which is equivalent to a superconducting quantum interference device (SQUID). SQUIDs are very sensitive to magnetic flux passing through the loop due to flux quantization of the superconducting material. The properties of the junctions are then a function of external magnetic flux Φ_{ext} in this loop, and so is the transition frequency of the qubit. For Transmon qubits, this theoretically results in [97]:

$$f_{01}(\Phi_{\text{ext}}) \propto \sqrt{\left|\cos\left(\frac{\pi\Phi_{\text{ext}}}{\Phi_0}\right)\right|} \,. \tag{2.32}$$

However, due to fabrication variations, the Josephson junctions in the SQUID loop will not be perfectly identical. This will result in a slightly deviating dependency with a reduced tunability range not reaching zero frequency. More details can e.g. be found in the supplementary material of [41].

By placing a bias current line close to the SQUID loop and applying a DC current through it, a controlled magnetic field can be created in the vicinity of the qubit. Thereby, the transition frequencies of individual qubits can be tuned independently by adapting these bias currents. Taking the Biot-Savart law into account, it is clear that there is a linear dependency between the current through the bias line and the magnetic flux Φ within the SQUID loop.

2.2.6 Example Implementation

To illustrate typical scales for properties of superconducting qubits, one qubit is presented in the following as an example. It is a concentric Transmon qubit that has been designed and fabricated at the National Institute of Standards and Technology (NIST) in Boulder in the United States. The qubit properties mentioned in the following are taken from characterizations performed in [98]. A microscopic

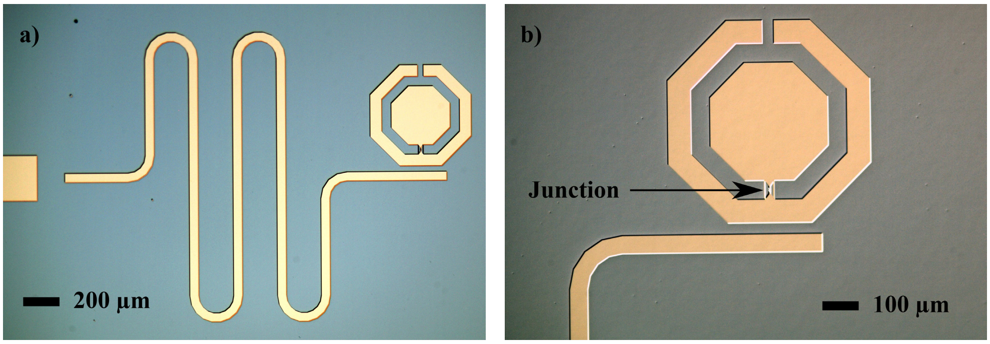


Figure 2.7: Microscopic pictures showing a concentric Transmon qubit. Light areas are superconducting TiN layers. The remainder of the chip shows the silicon substrate. (a) Overview over the whole sample. The meandered structure in the center is a $\lambda/2$ transmission line resonator used to dispersively read out the qubit state. It is capacitively coupled to the qubit and the input port on the left. (b) More detailed view of the Transmon qubit showing two capacitor islands that are connected by a single Josephson junction. [99]

photograph of the sample chip is shown in Figure 2.7. Its concentric design is advantageous to reduce the dipole moment of the structure and thereby the coupling to the environment via an electric field [41], [99]. The sample only has a single microwave port, hence all measurements are performed in reflection. A readout resonator is capacitively coupled to this port and the qubit. Its resonance frequency is $f_\mathrm{r} = 8.573\,\mathrm{GHz}$. The qubit is controlled via microwave pulses over the same port. Its transition frequency is $f_{01} = 4.755\,\mathrm{GHz}$. As the resonator acts as a band pass filter and mostly reflects the control pulses, their amplitude needs to be much stronger than the one for the readout [100]. The Transmon qubit is not a perfect two-level system. Indeed, it also shows higher energy levels. The quality of the qubit is characterized by its anharmonicity, i.e. the energy difference between the $|0\rangle - |1\rangle$ and the $|1\rangle - |2\rangle$ transition. In this case, it is $\alpha/h = f_{12} - f_{01} = -197.7\,\mathrm{MHz}$. This is large enough to reliably distinguish the different energy levels but still leads to some leakage into higher states due to thermal excitations [43]. The coherence times of the qubit show fluctuations in time [36] with values up to $T_1 \approx 30\,\mu\mathrm{s}$ and $T_2 \approx 50\,\mu\mathrm{s}$. The duration for an X gate, i.e. half a rotation around the Bloch sphere, is around 100 ns, depending on the strength of the control pulses.

2.2.7 Multi-Qubit Interaction

Different mechanisms to implement two-qubit gates with superconducting circuits exist. Which one can be used essentially depends on the physical design of the chip

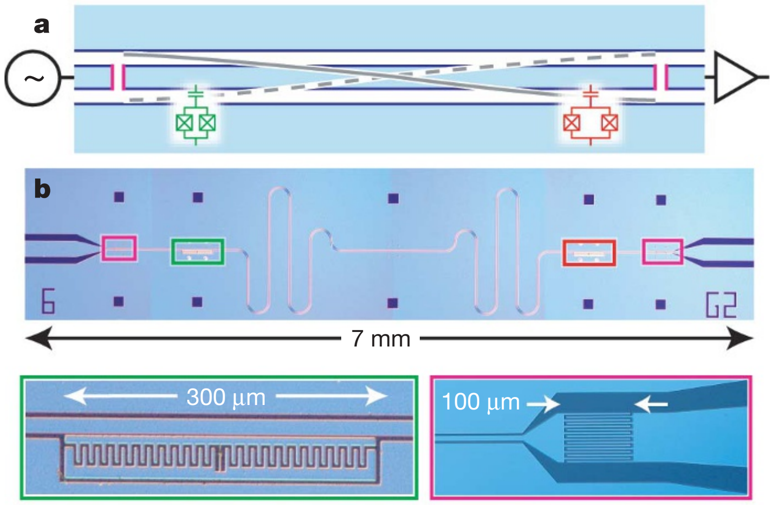


Figure 2.8: Two frequency-tunable Transmon qubits coupled to a common resonator acting as quantum bus. (a) Schematic representation of the sample. (b) Microscopic picture of a chip. The resonator is a coplanar waveguide interrupted by coupling capacitors (purple boxes). Two Transmon qubits are located at opposite ends of the resonator where the electric field shows maximum displacement and thus high coupling to the qubits. [97]

employed. One class is based on tuning the qubit's transition frequencies. Another one is all microwave induced [101].

For the first type, one requires frequency-tunable qubits which are coupled to a common resonator. An exemplary image of such an architecture is depicted in Figure 2.8. In this case, the resonator can act as a quantum bus where the qubit-qubit interaction is mediated by the exchange of a virtual photon [97]. Different gate schemes exist. By tuning the qubit's transition frequencies into resonance, they start to periodically exchange their excitations between $|01\rangle$ and $|10\rangle$. Carefully choosing the interaction time, i.e. the time of both qubits being in resonance, one can create an iSWAP gate. Alternatively, by cutting the time in half, one obtains a $\sqrt{\text{iSWAP}}$ gate [102].

Another gate that can be implemented with superconducting Transmon qubits is a cPhase gate. It leverages that the Transmon is no perfect two-level system but also has higher states. When adiabatically tuning the $|0\rangle - |1\rangle$ transition frequency of one qubit close to the $|1\rangle - |2\rangle$ transition frequency of the other one, the basis states will acquire a state-dependent phase. By engineering the rising and falling edge of the flux pulses to tune the qubit, as well as the interaction time, one can engineer a cPhase gate [42].

A different type of two-qubit gates are based on couplings solely induced by microwave drives [103]. A great advantage of these gates is that they do not require control over the qubit transition frequencies. Therefore, fixed frequency qubits can be used which eliminates one source of noise and decoherence. This scheme can also reduce the number of wires necessary to control a certain amount of qubits. One gate within this category is the cross-resonance gate [104], [105]. It requires separated microwave control lines for each qubit but no flux lines to tune the qubit. The scheme is conceptually simple: One irradiates a microwave pulse on the control line of one qubit but with the frequency matching the second qubit. If the qubits are weakly coupled to each other on the off-diagonal (e.g. XX-coupling), this coupling will be amplified by the cross-resonance pulse. The pulse thus effectively turns on the coupling between the two qubits and its strength will depend on the amplitude of the pulse (compare Eq. (14) in [104]). Furthermore, by changing the phase of the pulse, one can select different two-qubit gates. One example is the ZX gate which, together with one-qubit rotations, gives a universal gate set for quantum computation. The same way, a $\sqrt{ZX}$ gate can be achieved, which is related to the CNOT gate by only one additional $\pi/2$ rotation per qubit [104]:

$$\text{CNOT} = \left[R_z(-\frac{\pi}{2}) \otimes \mathbb{1}\right] \sqrt{ZX} \left[\mathbb{1} \otimes R_x(-\frac{\pi}{2})\right] \quad . \tag{2.33}$$

Yet another two-qubit gate is the parametric iSWAP gate. It can be used with fixed-frequency qubits but requires a tunable coupler between them [61]. An exemplary image of such an architecture is shown in Figure 2.9. Typically, these couplers are superconducting qubits themselves but with a SQUID-based junction and thus flux-tunable. With this architecture, it is possible to parametrically turn on the coupling between the two qubits via the coupler. This is achieved by oscillating the coupler at the qubit-qubit detuning, resulting in an oscillation between the states $|01\rangle$ and $|10\rangle$. The oscillation time will determine the extend of the exchange, making it possible to obtain both an iSWAP and a $\sqrt{\text{iSWAP}}$ gate.

2.2.8 Quantum Software Tools

When working with superconducting qubits, different software tools exist to support the researchers and users to perform experiments, write quantum algorithms and simulate quantum computations. Specialized measurement frameworks are used to control laboratory instruments, perform experiments and persist the acquired data. One such framework designed for superconducting qubits is the open-source quantum measurement suite Qkit [106]. It is developed and actively used by the Institute of Physics at the Karlsruhe Institute of Technology. Qkit provides a

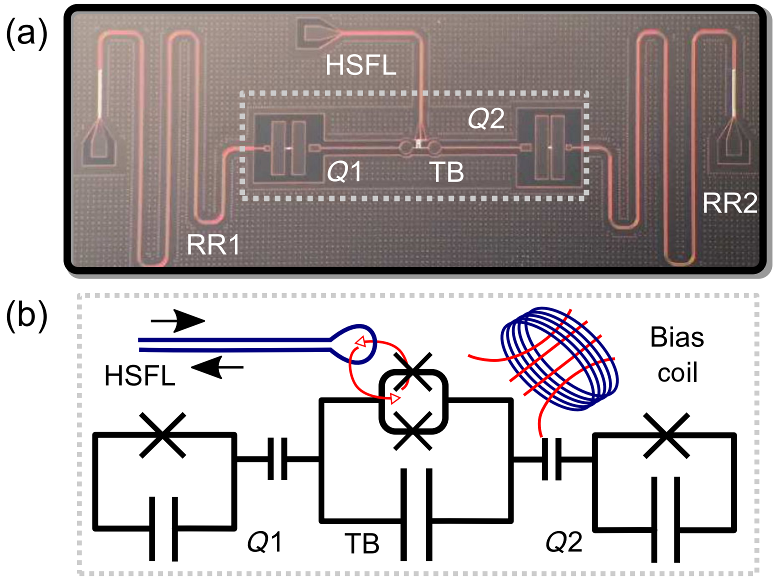


Figure 2.9: Two fixed-frequency Transmon qubits coupled to a third frequency-tunable qubit acting as tunable coupler. (a) Microscopic picture of a chip. Each Transmon qubit (Q1 and Q2) has its own readout resonator (RR1 and RR2). Both are coupled to a third qubit (TB) which is frequency-tunable using a high-speed flux line (HSFL) as well as a bias coil to adjust the working point of the coupler. (b) Schematic representation of the same circuit. [61]

collection of drivers to control laboratory instruments. Furthermore, it takes care of storing data in a standardized format and provides an interface to use this storage and work with the data. Additional modules encapsulate different measurement functionalities, data analysis capabilities and a graphical user interface to inspect the acquired data.

On a higher abstraction level, the open-source quantum development kit Qiskit [107] exists to work with quantum computers at the algorithm level. It is developed by IBM and widely used in the community when working with high-level applications of quantum computing. Qiskit provides tools to create quantum algorithms by combining quantum gates into a circuit (compare Section 2.1.4). These circuits can then be executed on various backends, mainly simulators and online systems provided by IBM. The quantum development kit is provided in Python which is the de facto language of choice when working in the field of quantum computing.

Qiskit is divided into multiple packages called Qiskit Aer, Terra, Ignis and Aqua. Aer provides low-level access to quantum hardware via pulsed experiments. It also contains various classical simulators to simulate the behavior of a quantum processor classically. Different noise models to make these simulators more realistic and investigate the impact of different noise sources are also provided.

Terra includes the generation of quantum circuits using a gate-level programming language. It also provides methods to optimize such circuits to be executed with minimal possible error on an actual quantum processor. Terra also contains the interface to real quantum hardware provided by IBM, so-called backends. Cloud backends of IBM can be accessed by a so-called provider that establishes a connection to the IBM Quantum Experience. Multiple superconducting quantum processors of IBM are available online to be selected.

Ignis is used to perform qubit characterization experiments and to apply error correction and mitigation techniques. It also contains different benchmarks to verify the operation of an existing quantum processor.

Finally, Qiskit Aqua provides high-level access to a number of quantum algorithms which have been written in Qiskit Terra but can be used without the necessity to rewrite the algorithms again. This includes different algorithms for chemistry simulations, finance applications, machine learning, and optimization problems. As the different Qiskit packages interlock, one can write a high-level algorithm using Aqua which can then be executed using a Qiskit Terra provider.

2.3 Typical Experiment Setup

Superconducting qubits are typically read out and controlled using microwave pulses with frequencies up to 10 GHz. Additionally, bias currents and fast current pulses might be necessary to tune the qubits or perform certain operations on them. Due to the utilization of superconductors like aluminum ($T_C = 1.2\,\mathrm{K}$ [108]) and the energy spacing of existing qubits, low temperatures on the order of 10 mK are required [31]. Therefore, the chip on which the superconducting circuits are fabricated is located inside a cryostat, most commonly a dilution refrigerator. A schematic overview of a typical experiment setup is depicted in Figure 2.10.

The low temperature requirement imposes challenges concerning the microwave signal strength and noise thermalization. With control electronics being operated at room temperature, noise at room temperature is added onto the microwave signals. Thus, additional attenuation inside the cryostat is essential to thermalize this noise to the target temperature [31]. At the same time, the signal strength will also be reduced to a level of only a few photons on the chip. To obtain good signal fidelity, it is therefore paramount to have a good amplification chain for the return signal. Each amplifier has a specific noise temperature which needs to be taken

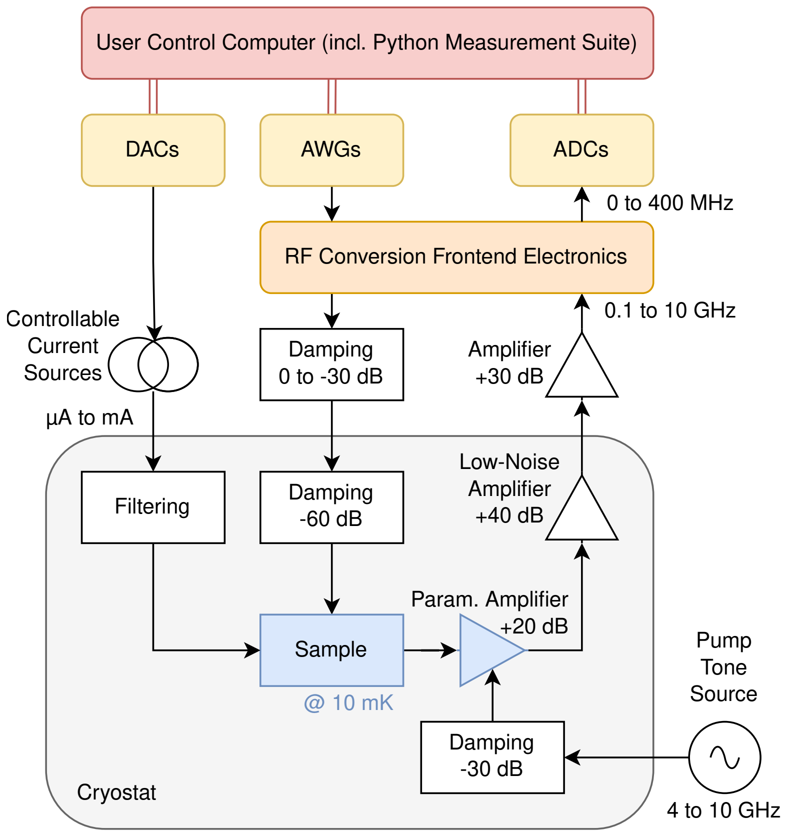


Figure 2.10: Schematic overview of a typical experiment setup for superconducting circuits.

into account. The first amplifier after the sample is often a specific superconducting circuit itself. This makes it possible to perform parametric amplification while only adding little more than the minimum noise allowed by quantum mechanics [109]. Examples of such quantum-limited amplifiers are traveling wave parametric amplifiers (TWPAs) [110], Josephson parametric amplifiers (JPAs) [111], or dimer Josephson junction array amplifiers (DJJAAs) [112]. While the specific mode of operation differs, all of them need an external pump tone that acts as energy source for the parametric amplification process. After this first amplification, classical low-noise amplifiers are employed to further boost the signal. In most of the cases, these are high-electron-mobility transistor (HEMT) amplifiers operated around 4 K [113].

Depending on the necessary signal levels, further damping and amplification can take place at room temperature. There, also filtering and possible frequency conversion within the radio frequency (RF) conversion frontend electronics is possible. Often, these are discrete, SMA-connectorized components which are manually installed and combined. The schematics of such a frontend are further discussed in Section 3.3 and thus not covered in more detail here.

Common laboratory equipment to generate and analyze microwave pulses are arbitrary waveform generators (AWGs) with gigasample precision and 12 to 16 bit resolution, and analog-to-digital converters (ADCs) with similar properties. The evaluation of the digitized signals happens on a separate computer. For generating currents, controllable current sources combined with digital-to-analog converters (DACs) might be used. For fast pulses, also AWGs with DC coupling can be employed directly. DC bias currents most likely will require additional filtering within the cryostat to remove noise from the signal which would disturb the superconducting circuits. A photograph of a typical measurement setup is shown in Figure 2.11.

All electrical devices are controlled by a separate user control computer. In most laboratories, Python is used within Jupyter Notebooks [114] to perform the configuration, measurement, and data processing. Different measurement suites exist to simplify this process. A prominent example is Labber which was originally developed at MIT and is now commercially distributed by Keysight [115]. Another example is Qkit [116], a measurement suite developed at the Institute of Physics (PHI) at Karlsruhe Institute of Technology (KIT) and introduced in Section 2.2.8.

The control platform presented in this thesis replaces the electronics to generate and analyze microwave pulses, i.e. the AWGs, the ADCs, the computer used to evaluate the digitized signals, as well as the RF conversion frontend. The architecture and capabilities of this platform are discussed in detail in Chapter 3.

2.4 FPGAs and Heterogeneous Systems

With increasing processing, timing and data throughput demands, pure software-based approaches become impractical and custom hardware is necessary. One option for such a situation are application-specific integrated circuits (ASICs). As these incur a high fixed cost to develop and manufacture as well as relatively long fabrication times, they are not suited for (rapid) prototyping and small-volume productions. Instead, programmable logic devices (PLDs) can be used which are integrated circuits that can be configured by the user after manufacturing. One type of PLDs are field-programmable gate arrays (FPGAs) which are introduced in the following section [117, p. 713 ff.]. These can be combined with classical processing

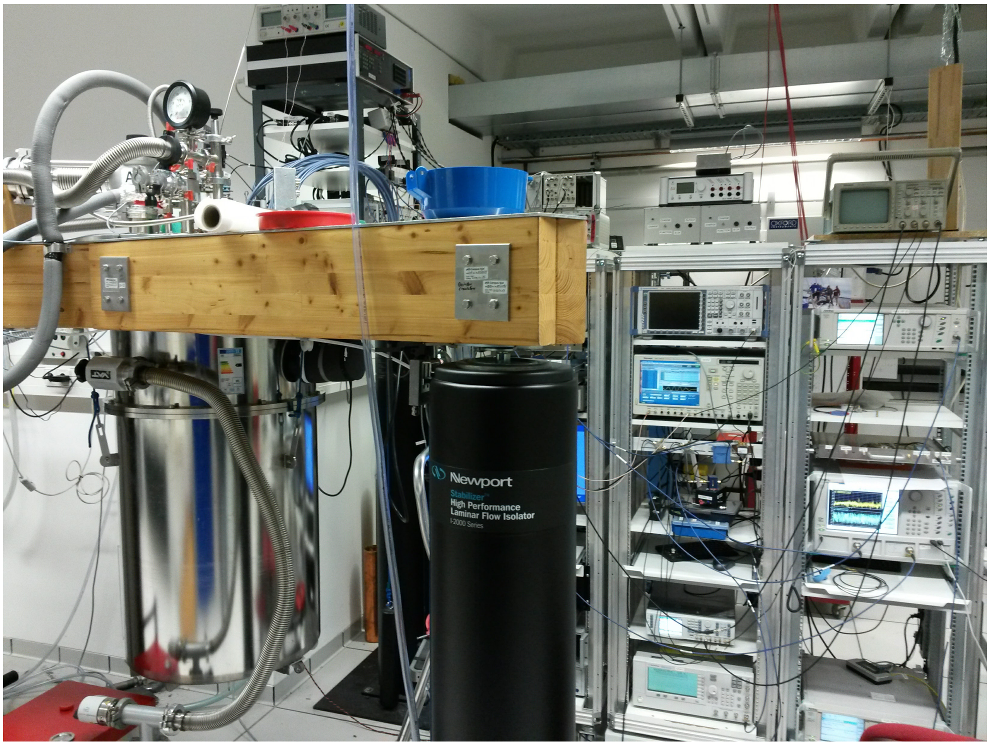


Figure 2.11: Photograph of a typical measurement setup for superconducting circuits. On the left, a closed-cycle dilution refrigerator is visible. Above it, multiple voltage supplies as well as room temperature amplifiers can be spotted. On the right, multiple racks with laboratory equipment are visible. These include RF frontend electronics as well as AWGs and current sources, but also more generic measurement equipment like vector network analyzers (VNAs) or spectrum analyzers. The picture was taken in 2017 inside the laboratory of the Institute of Physics (PHI) at the Karlsruhe Institute of Technology (KIT).

units to form a heterogeneous system-on-chip (SoC). Going even one step further, for applications where RF microwave signals need to be generated and processed, also DACs and ADCs can be directly integrated into such systems.

2.4.1 Field-Programmable Gate Array

FPGAs are integrated circuits produced in cutting-edge transistor technology which can be configured to act like a custom logic circuit. As the name field-programmable gate array suggests, these can be configured by the user in the field. FPGAs contain many configurable logic blocks (CLBs) which can be programmed and connected based on the needs of the user and the circuit to realize. These blocks provide all the basic components needed to form a wide variety of circuits. They are embedded in a versatile interconnection network that can be configured to connect different cells and their elements among each other, leading to the name "gate array". The FPGA can furthermore be interfaced with peripheral electronics using input/output (I/O) connections [117, p. 713 ff.].

The FPGAs used in this thesis are manufactured by Xilinx and based on their 16nm FinFET UltraScale architecture [118], [119]. Besides standard logic cells, common resources inside the FPGA are the following:

Look-up tables (LUTs) have n binary inputs and can represent any arbitrary n-digit binary function $f : \{0, 1, \ldots 2^n - 1\} \to \{0, 1\}$. The output values are stored inside static RAM (SRAM) cells within the LUT and the input configuration defines the address of the result to take. In a Xilinx UltraScale architecture, $n = 6$ inputs are used. LUTs can also be combined to form distributed random access memories (RAMs). [120]

D-type flip-flops capture the value of a data input (called D) at a defined section of an input clock cycle, e.g. at a rising edge. The captured value then becomes the output Q of the flip-flop. For other sections of the clock cycle, the output stays unaffected. This can be used to store and delay digital signals. In the UltraScale architecture, the flip-flops can also be configured to act as level-sensitive latches. [120]

Multiplexers are multiple-input, single-output switches that select the input based on a separate select signal. They can be utilized to route different input signals onto the same output and switch between them. Likewise, one input can also be forwarded to different outputs depending on the separate select signal. The Xilinx UltraScale architecture offers eight multiplexers with a 4:1 mapping per CLB. They can be further combined to result in a mapping of up to 32:1. [120]

Digital signal processing (DSP) slices perform efficient arithmetic operations, from adding over accumulation and logic operations up to multiplication. Common use cases are for filter designs, fast Fourier transforms, or complex multipliers. Xilinx' implementation of the DSP slices offers signed arithmetic operations with a two's complement multiplication of up to 27×18 bit. [121]

Block RAMs (BRAMs) provide versatile data storage on the FPGA. As these are RAMs, data can be written to and read from arbitrary positions inside the available memory address range. In a Xilinx UltraScale device, each BRAM stores up to 32 kbit of data. They can also be configured as two independent 18 kbit memory blocks. Furthermore, multiple BRAMs can be cascaded to form larger memories. Different operation modes can be distinguished: In the true dual-port mode, the memory can be accessed completely independently by two separate ports, including different clocks. Both ports offer full read and write access. In the simple dual-port mode, one port is exclusively used for writing, the other only for reading. In both modes, read as well as write accesses are executed synchronously in one cycle. [122]

I/O blocks (IOBs) connect external devices and logic to the FPGA fabric via I/O pads of the chip. These can be configured for different voltage levels and behaviors. Different types of blocks exist for slow signals, differential signals, and high-speed serial signals. The latter requires a serializer and deserializer (SerDes) so one can serially transmit and receive data with higher data rates than the design clock rate on the FPGA [123]. To achieve this, the data is converted between a parallel representation inside the FPGA and a serial representation at the I/O pad. In a Xilinx UltraScale device, IOBs are combined as I/O banks. Different banks exist depending on the application, called HP I/O for high-speed interfaces, HR I/O for flexible signal support, and HD I/O for low-speed interfaces. [124]

Integrated hard IP blocks for special purpose applications provide functionality predefined in the chip design of the FPGA. Examples are interface blocks for PCI Express, DDR memory control, 100G Ethernet, and Interlaken, a scalable chip-to-chip interconnect protocol for multi-gigabyte per second data transmission. [119]

2.4.2 Multi-Processor System-on-Chip

While FPGAs are perfectly suited to implement deterministic logic circuits with high data throughput and nanosecond precision, more versatile applications with less stringent requirements might be better handled by running software on a processing system (PS). Therefore, modern approaches often combine the benefits of both worlds in a heterogeneous system. These integrate an FPGA with one or multiple PSs on a single chip [118]. Due to the close interconnection, communication between both components can happen with latencies on the order of hundreds of nanoseconds.

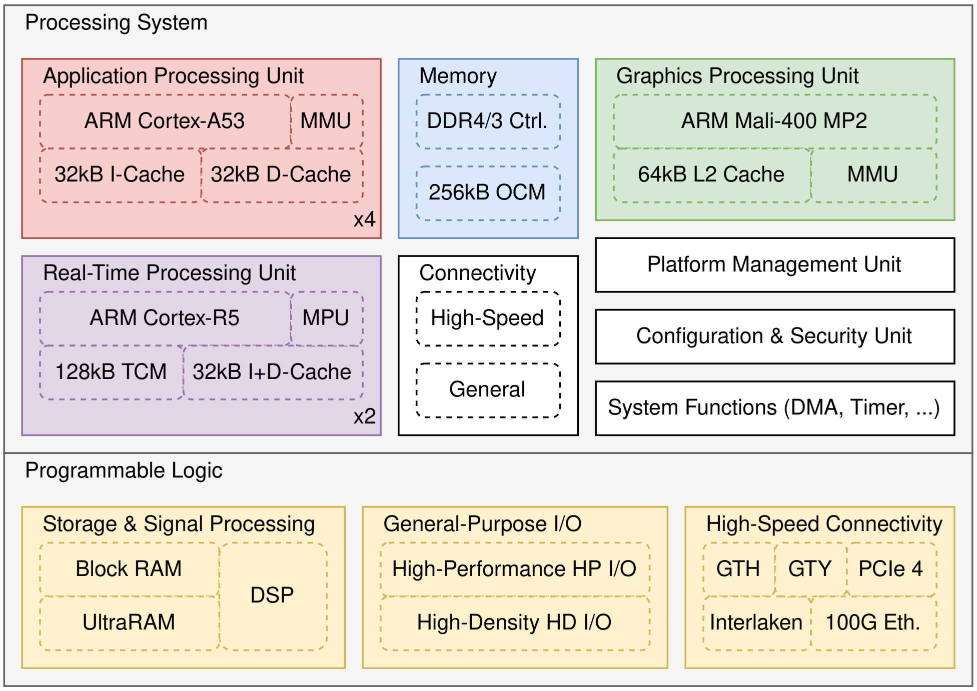


Figure 2.12: Top-level block diagram of the Zynq UltraScale+ EG MPSoC. [118]

In this thesis, the Xilinx Zynq UltraScale+ multi-processor system-on-chip (MPSoC) architecture is used [118]. The top-level block diagram of this heterogeneous system is depicted in Figure 2.12. The programmable logic (PL) contains the typical elements of an FPGA, as well as additional general-purpose and high-speed I/O interfaces. In the PS part, two processing units are implemented: an application processing unit (APU) consisting of a four-core ARM Cortex-A53 processor and a two-core ARM Cortex-R5 as real-time processing unit (RPU). It furthermore contains the graphics processing unit ARM Mali-400 MP2, as well as a DDR4 memory controller, two 128 kB tightly-coupled memories (TCMs) for the Cortex-R5 cores, and a 256 kB on-chip memory (OCM). The PS also offers dedicated units for platform management, configuration, and security, as well as versatile interfaces to connect to peripheral devices. With this rich set of heterogeneous capabilities, the Zynq UltraScale+ MPSoC is well suited to accommodate and partition tasks with a wide range of requirements.

2.4.3 Radio Frequency System-on-Chip

Telecommunications applications, as well as radar and others, require the generation and detection of microwave signals. To fulfill typical data processing requirements for these applications, either an FPGA-only or a heterogeneous MPSoC approach is employed. The PL is then connected to separate DACs and ADCs which translate the digital signals to analog microwave signals and back. Going one step further, one can also combine these converters into the heterogeneous system with the FPGA, forming a radio frequency system-on-chip (RFSoC) [125].

Driven by the development of the 5G standard with a frequency range of initially up to 6 GHz, high-precision, multi-gigasample DACs and ADCs entered the market to enable direct-synthesis of the required signals [125]. These benefit strongly from a tight integration with an FPGA, as otherwise the data throughput to and from the converters becomes a challenging bottleneck. Xilinx offers a Zynq UltraScale+ RFSoC family [125] with 14 bit DACs operating at up to 10 GHz sampling frequency and 12 or 14 bit ADCs operating at up to 5 GHz sampling frequency.

The chip used in this thesis is a Zynq UltraScale+ ZU28DR, a first generation RFSoC, with eight 14 bit DACs and eight 12 bit ADCs. The DACs can be operated at up to 6.554 GHz and the ADCs at up to 4.096 GHz. The chip also contains an FPGA with 930 000 system logic cells, 4272 DSP slices and 60 MB of memory. On the PS side, it contains all the units the Zynq UltraScale+ MPSoC architecture offers, as described above.

The data exchange between the PL and the so-called RF data converter [126], which contains the ADCs and DACs, is handled using AXI4-streams. Multiple samples are transmitted each cycle and the signal can also be represented as in-phase and quadrature (I/Q) components. Control and configuration of the RF data converter block is exposed via an AXI4-Lite register interface.

Xilinx offers a variety of special DSP functionality for their converter blocks. Functional diagrams for both DACs and ADCs are shown in Figure 2.13. A FIFO is used to translate between the data signal rate and the converter clock frequency. Interpolation and decimation filters can be applied to end up at the sampling frequency. Each ADC and DAC signal path contains an I/Q mixer which has an internal numerically controlled oscillator (NCO) and, when activated, performs a digital up- or down-conversion (DUC/DDC) with an arbitrary frequency and phase offset. A crossbar enables flexible configuration and connection between neighboring signal paths. For example, results of multiple mixers can be combined on one DAC output for multi-band operation or the resulting I/Q values of one mixer can be split up to two separate outputs. A quadrature modulator correction

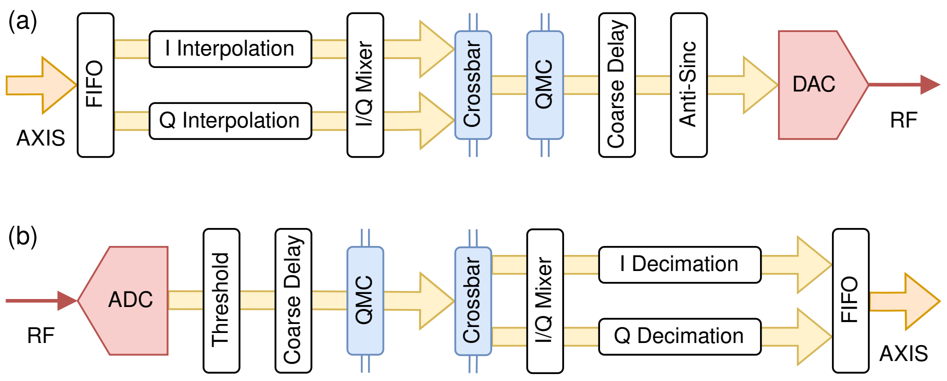


Figure 2.13: Functional diagrams of single signal paths within the RF data converter. Blue elements are interconnected between neighboring signal paths. (a) Digital-to-analog conversion path. (b) Analog-to-digital conversion path.

(QMC) makes it possible to adjust phase and amplitude of the signal, e.g. to calibrate external I/Q mixers. A coarse delay block can furthermore adjust for delay mismatches between multiple outputs by delaying the signal by up to seven sampling clock cycles. As final element before signal conversion, the DAC signal path has an inverse sinc filter to compensate the analog output response of the DAC if enabled [126, p. 104 f.]. The ADC exposes flags to indicate an over-voltage situation where the converter is shut-down to prevent damage and over-range situations where the input signal saturates the full-scale digital signal range. Right after digitization, a built-in threshold detection can furthermore be configured in the RF data converter [126].

The converters thereby offer a flexible and customizable way to generate and process analog signals and exchange them with the PL. Different operation schemes are possible, from multi-band operation, over I/Q signal generation and detection up to standard real signal handling with and without DUC and DDC.

2.5 Summary

Quantum computing is a promising field with diverse applications. The fundamental building blocks of a quantum processor are qubits. Different physical implementations of qubits exist, like photons or ion traps. This thesis focuses on qubits realized by superconducting circuits. A variety of different qubit designs belong to this category, including charge, Transmon, and Fluxonium qubits. They are controlled and read out using microwave pulses with frequencies of up to

10 GHz. Experiments with superconducting qubits consist of well-defined pulse sequences. Such experiments can, e.g., be conducted using laboratory equipment and the quantum measurement suite Qkit. Multiple qubits form a quantum register on which quantum algorithms can be executed. Such algorithms are composed of quantum gates acting as elementary operations on one or multiple qubits. One possibility to write such algorithms is a language provided by the IBM quantum development kit Qiskit which is widely used in the field of quantum computing.

To perform experiments with qubits and run quantum algorithms on actual quantum processors, sophisticated control electronics is required to bridge the gap between both worlds. FPGA- and RFSoC-based systems have a great potential in this area. The next chapter discusses in detail how they can be used to implement control electronics for superconducting qubits.

3 Platform Architecture

This chapter describes the architecture of the control electronics developed in this work, the QiController. The word is a combination of "Qi" for quantum interface and controller, as the system realizes the quantum-classical interface by controlling the connected quantum bits (qubits). First, requirements for the QiController are identified based on which a concept is deducted. Then, the different components of the system are introduced in more detail and a perspective on scalability is provided. Finally, results of characterization measurements with the QiController are given.

3.1 Functional Requirements

As a first step, it is crucial to identify requirements the QiController needs to meet in order to provide a flexible system to control and read out superconducting qubits. However, these requirements largely differ depending on the superconducting qubit chip employed as already discussed in the previous chapter. They are affected, e.g., by the type of qubit utilized, the number of qubits, as well as the coupling mechanism between the individual qubits for two-qubit gates. New superconducting qubit types are regularly proposed and research groups around the world strive to further improve their properties [90], [127]. Furthermore, the design of multi-qubit chips and quantum processors is still considered to be fundamental research in physics [29], [128]. For all these reasons, precise figures for the requirements often cannot be provided as the field is rapidly changing. Instead, this section elaborates on some guiding frames that a versatile control electronics should cover in order to support most of the superconducting qubit chips currently available.

Qubit Control

To control the state of individual superconducting qubits, microwave pulses with arbitrary shape and gigahertz frequencies have to be generated. Depending on the qubit type, these frequencies vary from tens of megahertz to roughly 10 GHz

[35], [36]. Typical pulse durations are between a few and hundreds of nanoseconds [40]. Arbitrary pulse shapes with nanosecond precision are required for optimal control protocols [39]. Other commonly used pulse shapes are rectangular pulses, Gaussian-shaped pulses, or pulses based on the derivative removal by adiabatic gate (DRAG) technique [129]. The latter is especially important for short pulses to eliminate leakage to higher qubit states in weakly nonlinear systems like a Transmon. It requires the ability to independently shape the in-phase and quadrature (I/Q) amplitude of the control pulses. To set the rotation axis around the Bloch sphere of the qubit state, precise control over the phase of control pulses is necessary [94]. Typical phases are multiples of 90° as these resemble the x and y axis on the Bloch sphere. Even small phase deviations can lead to an error of the performed operation which will add up when more operations are performed. For the same reason, it is important that the phase noise of the control signal is small. Otherwise, this will contribute to an effective reduction of the qubit's decoherence time T_2 [31], [32]. Per qubit, multiple different pulses are likely required to perform rotations around different axes and with different angles. Additionally, having the possibility to perform virtual Z gates with arbitrary rotation angles is beneficial as these can be implemented instantaneous and free of physical errors [94].

Qubit Readout

To read out the qubit state, microwave pulses at the frequency of the readout resonator need to be generated and acquired [37], [38]. The frequencies are typically between 4 and 10 GHz and separated from the control frequencies to minimize undesired leakage and cross-talk [35], [36]. Except for some special readout schemes, the pulse shape is only of secondary importance. Often, rectangular or Gaussian-shaped pulses are employed. Typical pulse lengths for the readout are on the order of hundreds of nanoseconds. Longer readout pulses yield a higher signal-to-noise ratio (SNR) but the qubit state might flip during the measurement if the energy relaxation time T_1 is on a comparable timescale. Too long pulses will thus increase the readout error and nullify the better SNR [31]. State-of-the-art readout pulses which still preserve high fidelity are on the order of 50 ns for appropriate chip designs [130].

The qubit state is encoded in the amplitude and phase response of the returning readout pulse [37], [100]. Therefore, state measurements also involve microwave signal recording and demodulation. As the returning signal is only a few photons strong and thus has to be amplified intensively, the SNR is limited and good filtering is important [100]. The resulting phase and amplitude information then needs to be translated into the measured qubit state. For some operations, e.g.

in quantum error correction schemes, it is furthermore necessary to perform conditional pulses depending on this state [46], [48]. With single pulses on the order of tens to hundreds of nanoseconds, response latencies for such measurement-based, closed-loop feedback operations should also be on the same order of magnitude.

Multi-Qubit Interactions

The implementation of multi-qubit operations strongly depends on the chip architecture used. So far, it is not obvious which approach will be the dominant one for future chip designs. Some approaches require fast flux pulses generated by microampere currents to effectively turn on the interaction of two neighboring qubits [41], [42], [97]. Other approaches use microwave pulses to mediate the interaction [61], [103], [105]. As the field is still very diverse, deriving clear requirements for the electronics is not yet possible. However, a versatile control electronics should keep these different approaches in mind and be extensible to support these when required, e.g. by integrating and triggering additional laboratory equipment to create flux pulses.

Experiments & Quantum Algorithms

Pulses and measurements act as gate operations on the stationary qubits. Multiple such operations are concatenated to perform a quantum algorithm [131]. Performing computational tasks and experiments therefore requires executing well-defined sequences of pulses and state measurements. The delays between multiple pulses can also have a significant impact on the results, e.g. due to decoherence or intentional detuning of control pulses [100]. As experiments and algorithms are typically repeated many times to collect sufficient statistics for the results, reproducibility on the nanosecond timescale is another important requirement. Advanced experiments can furthermore require complex control schemes, as well as sophisticated online processing and data reduction capabilities [45].

Scalability

When controlling many qubits, the requirements to the control electronics largely depend on the chip architecture used. It especially dictates the number of microwave channels required to control a number of qubits N. For some chips, each qubit has two separate input microwave lines for readout and control, so $3N$ lines in total when including the return lines for the readout. For other chips, frequency-division multiplexing (FDM) is used to combine multiple signals onto one microwave line. The concept of FDM is introduced in Appendix A.3. For single-qubit chips, often manipulation and readout pulses are fed into the chip using the same microwave port as seen in the example in Section 2.2.6. The same principle can be applied to a multi-qubit chip where each qubit has its own readout resonator which is also used for the control pulses, thereby requiring $2N$ lines [132]. Other chips combine the readout signals of all or some qubits onto a single line while the manipulation signals still use a separate line per qubit [133]. If all readout signals are multiplexed, one requires $N + 2$ microwave line. However, with growing qubit count, the accessible bandwidth becomes a limiting factor. A typical spacing between multiple resonators is at least tens of megahertz or higher [29], [133]. Therefore, larger architectures using FDM usually only group roughly six to eight qubits together for a single readout line [29]. If qubits are organized in $G < N$ sub-groups that are separately multiplexed, $N + 2G$ lines are required. Finally, some architectures combine both readout and manipulation for all qubits onto a single microwave line resulting in only 2 microwave lines. While this approach is the most attractive one in terms of microwave lines wired through the cryostat, it also leads to non-negligible cross-talk between different qubits [134]. To summarize, multi-qubit control electronics needs to support FDM and provide flexibility for the user to decide which signals should be combined onto one or multiple microwave lines connected to the cryostat.

3.2 QiController Concept

As the design of a superconducting quantum processor is still active research and the way to control it will likely change in the future, the QiController is not designed according to one specific chip in mind. Instead, its goal is to facilitate experiments with superconducting qubits and first prototypes of quantum processors with moderate number of qubits. This is also reflected in the design choices of the system leaving as much flexibility as possible for the user to adapt the control and readout to the investigated qubit chip.

Hardware Selection

A field-programmable gate array (FPGA) is best suited to deterministically assemble and process the required pulses with nanosecond accuracy. Compared to an application-specific integrated circuit (ASIC), FPGAs can still be adapted in the field to changing requirements that might arise in the ongoing research with superconducting qubits. As the FPGA handles signals digitally, high-speed and

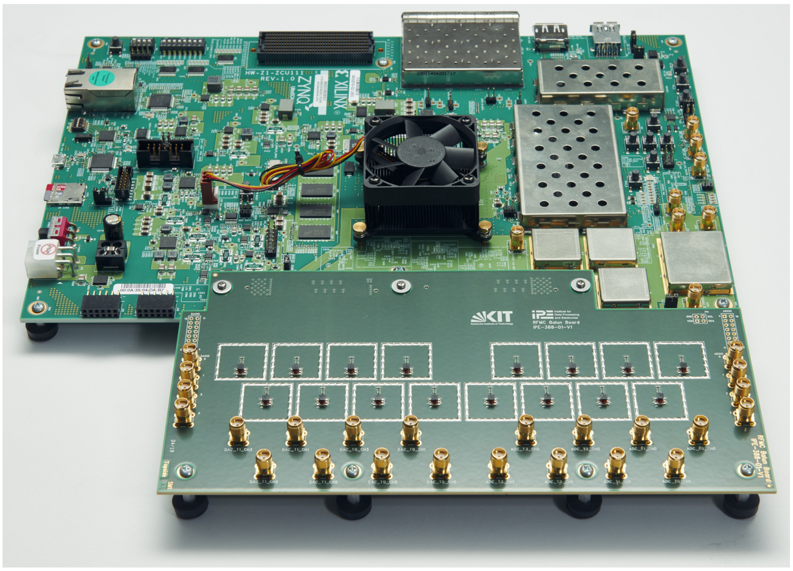


Figure 3.1: Photograph of the QiController consisting of a Xilinx ZCU111 evaluation board and a custom-built balun board.

high-precision digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) are necessary to generate and acquire the required microwave pulses. The Xilinx Zynq UltraScale+ radio frequency system-on-chip (RFSoC) architecture is used as a basis for the QiController as it tightly integrates eight multi-gigasample DACs and ADCs with a large FPGA and two processing units. This provides great flexibility to partition tasks with different requirements onto different parts of the chip. More details to the RFSoC architecture are given in Section 2.4.3.

The converters are operated at 4 GHz sampling frequency and handle the microwave generation and digitization in the complex baseband. Using decimation and interpolation filters, a per-channel data rate of 1 GSPS and thus 1 ns time resolution is obtained within the programmable logic (PL). This simplifies the internal signal handling and is sufficient for most experiments with superconducting qubits. For the PL design, a clock rate of 250 MHz is selected. To reach the data rate of the converters, four samples have to be processed each clock cycle. The clock is directly derived from the converter clocking to have a stable clock relation without dephasing.

As hardware, the Xilinx ZCU111 evaluation board is selected as it provides a readily available and flexible board hosting a ZU28DR RFSoC. To utilize all the eight ADC and DAC channels, a custom balun board is designed to convert the differential outputs and inputs from the RFSoC to single-ended SMA connectors. These can then be connected to a radio frequency (RF) frontend or directly to the experiment. A picture of the QiController comprising ZCU111 and custom-built balun board is shown in Figure 3.1.

Overall Architecture

An overview of the QiController architecture is presented in Figure 3.2. Tasks which require nanosecond accuracy are handled within the PL. There, the main functionality is provided by digital unit cells. They generate digital microwave pulses which are routed via AXI-streams to a separate cell multiplexer module. In this module, they are combined and distributed to the available DAC channels. Likewise, returning digitized microwave signals from the ADCs are distributed and split up onto the belonging cells. The necessary frequency conversion between RFSoC and qubits is happening inside an RF frontend electronics which is further structured into analog unit cells. Besides the microwave pulses, also digital trigger signals can be issued which will be routed to digital output pins. These are added to further expand the flexibility of the system, as additional laboratory equipment can be integrated and triggered for extra functionality.

A special cell coordinator is connected to all digital unit cells to ensure synchronicity and exchange information. In particular, it is able to start any subset of digital unit cells simultaneously. On the processing system (PS), online data processing and advanced control is implemented within the real-time processing unit (RPU) using the Taskrunner framework [47]. The application processing unit (APU) provides a user interface to interact with the platform via Ethernet. It is implemented using a modular communication server called ServiceHub [135]. The PS controls and configures the PL using a register-based AXI4-Lite bus where the PL modules are mapped into the physical memory address range of the PS. Access is performed by simple memory read and write operations using the AXI HPM FPD interface. To communicate between both processors, a shared memory is employed.

Partitioning of Functionality

Based on the architecture introduced above, experiments running on the QiController are always partitioned in a similar way. Pulse generation, detection, sequencing and result storage, as well as simple parameter variations, happen with nanosecond precision in the digital unit cells in the PL. Their execution is synchronized by the cell coordinator which is triggered by the Taskrunner. The Taskrunner controls the execution of the PL, performs more complex parameter variations between multiple executions and collects the result data that is generated within the digital

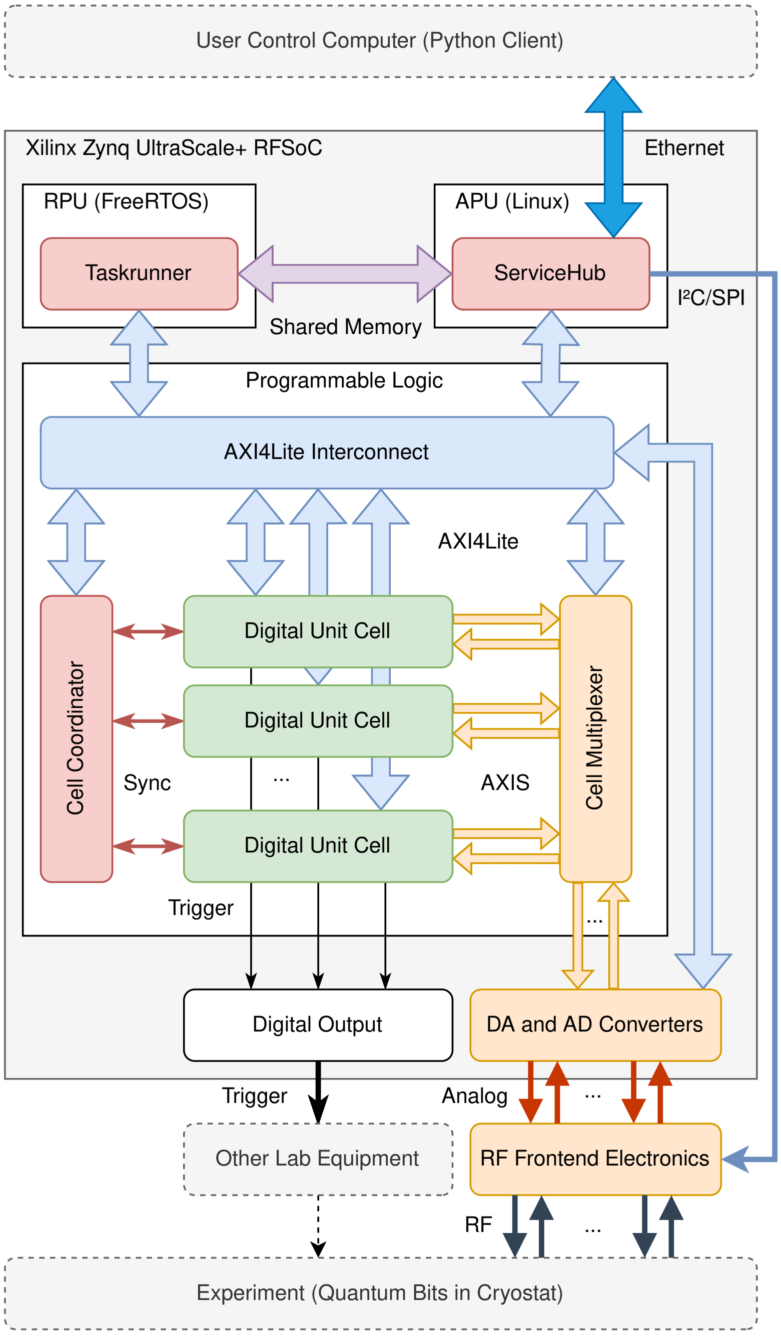


Figure 3.2: Architecture of the QiController. Everything sketched with solid lines is part of the heterogeneous electronics platform.

unit cells. Also, further data aggregation, sorting or online evaluation are possible, depending on the requirements of the experiment. This includes, but is not limited to, averaging of data, collection of individual qubit states or single measurement values, as well as counting different qubit state outcomes. The resulting data is then sent via the ServiceHub to the user client where the user can either persist or further process the data offline. During the experiment, no connection between client and QiController is required. However, the user can monitor the process of the execution and, depending on the experiment, stream some result data already before the execution finished.

Qubit Interface (Digital Unit Cell)

Different PL modules are developed to generate and analyze microwave pulses, to persist data, and to coordinate their execution in a well-defined sequence of operations with the required precision and timing accuracy. These are bundled inside a so-called digital unit cell which contains all functionality required to interface a single superconducting qubit. The microwave signals are digitally represented in the complex-valued baseband as I/Q components with 16 bit resolution each. This is required to perform arbitrary pulse shaping and to have full phase and sideband control of the up-converted signal. As discussed, four samples are processed each cycle to obtain the converter data rate of 1 GSPS.

Frequency Conversion (Analog Unit Cell)

With this data rate, microwave signals in the baseband, with frequencies of up to $f_s/2 = 500\,\text{MHz}$, can be generated. To cover the full frequency range from tens of megahertz up to ten gigahertz, a separate RF frontend is necessary to convert the frequencies. Frontends built out of discrete components are common in typical experiment setups but face high relative cost, long assembly time, high space requirements, and increased susceptibility to errors. Therefore, a custom printed circuit board (PCB) is developed to mitigate these challenges. It forms a so-called analog unit cell and handles the frequency up- and down-conversion with the same reference oscillator for a single RF output and input.

Scalability

To control multiple qubits with a single QiController, digital and analog unit cells are replicated. The concept is illustrated in Figure 3.3 with a three-qubit chip

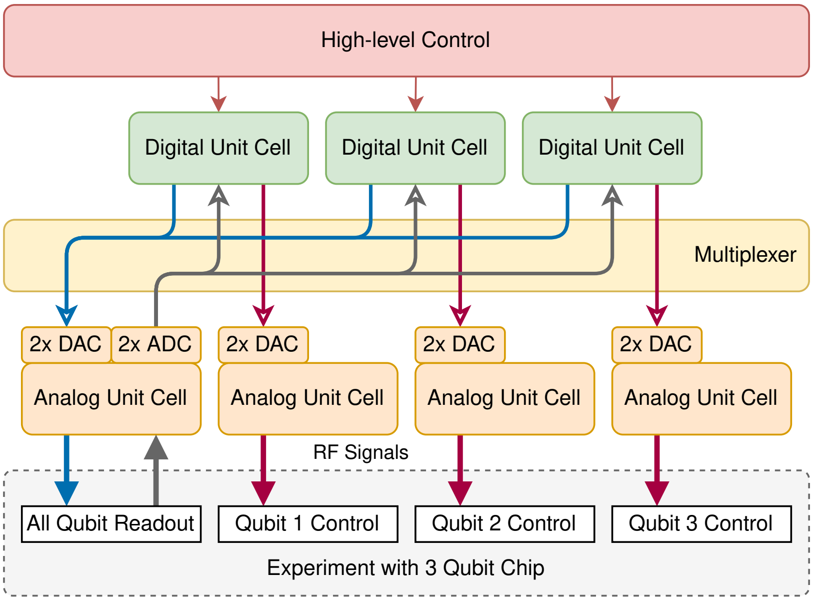


Figure 3.3: Abstract concept of single-platform scalability with digital and analog unit cells. For illustration, a chip with multiplexed readout but separate control pulses for three qubits is sketched.

where the readout is frequency-multiplexed while the control pulses are not. As introduced, digital unit cells are implemented on the FPGA and contain all digital logic required to control and read out a single qubit. For three qubits, therefore three digital unit cells are necessary. Analog unit cells translate the baseband signals to and from the required frequency range. Each analog unit cell interfaces with two DAC channels to up-convert the generated I/Q signal. It might furthermore also interface with two ADC channels to down-convert a returning signal with the same frequency reference. In the above example, one analog unit cell is used to up- and down-convert the frequency-multiplexed readout signals. As the manipulation signals are not multiplexed, three additional analog unit cells are required to up-convert their frequencies. The mapping between digital and analog unit cells happens on the FPGA within the cell multiplexer module.

Online Data Processing and Advanced Control (Taskrunner)

Nanosecond-accurate creation of pulse sequences and qubit state measurements, as well as simple parameter variations, are handled on the FPGA. More complex tasks with less strict timing requirements are implemented on the RPU of the RFSoC. For example, this can be the collection and online post-processing of experiment result data, or complex parameter changes between pulse sequences. A great advantage of this partitioning is that it combines the flexibility of a software-based approach with a tight and low-latency interconnection to the capabilities provided on the FPGA. This minimizes the overhead compared to an execution on a separate control computer. To provide convenient access to the RPU, the Taskrunner framework [47] is designed. It complements the PL with versatile, low-latency real-time control, data aggregation and evaluation features. The RPU is not used by any other software to guarantee a mostly time deterministic interaction with the PL (compare [47], [135]). With the Taskrunner, a user can load arbitrary C code onto the RPU which will be compiled on-the-fly to reduce external dependencies and improve usability.

User Interface (ServiceHub)

The control electronics needs to provide a well-defined interface to exchange control signals, configuration values and measurement data with the user in a convenient way. To provide an extensible interface, a modular communication server called ServiceHub [135] is designed which is running on the APU of the RFSoC. It is closely connected to and controls the PL, the RPU, as well as peripheral devices like analog unit cells. The user interface is implemented using remote procedure calls via an Ethernet connection. This provides flexible means for the user to configure and control the QiController from a remote control computer. The ServiceHub runs on a Yocto-based Linux operating system which also initializes the whole platform at boot time. Software developments targeting the APU can therefore leverage the rich ecosystem of Linux. As many physics laboratories use Python, a Python client is provided to interface with the platform. It wraps the remote procedure calls exposed by the ServiceHub. More information on the client is given in Chapter 4.

The following sections will present the different parts of the QiController in more detail.

3.3 Analog RF Frontend

To convert the baseband signals generated by the RFSoC to the desired frequencies of the qubit chips and back, an analog RF frontend is required. Such frontends can be assembled using discrete components, which is a popular approach for experiment setups. However, notable downsides include high space requirements and costs, long assembly time, and increased susceptibility to errors. This also impairs scalability when multiple signals need to be converted. Therefore, the

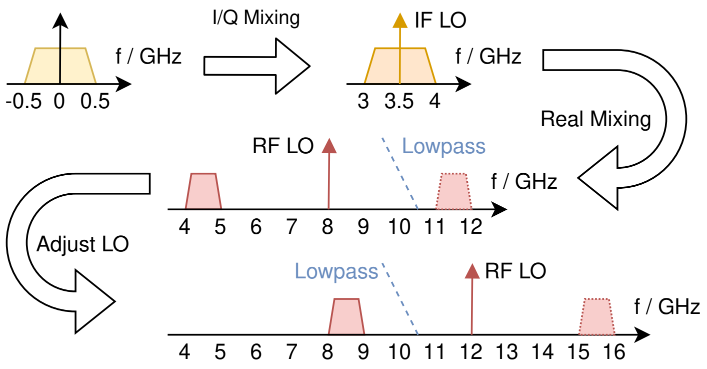


Figure 3.4: Concept of the two-stage frequency up-conversion with the relevant frequency bands. Frequency down-conversion happens after the same scheme but in reverse.

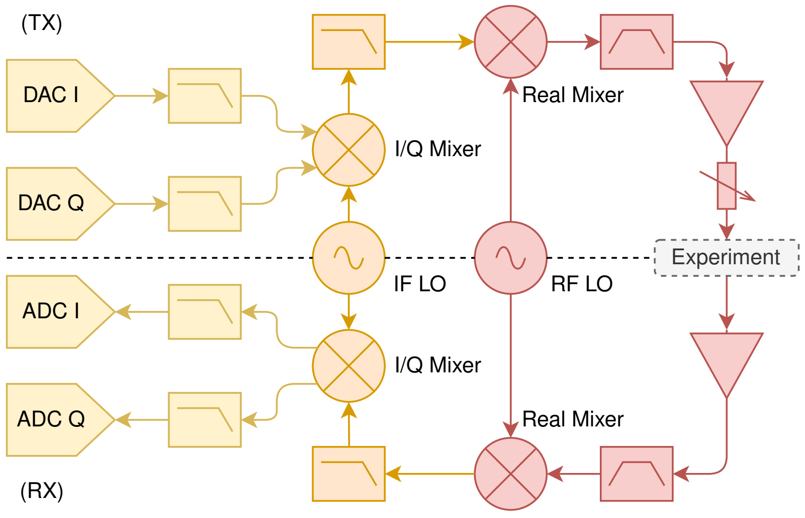


Figure 3.5: Functional schematic of the analog RF frontend PCB. It is separated into a transmitting (TX) and a receiving part (RX). Depending on the requirements, only the TX or both parts are used. Colors indicate different frequency domains, that is baseband (yellow), intermediate frequency (IF) (orange), and RF (red).

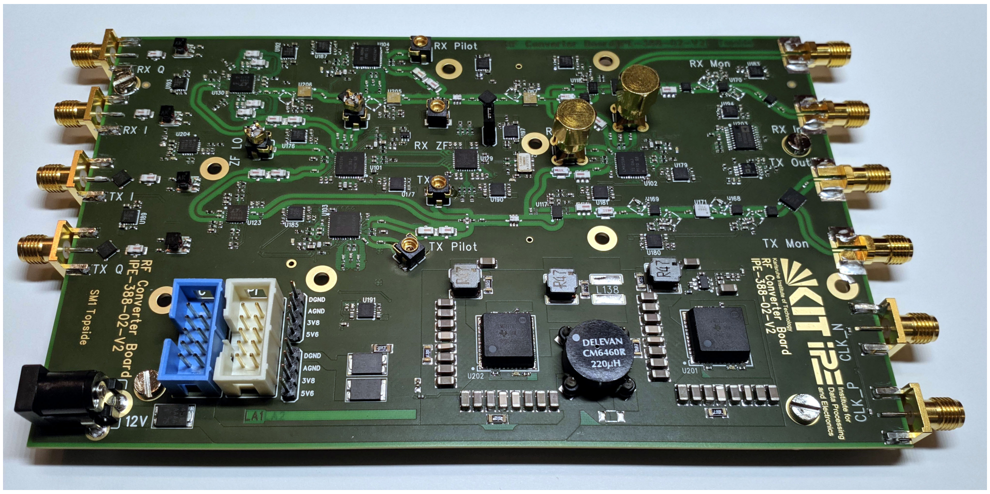


Figure 3.6: Photograph of the analog RF frontend PCB.

functionality to up- and down-convert a single signal is integrated onto a PCB, forming an analog unit cell. To decrease cost while maintaining high signal quality, a superheterodyne, two-step mixing process is used [136, p. 678]. The concept of it is sketched in Figure 3.4. A functional schematic of the analog RF frontend PCB is given in Figure 3.5. In this approach, the complex-valued baseband signals are first converted to a fixed IF using an I/Q mixer (see Appendix A.1). The IF signal is then further up-converted to the desired RF frequency using a real mixer. The same two-step process is performed in reverse when translating the RF signal back to the baseband for readout.

To cover a wide RF frequency range of 4 to 9 GHz with a single setup, the IF frequency of the local oscillator is chosen at 3.5 GHz. With the 1 GHz digital data rate in the complex baseband, one obtains an accessible frequency range at this stage between 3 and 4 GHz. A great advantage of this approach is that high-quality and low-cost telecommunication chips are readily available for this frequency range. The IF signal is then up-converted by a real mixer, which produces both the sum and difference of the frequencies. The mirror sideband, being the frequency sum in this case, is easily rejected using a low pass filter. This can efficiently be achieved as the difference between both bands is 6 GHz due to the first up-conversion to the IF band. A low-pass-filter with 3 dB cut-off frequency of 9.1 GHz is used. This upper frequency is sufficient for most applications, and expansions are easily possible by exchanging the filter.

The PCB was developed during the master thesis of Robert Gartmann [137]. A photograph of the second version of the PCB is given in Figure 3.6. Besides the

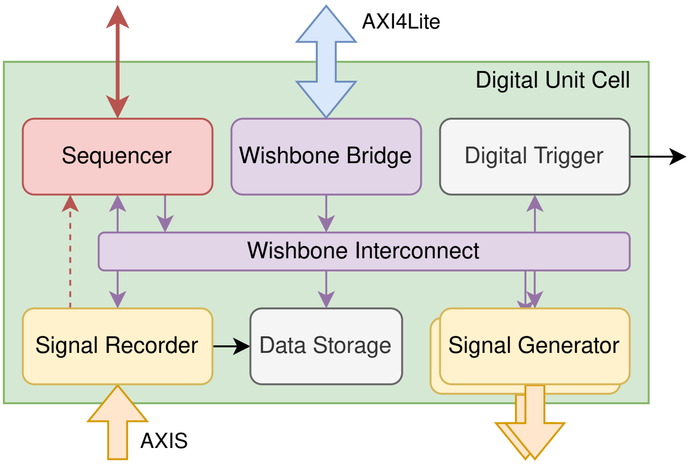


Figure 3.7: Architecture of a single digital unit cell. Arrows of the Wishbone (WB) bus indicate the direction from master to slave.

required RF electronics, it also integrates the power supply as well as an I^2C and SPI interface to control and configure the components. It can either be connected to a separate Raspberry Pi and controlled via Ethernet, or to the QiController which then provides access via the ServiceHub.

3.4 Programmable Logic

The core of the QiController is the PL which hosts the entities to control and read out the superconducting qubits. Its general structure was already depicted in Figure 3.2. The detailed functionality of the different modules as well as the inner structure of the digital unit cell is covered in the following sections. The whole design is operating with a 250 MHz clock derived from the converter clock. Its resource utilization, as well as for the single modules, is given in Section 3.7.1.

3.4.1 Digital Unit Cell

The digital unit cell contains all the relevant logic to generate and analyze pulses in the complex-valued baseband for a single qubit and to control peripheral devices for additional capabilities. Its architecture is sketched in Figure 3.7.

Two signal generators create the required pulses to control and read out the qubits as AXI-streams. A signal recorder takes the digitized signal from the ADCs and demodulates it to obtain the qubit state. A dedicated data storage collects the resulting data from the signal recorder and the sequencer. A digital trigger block can generate digital signals to address and trigger external lab equipment. All modules are controlled and activated by the sequencer which orchestrates their execution in single-cycle steps (4 ns). For fast conditional responses, the signal recorder directly reports all measured qubit states back to the sequencer which can then act accordingly. In all other cases, the modules communicate exclusively via a WB bus [138]. This also applies for trigger signals from the sequencer to control the execution of the other modules.

3.4.2 Wishbone Bus Infrastructure

Wishbone (WB) was selected as bus architecture because of its simplicity and resource efficiency compared to a full-featured AXI interface. A custom implementation is used to ensure deterministic timing which is essential for the targeted application. The WB bus inside the digital unit cell features a 16 bit address width and a 32 bit data width. It is also used by the PS to configure and access the modules in the unit cell. Therefore, an AXI4-Lite to WB bridge is utilized to translate the register accesses for the internal bus. It also performs an address translation from byte-based addressing as used by the AXI4-Lite bus to register-based addressing used by WB.

All slave modules have a special WB register interface implemented that guarantees a deterministic response time of 2 cycles without stalling. For the generation of the interface, a script was implemented that translates an abstract register definition file to the required VHDL entity. The register definition is stored as comma-separated values (CSV) and can be edited with a normal text editor or a spreadsheet program. This eases development and ensures that all interfaces are implemented in the same way.

A custom WB interconnect allows for two masters and up to seven connected slaves. Both sequencer and WB bridge are connected as masters. In case of a conflicting access, the sequencer always takes priority in order to keep deterministic timing during executions. With the interconnect, read and write operations take exactly 4 cycles from the sequencer to return a response. The timing diagram of such an operation is sketched in Figure 3.8. As the sequencer takes priority, the access from the WB bridge will take one extra cycle for buffering and might be stalled if the sequencer is currently accessing the bus. With the deterministic latency in mind, the interconnect was modified to allow a single pipelined register access each cycle

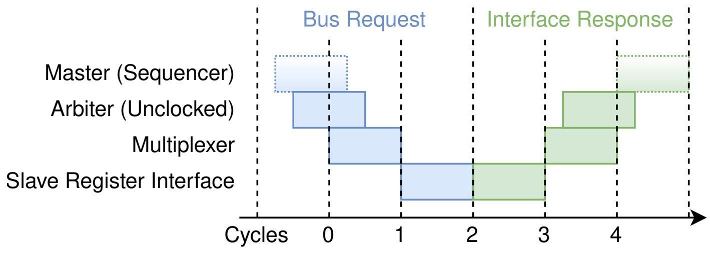


Figure 3.8: Timing diagram for a WB bus access from the sequencer to a slave register interface.

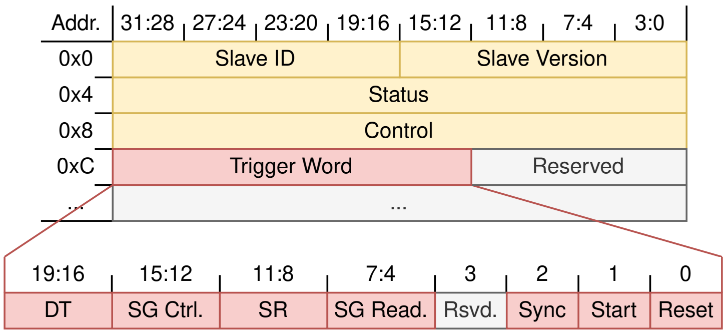


Figure 3.9: Common start of the register interfaces of all modules inside the digital unit cell. The address offset is given in bytes. Signal generators (SG), signal recorder (SR), and digital trigger (DT) are abbreviated.

on the bus. This even applies if a previous operation originating from one of the masters has not finished yet as the deterministic timing still allows to route the multiple requests independently. This way, one can ensure that the sequencer can always issue trigger commands on the WB bus with deterministic access latency.

The multiplexing between the modules happens according to the highest three address bits. While `000` up to `110` represent the according connected module, `111` acts as broadcast modifier. In this case, the bus operation will be forwarded to all connected slave modules at the same time. This feature is utilized to provide means to trigger all the connected modules with a common trigger word at the same time without utilizing a separate trigger infrastructure. The register interfaces of all slave modules are therefore starting in a similar way with an info, a status and a control register. Afterwards, a broadcast register follows with a 20 bit trigger word field that can be strobed by a write access. The remaining registers can be freely and independently used depending on the demand of the different modules. The common structure of the register interfaces and the trigger word is also depicted in Figure 3.9. Special trigger commands are shared between all modules to reset them, mark the start of an execution, and to synchronize the numerically controlled oscillators (NCOs) inside the two signal generators and the signal recorder.

3.4.3 Sequencer

The sequencer is the core of the digital unit cell. Its purpose is to control and orchestrate the execution of experiments with single cycle accuracy. It controls all connected slave modules and can, e.g., schedule pulses or start a recording. Via the WB infrastructure, it can furthermore even change configurations of other modules within the unit cell or read out data from their register interfaces. The user can define a sequence of operations in 4 ns steps using the RISC-V instruction set architecture (ISA) [139] together with a custom special-purpose set. The RISC-V ISA was selected as it is state-of-the-art, easily extensible, very flexible, and hardware efficient. It furthermore provides a rich ecosystem and is well established in the scientific community.

From the modular instruction sets of the RISC-V ISA, most of the base integer and multiplication set, as well as a custom special-purpose set for the sequencing is implemented. In total, 33 instructions are available for the sequencer as well as 32 registers (one being the `x0` register tied to zero). The following operations are part of the special-purpose set:

`TRIG`: writes the given trigger word to the broadcast register of all connected WB slave modules, compare Figure 3.9.

`WAIT-IMM`: delays the execution by the given number of clock cycles.

`WAIT-REG`: delays the execution by the number of clock cycles given in the defined register.

`WAIT-REG-TRIG`: same as `WAIT-REG` but reduces the wait time given in the register by one cycle. This can be used to wait a register-defined time and take the one cycle to execute a previous command into account. The main use case is to perform a trigger operation and then wait a time defined in a register until it completes. As `TRIG` and `WAIT-REG` are two separate commands, the total delay would be one cycle too long. Therefore, the separate `WAIT-REG-TRIG` is implemented.

`SYNC-EXT`: waits for external input before continuing with the program execution. E.g., this can be the resulting qubit state returned by the signal recorder.

`SYNC-START`: ends the execution of the sequencer and returns to an idle state where it waits for a new start command.

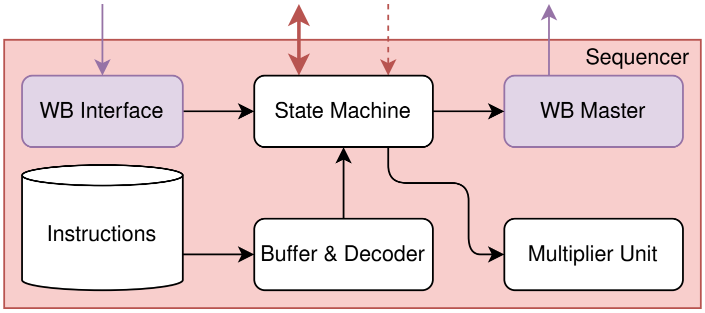


Figure 3.10: Internal structure of the RISC-V-based sequencer within the digital unit cell.

A list with the implemented instructions from the base integer and multiplication set is given in Appendix B.1.

Most instructions are optimized to execute in only one cycle for highest performance. Only multiplication takes 6 cycles in order to relax the timing requirements for the 32 bit times 32 bit multiplication. Instructions entailing a jump in the program counter take 3 cycles. These are branch instructions if the comparison yields true, as well as the unconditional `JAL` jump operation. The different wait operations take as long as specified in the command and the sync commands might wait an undetermined time on external input.

The internal structure of the sequencer is shown in Figure 3.10. Up to 1024 instructions can be stored in the internal block RAM (BRAM) of the sequencer. With typical experiments requiring tens of instructions or less to be executed, this is sufficient for nearly all imaginable experiments. A state machine fetches the instruction words from this memory via a buffering and decoding stage. It then performs the adequate operation depending on the RISC-V instruction obtained. Multiplication is handled by a separate multiplier unit realized with four digital signal processing (DSP) slices. The sequencer has both a WB master and slave interface. As for every other module, the slave interface is used to configure and control the sequencer. The master interface is used to trigger the connected slave modules and can also reconfigure or fetch data from them. This can be used to change pulse properties like frequency, phase or amplitude, or to vary the settings for the recording, just to name two examples. One thereby gains a lot of additional flexibility and capabilities what the sequencer can handle directly in hardware. The according load and store operations will take exactly 8 cycles as they wait for the acknowledgment of the WB bus and a possible read data value. Of these 8 cycles, 4 cycles account for the deterministic latency of the bus and the remaining 4 for processing the operation in the sequencer, applying the output to the bus and processing the return signals from the bus.

Contrary, trigger commands are applied to the bus using pipelined block write operations and the sequencer does not wait for a response but directly continues with the next command. This way, trigger commands can be issued each cycle without stalling. However, after trigger operations, the bus is still processing while the sequencer already executes the next operation. If this is a normal load or store operation, it will first have to wait until the bus is not busy anymore. In this case, the latency to perform such an operation might also be longer than 8 cycles. This could be further optimized by always applying pipelined block write operations and exploiting the deterministic latency of the modified WB bus. For now, the WB master interface of the sequencer was designed in a way that it can also be connected to any normal WB interface without causing any compatibility issues.

A more detailed coverage of the implementation aspects of the sequencer can be found in the master thesis of Rainer Illichmann [140].

3.4.4 Signal Generator

The signal generator handles the generation of pulses in the complex-valued baseband with a common frequency and well-defined, stable phase relation. Users can load 15 different pulse configurations into the module, corresponding to the available 4 bit trigger command value. The sequencer uses this value to start the generation of the corresponding pulse in the signal generator. Pulses are defined by their envelope which is then multiplied within the module by an NCO signal acting as global phase reference and defining both pulse phase and frequency. It is not possible to store the baseband pulses directly as a well-defined and stable phase relation between consecutive qubit control pulses is crucial. The structure of the signal generator is presented in Figure 3.11.

All configuration, as well as the trigger from the sequencer, is fed to the module via the WB interface. The 4 bit trigger command can select one of 15 pulse configurations, which are called trigger sets. Trigger value 0 is reserved for the no operation case. For each trigger set, the following properties can be individually specified:

- The duration of the pulse in cycles. As four samples are handled each cycle, sample-accurate, sub-cycle precision of the pulse can be achieved by padding the envelope values with zeros to fill up a full cycle.

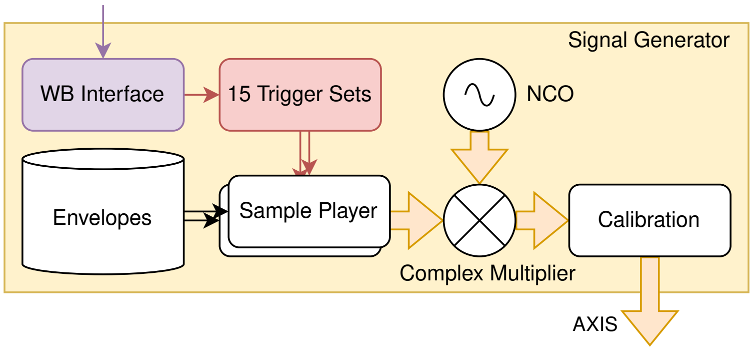


Figure 3.11: Internal structure of the signal generator within the digital unit cell.

- The phase offset of the pulse relative to a global phase reference inside the module. Using this property, the rotation axis on the equator of the Bloch sphere can be selected.
- I and Q envelope sample values of the pulse.
- Option to hold the last envelope sample value until another trigger is received. This results in continuous wave (CW) operation and can also be used to create variable length pulse shapes like a trapezoid.
- Option to persist the phase offset in the global phase reference. This enables the user to perform a virtual Z rotation.

The envelope values are stored inside the envelope memory which is an 8 kB large true dual-port BRAM. It can store 4096 real-valued samples (16 bit each) corresponding to up to about 4 µs of pulse data if I and Q share the same envelope samples. As most pulses are in the order of tens to hundreds of nanoseconds, this is enough for most applications and multiple pulses to be stored. Trigger sets define the address offsets to the envelope values in the BRAM. These can be the same address if the envelopes for I and Q are identical, or different, if not (e.g. for DRAG pulses).

When a trigger arrives and selects one of the 15 trigger sets, the configuration of this trigger set is activated and the execution of the sample players is started. Two sample players are instantiated, one for the I, and one for the Q envelope. They simultaneously read the I and Q envelope values out of the dual-port BRAM. After the duration of the pulse has passed, the sample players either turn off the envelope output or keep the last sample value, depending on the option given in the trigger set.

The I and Q envelope values are then forwarded to a complex multiplier. There, they will be multiplied by the oscillating complex quadrature signal of the NCO. The NCO is based on the direct digital synthesis technique and acts as global phase reference for all pulses generated by this signal generator. It has a configurable frequency and a global phase offset. The latter can be temporarily shifted for a single pulse by the configuration in the trigger sets. It can also be intentionally changed to perform arbitrary virtual Z rotations. Synchronization of the frequency references between multiple modules is performed by simultaneously resetting the internal phase of all NCOs to zero. This is achieved by the special sync trigger command (compare the structure of the trigger word in Figure 3.9).

At the output of the multiplier, one obtains the digital pulse in the baseband. Before it is forwarded towards the DACs as AXI-stream, the I and Q quadratures can be independently calibrated by a scaling factor. Factors from zero to two (excluding) are possible with high precision as the factor is implemented as a 1:15 fixed point value, one for the I and one for the Q component. If the multiplication product exceeds the valid range of the 16 bit data width, it will be clipped and a saturation flag will indicate this to the user until it is manually cleared. From receiving the trigger at the WB register interface until the pulse is output as AXI-stream, the signal generator needs in total 13 cycles. A breakdown of this delay is given in Appendix B.2. It also includes two pipeline stages to relax timing constraints.

3.4.5 Signal Recorder

The signal recorder receives the digitized microwave signals from the converters in the complex-valued baseband. After an initial signal conditioning, a digital down-conversion (DDC) follows using a complex mixer followed by an accumulator acting as boxcar integrator. The demodulated I and Q value are further evaluated to obtain the estimated state of the qubit. All results are forwarded to a separate data storage module. There, the desired results are selected, aggregated and consecutively persisted in one or multiple BRAMs for later retrieval by the PS and the user. More information on the data storage is given in the next section. The structure of the signal recorder is depicted in Figure 3.12.

A typical readout operation will trigger a signal generator to play a pulse with a certain length. After some latency and electrical delay through the setup, a modified version of the pulse will arrive at the recording module. Then, the pulse has be demodulated and evaluated by the signal recorder to obtain the amplitude and phase response of the investigated system. To cover this, the sequencer will not only trigger the signal generator but simultaneously also activate the signal recorder. In the signal recorder, a trigger delay is specified to compensate for the latency of

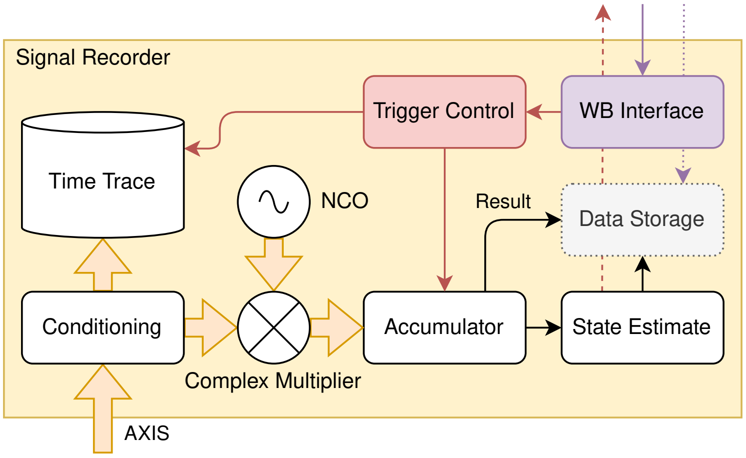


Figure 3.12: Internal structure of the signal recorder within the digital unit cell.

the pulse through electronics and experiment setup. By adjusting the duration of the recording to the length of the pulse, the accumulator will then take the whole returned pulse into account. Using these two parameters, the user can define a rectangular recording window which will be evaluated by the DDC.

Due to mixer and other imperfections in the analog setup, the raw I and Q data from the converters might not be well balanced in amplitude or have a 90° phase relation. To correct for this, a signal conditioning is performed on the raw input data. It consists of a multiplication with a 2×2 conditioning matrix M_{cond} following after an offset subtraction to remove a potential DC offset:

$$\begin{pmatrix} I_{\text{out}} \\ Q_{\text{out}} \end{pmatrix} = M_{\text{cond}} \left[\begin{pmatrix} I_{\text{in}} \\ Q_{\text{in}} \end{pmatrix} - \begin{pmatrix} I_{\text{offset}} \\ Q_{\text{offset}} \end{pmatrix} \right] \quad . \tag{3.1}$$

The corrected raw time trace is then stored inside a separate BRAM whenever a recording is performed. The memory thus always contains a snapshot of the raw data used for the last demodulated I and Q value. Due to the limited size of the BRAM, it can only store up to 1024 ns of raw data. If longer recording durations are set, only the beginning of the incoming raw time trace will be persisted in the BRAM. For most applications, the recording pulse is considerably shorter than this length. The stored time trace can later be used for debugging purposes or to visualize the raw input.

At the same time, the conditioned signal will also be down-converted by complex multiplication with a reference oscillation from an NCO. The NCO is of the same type as in the signal generator. To obtain reproducible results, it is important that both NCOs are synchronized at the beginning of an experiment to have a stable phase relation. The phase offset of the NCO in the signal recorder can furthermore be adapted to rotate the down-converted signal in the I/Q plane to a desired position. The frequency of the NCO is chosen to be the negative value of the NCO used in the signal generator of the corresponding readout pulse. The complex multiplier thereby shifts the frequency of the input signal carrier to DC. To make configuration easier, the quadratures of the NCO are swapped in the recording module which effectively negates the frequency. Both readout signal generator and recorder can thus be configured with the same frequency value.

Afterwards, a low-pass filter and decimation are necessary to average the resulting I and Q component. These components then correspond to the amplitude and phase response of the resonator. In the signal recorder, a boxcar integrator is implemented by using an accumulator to add up the samples over an adjustable time window. This recording duration can be freely chosen by the user and typically is a trade-off between a desired fast readout operation and a sufficient signal-to-noise ratio. Alternatively, a FIR filter and decimation could be used for a better low-pass characteristic. At the same time, the accumulation yields a smaller latency to obtain a result which is the reason why it was preferred.

The accumulation will add up the incoming 16 bit I and Q values separately for the specified time window in two 32 bit accumulator registers. Therefore, a minimum of $2^{16} = 65\,536$ samples can be averaged without creating an overflow, corresponding to recording durations of over 65 µs. The values will then be reduced back to 16 bit. An overflow of the values during this operation can be prevented by shifting the values by a user-defined number of bits before reducing the bit size. This value shift is, by default, automatically calculated by the drivers based on the recording duration so an overflow cannot happen. Especially if the digital input range of the ADC is not maxed out, it can be beneficial for the signal quality to intentionally reduce this shift to increase the obtained signal strength. A status flag will indicate if an overflow happened when reducing the bit size after the value shift. This gives the user the feedback that the obtained data is corrupted and the value shift should be increased.

While conditioning and complex multiplication are performed continuously, the boxcar integration as well as the storage of the raw time trace are only activated when the signal recorder receives a trigger signal. Once the recording duration has passed, the accumulated I/Q result value is passed to the data storage, as well as to a state estimation routine. In the latter, the result is transformed into a binary information of `0` or `1` corresponding to the estimation of the measured qubit state. This state result will be directly returned to the sequencer which can be programmed

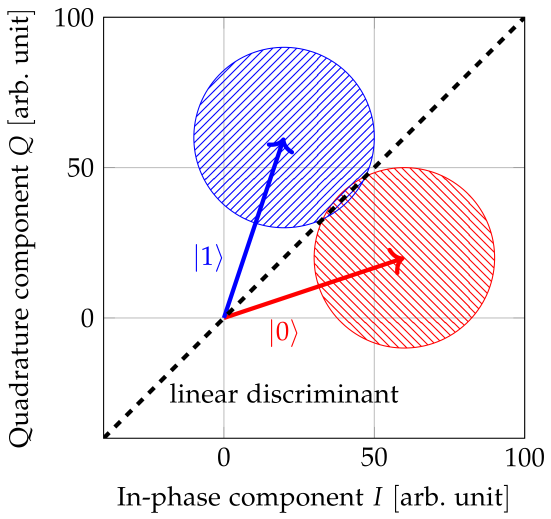

100
50
0
0
50
100
Quadrature component Q [arb. unit]
In-phase component I [arb. unit]
$|1\rangle$
$|0\rangle$
linear discriminant

Figure 3.13: Illustration of the I/Q plane with noisy measurement results of the readout pulse indicated by two areas for the qubit states $|0\rangle$ and $|1\rangle$. The dashed line is the linear discriminant to distinguish between both states.

to wait for this value and store it in a register using the `SYNC-EXT` operation. The state will also be passed to the data storage where it can be aggregated and saved for later retrieval. To distinguish between states $|0\rangle$ and $|1\rangle$, in most cases, it is sufficient to determine a linear discriminant of the form

$$\boldsymbol{a}^T \boldsymbol{m} + b = 0 \tag{3.2}$$

for the I/Q plane of the result values. By checking on which side of the discriminant a measurement result $\boldsymbol{m} = (I, Q)^T$ is located, the qubit state can then be inferred. This is illustrated in Figure 3.13. The discrimination can be realized by evaluating the left hand side of Equation 3.2 and checking if the result is larger or smaller than zero. The values $\boldsymbol{a} = (a_\mathrm{I}, a_\mathrm{Q})^T$ and b are calibrated and configured by the user as 32-bit signed integers. As only integers are allowed, special care has to be taken that rounding errors do not play an important role. This can be easily prohibited by multiplying Equation 3.2 by a factor $k \gg 1$. As the equation still describes the same linear discriminant, the coefficients a_I, a_Q and b can also be multiplied by this common factor k without changing the discrimination. By choosing k appropriately, the relative errors when rounding the coefficients become small. If a result is located directly on the discrimination line, it is, for simplicity, attributed to the positive case. This way, the state estimation can simply calculate the left hand side of Equation 3.2 and use the sign bit of the result as estimated qubit state.

For simple experiments, the signal recorder also provides an averaging functionality. Single results obtained from the accumulator will be further summed up until the module is reset externally. This is especially helpful if a single measurement should be performed and repeated many times to obtain an averaged I and Q result value.

Different operation modes of the signal recorder can be distinguished, based on the received trigger value. The following values are implemented:

`RESET`: resets all internal data of the module, including the averaged result.

`SINGLE`: performs a single measurement.

`ONESHOT`: performs a single measurement but does not include it into the result averaging. A typical use-case are two consecutive measurements where the first one is only used internally and will result in a state estimation on which the sequencer will react. The second one is then to obtain a measurement result of the experiment.

`CONTINUOUS`: continuously performs consecutive measurements and returns the values to the storage module. This mode can be used to obtain a seamless stream of demodulated results without the need of the sequencer to trigger each single measurement. The continuous mode will continue until a `STOP` trigger is received. This mode provides the basis for continuous operation over long periods, e.g. to observe state changes of the qubit over time (quantum jumps).

3.4.6 Data Storage

The data storage provides a configurable and flexible way to store experiment data from both the signal recorder and the sequencer. It contains multiple memory containers which can be filled with data independently and in parallel, e.g. to partition measurement results, or to persist values from the sequencer in parallel to the results of the recording module. From these memory containers, the data can later be retrieved by the user.

When the signal recorder performed a measurement, it passes the single results and belonging estimated qubit states to the data storage. To be efficiently stored, qubit states will be concatenated to 32 bit words. Either 32 or 10 states are combined, depending if only one bit is used or also higher states are accounted for (3 bit information per state). The user can then select between the following data sources to be stored in the memory containers:

- Single I and Q result values
- Single estimated qubit states

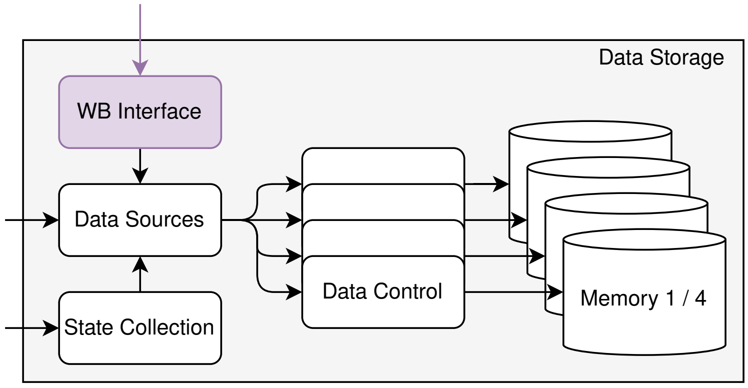


Figure 3.14: Internal structure of the data storage within the digital unit cell.

- Concatenated qubit states (either 10 or 32 per register)
- Input data from the WB interface

The last data source is a special register in the WB interface to which the sequencer can write to append values to the memory blocks. This way, the sequencer can also perform manipulations on the results before storing them or persist some additional values in the data storage.

The structure of the data storage is shown in Figure 3.14. It comprises four memory containers, each with a separate dual-port BRAM to store data. Each container provides an interface to consecutively append 32 bit values to the memory until it is full after 1024 values have been stored. It furthermore contains an option to use the memory as circular buffer and wrap the address instead of rising an overflow flag if the memory is full. In this mode, it can endlessly write data to the memory trusting that the PS fetches it again fast enough before being overwritten. The second port of the dual-port BRAM is mapped into the WB interface for direct read and write access from sequencer and PS. If a memory container is otherwise unused, the sequencer can also utilize it via the WB interface as a memory extension, e.g. for arrays. The memory containers can be configured via the WB interface and also expose status signals the same way, like empty, full, and overflow flags, as well as the current data size.

3.4.7 Digital Trigger Output

While the system covers most aspects to control and read out superconducting qubits, for some experiments it can be necessary to digitally trigger external

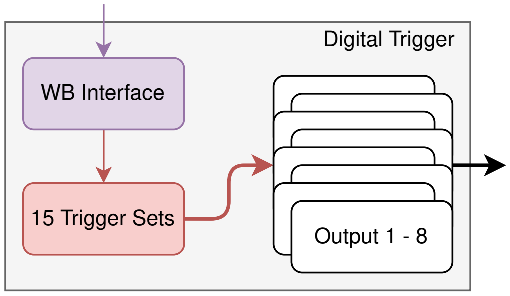


Figure 3.15: Internal structure of the digital trigger output inside the digital unit cell.

measurement equipment for additional functionality. A common use case are current pulses for special qubit gates. For these situations, the digital unit cell contains a digital trigger output module capable of generating trigger signals for external devices. These signals are routed to GPIO pins of the platform.

The structure of the digital trigger is depicted in Figure 3.15. Each unit cell has 8 digital outputs available that can be strobed. The module provides 15 trigger sets to define the behavior of the trigger signals. Using a bit mask, one can define which outputs to activate for a given trigger set. One can also control the number of cycles how long the outputs should stay asserted. A special option can be used for continuous activation until another trigger value from the sequencer is received. Each output can be individually inverted and a trigger delay specified. This delay is especially important to synchronize the action of external devices with the operation of the system.

3.4.8 Cell Coordinator

The cell coordinator synchronizes the execution of the digital unit cells. When started externally, the sequencers within the digital unit cells run independently. However, for experiments requiring multiple cells, the execution needs to be aligned in time. This is achieved by the cell coordinator which is connected to all digital unit cells in a star point architecture. Using this structure, the cell coordinator can synchronously start the sequencers of any subset of the available digital unit cells. It also aggregates their busy flags and exposes them to the user.

The structure of the cell coordinator is sketched in Figure 3.16. Via an AXI4-Lite register interface, the status of the digital unit cells can be monitored. The user can also trigger the synchronous execution of some or all digital unit cells by writing a trigger mask to the interface. The start trigger is then forwarded to the sequencers

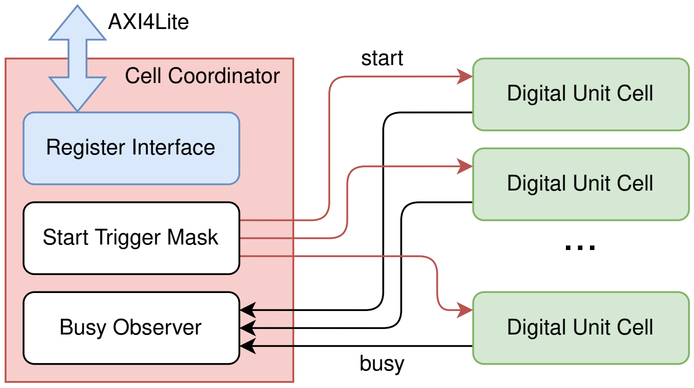


Figure 3.16: Structure of the cell coordinator connected as star point to all digital unit cells.

of the relevant unit cells. As the delays of the individual logic signals are matched, a synchronous start of execution is guaranteed. To stay synchronous during the execution, one can insert appropriate wait statements in the program instructions of the different sequencers.

In future, the cell coordinator could be extended to implement data exchange between unit cells, like measured qubit states or register contents of the sequencers. Also, means for an explicit synchronization between multiple unit cells during the execution could be added. This can be necessary when, e.g., one cell performs a conditional execution and the timing cannot be inferred beforehand. However, the structure of the exchange between digital unit cells will also likely change depending on the evolving requirements concerning multi-qubit chips. As it is not required for most experiments, it was decided to not implement it yet but first let the research field evolve further. With more detailed requirements being known in future, the star point structure might also be replaced by another architecture which better suits the requirements of future quantum processors.

3.4.9 Cell Multiplexer

The cell multiplexer connects the digital unit cells with the DAC and ADC channels and can combine and split up multiple signals using FDM (see Appendix A.3). The required routing of the signals between converters and cells depends on the design of the chip and the remaining setup. The signals are either generated for separate microwave lines, or need to be frequency-division multiplexed on a single line. With the cell multiplexer, one can flexibly route the signals from the different digital unit cells to the converters and back. Its functionality was already depicted in Figure 3.3.

All signals from the converters and the digital unit cells are connected as AXI4-streams. Frequency-division multiplexing is achieved by digitally adding multiple signals together. As these are generated by the digital unit cells with different frequencies, the sum of the signals is a multi-tone signal which will then be synthesized by a DAC. During the addition, an overflow can happen if the resulting signal exceeds the valid range. For this case, the cell multiplexer has an overflow detection which will set a flag and alert the user who then can reduce the individual signal amplitudes. Returning signals from the ADCs are split up onto the relevant digital unit cells. Multiplexed signals are simply routed to multiple cells. The discrimination between different carriers happens inside the signal recorders which perform DDCs with different intermediate frequencies. Other signal contributions are thus suppressed if their frequencies are sufficiently spaced apart to be removed by the low-pass filtering.

3.4.10 Platform Information and Management Core

The platform information and management core (PIMC) is an FPGA module designed to provide standardized access and recognition of different hardware images. It provides an AXI4-Lite register interface to query the status of the running image, as well as to perform a global reset operation. The status also includes a project and a platform ID to identify the built hardware image and the platform for which it is intended. This information can later be used by other software components to check for compatibility with the system. It also provides information of the build revision as well as the timestamp when the FPGA bitstream was created. A busy flag indicates if the platform is currently performing an operation. A ready flag tells the user if the platform is fully initialized and ready to use. This includes a check if all clocks are configured and running correctly, and if the connection to the converters is established and working. Finally, it also connects to the LEDs of the board to show this status information to the user. For this purpose, it also divides the clock down to a frequency of 1 Hz which will be used to let an LED blink at this frequency to show the design clock is working correctly. The PIMC is also actively used in other projects at the Institute for Data Processing and Electronics (IPE).

3.5 Software Stack

While the functionality to interact with the qubits is completely implemented within the FPGA to leverage its fast parallel processing and nanosecond accuracy, more complex and versatile tasks are performed in software. By utilizing a multi-processor system-on-chip (MPSoC) platform, the processors are tightly linked to the FPGA and can interact with it. Still, a latency of a few hundreds of nanoseconds occurs for the communication between PL and PS. This is too slow for time-critical tasks like pulse sequencing but can be used for more high-level control over the setup.

Two different subsystems have been developed during this thesis and will be explained in more detail in the following sections: First, the ServiceHub which runs on the APU and provides an interface between the user and the platform. Second, the Taskrunner on the RPU that complements the execution on the PL with versatile control and data aggregation capabilities.

3.5.1 ServiceHub

The ServiceHub framework [135] is a modular communication server that runs on the APU as Linux application. It provides a flexible abstraction of the platform's functionality and convenient means to interact, control and configure the different parts of the QiController. Besides facilitating configuration and control capabilities for the different PL modules, the interface is also used to transfer measurement data from the system to the user. An example would be to read the content of the memory blocks inside the storage modules or transferring data from the Taskrunner which is treated in the following section. The ServiceHub is based on C++ and access to the platform is implemented using remote procedure calls (RPCs) utilizing the open-source framework gRPC [141]. In gRPC, the interface is specified in language-agnostic protobuf definition files. From these files, client interfaces can be automatically generated in a large variety of supported languages, like Python, Go, C/C++, Rust, or C#.

The architecture of the ServiceHub is sketched in Figure 3.17. Its modularity is achieved by separating different functionality into different plugins. The plugins are then dynamically loaded by the ServiceHub according to a specified configuration. Each plugin has access to the ServiceHub infrastructure via a standardized interface. This provides means for plugin management and health checks, storage, logging and user configuration, as well as SysFS and device tree access. Each plugin also defines its own gRPC remote procedures that will be exposed to the user via the ServiceHub. Plugins can also interface with and control other plugins to coordinate between them or provide more complex calibration and high-level control routines.

For each type of PL module, a ServiceHub plugin exists which facilitates the user access to these modules. Plugins are separated into a part with the gRPC methods that will be exposed to the user, and one or more endpoints that implement the drivers to access the underlying hardware objects. As multiple unit cells are

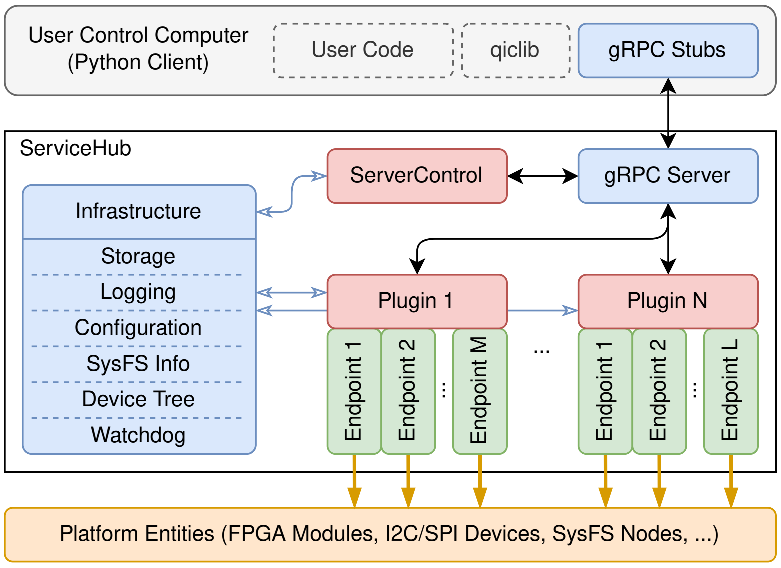


Figure 3.17: Architecture of the ServiceHub as modular interface between platform entities and user control computer.

implemented, e.g. plugins addressing the contents of a unit cell have multiple endpoints, like the plugin for the signal generator or the sequencer. In these cases, an endpoint index needs to be additionally passed to the remote procedure calls to select the endpoint index to use. Besides the PL modules, also other plugins exist, like for the RF frontend electronics that is controlled via SPI and I^2C, or for the Taskrunner.

Besides the plugins which are dynamically loaded depending on the hardware project used and the available entities within the platform, a ServiceHub control plugin is loaded. It provides a default service for the user which is always present whenever a ServiceHub is running on the system. It can be used to query a list of loaded plugins, the current status of the setup and the logs, as well as to control the platform, like reloading the ServiceHub or rebooting the system.

From a conceptual point of view, the ServiceHub exposes functionality to the user, and when called translates these requests to internal signals within the platform. One example would be changing the frequency of the NCO within the signal recorder, as is depicted in Figure 3.18. The signal recorder plugin exposes an RPC method called `SetInternalFrequency` for this purpose. Its function arguments are the endpoint index, as well as the frequency in hertz. Based on the index, the memory address range of the corresponding signal recorder within the AXI4-Lite

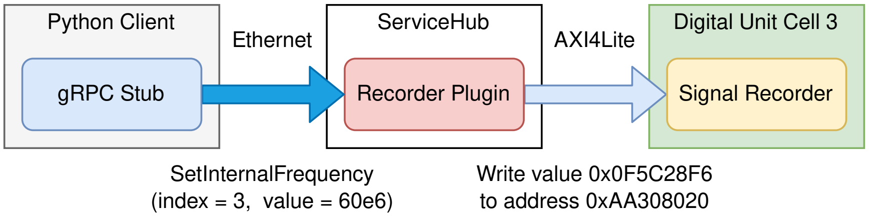


Figure 3.18: Conceptual view of a remote procedure call to change the frequency inside a signal recorder within the FPGA.

interface is selected. The frequency is translated inside the plugin from hertz to a phase increment value which is then written to the appropriate register within the FPGA using the AXI4-Lite bus. The plugins thus also encapsulate and abstract from implementation-specific details of the endpoints that are not relevant for the user.

3.5.2 Taskrunner

Complex parameter variations during experiments are commonly necessary to sweep a specified parameter range. Measurement results need to be further processed, evaluated, and aggregated. Both requirements can be highly experiment-specific and are not well suited to be implemented within the FPGA due to lacking flexibility and a high entry barrier. As an alternative, a software-based approach is pursued. Yet, largely deterministic timing and low latency control of the PL is required to reliably process an incoming data stream and reduce the time overhead for control schemes. Utilizing the APU is not convenient due to uncontrolled context switches by the Linux operating system, as they occur, e.g., by external requests via Ethernet. If the data buffer between FPGA and APU would overflow during such an interruption, data loss would be the consequence.

Based on these considerations, a separate subsystem is implemented on the real-time co-processor of the MPSoC architecture, the Taskrunner [47]. It provides convenient access to the RPU and enables users to dynamically execute arbitrary C code, so-called user tasks. Thereby, complex control schemes and flexible online data processing can be implemented directly in software.

The structure of the Taskrunner framework is sketched in Figure 3.19. It consists of three parts: the Taskrunner firmware on the RPU, a ServiceHub plugin, and a cross-compiler on the APU. The firmware on the RPU handles the execution of user tasks. It also provides an interface to exchange data and status information with the APU. The ServiceHub plugin provides an interface of the Taskrunner

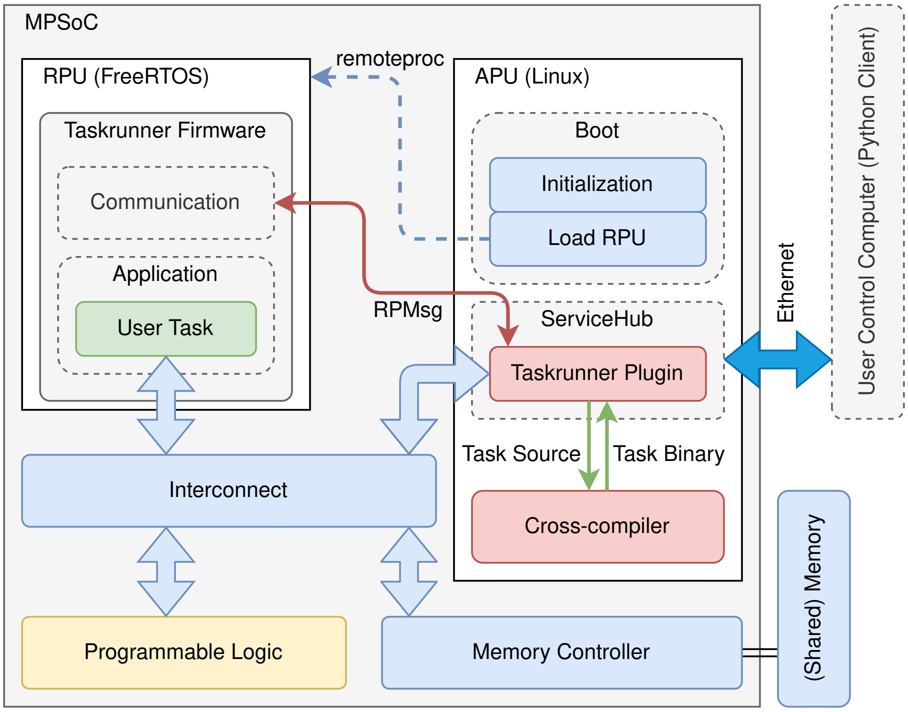


Figure 3.19: Structure of the Taskrunner framework inside the MPSoC architecture.

to the user client. It interacts with the interface of the firmware on the RPU and exposes its functionality to the client. To transfer data to the user client, a shared memory region inside the DDR4 memory is used. In it, so-called data boxes are created which are written by the user task in the RPU. These data boxes are used to store obtained or processed values. They are then fetched by the ServiceHub plugin on the APU and transferred to the client. As user tasks are conveniently specified using standard C language, the source code is compiled on the fly directly using a cross-compiler on the APU.

The Taskrunner firmware is hosted by the real-time operating system FreeRTOS. Both are loaded together during the Linux boot sequence from the APU. The firmware consists of two threads that are scheduled by the FreeRTOS scheduler. One handles the communication with the ServiceHub plugin on the APU, the other controls the execution of the user task, in the following called application thread. The communication thread is only scheduled when the plugin is requesting information from or sending data to the Taskrunner. Unwanted context switches can be inhibited by defining a critical section in the user task. Both Taskrunner firmware and user task are located inside the tightly-coupled memory (TCM) of the RFSoC to ensure the execution of operations with deterministic timing. The RPU is operated in lock-step mode, giving access to the full 256 kB TCM capacity.

The APU acts as host, controlling the RPU slave via remoteproc, based on the Open Asymmetric Multi Processing (OpenAMP) [142] framework. The communication between Taskrunner firmware and ServiceHub plugin is handled by the RPMsg protocol which utilizes inter-processor interrupts and a shared DDR4 memory. User tasks are transmitted as source code to the ServiceHub plugin on the APU where the cross-compiler for the RPU is invoked. A library is provided to abstract from a pure register-based access when interacting with digital unit cells on the FPGA. During compilation, the task is linked against the RPU firmware binary in order to provide the C standard libraries as well as an interface to interact with the Taskrunner. The resulting binary is loaded onto the RPU using the RPMsg interface where it is stored inside the TCM by the communication thread. Direct access from the APU to the TCM is not required.

Prior to starting the task execution, parameter values can be passed to the user task. The parameter list is copied into a designated 15 MB DDR4 memory region by the ServiceHub plugin. A wide range of parameters can be passed, like a list of delays, the number of average repetitions to perform, or some boolean information. When the user starts the task execution, the communication thread of the Taskrunner ensures that no task is currently running and notifies the application thread about the received start command. The application thread then calls the entry function of the user task in the context of the application thread.

The user task can utilize a provided function library to interact with the Taskrunner. The user-defined parameter list can be accessed to customize the task's behavior. A progress value can be specified and queried during execution to monitor the task progress. Error messages can be specified and will be appended to a queue that can be fetched by the Taskrunner plugin within the ServiceHub. Finally, functions to operate with data boxes are provided to exchange data with the user. The full list of available commands is given in Appendix C.1.

Data boxes are located in a 480 MB heap in the DDR4 memory. Custom memory management functions exist to request, finish and discard data boxes of an individually settable size. Finished data boxes can be queried by the client, also during the user task execution. The ServiceHub plugin then fetches a list of corresponding DDR4 memory addresses and sizes from the Taskrunner. After sending the data boxes to the client, the corresponding memory regions are freed and can be reused again.

When transferring data, two different operation schemes can be distinguished. For shorter experiments, it is sufficient to collect the data after the user task completed. The data boxes are finished at the end of the user task and the client checks if the task is done prior to fetching the data boxes. During task execution, only the progress value is exchanged with the client. Longer experiments, on the contrary, might exceed data sizes that can be handled by the 480 MB data box heap. Therefore, data boxes can also be finished during the user task execution and collected in parallel by the client. This prevents memory exhaustion and provides the means to directly visualize the evolving measurement results while the experiment progresses.

3.5.3 Boot and Initialization

The heterogeneous platform consists of a lot of different components that need to be initialized when starting the system. When the power is turned on, a first stage boot loader (FSBL) is started. It configures the RFSoC and programs the FPGA bitstream with the components discussed in Section 3.4. Then, it starts U-Boot which will select the appropriate boot device and initialize the Linux kernel located on it, e.g. on an SD card. During Linux boot, peripheral devices get configured. This also includes the external Texas Instruments LMX2594 clock chips that supply the sampling frequency to the DACs and ADCs. They get configured to operate at 4 GHz output frequency. Only after setting up the converters, the PL design is functional, as its main clock is derived from the converter clock. To ensure proper initialization, a reset of the PL design is triggered once the clock is ready.

The Taskrunner is also initialized during Linux boot. An init script first ensures that the RPU is stopped. Then it copies the Taskrunner binary including FreeRTOS as firmware into the TCM. After copying the firmware, the RPU is started and it is checked that the processor is running. Finally, a Linux kernel module handling the inter-processor communication between APU and RPU is loaded. After the Linux boot process finished, the ServiceHub is started and the required ServiceHub plugins are loaded, depending on a given configuration file. The plugins themselves can perform further initialization routines. For example, the plugin for the Taskrunner will open a connection via RPMsg to the RPU and the firmware running on it.

3.5.4 System Reliability

As the platform is designed to be operated in the laboratory, it is important that it runs as reliable as possible and only with minimal downtime. One measure to avoid an otherwise necessary in-person maintenance is the integration of a fail-over boot mechanism. Whenever the normal operation fails or the system gets stuck during Linux boot or afterwards, a separate fail-over system is loaded instead. In consists of a minimal Linux setup without the qubit control functionality. Yet, it enables remote access via Ethernet to the system and thus the possibility of a remote maintenance. The fail-over system is stored on the QSPI flash of the ZCU111 board and loaded as random access memory (RAM) disk to prevent overwriting the fail-over. This way, also the SD card containing the full image can be mounted and altered to restore operation. U-Boot is expanded by a boot loop detection counter which will trigger a boot from QSPI if a loop is detected, and from the SD card otherwise. During normal operation, a watchdog supervises the ServiceHub and triggers a restart of the system if it is not responding anymore. This way, a non-functional system will cause multiple restarts which eventually lead to an initialization of the separate fail-over system. Furthermore, the system can also be triggered manually during the U-Boot startup process for manual maintenance when necessary. Robin Bauknecht covered the topic of system reliability in more detail in his master thesis, including an analysis of possible failure scenarios and belonging mitigation strategies [143].

3.6 Scalability

Current systems mostly consist of only few superconducting qubits that need to be controlled and read out. However, when targeting a useful quantum computer, many qubits need to be combined into a quantum processor. This will require reinventing the complete hardware and software stack. Yet, also for the intermediate regime of tens of superconducting qubits, technical solutions are required. Concepts for this intermediate regime are described in the following sections.

3.6.1 Single Platform

As already motivated and introduced in Section 3.2, multiple qubits can be controlled by a single QiController as the architecture is divided in digital and analog unit cells. A sketch of this concept is presented in Figure 3.3. The provided experiment description language QiCode, described in Section 4.4, is also designed from ground-up to be a scalable experiment description language and to support multi-qubit experiments as well as quantum algorithms. The modular concept of both software and hardware makes it possible to control an arbitrary number of qubits, as long as the FPGA resources are sufficient and the qubits can be controlled with the available amount of DAC and ADC channels. With the resource utilization given in Section 3.7.1, it is clear that a total of 15 digital unit cells can be currently implemented on the FPGA. A more stringent limit is typically imposed by the available DAC channels. When both readout and manipulation signals are separately multiplexed, any number of qubits can be controlled with four DAC and two ADC channels. However, in many cases, only the readout is multiplexed while the manipulation is not. In this case, the eight available DAC channels of the Xilinx ZU28DR RFSoC on the ZCU111 can only be used to interface with a total of three separate qubits.

An alternative approach would be to use a chip with more DAC channels. For example, the Xilinx ZU49DR RFSoC offers 16 DAC and ADC channels. With it, up to seven qubits can be read out in frequency multiplex when controlled via individual microwave lines. Another approach would be to perform a first frequency up-conversion already in the digital domain to replace the separate I/Q signals of the modules already by a real signal before converting them to the analog domain. As introduced in Section 2.4.3, the DAC tiles offer specialized DSP components like a digital I/Q mixer that can be used to perform this conversion. A digital up-conversion (DUC) of the I/Q signal to a real signal in the gigahertz domain can be established this way. With a converter sample rate of 4 GHz, the signal could be positioned in the second Nyquist zone around 3 GHz. Using a similar procedure as explained for the analog unit cells in Section 3.3, the signal could be further converted to the target frequency. With the ZU49DR, the DACs could also be clocked with up to 10 GHz, thereby allowing direct RF synthesis of signals with the desired frequencies. In both cases, the number of channels could be cut in half. Using multiplexed readout pulses that also utilize a DUC, up to seven qubits can be controlled with the ZU28DR and up to 15 qubits with the ZU49DR. However, a reduced phase-noise performance is to be expected compared to analog I and Q generation at lower frequencies which will later be up-converted by an I/Q mixer. Therefore, it first needs to be established if this scheme can be used to perform high-quality qubit control pulses.

3.6.2 Multiple Platforms

As seen in the last section, each QiController can control a limited number of qubits, on the order of ten. If more qubits need to be controlled, e.g. to interface with a medium-scale quantum processor, multiple QiController can be combined. The concept on how to scale up these systems is sketched in Figure 3.20. Each QiController interfaces with a subset of qubits in the experiment and might additionally control external laboratory equipment. All platforms are connected to and configured from a single user client via a regular network infrastructure.

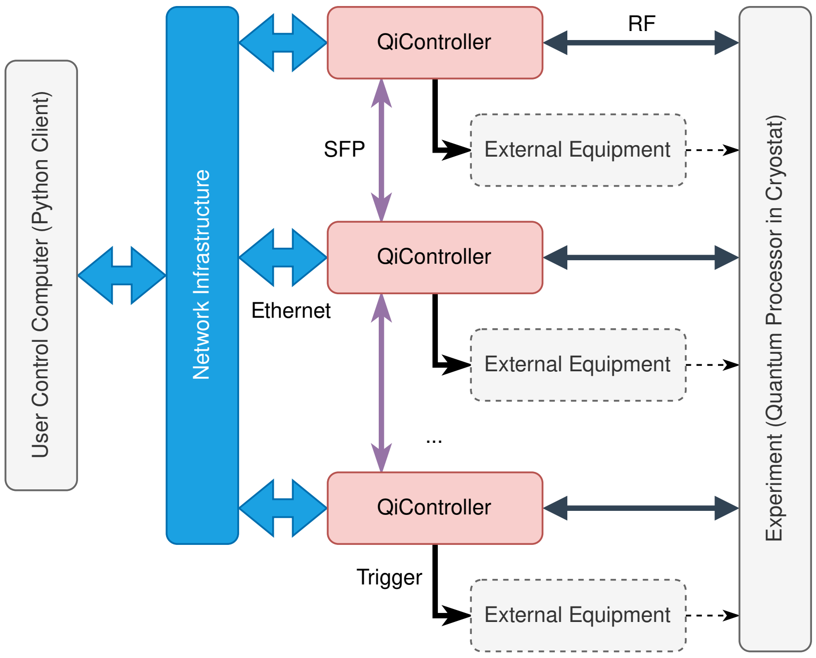


Figure 3.20: Concept of scaling the qubit control system up to multiple platforms.

When multiple platforms are operated in parallel, it is very important that they act synchronously, including a rigid clock synchronization. Pulses scheduled on multiple platforms have to be generated and output at the same time and with matching frequencies, no matter on which platform they are generated. Additionally, fast data transfer between the platforms is beneficial in order to, e.g., provide means for inter-platform feedback operations.

To fulfill these requirements, a multi-platform synchronization scheme is outlined in the following. It is based on daisy-chained SFP connections between the QiControllers. Clock and time synchronization can be handled by a scheme inspired by the White Rabbit project [144]. White Rabbit is a collaborative project including CERN, GSI, and others to develop a deterministic network for sub-nanosecond time synchronization and general purpose data transfer based on Ethernet.

When scaling up superconducting qubits, they are typically arranged in a two-dimensional grid with connections to all four neighboring qubits. Therefore, a QiController should ideally also be able to interface with a unit cell of this grid that can be repeated along both axes of the chip. Data transfer between multiple QiControllers is especially relevant for the control of neighboring qubits on the

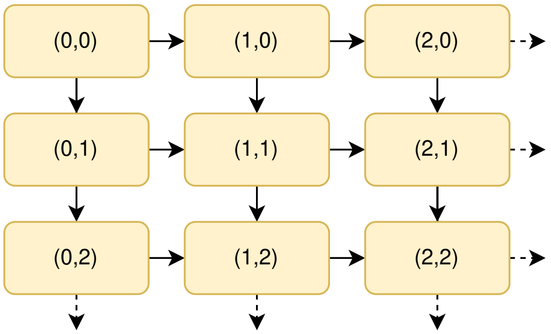


Figure 3.21: Topology of the synchronization connections between multiple QiControllers. The arrows point from master to slave.

quantum processor at the boundaries of these grid unit cells. In a classical daisy chain architecture, it would not be possible to minimize the latency for data transfer in all four directions. Thus, a modified synchronization and data exchange topology is proposed that is oriented along the needs of a near-term quantum processor. It itself forms a two-dimensional grid where each node is a QiController that has a longitude and a latitude index as identifier. To provide a deterministic chain of synchronization, each platform has two master ports and two slave ports for synchronization. The topology is visualized in Figure 3.21. In this structure, the latency overhead for a data exchange would grow linearly with the distance of the QiControllers in the two-dimensional grid. It thereby minimizes the latency between the control logic of neighboring qubits on the quantum processor.

Multi-platform synchronization will also require adaptations of the PL of the individual QiController, as shown in Figure 3.22. The design needs to be extended by a global time counter which will be incremented each cycle and can be adjusted via the board synchronization mechanism. The cell coordinator also has to be extended so it can start all or a subset of the digital unit cells when the global time reaches a certain value. The board synchronization mechanism will implement the communication and time synchronization via the SFP connection. It also holds its identifier which is the two-dimensional position index of the topology shown in Figure 3.21. It defaults to $(0, 0)$ and is automatically updated if it is synchronized from another master. If the master has the id (i, j), the slave will get either id $(i + 1, j)$ or $(i, j + 1)$, depending on the direction along the grid and therefore the interface used. Synchronization of the slave should ideally happen automatically whenever an SFP connection with another master interface is established.

As soon as the global time is synchronized across the whole network of QiControllers, and each platform has its unique identifier, data transfer would be possible. Utilizing

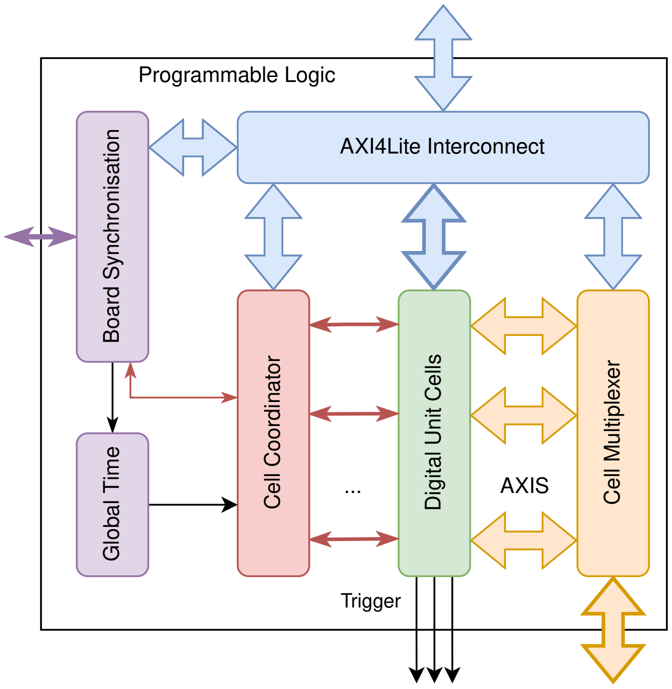


Figure 3.22: Adaptions to the PL of a single platform to support multi-platform synchronization.

the two-dimensional index, the routing of information is easy. When the cell coordinator is extended to facilitate the exchange of register contents between digital unit cells, the same interface could be further extended to integrate multi-board data transfer. Each digital unit cell of the complete setup can be uniquely identified by the two-dimensional index of the QiController in the grid and the index of the digital unit cell within the platform.

To conclude, a clear vision exists demonstrating that and how the QiController can be further scaled. However, a detailed concept will still need to be elaborated and consecutively implemented. It also depends on ongoing developments regarding the architecture of superconducting quantum processors. The provided vision and sketched approaches can be used as a starting point for this future work.

Table 3.1: Resource utilization on a Xilinx XCZU28DR RFSoC. Categories are configurable logic blocks (CLBs), block RAMs (BRAMs), and digital signal processing (DSP) slices.

Entity	CLB	BRAM	DSP
Available resources	53 160	1080	4272
Full design	74.69 %	97.2 %	32.3 %
Single cell	(5.01 ± 0.17) %	6.48 %	2.15 %
Sequencer	(1.57 ± 0.03) %	0.09 %	0.09 %
Signal generator	(0.84 ± 0.05) %	2.04 %	0.47 %
Signal recorder	(1.45 ± 0.08) %	1.94 %	1.12 %
Data storage	(0.30 ± 0.02) %	0.37 %	0 %
Digital trigger	(0.43 ± 0.02) %	0 %	0 %
WB infrastructure	(0.63 ± 0.04) %	0 %	0 %

3.7 Characterization Results

To compare the performance of the QiController with other devices, several characterization measurements have been performed. A Xilinx ZCU111 evaluation board is used as QiController without analog frontend.

3.7.1 Resource Utilization

The Xilinx ZU28DR RFSoC on the ZCU111 offers resources for up to 15 digital unit cells. The resource utilization of the design is given in Table 3.1. The amount of cells is limited by the amount of available BRAMs, mainly due to the resources required for the NCOs inside the signal recorders and generators. These account for 1.48 % BRAM usage per NCO. When more digital unit cells are needed, special Ultra RAM blocks available inside the RFSoC could be used in addition to normal BRAM blocks. These are special memory blocks within the Xilinx UltraScale+ architecture offering additional memory capacity within the FPGA while being more optimized but less flexible than normal BRAM. On the other hand, the ZU28DR offers only eight DAC channels. For most experiments where these are sufficient, 15 unit cells are more than enough as already discussed in Section 3.6.1. Yet, when utilizing another RFSoC with more channels, this optimization might become useful to allow for more unit cells on the system. However, also the available configurable logic blocks (CLBs) will pose a limitation with only roughly 5 additional unit cells that can be implemented then.

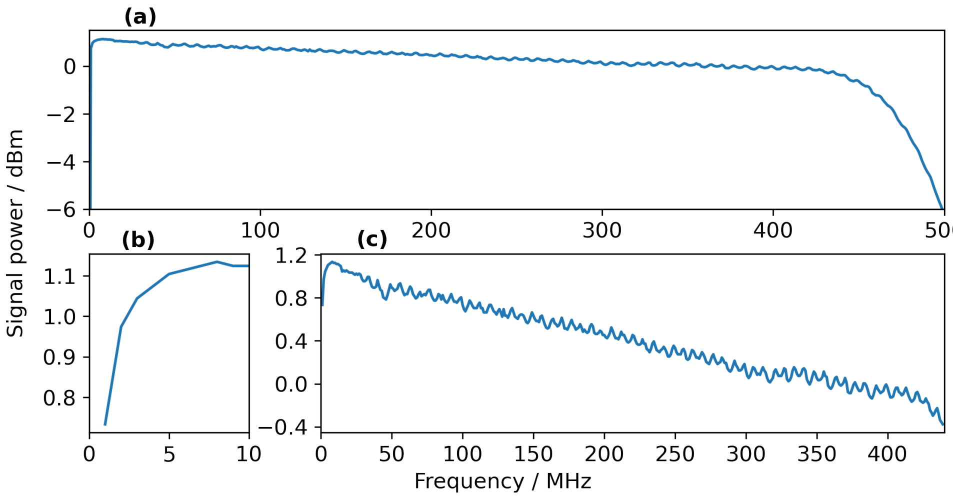


Figure 3.23: Output power over frequency of a baseband signal generated with the QiController. (a) Complete frequency range which can be generated, from 0 to 500 MHz. (b) Low-frequency region where the DC cut-off of the balun becomes visible. (c) Frequency region with nearly linear dependency of the power.

3.7.2 Analog Signal Properties

When characterizing the quality of generated microwave signals, important properties are output power, phase noise performance, and spurious free dynamic range. The measurements have been performed with a Rohde & Schwarz FSWP50 phase noise analyzer which also supports a spectrum analyzer mode, and a Tektronix MDO4104C oscilloscope. Depending on the measurement, one of the devices is directly connected to an SMA output port of the QiController's balun board via an SMA cable. Due to the internal data rate of 1 GSPS within the PL, the generated microwave signals are limited to frequencies below $f_s/2 = 500\,\mathrm{MHz}$.

Signal power levels are typically expressed in decibel-milliwatts, a logarithmic scale defined as $L_P(\mathrm{dBm}) = 10\log_{10}(P/1\,\mathrm{mW})$. When comparing such power levels, the level difference of a signal with respect to a reference, called carrier, is given as decibel-carrier: $\Delta L_P(\mathrm{dBc}) = L_P(\mathrm{dBm}) - L_{\mathrm{carrier}}(\mathrm{dBm})$.

Output Power

The maximum output power of the microwave signals depending on the output frequency is depicted in Figure 3.23. No analog frontend was used in this case.

For frequencies below 5 MHz, one can see a slight drop in signal power. This is due to the passive baluns used to convert the differential signals from the DACs to single-ended signals required for the experiments. As the baluns feature an inherent AC coupling, DC signals cannot pass through the balun. Up to 420 MHz, the signal power is very stable with a slight linear decrease of roughly 1 dBm over the whole range. For even higher frequencies, the signal power drops more significantly, as the signal approaches half the sampling frequency, $f_S/2 = 500\,\mathrm{MHz}$. All these behaviors are as expected. Additionally, one can see a small periodic oscillation in the signal power with peak-to-peak variations below 0.1 dBm and a period duration of roughly 7 MHz. The origin of this oscillation is unclear. It might be a parasitic resonance on the balun board. However, the oscillation is small compared to the general slope in dependence of the frequency and is not critical. The highest measured output power is 1.14 dBm at 8 MHz. At the oscilloscope, this corresponded to an output signal amplitude of $652\,\mathrm{mV_{PP}}$ which is a bit lower than expected from the spectrum measurement. The difference to 1.14 dBm, equivalent to $721\,\mathrm{mV_{PP}}$, could be caused by worse cable losses and an impedance mismatch in case of the oscilloscope.

Phase Noise Performance

The phase noise of output signals with different frequencies is shown in Figure 3.24. Offsets larger than the carrier frequency have been excluded from the plot. The corresponding root mean square (RMS) time jitters are similar for all carrier frequencies and around 210 fs. The phase noise increases with increasing carrier frequency which is as expected, as time jitter and phase noise are linked via the carrier frequency ($\delta\varphi = f \cdot \delta t$). All signals show a phase noise contribution at 426.58 kHz carrier offset, potentially indicating a contribution of noise from the onboard clocks or switching regulator noise. For selected carrier offsets, the phase noise values are also given in Appendix D.1. In general, the phase noise performance is good and comparable to other devices. By measuring with a variety of frequencies and across a wide span of offset values, it is furthermore verified that no critical outliers exist.

Frequency Stability

The ZCU111 has an internal, temperature-compensated 12.8 MHz reference oscillator (TCXO) from which the converter clocks and the PL design clock are derived. To assess the frequency stability of the system, a 100 MHz tone is continuously generated by the platform and repeatedly measured by the FSWP for more than 16 hours. The FSWP itself is referenced to an external, high-precision reference clock.

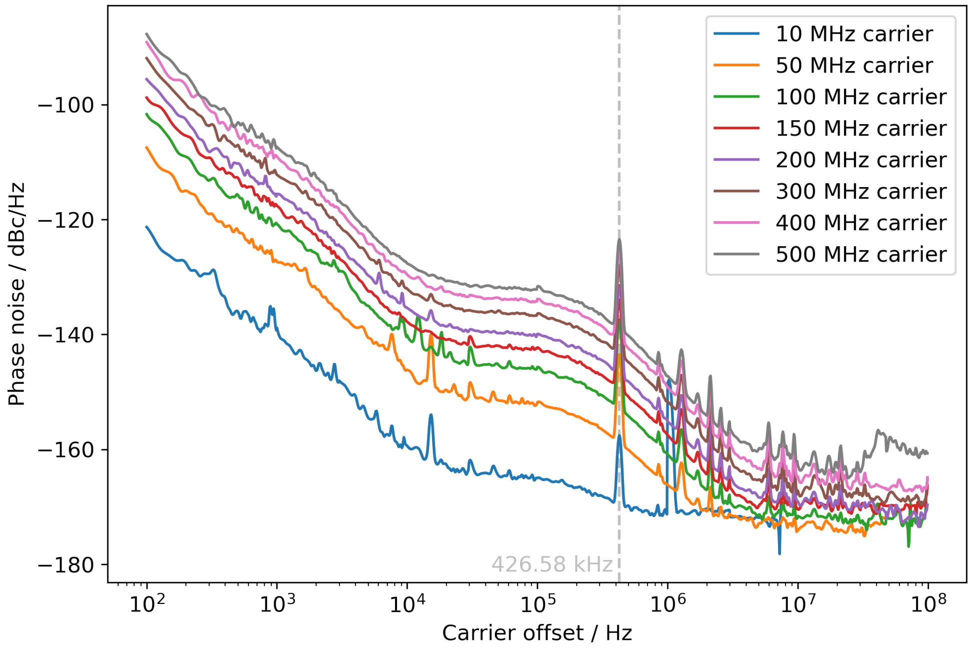


Figure 3.24: Phase noise of the output signal for different carrier frequencies.

The measurement result is depicted in Figure 3.25. The absolute frequency mismatch is on average 22 Hz. In an environment with stable temperature surrounding, the frequency jitters with a standard deviation of 64 mHz. However, actively changing the room temperature, in this case by opening a window, resulted in a frequency drop by up to 4 Hz and consecutive increased frequency jitter with 730 mHz standard deviation. If an external reference clock is supplied to the ZCU111 which replaces the internal reference oscillator, no frequency deviation can be detected anymore and the absolute frequency mismatch is less than 100 mHz. The RMS jitter for frequency offsets between 5 Hz and 10 kHz is reduced from (1480 ± 130) fs to (816 ± 12) fs. This is also visible in the phase noise for small carrier offsets, compare Figure 3.26.

Spurious Free Dynamic Range

The three most dominant spurious contributions to the output signal in dependence of the output frequency of the desired signal carrier are presented in Figure 3.27. The spurious free dynamic range (SFDR) is below −62 dBc for carrier frequencies of up to 413 MHz. At higher frequencies, the signal from the second Nyquist

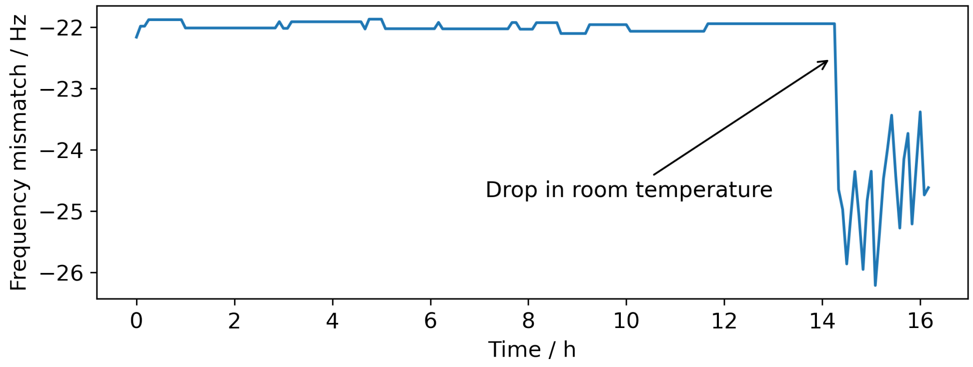


Figure 3.25: Frequency stability of the output signal without external reference.

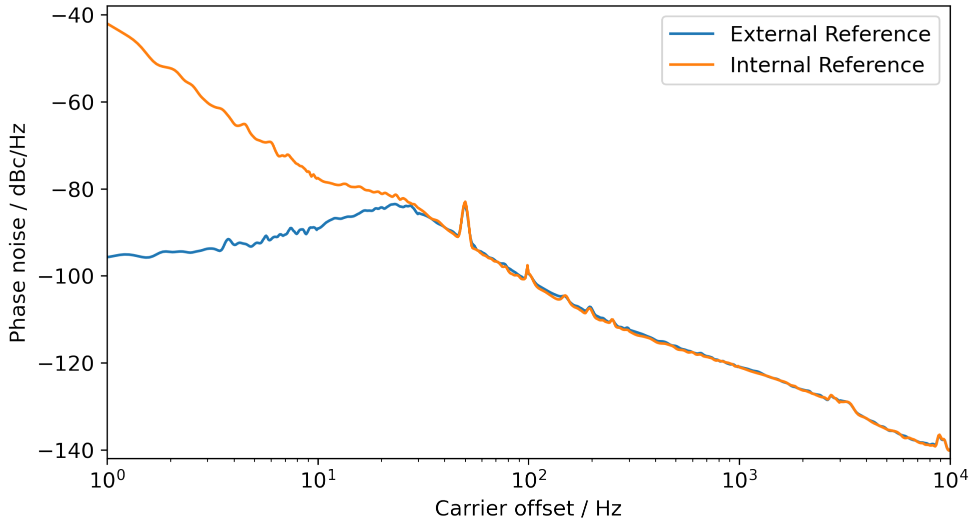


Figure 3.26: Phase noise of the output signal for low carrier offsets with and without an external reference.

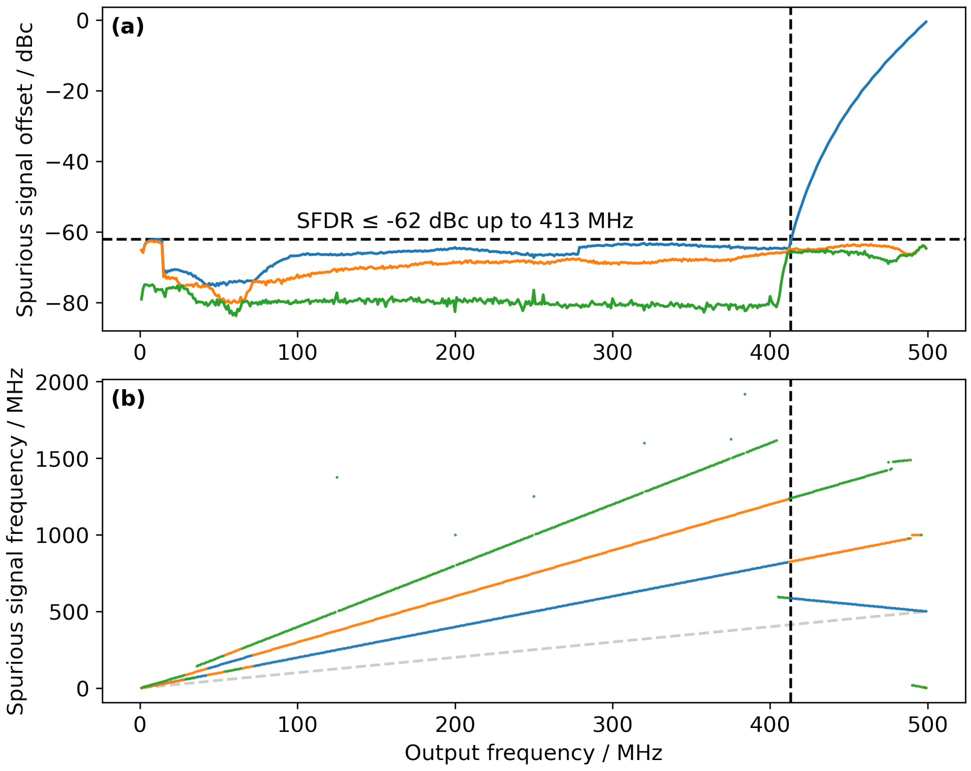


Figure 3.27: (a) Signal strength of the top three spurious contributions in dependency of the output frequency of the desired signal carrier. The highest contribution (sketched in blue) is equivalent to the spurious free dynamic range (SFDR). (b) Frequencies of these top three spurious contributions in dependency of the carrier output frequency. The dashed gray line indicates the frequency of the carrier itself. The colors match between both subplots.

zone becomes dominant as the frequency approaches half the sampling frequency, $f_s/2 = 500\,\text{MHz}$. The SFDR is limited by the harmonics of the signal, as can be clearly seen when looking at the frequencies of the spurious contributions. To get a more refined picture, some device manufacturers provide a separate value for the contributions by harmonics, the harmonics distortion, and an SFDR where these harmonics have been excluded. As the resolution of the SFDR without harmonics needs to be much better due to lower power levels, the spectrum measurements have been repeated with lower resolution bandwidth (100 Hz instead of 10 kHz). In this case, the noise floor of the spectrum was located below −105 dBc. Figure 3.28 shows the results of this second measurement, with excluded harmonics. In this case, the SFDR is even lower and below −84 dBc.

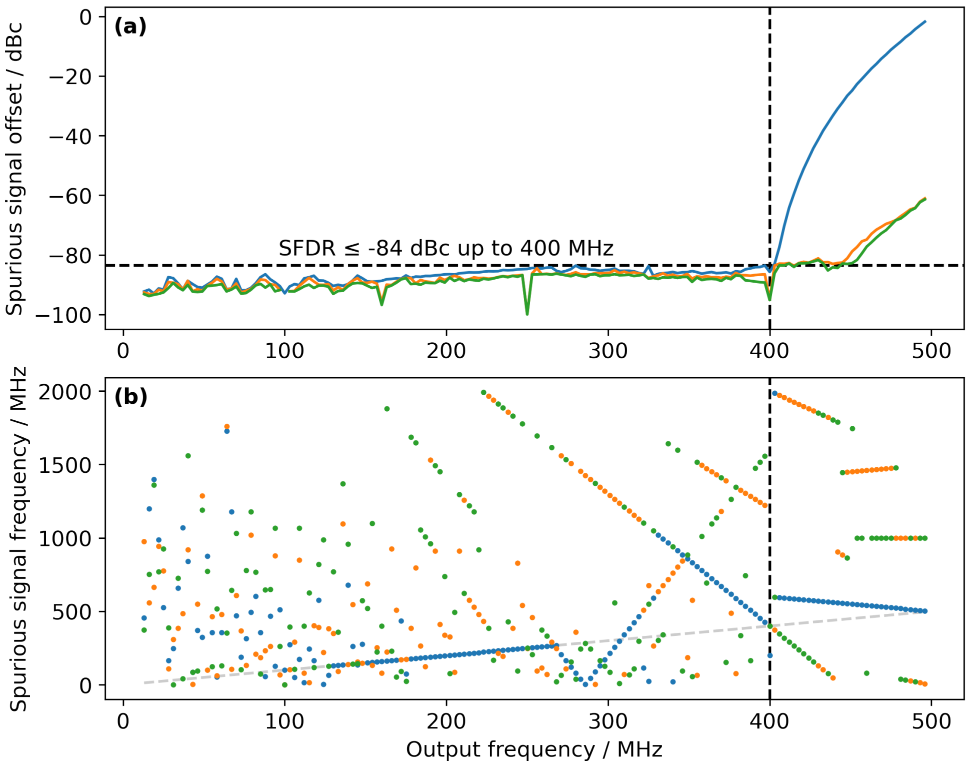


Figure 3.28: Similar measurement as in Figure 3.27 but without harmonic contributions and with decreased resolution bandwidth to improve the measurement noise floor. (a) Signal strength of the top three spurious contributions and (b) their frequencies.

Comparison to Commercial Devices

To get a better impression, the obtained properties are compared to datasheet values of three commercial arbitrary waveform generators (AWGs) which are used in the field. One device is the HDAWG by Zurich Instruments [145] which is also advertised for applications with superconducting qubits. Other similar devices are the Tektronix AWG5200 [146] and the Active Technologies AWG5062 [147]. Phase noise performance, as well as harmonics distortion and SFDR are summarized in Table 3.2. Not all properties are given in the data sheets for all devices. If multiple values are available, the one with the most similar settings to the QiController characterization is selected. In general, all properties of the QiController are competitive to the commercial devices. Only the harmonics distortion of the Active Technology AWG5062 is slightly better than the one of the QiController. In all other cases, the QiController even exceeds the values of the commercial devices.

Table 3.2: Comparison of analog signal properties of the QiController to commercial AWGs. Missing values are not given in the data sheet of the devices. PN stands for phase noise and is presented for a 100 MHz tone at the given offsets.

	IPE	Zurich Instrum.	Tektronix	Active Technol.
Property	QiController	HDAWG	AWG5200	AWG5062
PN @ 10 kHz	−140 dBc/Hz	−135 dBc/Hz	-	−123 dBc/Hz
PN @ 1 MHz	−161 dBc/Hz	−148 dBc/Hz	-	-
Harmonics	−62 dBc	−53 dBc	-	−70 dBc
SFDR	−84 dBc	−80 dBc	−70 dBc	−75 dBc

Analog RF Frontend

Additionally, the impact on signal performance when using the analog RF frontend is evaluated. A single analog unit cell is connected to the output of the ZCU111 board which generates a 100 MHz I/Q baseband signal. The frontend is configured to output the resulting up-converted RF signal at 6 GHz. In this case, the highest spurious contribution is given by local oscillator leakage of the I/Q mixer conversion stage at around −30 dBc. Without further I/Q mixer calibration, the sideband is roughly −40 dBc suppressed. These values are comparable to other setups using I/Q mixers. The RMS jitter of the 6 GHz output signal measured between 100 Hz and 1 GHz offset is 1.4 ps. The phase noise is −104 dBc/Hz at 10 kHz and −122 dBc/Hz at 1 MHz offset. When comparing these values to the values from the baseband signal, the different carrier frequencies need to be considered. The same jitter will result in a phase noise difference of 36 dBc/Hz between both frequencies. This matches well to the measured values and indicates that the analog RF frontend does not significantly impair the phase noise of the signal.

3.7.3 Digital Signal Properties

Apart from the microwave signal properties, latencies and data communication properties are of interest.

Feedback Control Latency

The QiController is capable of reacting to a previously measured qubit state online during the execution of the experiment. An important figure when performing such feedback operations is the minimal latency between the readout pulse and a pulse which is depending on the result of this readout. It can be measured by operating

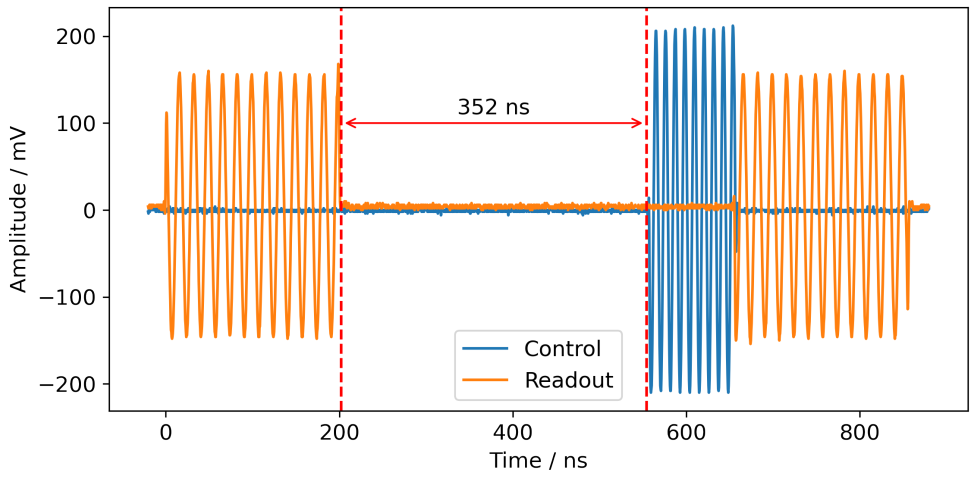


Figure 3.29: Oscilloscope recording of a measurement-based, closed-loop feedback operation with the QiController. The delay between the first readout pulse and the conditional control pulse is the platform-specific feedback latency.

the QiController in loop-back mode. An oscilloscope recording to visualize the latency is presented in Figure 3.29. There, a 100 ns control pulse is conditionally executed as fast as possible depending on the result obtained from a preceding measurement pulse. Based on this oscilloscope recording, the intrinsic latency is determined as 352 ns. Of this, 276 ns are attributed to the delay between signal generator and recorder, i.e. the delays through the cell multiplexer, the converters, as well as the electrical delay through the analog signal paths. This is represented by the trigger delay which has to be adjusted in the signal recorder as introduced in Section 3.4.5. The remaining 76 ns, i.e. 19 logic cycles on the FPGA, are due to processing of the result in the signal recorder to obtain a state, and the decision logic within the sequencer. A more detailed analysis of the individual contributions is given in Appendix B.3.

Taskrunner Timing Performance

When interacting with the platform, benchmarking the speed and overhead for communication operations is of interest. The modules on the FPGA are accessed using single AXI4-Lite register read and write operations. Such operations can be either performed encapsulated via the ServiceHub and the gRPC interface from the Python client, or directly by a user task running on the Taskrunner. In the latter case, the communication overhead via gRPC only occurs once before an

Table 3.3: Timing performance for typical operations running on the Taskrunner or the Python client. On the Taskrunner, the TCM is used for memcpy and array multiplication. The array multiplication uses two 32 bit arrays, each with 1024 elements. 100 000 repetitive measurements have been performed each to obtain the presented average durations and standard deviations. [47]

Operation	Taskrunner	Python client
AXI register read	(306 ± 2) ns	(590 ± 40) µs
AXI register write	(323 ± 2) ns	(620 ± 40) µs
Sequencer status poll	(324 ± 2) ns	(580 ± 60) µs
memcpy of 1024 AXI registers	$(312\,401 \pm 4)$ ns	(1500 ± 130) µs
Array multiplication	10 270 ns	(1.5 ± 0.8) µs

experiment is started. Afterwards, only the substantially smaller latencies of the register accesses arise. A comparison between execution within the Taskrunner and from the Python client for different operations is given in Table 3.3. The data of this section is mainly extracted from the corresponding paper on the Taskrunner [47]. Accesses to the AXI4-Lite register interfaces originating from the Taskrunner take on the order of 300 ns. The same operations performed from the APU yield similar values, with (299 ± 4) ns read and (210 ± 2) ns write latency when accessed by a bare metal application. When these operations are performed from within a Linux operating system, additional outliers on the order of 10 µs can be observed which are likely caused by other processes running at the same time [135]. In contrast, the same accesses from the Python client are on the order of 600 µs due to the required communication via Ethernet. Only for the array multiplication, where two 32 bit integer arrays with 1024 elements each are multiplied, the Python client is faster as it can leverage a stronger desktop processor and the Python numpy library. Constant execution time of the array multiplication was observed on the Taskrunner indicating deterministic timing behavior as expected when only RPU and TCM are involved. Only the first two executions took 14 ns longer which is caused by the branch prediction feature of the ARM Cortex-R5 processing unit in the loop used for the calculation.

The overhead incurring when loading tasks with different complexity onto the Taskrunner by the user is shown in Table 3.4. In all cases, the duration to process the source file is dominated by the online compile time due to the use of optimization flags. It does not pose an issue as compilation only occurs once during configuration, and not during the experiment itself. The duration to load a source task also shows a clear dependency on the complexity of the task. While the empty and basic tasks only consist of a single function, the complex task implements multiple functions to perform calculations like an FFT.

Table 3.4: Durations to transfer different tasks from the Python client onto the Taskrunner. Source code will be compiled on the fly on the platform before the generated binary is then transferred to the Taskrunner. The binary file is directly transferred from Python to the Taskrunner. Three tasks are distinguished based on their complexity (also reflected in the number of code lines). The durations to load the source code are each averaged over 100 iterations while the durations to transfer the binary file, due to lower values and higher relative fluctuations, are averaged over 10 000 repetitions. [47]

Task	Lines	Source code	Binary file
Empty	6	(184.8 ± 0.5) ms	(1.25 ± 0.09) ms
Basic	39	(260.3 ± 0.6) ms	(1.55 ± 0.10) ms
Complex	386	(1107.2 ± 1.8) ms	(2.99 ± 0.10) ms

Aggregation of larger amounts of data is also handled by the Taskrunner, as the capacity of the BRAMs on the FPGA is limited. Therefore, the speed to transfer data boxes from the Taskrunner to the Python client is another important benchmark. Requesting and transferring 10 kB of data takes (4.3 ± 1.6) ms. With increasing data box size, the relative overhead decreases and thus the transfer speed increases. Transferring a 100 MB data box requires (2.32 ± 0.16) s corresponding to 345 Mbit/s or 43 MB/s. This is likely limited by the necessary copy operation on the A53 to convert the data into the right gRPC format, including the required marshaling of the 4 B integers to make the data platform-independent. This is supported by the fact that, during the transfer, one of the cores of the ARM processor is fully utilized. Indeed, when transferring byte arrays instead where no marshaling is required, speeds of 940 and 934 Mbit/s for read and write accesses can be observed [135]. These speeds are limited by the 1 Gbit/s Ethernet connection [148].

3.8 Summary

The QiController is a versatile control electronics specifically targeting superconducting qubits. It can create microwave pulses with arbitrary shapes on the nanosecond scale, precise phase control, and frequencies up to over 9 GHz. Phase noise and spurious contributions of the analog output signals are competitive to high-end commercial AWGs. Readout pulses can be acquired and their amplitude and phase decoded to extract the measured qubit state. Nanosecond-accurate sequencing of pulses and measurements enables the reliable and reproducible execution of experiments and quantum algorithms, including fast feedback operations. For complex experiment schemes, extensive online processing is provided using the Taskrunner framework. Users can access the platform via Ethernet using a modular communication server, called ServiceHub. The QiController is based on the RFSoC architecture, combining high-speed converters with processing units and an FPGA. The FPGA firmware is structured into digital unit cells, each controlling a single qubit. Analog unit cells, realized as custom PCBs, up- and down-convert the microwave signals between the RFSoC and the quantum processor.

While the QiController provides unique capabilities and great flexibility, programming such a complex system is nontrivial. Therefore, providing a suitable interface to control the platform is essential which is covered in the next chapter.

4 Platform Interface and Control

The QiController provides a versatile and flexible quantum bit (qubit) control platform. To leverage its capabilities, convenient software access to it is of key importance. Great effort was put in designing and implementing a user-friendly and intuitive interface that allows the user to control the whole setup. The following sections detail the considerations taken into account, provide some insights into the interface and typical use cases, and introduce integrations into third-party frameworks.

4.1 Philosophy

The QiController is meant to be used as laboratory equipment specifically designed to meet the demands of the superconducting qubit research community. It consists of numerous different modules: Taskrunner, cell coordinator, multiple digital unit cells, each with sequencer, signal recorder and two signal generators, among others. All modules can be individually controlled and configured using their register interfaces to obtain the desired output and functionality.

To facilitate the usage of the QiController, different layers of abstraction to control and configure the platform exist. The ServiceHub plugins already provide a first layer by encapsulating the register interface accesses inside gRPC remote procedure calls. These abstract from the hardware addresses and also include interpretations and conversions of the register contents, e.g. for the frequency. A Python client is provided to facilitate access to the low-level configuration of the platform in a Python-style class by wrapping the remote procedure calls. The low-level access provides many degrees of freedom. However, it also requires expert knowledge of all the modules within the platform as well as their specific interactions among each other. Directly configuring all the modules of this complex system is therefore inconvenient for regular users, and also error prone.

To provide easy and intuitive access to the capabilities of the QiController, another layer of abstraction is added. It is realized by a Python-based, high-level experiment description language called QiCode. Its goal is to enable regular users to functionally and intuitively describe the cells' control flow and output. The concept as well as the technical details around QiCode are introduced in Section 4.4. The language leverages the advanced partitioning of functionality within the QiController which was already discussed in Section 3.2.

The underlying workflow can be generalized as follows: The user describes the experiment they want to perform using the QiCode description language. The description will then be compiled into the configuration for the QiController, including the RISC-V instructions for the different sequencers, and loaded into the modules on the programmable logic (PL). Depending on the requested result data format, an appropriate task is loaded onto the Taskrunner. Predefined tasks exist for the most common data formats, but a custom task can also be provided if special data aggregation or evaluation is required. After the experiment execution on the QiController finished, the resulting data is structured and stored in the reference of the QiCode description, where it can be accessed by the user.

4.2 Python Client qiclib

QiCode is provided together with the Python gRPC client and a collection of standard experiment descriptions and routines in a Python package called qiclib. qiclib is an abbreviation for QiController (qic) and library (lib). It is the interface to the QiController and enables the user to control the platform to any extend possible. This ranges from direct low-level access to the PL modules up to high-level access using QiCode and even integration in other quantum programming frameworks like Qiskit.

The qiclib package is structured in multiple submodules. One submodule provides the functionality of QiCode and includes the compiler. Another submodule contains predefined experiments that can be executed without any specific platform knowledge. Other submodules exist with utility functionality, the generated gRPC interface and extended measurement scripts. The core submodule of qiclib, however, contains the hardware representation of the QiController. It is used to configure and interact with the platform and its containing entities and will be introduced in more detail.

The main component to interact with the platform is the `QiController` class. A connection to the platform can be established by creating an instance of this class and passing the ip address of the QiController. The class instance then provides access to all the functionality of the platform. Its structure is depicted in Figure 4.1. The QiController class contains multiple submodules, one for the Taskrunner, another

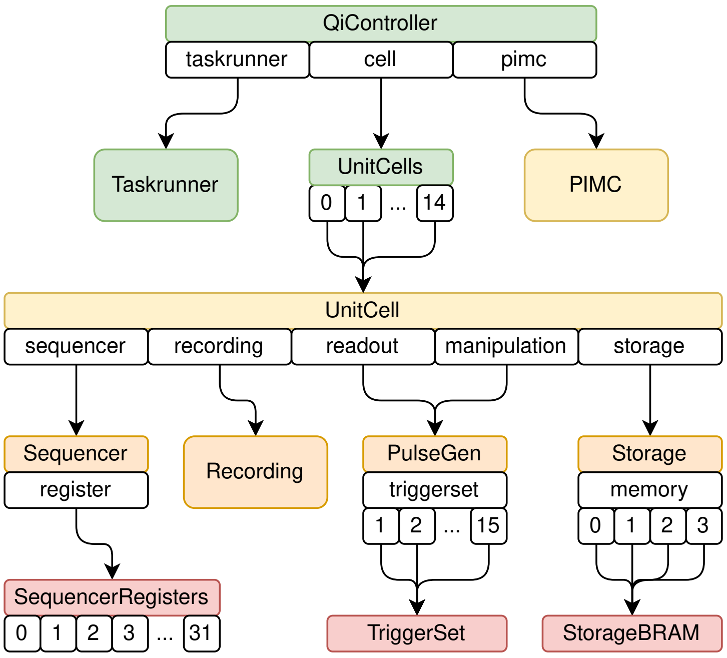


Figure 4.1: Structural diagram of the `QiController` class. Colored items are classes while white boxes are properties of this class. Only properties containing instances of other classes are shown. The color of the classes resembles the level of their abstraction. Green items are whole (sub)systems. Yellow classes correspond to actual PL modules. Orange modules are the contents of the digital unit cell, and red classes are used to access parts of these modules.

one for general status information by the platform information and management core (PIMC), and a third one to represent the digital unit cells of the platform. Each unit cell contains submodules for sequencer and signal recorder (called `recording`), as well as for the data storage and both signal generators. The latter are called `readout` and `manipulation` to distinguish between their intended usage. Each submodule within the digital unit cell implements the gRPC calls to interact with the corresponding plugin within the ServiceHub on the QiController. They encapsulate the remote procedure calls in a Python-style class to ease access and make it easy to familiarize oneself with the platform.

As an example, setting the pulse duration of the manipulation pulse stored in trigger set 12 of the fifth digital unit cell is accomplished by:

Code 4.1: Accessing the configuration of the QiController using nested properties.

```python
qic = QiController("ip-address") # Only needed once when initializing the board
qic.cell[4].manipulation.triggerset[12].duration = 80e-9 # seconds (80 ns)
```

The assignment will then be handled within the `TriggerSet` class where it is translated to a gRPC request of the `PulseGen` class that communicates with the ServiceHub plugin of the signal generators on the QiController.

In general, most accesses to the platform are realized using the getter/setter pattern where one does not call any method explicitly but just assigns the values to the appropriate property which will be transparently propagated to the QiController. Likewise, reading out a value happens by simply accessing the property which will cause a gRPC request in the background whose result is then returned as value of the property. Only more complex operations are implemented as methods to indicate to the user that these operations might cause some considerable overhead. The same applies for operations that not just configure the modules but trigger an action, like starting the sequencer or resetting a module.

4.3 Typical Use Cases

The following section provides examples on how to use the platform with the high-level experiment description language QiCode. A typical measurement procedure with the QiController is described to demonstrate its capabilities. For each example, also the underlying mechanics are explained to understand how the platform is operating.

4.3.1 Setup and Initialization

Connection and Sample Object

At the beginning of each experiment, one has to include the qiclib package as well as the QiCode language. The QiController can then be instantiated just by specifying its ip address:

Code 4.2: Instantiating the QiController with qiclib.

```python
import qiclib as ql
from qiclib.code import *

qic = ql.QiController('ip-address')
```

In the background, a gRPC connection to the ServiceHub will be established and some key parameters of the platform are read out to determine if the Python drivers are compatible with this version of the QiController.

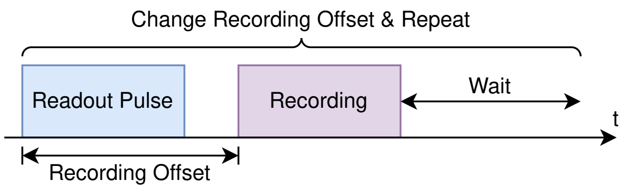


Figure 4.2: Experiment to calibrate the recording offset, i.e. electrical delay.

In QiCode, qubit properties are stored inside a QiCells object. Each element in this object represents a single qubit which will later be mapped onto a digital unit cell of the QiController. The QiCells sample is typically the first object one creates after instantiating the platform driver. At the beginning, it will be populated with the basic parameters to perform a readout, like pulse length, recording duration, and the intermediate frequency of the readout signal:

Code 4.3: Creation of sample object to describe qubit properties.

```
sample = QiCells(1) # 1 cell for 1 qubit
sample[0]["rec_pulse"] = 416e-9 # s readout pulse duration
sample[0]["rec_length"] = 400e-9 # s recording window duration
sample[0]["rec_frequency"] = 60e6 # Hz readout pulse frequency
sample[0]["manip_frequeny"] = 85e6 # Hz control pulse frequency
```

Later, in the course of the qubit characterization experiments, further values can be added to this sample object.

Electrical Delay Calibration

The first step is to determine the electrical delay of readout pulses through setup and cryostat. The signal recorder offers a trigger offset to compensate for this delay. Then, the recording will capture the whole readout pulse and start when it arrives at the module. The offset can be determined by varying it over a large range to maximize the amplitude of the recorded signal. Such an experiment is schematically depicted in Figure 4.2, corresponding to the following QiCode snippet:

Code 4.4: Experiment to calibrate the recording offset, i.e. electrical delay.

```
with QiJob() as calib_offset:
    q = QiCells(1) # This QiCode snippet controls one qubit
    offset = QiVariable() # Define a variable for the offset which will be changed
    with ForRange(offset, 0, 1024e-9, 4e-9):
        PlayReadout(q[0], QiPulse(q[0]["rec_pulse"], frequency=q[0]["rec_frequency"]))
        Recording(q[0], q[0]["rec_length"], offset, save_to="result")
        Wait(q[0], 2e-6) # Give the resonator excitation some time to decay
```

The defined QiJob acts on one qubit and uses the same property names that have been defined above for `sample`. It is important to note that these are still different objects. The key is that jobs can be an abstract representation of an experiment using such named properties as placeholders. The same job can be executed for different sample objects. The named properties will only be replaced once the job is executed and the user has specified which sample to use. In this case, performing the experiment on the QiController and obtaining the best offset entails:

Code 4.5: Execution of recording offset calibration experiment and sample update.

```
calib_offset.run(qic, sample, averages=1000) # Compile + run the QiJob 1000x with sample
data = calib_offset.cells[0].data("result") # Extract results as [I list, Q list]
sample[0]["rec_offset"] = 4e-9 * numpy.argmax(numpy.abs(data[0] + 1j * data[1]))
```

When the `run()` method is called on the `calib_offset` job, the QiCode compiler will first extract the necessary configuration for the PL out of the description and generate the RISC-V instructions for the sequencer. During this process, also the properties of the `sample` object are mapped onto the QiCells object defined within the QiJob. Then, the whole configuration is loaded onto the platform. The run method also accepts a `data_collection` parameter which defines how the results will be collected and processed by the Taskrunner. By default, for each recording performed within the experiment description, the results are collected as I and Q values. These results will then be averaged point-wise in the Taskrunner.

For each iteration, the Taskrunner starts the sequencer which performs the parameter variations. In this example, the trigger offset is written into the register interface of the signal recorder via the Wishbone (WB) bus at the beginning of each for loop repetition. At the end of the sequencer execution, the data storage holds all measurement results that have been generated during this single execution, so one I and Q value for each loop repetition. This result data is collected and averaged by the Taskrunner. Then, it is transmitted as data boxes to the Python client where it is stored inside `calib_offset.cells[0].data("result")`. This corresponds to the `save_to="result"` parameter specified in the `Recording` command executed on cell 0 of the QiJob.

In the above code, the offset with the highest amplitude is determined in Python and directly stored inside the sample object. From there, it can be reused for the following experiments and characterizations. The same experiment, including the update of the sample object, can be performed by calling `ql.init.calibrate_readout(qic, sample, averages=1000)`. It also contains further calibration routines, like resetting the phase, and visualizes the results as plots for the user.

Measurement Gate

In QiCode, gates are called QiGates and can be utilized as reusable building blocks. Similar to a QiJob, QiGates can act as templates and use the placeholder properties of the cells on which they should act. With the determined offset, one can define a reusable measurement gate:

Code 4.6: Generic measurement gate to encapsulate readout pulse and recording.

```
@QiGate
def Measurement(cell: QiCell, save_to=None):
    """Output readout pulse and trigger recording module to process result."""
    PlayReadout(cell, QiPulse(cell["rec_pulse"], frequency=cell["rec_frequency"]))
    Recording(cell, cell["rec_length"], cell["rec_offset"], save_to=save_to)
```

Depending on the cell that is passed to the gate, the performed operation can be quite different, with different pulse length, trigger offset, or frequency.

To verify that the electrical delay is configured properly, the raw data that is captured during the recording window can be visualized. QiCode offers a predefined `data_collection` mode for this, namely `"raw"`. The QiCode can stay the same as for any other experiment. In this case, a simple readout is sufficient:

Code 4.7: Simple readout experiment to obtain raw data of recording window.

```
with QiJob() as readout:
    q = QiCells(1)
    Measurement(q[0], save_to="result")
    Wait(q[0], 2e-6)

readout.run(qic, sample, averages=1000, data_collection="raw")
raw_i, raw_q = readout.cells[0].data("result")
```

The obtained result is the raw time trace of the I and Q input at the signal recorder which was collected and averaged 1000 times by the Taskrunner before being sent to the Python client.

4.3.2 Pseudo-VNA Measurement

After the readout has been configured, the qubit can be characterized. As a first step, the readout resonator needs to be located. Typically, the frequency is already known from previous spectroscopic measurements. Thus, local oscillator frequency for analog frequency conversion as well as the intermediate frequency stored in the `sample` can already be chosen appropriately. However, it should still be verified with the platform itself that the right readout frequency is used. For this, a pseudo vector network analyzer (VNA) scan can be implemented on the QiController:

Code 4.8: Pseudo-VNA experiment to search the readout resonator frequency.

```
with QiJob() as pseudo_vna:
    q = QiCells(1)
    fr = QiVariable()
    with ForRange(fr, q[0]["rec_frequency"] - 10e6, q[0]["rec_frequency"] + 10e6, 0.1e6):
        PlayReadout(q[0], QiPulse(q[0]["rec_pulse"], frequency=fr))
        Recording(q[0], q[0]["rec_length"], q[0]["rec_offset"], save_to="result")
        Wait(q[0], 2e-6)
```

Running the QiJob is performed the same way as for the electrical delay calibration. The execution on the QiController is also similar. Instead of changing the trigger offset, in this QiJob, the numerically controlled oscillator (NCO) frequencies of both signal generator and recorder are adapted. This happens using a broadcast write to both modules via the WB bus to keep them synchronized. From the obtained I and Q values, the amplitude and phase response can be calculated, equivalent to an S_{21} measurement of a VNA. Then, the resonator frequency can be precisely extracted and the intermediate frequency of the readout signals can be updated accordingly.

4.3.3 Single-Qubit Characterizations

Two-tone Measurement

As next step, the qubit control frequency f_{01} needs to be determined. This can be achieved by performing a two-tone measurement. Two signals are applied to the setup, one at the frequency of the readout resonator and one with changing frequency. The qubit will be excited when the changing frequency matches its transition frequency f_{01}. At this frequency, the excited qubit causes the readout resonator to shift which can be measured with the first signal. The QiJob could look similar to the `pseudo_vna` scan, but to demonstrate the flexibility of QiCode, a different approach is chosen here. It is also depicted in Figure 4.3.

Code 4.9: Two-tone experiment with continuous tones to find the qubit control frequency f_{01}.

```
with QiJob() as two_tone:
    q = QiCells(1)
    fcontrol = QiVariable(name="fcontrol")
    PlayReadout(q[0], QiPulse("cw", frequency=q[0]["rec_frequency"]))
    Play(q[0], QiPulse("cw", frequency=fcontrol))
    Wait(q[0], 20e-6) # Wait for qubit to decohere in a driven 50/50 superposition
    i = QiVariable()
    with ForRange(i, 0, 10000):
        Recording(q[0], q[0]["rec_length"], q[0]["rec_offset"])
        Wait(q[0], 100e-9)
    Play(q[0], QiPulse("off")) # End continuous control tone
    PlayReadout(q[0], QiPulse("off")) # End continuous readout tone
```

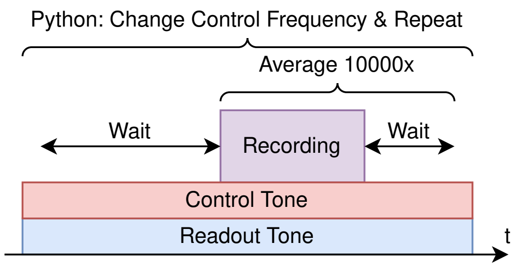


Figure 4.3: Two-tone experiment with continuous tones to find the qubit control frequency f_{01}.

In this case, two continuous tones are applied, one at the readout output using `PlayReadout`, and another one at a separate control output using `Play`. The control frequency is given as a named variable called `"fcontrol"`. Its value can be defined by the user and adapted between multiple repetitions by writing a different value into the corresponding sequencer register. After both continuous tones are turned on, one waits some time so the qubit decoheres while being continuously excited. If the control frequency matches the transition frequency, the qubit will end up in an equal superposition of all possible states. Afterwards, repetitive recordings are performed by the sequencer before the tones are turned off again at the end. As no `save_to` parameter is supplied for the recording, the single values will not be persisted. However, the signal recorder contains a simple averaging functionality which will add all 10 000 single measurement results that are obtained during one sequencer run. This averaged value will then be returned. Executing this job now also entails controlling the frequency from the Python client:

Code 4.10: Execution of the two-tone experiment with parameter variations in Python.

```python
exp = two_tone.create_experiment(qic, sample) # Compile QiJob
exp.configure() # Load configuration onto QiController
results = []
# Investigate frequencies close to the estimated frequency from the sample
frequencies = numpy.arange(-5e6, 5e6, 0.1e6) + sample[0]["manip_frequency"]
for fcontrol in frequencies:
    exp.init_variable("fcontrol", fcontrol) # Set register in sequencer
    data = exp.record() # Start the execution and obtain the averaged I/Q result
    results.append(data)
```

In this case, the QiJob is not directly executed, but the compile step and the execution happen separately. An experiment object will be created that contains the compiled configuration values from the QiJob. These are then loaded onto the QiController using the `configure()` method. Part of the compiled information is a mapping between variable names and register indices in the different sequencers. Therefore, it is possible to initialize variables with a custom value, which will then

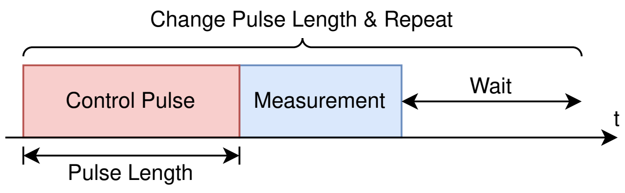


Figure 4.4: Rabi experiment to determine the control pulse length for a π rotation.

be written to the according registers. In this case, the value of the control frequency is adjusted between consecutive QiJob executions. By calling the `record()` method, the execution is started and the resulting average value is collected. After the experiment, one obtains the resonator response depending on the control frequency. The highest resonator shift will occur when the control frequency matches the qubit transition frequency f_{01} which can thereby be extracted from this measurement. The same experiment execution can also be performed using a custom task on the Taskrunner. This would reduce communication overhead and speed up the experiment. Alternatively, a pulsed experiment can be performed where the frequency change is handled directly within the sequencer, similar to Code 4.8.

Rabi Experiment

Now that the qubit control frequency is known, the time to perform a π rotation around the Bloch sphere can be determined. The Python driver provides a collection of common experiment descriptions available via `ql.jobs`, including one for a Rabi experiment. This experiment performs a control pulse with variable length and then determines the resulting qubit state, see Figure 4.4. The Python code providing the QiJob looks like the following (available as `ql.jobs.Rabi`):

Code 4.11: Method providing a Rabi experiment QiJob object for the user.

```
def Rabi(start: float, stop: float, step: float):
    with QiJob() as job:
        q = QiCells(1)
        length = QiVariable()
        with ForRange(length, start, stop, step):
            Play(q[0], QiPulse(length, frequency=q[0]["manip_frequency"]))
            Measurement(q[0], save_to="result") # Gate defined earlier
            Wait(q[0], 5 * q[0]["T1"])
    return job
```

By passing the values of the for loop to the `Rabi` method, a corresponding QiJob is created and returned. This job also contains a `Wait` statement which depends on the T_1 time of the qubit and shall ensure that the qubit state relaxes back to thermal equilibrium, i.e. close to $|0\rangle$, before continuing with the next repetition. If the value is not clear yet, e.g. for a new qubit that has not been used yet, a conservative value should be chosen. In this experiment, the length of the manipulation pulse depends on a variable. The sequencer will then turn on a continuous tone when the pulse should start and turn it off again after the defined length of the pulse. In this QiJob, the `Measurement` gate defined earlier is reused. The result values can be fitted to a damped sine in order to extract the ideal pulse length to perform a π rotation.

Qubit Control Gates

With the π-pulse time being defined and stored as `"pi"` property inside the sample, gates for π- and $\pi/2$-pulses can be created:

Code 4.12: Gates for qubit control pulses performing a π and $\pi/2$ rotation.

```
@QiGate
def PiPulse(cell: QiCell, phase: float = 0.0, detuning: float = 0.0):
    """Output manipulation pulse to rotate qubit state around Bloch sphere by pi."""
    Play(cell, QiPulse(
        length=cell["pi"],
        phase=phase,
        frequency=cell["manip_frequency"] + detuning
    ))

@QiGate
def PiHalfPulse(cell: QiCell, phase: float = 0.0, detuning: float = 0.0):
    """Output manipulation pulse to rotate qubit state around Bloch sphere by pi/2."""
    Play(cell, QiPulse(
        length=cell["pi"] / 2,
        phase=phase,
        frequency=cell["manip_frequency"] + detuning
    ))
```

The $\pi/2$-pulse could alternatively also be realized by cutting the amplitude in half instead of the length, by using the `amplitude=0.5` argument within the pulse. By changing the phase of the pulses, the rotation axis can furthermore be defined. A phase of $\pi/2$ for example would correspond to a rotation around the y instead of the x axis. The detuning argument is useful when applying off-resonance pulses, like for the Ramsey experiment.

With these gates and properties of the qubit being defined, further characterizations can be performed. This includes a T_1 measurement to determine the energy relaxation time, a Ramsey experiment that can both be used to fine-tune the qubit transition frequency and to determine the decoherence time T_2. With a spin echo experiment, slow fluctuating noise - when compared to the pulse sequence length - is filtered out and an improved T_2^* time is obtained. As these experiments are built and executed similar to the previous ones using the available QiGates, they are not explicitly shown here.

Once T_1 is known, one can passively initialize the qubit state by letting it thermalize with the environment. As this process is an exponential decay, after time T_1, the population of the excited state $|1\rangle$ will be reduced to a percentage of $1/e$. When waiting five times T_1, the probability to still be in the $|1\rangle$ state is only 0.67 %. Of course, this neglects thermal noise which can also excite the qubit from the ground state again. However, waiting five to ten times the T_1 time is a sufficient way to initialize the qubit state by bringing it into thermal equilibrium. Therefore, one can furthermore introduce a thermalization gate:

Code 4.13: Gate to let the qubit state thermalize as passive initialization.

```
@QiGate
def Thermalize(cell):
    """Wait 5 times the T1 time of the qubit so it can return to the ground state."""
    Wait(cell, 5 * cell["T1"])
```

4.3.4 Single-Shot Resonator Response Plots

Single-shot readout is the capability to extract the qubit state from a single measurement. It is an important requirement for quantum computing. The signal recorder contains a state estimation that needs to be calibrated so it can attribute different I and Q result outcomes to the respective qubit state. As the two qubit states lead to different amplitude and phase responses, this is visible in the in-phase and quadrature (I/Q) plane of the result values. With good enough signal-to-noise ratio for single-shot readout, two separable regions can be distinguished in which all result values are located. One represents the $|0\rangle$ state, and the other one the $|1\rangle$ state. The regions can be visualized by performing many single-shot readouts and plotting a histogram of the results in the I/Q plane. QiCode offers a special `data_collection` mode called `"iqcloud"` for this purpose:

Code 4.14: Single-shot resonator response plot for both qubit states.

```
with QiJob() as states:
    q = QiCells(1)
    # State |0>
    Measurement(q[0], save_to="result")
    Thermalize(q[0])
    # State |1>
    PiPulse(q[0])
    Measurement(q[0], save_to="result")
    Thermalize(q[0])
states.run(qic, sample, averages=100000, data_collection="iqcloud")
state0, state1 = states.cells[0].data("result") # One iqcloud per recording
# Both state0 and state1 are a tuple: (I list, Q list)
```

In this case, the average count tells the underlying task in the Taskrunner how many data points to collect for each measurement within the QiJob. The job performs two measurements, one for the ground state and one for the excited state. Separate histograms can be created for both measurements, as the data is separately collected for each of the measurements within the job. Again, the different data collection mode is handled by a predefined task in the Taskrunner that collects all single measurement results, groups them by measurement within the QiJob and sends them back without averaging to the Python client. There, the data can be further processed, e.g. to create a histogram as mentioned.

4.3.5 Multi-Qubit Characterizations

If a sample contains multiple qubits, these can be described as individual cells in a QiCells object. For each cell, one individually defines all the properties needed to perform experiments with this particular qubit, from frequencies, over pulse lengths, to recording durations. For example, a sample object for five qubits could be initialized like this:

Code 4.15: Creation of a sample object for a multi-qubit chip.

```
sample = QiCells(5, cell_map=[0,2,4,6,8])
sample[0]["rec_frequency"] = 30e6
sample[1]["rec_frequency"] = 80e6
sample[2]["rec_frequency"] = 130e6
# [...]
```

If the qubits are read out using frequency-division multiplexing (FDM), different intermediate frequencies will be required to end up at the individual resonator frequencies with a single local oscillator and I/Q mixer. Another relevant information is the mapping of physical qubits as described by the sample object, and the digital unit cells within the QiController. For this, a cell map can be passed that states for each cell within the QiCells on which digital unit cell of the QiController the execution should happen. In the given example, the qubit represented by `sample[0]` is connected to digital unit cell 0, and `sample[4]` to digital unit cell 8. As this mapping is specific to the setup wiring and the sample, but not to the actual experiment, it is defined together with the sample properties.

To perform single-qubit characterizations for such a multi-qubit sample, the mapping between the QiJob cell and the QiCells sample object can furthermore be defined. Details regarding the mapping of cells is provided when formally introducing the QiCode concept in Section 4.4 and is visualized in Figure 4.8. As example, the electrical delay calibration from above, when performed for the fourth qubit, will look like this:

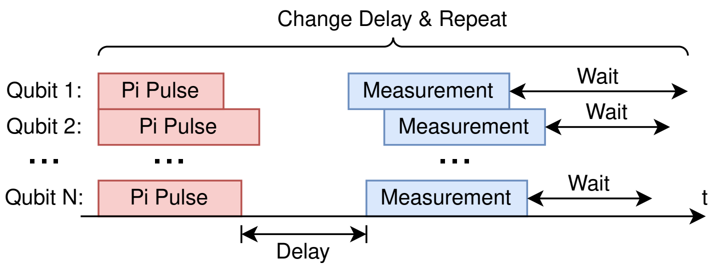


Figure 4.5: T_1 experiment which is performed on N qubits simultaneously.

Code 4.16: Execution of the recording offset calibration for the fourth qubit.

```
calibrate_offset.run(qic, sample, averages=1000, cell_map=[3]) # 4th cell of sample
data = calibrate_offset.cells[0].data("result") # 1st cell of QiJob
sample[3]["rec_offset"] = 4e-9 * numpy.argmax(numpy.abs(data[0] + 1j * data[1]))
```

The job can stay unchanged as it acts as a template that can be executed on any qubit thanks to the placeholder cell properties used to define it. By default, always the first cells within the QiCells sample object are used. A QiJob requiring five cells will therefore use the first five cells of the sample object, if no cell map is given. Hence, without the cell map, the above calibration would always be executed for the first qubit defined within the sample.

As an example of a multi-qubit characterization, a T_1 measurement happening simultaneously on all qubits can be described with only nine lines of code using some already-defined QiGates:

Code 4.17: T_1 experiment which is performed on all qubits simultaneously.

```
with QiJob() as multi_t1:
    q = QiCells(len(sample)) # Control as many cells as the sample has defined, i.e. 5
    length = QiTimeVariable()
    with ForRange(length, 0, 4e-6, 100e-9):
        for cell in q:
            PiPulse(cell) # Excite qubit into state |1>
            Wait(cell, length) # Wait variable time for qubit energy to decay
            Measurement(cell, save_to="result") # Measure the obtained state
            Thermalize(cell) # Reset qubit into its (thermal) ground state
```

As seen in the earlier examples, subsequent commands on the same cell will happen after each other. With multiple cells involved, the timing is not always intuitively clear. By default, commands on different cells will happen independently and in parallel, even if they are defined after each other in the QiJob. This means that each cell has its own timeline in QiCode, as can also be seen in Figure 4.5. However, explicit and implicit synchronization mechanisms exist. The latter will happen, e.g., at the beginning of a gate acting on multiple qubits where the timelines of the affected cells are automatically aligned. Another example is the loop body of the `ForRange`. Here, at the beginning of each iteration, the timelines of all cells present in the body are implicitly aligned. The same mechanism holds true for other control structures in QiCode as well. Within the body, the commands on different cells will then again be executed independently and in parallel. Therefore, a normal Python for loop can be used to generate the pulse sequences within the body for the different qubits. Python is an interpreted language and therefore, the for loop will be executed directly while creating the QiJob. This leads to the sequences for all qubits to be subsequently added within the `ForRange` loop body. As the defined gates use the cell properties to determine all the parameters, we can use the same gates for the different qubits, and the correct parameters will be later inserted just before the compilation. This is a great advantage of QiCode and makes it very flexible and reusable, like the example of the multi-qubit T_1 experiment illustrates.

Similar to the single-qubit experiments, the experiment description will first be compiled into the configuration for the QiController when the `run()` method of the QiJob is called. In this case, the compiler separates the different pulses and sequencer instructions for the individual digital unit cells. They are then loaded onto the QiController and all utilized cells are synchronously started using the cell coordinator. After the execution of all cells finished, the data from each cell is separately collected by the Taskrunner which, again, also performs averaging and restarts the utilized cells. Finally, the data is transmitted to the Python client where it can be accessed by the user in the same way as for the single-qubit experiments:

Code 4.18: Execution of the multi-qubit T_1 experiment.

```
multi_t1.run(qic, sample, averages=1000)
# Results are stored inside the job:
multi_t1.cells[0].data("result") # Data of first qubit
multi_t1.cells[1].data("result") # Data of second qubit
# [...]
```

For the user, the whole data collection process is transparent and will be automatically handled by an appropriate task within the Taskrunner.

4.4 Experiment Description Language QiCode

QiCode is a high-level experiment description language to control the QiController. While the last section already presented multiple use cases to illustrate its capabilities, this section will provide a more technical and complete overview.

4.4.1 Concept

Goals and Requirements

QiCode was designed from the beginning with the needs and requirements of the superconducting qubit research community in mind. Based on discussions with and input from experimental physicists, the following goals and requirements have been identified for the new experiment description language:

> **High-Level**
> no platform-specific, low-level knowledge required

> **Versatile and Flexible**
> support a wide variety of experiments

> **Easy to Use**
> facilitate working with the QiController

> **Intuitive**
> new users can easily learn how to use the QiController

> **Multi-Qubit Support**
> promote experiments with arbitrary number of qubits

> **Reusable Abstractions**
> provide means to create reusable building blocks and abstract from qubit chip

> **Reproducible Execution**
> complete, self-contained description with all configuration

> **Flexible Data Handling**
> customizable modes to collect measurement data

> **Custom Control Flows**
> advanced experiments with if/else clauses, variables, and loops

> **Integration into Existing Workflows**
> description, execution and data retrieval directly in Python (widely used)

> **Informative, User-Friendly Error Messages**
> instructive and understandable information about limitations, e.g. the sample rate

The QiCode language is specifically targeted and designed for quantum computing experiments and algorithms. It provides means to define reusable gates and abstract experiment descriptions that can act as template and be reused for different qubits. An overview of its structure and concept is shown in Figure 4.6.

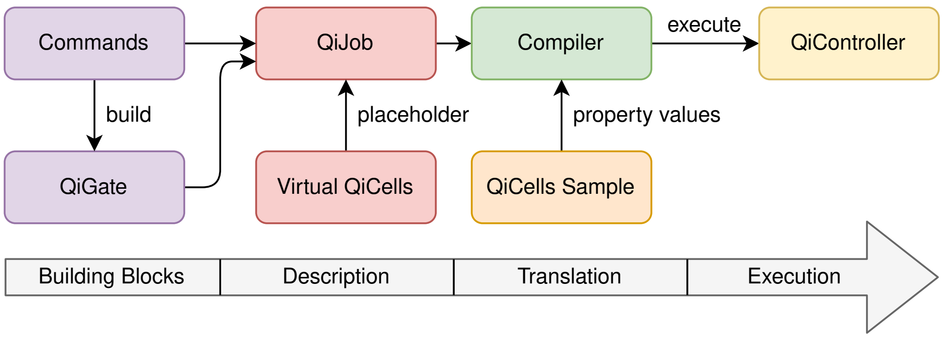


Figure 4.6: Abstract concept of the QiCode experiment description language.

Multi-Qubit Sample Representation: QiCells

An important concept in this scheme are `QiCells` objects. These hold the physics parameters to interact with a superconducting qubit chip. An instance of this class is therefore called sample. The sample can consist of one or more cells, i.e. qubits, and each cell defines all relevant properties for one qubit. This can be pulse lengths, frequencies, but also other experiment-related parameters. The naming convention of these properties is left to the user who can define any properties necessary. The sample can furthermore also store qubit-related characteristics, like decay times and further information which is not needed for the experiments. As the sample can be exported as JSON file and imported again later, this can be useful.

Abstract Experiment Description: QiJob

An experiment description is called `QiJob`. Each QiJob defines a virtual QiCells object which will be used within this job. This way, the user defines on how many cells/qubits a QiJob will act. Within the job, one can access properties of this virtual QiCells object. As it is a different object than the sample, no values will be set within the virtual QiCells object when the job is created. Instead, a reference for this property is stored as placeholder which will later be matched to the sample when one executes a job. The QiJob therefore acts as a template experiment which can later on be executed on different qubits with different properties by simply changing the sample. Adaptions to the QiJob are not necessary as the properties are defined in an abstract way. As long as the actual values are set in the sample which is passed to the QiJob, it can be executed on the QiController. Otherwise the QiCode compiler aborts and names the missing properties so the user can add them to the sample.

Modular Building Blocks: QiGates

While already the QiJobs are designed to be reusable, also smaller patterns might occur again and again in one or across multiple experiments. For such cases, QiGates have been designed as modular building blocks. These encapsulate one or multiple QiCode commands which then can be reused multiple times and also for different cells and experiments. Prominent examples are a π-pulse or a measurement operation, consisting out of a readout pulse and a consecutive recording operation. QiGates are regular Python methods which are prepended by a `@QiGate` decorator. While this is not strictly enforced, one typically passes as first argument the cell on which the gate should act before supplying additional parameters, e.g. `PiPulse(cell, phase=0.0)`. As all QiCode commands are structured the same way, this improves the readability of the experiment description. Within the gates, the cell properties can be accessed the same way as within the QiJob.

Advanced Control Flow: Context Managers

QiCode is written in Python. As Python is an interpreted language, some syntax is reserved and cannot be used for QiCode, like if/else or for statements. Instead, context managers (started by `with` statements) are used to implement control flows for QiCode within Python. This even makes it possible to combine QiCode which will only be evaluated during a later compilation step, and Python control flow to dynamically generate and change a QiJob while it is created. One example was already given in Code 4.17 where the QiCode commands for one qubit are repeated multiple times with a Python for loop to end up with a QiJob that controls multiple qubits. To distinguish normal Python syntax from QiCode, in QiCode all methods are written as Pascal case, i.e. with initial capital letter and capital letters for each consecutive word. In contrast, by convention, Python methods are normally written as snake case, i.e. with lowercase letters and spaces not omitted but replaced by underscores.

Accurate Timing: Implicit and Explicit Synchronization

Precise timing is of great importance in qubit experiments. Each command in QiCode will wait until the action it triggered is over. Hence, the command to play a pulse will wait the duration of this pulse before the next command is executed. This makes it possible to seamlessly sequence multiple operations. If one, for example, performs two consecutive pulse commands one one cell, these will be executed sequentially after each other without any gap between them. Different

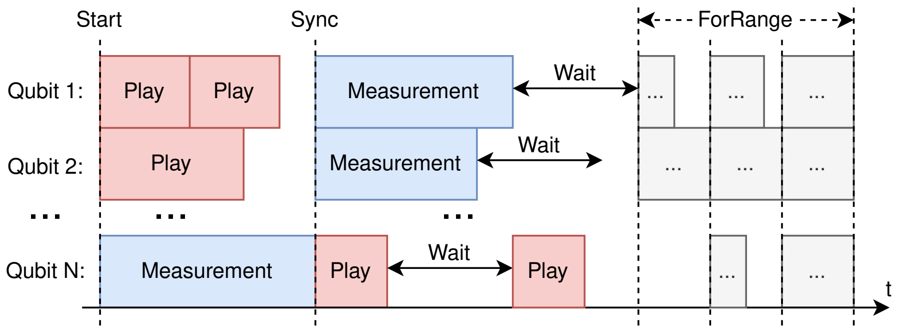


Figure 4.7: Timing concept in QiCode showing the timelines for N cells, i.e. qubits. The depicted operations are only exemplary to visualize the different time synchronizations across cells. The changing loop body of the ForRange is only hinted.

cells, however, have their own, independent timeline. The timing concept in QiCode is visualized in Figure 4.7. At the beginning of the execution, the timelines are synchronized across all cells. The first command of each cell will thus be executed at the same time. QiCode will also automatically align the timelines of multiple cells at the beginning of control blocks, like for loops, if/else clauses, or QiGates. This alignment will only take into account the cells that are used within these structures. A command to explicitly synchronize the timelines of multiple cells, `Sync`, is also provided so the user can reestablish a well-defined time relation where necessary. An example of a multi-qubit experiment, including its time relation, is given in Section 4.3.5 and also depicted in Figure 4.5.

Flexible Execution: Compilation and Cell Mapping

Once the QiJob is defined, it can be compiled into the configuration of the platform and executed. The details of these steps are provided in Section 4.4.3. For the compilation, the sample object needs to be provided, which will replace the template properties within the job. Furthermore, additional parameters regarding the execution can be specified, like the number of averages or repetitions to perform, and the format in which the resulting data should be collected and sent back to the user. Multiple predefined data collection modes are provided which can be further customized by passing a user-defined Taskrunner task for online data processing and retrieval. When multiple qubits are in play, the user can furthermore define on which qubit(s) the QiJob should be executed by providing a custom cell mapping. An overview over the general cell mapping between QiJob, QiCells sample object and QiController is illustrated in Figure 4.8.

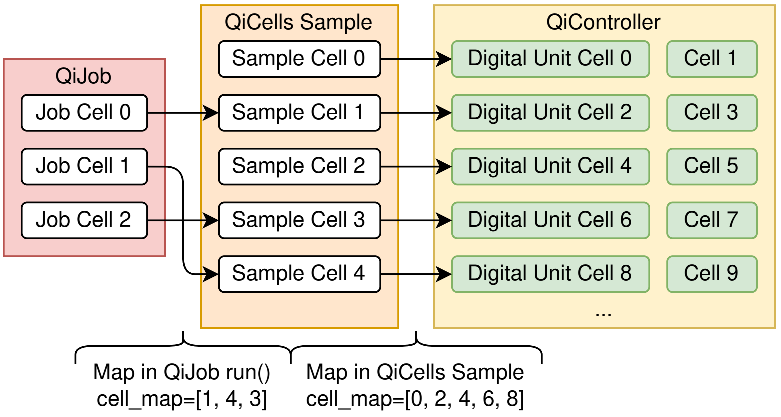


Figure 4.8: Cell mapping concept between QiJob, QiCells sample object and QiController. As an example, the QiJob utilizes three cells, and five qubits are defined as cells within the QiCells sample object.

4.4.2 Available Commands

QiCode provides a set of commands to describe the experiments and the control flow on the QiController. While Section 4.3 already showed some of them in action, this section will introduce the available commands and provide a more technical insight, how they are used and what their effect is on the platform.

> `QiCells(number)`
declares the number of cells to use for this experiment. Each QiJob needs to have exactly one QiCells declaration, typically right at the beginning. Its reference also acts as placeholder within the QiJob to define which command should act on which cell. Outside of the QiJob, QiCells objects act as representation of physical samples which can be filled with properties describing the relevant parameters of the qubits. During compilation, the internal placeholder QiCells object will be replaced by the provided sample with the actual parameters. This makes the QiJob reusable for different samples with different properties without needing any adaptations.

> `QiVariable(value_type, value, name)`
declares and creates a variable in QiCode. It will be assigned during compilation to sequencer registers of the cells in which the variable is used. If no type is provided, it will be intelligently inferred based on the usage within the QiJob. For example, if the variable is used as a pulse length, the type will automatically be inferred as a time. Available types are integer, time, state, and frequency and can be specified using the `QiType` enumeration. Depending on the type, the value of the variable

needs to be differently converted for the platform, e.g. a time will be converted from seconds to cycles. Multiple aliases exist for variables with predefined types, like `QiTimeVariable` and `QiStateVariable`. If a value is given the variable will be initialized to this value. Otherwise, it stays uninitialized. When a name string is provided, one can later access the register within the sequencer using the name, as the mapping between name and register index is stored. This can be used to initialize the variable to different values before starting the execution. If neither a name nor an initial value is provided, QiCode will complain if the variable is used somewhere before being initially assigned within the code.

> `Assign(destination, calculation)`
performs a calculation and stores the result in the QiVariable provided as destination. Any type of mathematical operation is possible that can be implemented within the sequencer and is not forbidden by type constraints. That means that it is not allowed to add a time and a frequency, for example. All common operations are available with the exception of the division which is not implemented in the sequencer. However, bit shifts are possible to divide numbers by a power of two. Also nested calculations are possible which will then be separated into elementary operations by the compiler. The operands of the calculations can be QiVariables as well as constant integer and float values.

> `Store(cell, variable, save_to)`
stores the value of the variable into the result container of the specified cell with the name given as `save_to`. The name is provided as string and can be used to identify the data after the experiment.

> `Wait(cell, delay)`
delays the execution of the commands within the cell by the given amount of time. The time can be given as number specifying the seconds or as QiVariable. On the sequencer, it will be implemented using the wait instructions of the custom special purpose instruction set.

> `Shape(name, function)`
defines a parametrized shape with a provided name. The provided function has to be of the form $[0,1) \rightarrow \{x \mid x \in \mathbb{C}, |\operatorname{Re}(x)| \leq 1, |\operatorname{Im}(x)| \leq 1\}$. It can be provided as lambda function or as normal method. The input value will then be scaled to the full length of the pulse that should have this shape. A library with predefined shapes is also provided with QiCode and called `ShapeLib`. It contains shapes for rectangular, Gaussian, ramp, and other pulses.

> `QiPulse(length, shape, amplitude, phase, frequency)`
creates a pulse object that can be used in the `Play` and `PlayReadout` commands. The length can be either a fixed value in seconds or one of the two strings `"cw"`

or `"off"`. The strings will turn on and off a continuous wave (cw) tone with the given parameters. The shape is a `Shape` object and defaults to a rectangular pulse. The amplitude can be a value between $[-1, 1]$ and defaults to 1. The phase is given as float in radian and by default is 0. The frequency has to be provided in hertz. If no frequency is given, the frequency will be automatically taken from another pulse of the same cell and category (readout/control). Length and frequency can also be given as a `QiVariable`. The compiler will then integrate a store operation in the sequencer instructions to update these parameters within the register interface of the appropriate signal generators. For a variable length parameter, only the rectangular shape is currently supported as it is implemented as a cw pulse that will be stopped after the time specified in the variable has passed.

> `Play(cell, pulse)`
outputs the specified `QiPulse` using the control signal generator of the given cell. The pulse will later be assigned to a trigger set of the signal generator during the compilation and consecutively loaded. A trigger instruction will be added to the sequencer execution to play this trigger set on the control signal generator. Afterwards, a wait time is inserted to delay the execution until the end of the pulse.

> `RotateFrame(cell, angle)`
rotates the reference frame of the control pulses generated by `Play`. This corresponds to a counter-clockwise virtual Z rotation around the Bloch sphere by the given relative angle. The operation takes one cycle and is a normal trigger command of the control signal generator where the rotation is loaded as trigger set. There, the option to persist the phase offset is selected which will cause the rotation of the phase reference and thereby the reference frame whenever this trigger set is activated.

> `PlayReadout(cell, pulse)`
performs the same operation as the `Play` command but using the readout signal generator instead of the control one.

> `Recording(cell, duration, offset, save_to, continuous)`
performs a recording at the input of the cell with the signal recorder. The duration specifies the length of the recording window, the offset is the electrical delay between the start of a preceding readout pulse and the time when the pulse enters the signal recorder. The offset can also be a QiVariables and will then be updated by a store operation of the sequencer. The save to parameter, similar to the `Store` command, can be a string to specify the name of a result container where to store the result values. Furthermore, it can also be a `QiStateVariable` so one can react to the measured qubit state within the control flow. The continuous flag can be used to start or stop a continuous measurement with `True` and `False`, respectively. By default, only a single measurement will be performed. The command is typically

used directly after a `PlayReadout` command and will then be merged with it to be executed simultaneously. The offset will be separately handled within the signal recorder and not by the sequencer.

> `Sync(*cells)`
synchronizes the timeline between multiple cells which can be passed as arguments here. Normally, each cell has its own, independent timeline. However, some pulses and operations need to be aligned to work properly. For this scenario, the Sync command can be used. The compiler will synchronize the timelines by calculating the difference in execution time and adding appropriate wait statements.

Besides these commands, QiCode supports complex control flows utilizing the following Python context managers:

> `with If(condition):`
only executes the following indented QiCode block if the given condition is true. A condition can be a fixed value or a QiVariable that will be evaluated to true if it is greater than zero, as well as a comparison of two QiVariables or between a QiVariable and a fixed value. Calculations can be nested within the condition which will then be separated by the compiler and executed before the branch instruction is performed within the sequencer.

> `with Else():`
only executes the following QiCode block if the condition within a preceding `If` context manager evaluated to be false.

> `with ForRange(variable, start, end, step):`
loops the value of the given variable from start to end with increments provided as step. The following indented QiCode block will be repeated for each value of the variable. The compiler performs a few optimizations, including unrolling single executions of the loop when the variable represents a time and is zero or one cycle long in this loop execution. This is necessary so commands that should be zero cycles long can be skipped in the unrolled loop iteration. When variable length pulses are used, the pulse is split into a trigger to turn on the pulse, a wait command that waits the number of cycles in the variable minus one, and a separate trigger to turn off the pulse again. If the pulse should only be one cycle long, the wait command in between needs to be left out as it will take at least one cycle which makes the pulse at least two cycles long if it is present.

> `with Parallel():`
allows the parallel execution of multiple QiCode commands acting on the same cell, e.g. performing a control and a readout pulse at the same time. In the normal timeline, the pulses would always be executed after each other. With the parallel

context manager, these can be executed at the same time. Multiple consecutive `Parallel` blocks are merged by the compiler and run in parallel. Within the indented blocks, the timeline will be organized as usual, with sequential commands of one cell and independent timelines between multiple cells.

> `with QiJob() as job_reference:`
encapsulates all the other QiCode commands. These can only be used within this context manager. The only exception is the QiCells command which can be defined inside and outside the QiJob separately. The QiJob builds a control flow graph at runtime from the utilized commands in its body and stores it for the compiler. The job can be referenced, compiled, and executed using the `job_reference` of the context manager and its provided methods, as further detailed in the next sections.

At the beginning of each of the context managers, the cells which are used within its body will be automatically synchronized by QiCode.

4.4.3 Compilation

QiCode is designed in a way that a control flow graph (CFG) of the experiment description is directly created when the Python interpreter goes through the code. For this purpose, the `QiJob` context manager creates a global variable containing this CFG. Each command of QiCode will extend it when being interpreted by Python. This intermediate representation is then stored inside the QiJob reference. Calculations with variables inside QiCode are stored as abstract syntax trees (ASTs). Python makes it possible to overload all mathematical operations with custom methods that will be called when the code is interpreted. This way, the AST will be directly created using the Python interpreter, including the correct operator precedence.

The QiCode compiler is started with the sample object, the cell map between sample cells and QiJob cells, and the CFG of the QiJob. As a first step, the cells from the sample object are extracted and rearranged to be in the same order as the cells of the QiJob. These contain the properties of the sample which can be necessary to compile the job. Then, the properties within the QiJob are replaced by the values given within the sample object cells. If any unresolved properties remain after this step, an exception is raised and the user is notified about the missing properties.

To compile the intermediate representation of QiCode into the configuration for the QiController, additional information needs to be gathered. For example, commands need to be assigned to the cells on which they have to be executed, like variables and if statements for which it is not explicitly defined. To extract this information from the CFG, the visitor pattern is used. Visitors represent operations that will be performed on the structure of the CFG by iterating through the single nodes. In the visitor pattern, the operation is separated from the structure. This way, new operations can be later added without having to adapt the data structure.

Multiple visitors are implemented which will walk through the tree structure of the CFG and annotate the nodes where necessary. As an example, each command will be annotated with a list of cells that are relevant for this command. To decide which variable is used in which cell, a visitor will iterate through the CFG in reverse and track the usage of the variables through all computations and so forth. Some visitors will also perform additional code validity checks and raise exceptions if forbidden or unsupported code is used. For example, the control variable of a `ForRange` must not be altered within the loop body to prevent infinite loops and guarantee well-defined behavior. During this phase of information gathering, also the number of recordings performed for a result container during the execution of QiCode is determined. This information is later used during the execution to determine how much result data to expect and fetch from the PL.

Once the visitors collected the required information and annotated the CFG, the translation into sequencer instructions takes place. To assemble the instructions, another visitor is used. It iterates through the CFG and, for each command node in the graph and each relevant cell of this node, selects the appropriate sequencer instructions.

To organize this code translation phase, each cell has its own object representing an abstract sequencer and digital unit cell. It takes care of assembling the sequencer instructions by providing appropriate methods that are accessed by the visitor. It furthermore contains a register stack that holds all the unused registers of the sequencer which can still be used during the execution. If a command node requires a new register, it will be popped from the stack and can then be utilized until it might later be freed again and pushed back to the stack. The registers also keep track of their value when traversing through the CFG. This way, the program length can still be calculated when variables are used as pulse lengths or for wait delays. Monitoring the execution time of the code in clock cycles is also handled within the abstract sequencer class. As long as the current execution times of all the cells are known, they can be synchronized at any time by adding wait statements with the time differences.

The translation of the command nodes in the CFG to the sequencer instructions can be separated in single and compound statements. Single statements include `Play`, `Recording`, `Wait`, and so on. They can be directly translated into appropriate sequencer instructions. If pulses should be played, they will be assigned with a trigger set value and added to a dictionary for the signal generator to be loaded later on. The compiler also compares pulses. If multiple pulses with exactly the same properties are used within the description, only one pulse is loaded and then reused. Compound statements, represented in QiCode as context managers, require more thorough handling. If multiple cells are relevant within such a statement, a synchronization between these cells is added at the beginning of the statement so it will be executed at the same time within all sequencers, as one would expect when reading the QiCode. Depending on the compound statements, different sequencer instructions will be added.

For the `If` block, a branch instruction will be added. When the condition is false, the inner body will not be executed and thus needs to be jumped over. The condition for the branch instruction will thus be inverted before being added. The relative jump address will be determined after the body of the conditional clause was added to the instruction list. If an `Else` block follows, an unconditional jump is additionally introduced at the end of the `If` body to jump over the other one.

The bodies of multiple `Parallel` statements are evaluated and merged into a single sequence of commands. These are then added to the sequencer instructions of the relevant cells. In order to merge the commands, their timing needs to be known. Then, wait times between parallel statement executions can be calculated and the multiple command sequences appropriately combined.

To translate the `ForRange` loop, multiple different cases have to be distinguished. If the run variable is a time and the start or end of the execution are zero or one cycle, these loop iterations need to be unrolled. For loops are implemented by first assigning the start value to the run variable. Then, a branch condition is added to check if the run variable is greater (or smaller for negative step sizes) than the end value. If not, the loop body will follow. Afterwards, the run variable is incremented and an unconditional jump back to the branch condition is performed. Once the branch condition is met, i.e. the run variable exceeds the for loop range, a jump is performed to the instruction following the loop body and the unconditional jump. The compiler also builds a tree structure of all existing for loops within the QiJob. It will later be used to track the progress of the sequencer execution by reading out the register values during runtime. As an example, if a `ForRange` increments the run variable a hundred times, one can track the progress during the sequencer execution by checking the current value of the register associated with the run variable. When start, stop and step values are known, one can calculate which iteration the sequencer is currently performing. For a more complex nesting of loops, the value of the program counter is also necessary to determine the progress of the execution. QiCode will automatically evaluate this value during runtime and present it to the user.

The language was initially implemented during the master thesis of Rainer Illichmann. More technical details can therefore be found in his thesis [140].

4.4.4 Execution

Once the compilation is completed, for each cell of the QiJob, a sequencer instruction list is returned. Additionally, the mapping between variable names, for those which are explicitly named, and the register indices on the different sequencers is returned. It can be used to access the belonging registers and initialize the variables based on their names. With this information, an experiment class instance is created that contains all the necessary logic to load the configuration of this QiJob and execute it on the QiController. The QiJob reference contains two methods for this purpose:

> `job.create_experiment(qic, sample, averages, cell_map, data_collection)`
> creates the experiment class instance of the QiJob `job` and returns it for more advanced usage to the user. During the creation process, the compilation explained above will take place. `qic` is the reference to the QiController, `sample` is a QiCells object containing the information of the qubit chip needed for compilation of the QiJob. With `cell_map`, the user can provide a mapping between sample cells and job cells. Based on the second map within the sample, the job cells will be further mapped onto the digital unit cells of the QiController. This cell mapping concept is also depicted in Figure 4.8. `averages` and `data_collection` will change the Taskrunner configuration, i.e. how the data is collected and how often the job should be executed on the PL. Different predefined modes exist which are explained below. After creating the experiment instance, the user needs to manually call the methods to configure and execute the job. These can also be further modified and expanded by additional operations. However, directly accessing the experiment instance is only required for few advanced use-cases.

> `job.run(qic, sample, averages, cell_map, data_collection)`
> also creates the experiment class the same way as the first command. However, the experiment is never exposed to the user but directly executed within the run method which is sufficient for most QiJobs. The results of the experiment will be automatically stored within the job cells accessible as `job.cells`.

The following predefined data collection modes and belonging Taskrunner tasks are available:

`"average"` is the default data collection mode and will return results as averaged I and Q values. Multiple recordings inside the job are averaged separately.

`"amp_pha"` performs the same measurements as `"average"`. The obtained I and Q values are afterwards converted to amplitude and phase values in Python.

`"iqcloud"` does no averaging on the resulting I and Q values but returns all single values back to the user, grouped by recording in the QiJob. If, e.g., five recordings are performed within the QiJob, five groups of measurement data will be returned, each containing as many I and Q values as the user specified as average count.

`"raw"` returns the averaged raw time trace obtained within the signal recorders. As only one time trace can be stored within the signal recorder per execution, this mode is limited to one recording per experiment description and cell.

`"states"` collects the single measurement results as decimated state values instead of as I and Q. Otherwise, the mode is similar to `"iqcloud"` and can be used to measure quantum jumps.

`"counts"` is similar to the previous mode, but it interprets the qubits as a matching quantum register and the last measured qubit states across all cells as a binary number. It then counts how often which classical number occurred when repeating the job multiple times and only reports back these counted values.

The execution of the QiJob can be separated in multiple stages:

In the first stage, the QiController is configured. The pulses are loaded into their associate trigger sets within the signal generators of the different digital unit cells. The NCO frequencies of the modules are updated and the signal recorder is configured according to the parameters given within the job. Then, the instruction lists are written into the sequencers of the digital unit cells and the appropriate data collection task is loaded onto the Taskrunner. The task will later trigger the execution of the sequencers and collect the results in the requested format. Different tasks are used to switch between multiple data formats and aggregations as explained above. The tasks are written in a very generic way so one can pass them the mapping of the unit cells, the number of recordings performed per signal recorder and how many repetitions to perform as parameters. Alternatively, a user-defined task can be loaded, as detailed in the following section.

In the second stage, the execution of the Taskrunner is started. From now on, the execution of the QiJob will happen completely decoupled from the Python client on the QiController. One can monitor the progress of the execution using the values provided by the task and in more detail also by monitoring the values of the sequencer registers and its program counter. Once the Taskrunner finished, the collected data is transferred from the platform to the client.

In the third and last stage, the data is converted into the right format within Python and distributed to the respective job cells where the user can access it.

4.4.5 Custom Data Processing

For special experiments, users might want to provide their own user task for the Taskrunner to perform custom data processing and control flows. QiCode provides the possibility to annotate a QiJob with a custom user task and data processing for these situations:

> `job.set_custom_data_processing(file, parameters, converter, mode)` attaches a custom user task C file which will control the `job` execution on the Taskrunner. One can pass custom parameters to the task. If none are given, the same set of parameters which is used for all predefined data collection modes is provided by default. The converter is a method which takes the data boxes returned from the task as argument and can post-process them. It needs to return the data in a structured format based on which it will be assigned to the cells of the job and their result containers. The mode defines the data type in which the data boxes have to be collected. By default, they are transferred and interpreted as lists of signed 32 bit integer values.

As an example, the user task `state_collect.c` of the `"states"` data collection mode packages multiple states as single bits inside one 32 bit unsigned integer value. If one created a job to record many such states, it can be annotated as shown in the following to extract and unpack the boolean state information:

Code 4.19: Annotating a job with custom data processing, in this example to collect qubit states.

```python
# Data converter to unpack the 32bit values back to boolean states
def data_converter(databoxes):
    results = []
    # Each data box contains the states obtained from one cell
    for db in databoxes:
        # States are compressed as single bits in 32bit unsigned integers
        results.append([(val >> i) & 0x1 for val in db for i in reversed(range(32))])
    return results

# Attach custom data processing with Taskrunner to job
job.set_custom_data_processing(
    file="state_collect.c", converter=data_converter, mode="uint32",
)

# Run the experiment with custom data processing (default if attached)
# The average count is used by the task to define the number of states to collect
job.run(qic, sample, averages=10000)

# Get the data as list of 0s and 1s for the first cell (as processed by the data_converter)
states = quantum_jumps.cells[0].data("result")[0]
```

4.4.6 Current Limitations

Current limitations are imposed by hardware restrictions and complex implementation efforts. Whenever possible, a descriptive error message will be displayed if the user tries to create an experiment description that is not supported. The most basic limitation is probably that time variables and values can only be a multiple of 4 ns as this is the time of a clock cycle within the PL. When using different values these will be rounded to the nearest 4 ns multiple in most cases.

Another limitation concerns the use of the `Parallel` context manager. As one needs to analyze the parallel command executions and merge them into one, synchronization and control logic is currently not allowed inside these context managers as the timing is challenging to infer or even impossible in some cases. For the same reason, it is also not possible to save the qubit state into a variable within a parallel construction as this will cause the sequencer to indefinitely wait until the state is returned by the signal recorder.

When changing the NCO frequencies of signal generators or recorder, the phase relation to other parts of the experiment is lost. If the phase relation is important, e.g. for control pulses within a sequence to specify the rotation axis, the frequency must not be altered. This is due to the fact that the modules only have one global frequency value but the phase relation is always defined to the common start of all digital unit cells. When changing the frequency mid-term, the NCO will just continue with the last phase value of the old frequency which does not match the required phase the new frequency would have needed.

Division, as well as fixed or floating point numbers are not supported within QiCode as they are not implemented within the sequencer. However, the use cases where this would be necessary at all are quite rare and uncommon, or can be implemented afterwards inside the Taskrunner or the Python client.

As the results are temporarily stored within the data storage inside the digital unit cell, a QiJob is currently limited to 1024 recordings per execution and cell. Otherwise, the memory is not sufficiently large to store all the results. A similar restriction applies to the length of the raw time trace which can only be 1024 ns long. Again, however, this is sufficient for most use cases.

The digital trigger module is not yet integrated into QiCode and relevant QiCode design decisions have not yet been taken.

QiCode supports variable length pulses. As these are implemented as a continuous wave which is turned off again after a delay matching the variable length has passed, only rectangular pulses are supported. Variable length pulses with arbitrary shape can be implemented by reloading the envelope memory between different executions using the Taskrunner. This entails a significant overhead when compared to an implementation solely within the PL. One possibility for the latter would be to store the envelope with high resolution inside a block RAM (BRAM) and then subsampling it with a step size depending on the desired pulse length. However, as this step size would be reciprocal to the pulse length, an obvious implementation of a linear for loop within the sequencer is not possible. As this significantly diminishes the advantage of variable length pulses, it was not implemented.

Another limitation is the duration of wait times and pulse durations. Wait times are limited by the number of cycles fitting in a 32 bit register, i.e. $2^{32} - 1$ cycles, corresponding to roughly 17 s. The pulse duration, with except for variable length pulses, is limited by the number of envelope samples which can be stored. As 8192 B of memory are available and each sample is 16 bit large, this corresponds to up to 4096 ns pulse length. In this extreme case, only one pulse can be loaded onto the signal generator. However, for nearly all applications, either short pulses on the order of ten to hundreds of nanoseconds are required, or the pulses can be realized using the continuous wave functionality.

4.5 Qkit Integration

Qkit is an open-source quantum measurement suite used to control laboratory instruments, perform experiments and store the acquired data [106]. A more detailed introduction into Qkit is given in Section 2.2.8. Physicists in multiple laboratories are used to perform research with superconducting qubits using Qkit. Thus, to facilitate the usage of the QiController, an integration of the QiController into Qkit was developed.

Qkit collects measurement data from the instruments and persists it as files on the hard drive. For this purpose, Qkit requires a special instrument that provides the resulting data in a specified format. Each experiment in qiclib therefore provides a special readout instrument that mimics this. It will trigger the experiment execution and reformat the obtained data for Qkit. It also provides additional functionality required by Qkit, like a method to query a list with the frequencies of all utilized readout tones for multi-qubit readout. The instrument is available as `readout` property of the experiments and also packaged inside a special `qkit_sample` property. The latter is a Qkit sample object stub which can be passed as adapter to the Qkit measurement class. As an example, a Rabi experiment job written in QiCode can be integrated in Qkit the following way:

Code 4.20: Integration of a Rabi experiment written in QiCode into the Qkit environment.

```
job = ql.jobs.Rabi(0, t_max, t_step) # Create QiJob from predefined experiment library
exp = job.create_experiment(qic, sample, averages=10000) # Create experiment for QiJob

# From here, the code is nearly identical to an experiment without the QiController
m = Measure_td(exp.qkit_sample) # Create Qkit measurement object and pass adapter
# Define the parameter axis within Qkit which is performed on the QiController
m.set_x_parameters(exp.time_range(0, t_max, t_step), 'pulse_length', None, 's')
m.dirname = "Rabi" # Name the experiment data
m.measure_1D_AWG(iterations=10) # Perform the measurement, obtain results and repeat 10x
```

In the code snippet, `exp.qkit_sample` is the experiment-specific adapter which will be created and passed to the Qkit measurement class. At this point, during the creation, also the QiController will be configured. Once Qkit requests a recording from the instrument within this adapter, it will trigger the execution of the QiJob on the platform. The data is then returned in the special format required by Qkit so it can be automatically added to the measurement file. Qkit handles parameter variations by defining an axis. As these are completely handled by the QiController in this case, the Qkit axis is only given to store and visualize the data correctly. The `exp.time_range` method of the experiment class is used to obtain a list with all time values the `ForRange` will generate on the sequencer.

Qkit furthermore adds meta information to the obtained measurement data, like information about the current sample, the status and configuration of all connected instruments at the start of the experiment, as well as optional further information given by the user. To leverage these capabilities of Qkit, an interface between the QiController and Qkit was developed. All submodules of the platform within qiclib, as well as the QiController itself, are derived from a common base class named `PlatformComponent`. At creation, it stores a unique name of the submodule, as well as the gRPC connection handle to the platform. If Qkit is available, it furthermore creates a `QkitInstrumentProxy` which mimics the functionality of a normal Qkit instrument and exposes the properties of this submodule for Qkit to be automatically persisted. All properties which should be persisted are decorated inside the class of the submodule using `@platform_attribute`.

Qkit also provides a visual progress bar widget when used within Python Jupyter notebooks. If Qkit is available, QiCode and the experiments also utilize this progress bar to visualize the progress of the execution, including iterations, averages and the progress within the QiJob.

The integration into Qkit substantially improves usability of the QiController for users who are familiar with performing experiments in Qkit. The interface is designed to have a low footprint and be as simple as possible so it is easy to use with a low entrance barrier.

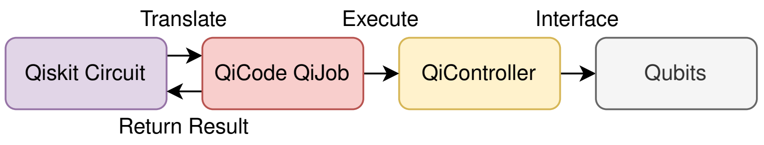


Figure 4.9: Abstract concept of the Qiskit integration with the QiController.

4.6 Qiskit Integration

Qiskit is an open-source quantum development kit for working with quantum computers at the algorithm level. It is widely used in the quantum computing community and also contains backends to interface with IBM quantum computers in the cloud. A general introduction into Qiskit is provided in Section 2.2.8.

Besides predefined IBM backends, custom backends can be provided to utilize Qiskit with own electronics and quantum processors. For the QiController, such a generic backend was developed to be used with all kinds of attached quantum processors. The goal of the Qiskit integration for the QiController is to be able to use as much of the same code and syntax from Qiskit as possible. The backend thus aims to integrate seamlessly in the remaining Qiskit workflow. This way, users familiar with Qiskit can use the rich ecosystem directly with the QiController without needing deep understanding of the platform. The abstract concept of the Qiskit integration is visualized in Figure 4.9. A Qiskit circuit is translated inside the backend to a QiJob and executed on the QiController. The obtained measurement results are then returned to Qiskit where they can be further analyzed.

Defining and executing Qiskit circuits on an IBM backend in Python looks like this:

Code 4.21: Defining and executing a quantum circuit on a simulator in Qiskit.

```python
from qiskit import *

circ = QuantumCircuit(3, 3) # 3 quantum and 3 classical bits
circ.h(0) # Hadamard gate on first qubit -> superposition
circ.cx(0, 1) # CNOT with qubit 1 as target and 0 as control
circ.cx(0, 2) # CNOT with qubit 2 as target and 0 as control
circ.barrier() # Barrier to measure all three qubits simultaneously
circ.measure([0,1,2], [0,1,2]) # Measure all three qubits, store states classically

backend = Aer.get_backend('qasm_simulator')
job = backend.run(circ, shots=1024)
result = job.result() # Obtains the results from the backend
counts = result.get_counts() # Gives the counts how often which result was measured
# Example for counts: {'000': 502, '111': 522}
plot_histogram(counts) # Visualize results
```

First, Qiskit is imported and a quantum circuit created. In this case it entangles three qubits by bringing the first qubit in a superposition and then performing a conditional not operation on the other two qubits with the first qubit as control. The final measurement stores the qubit states in the classical register. Due to the entanglement, one should only obtain two different measurement results, namely `000` or `111`. Using a simulator in Qiskit, this can be verified. A backend is created for the simulator and the circuit is executed on this backend with 1024 repetitions. The result is obtained from the backend and transformed to the counts of the different outcomes in the classical register. Finally, it is plotted as a histogram to visualize the result.

When using Qiskit together with the QiController, only the backend needs to be exchanged by a custom one provided by qiclib:

Code 4.22: Using the QiController backend to integrate seamlessly in Qiskit.

```python
from qiclib.packages.qiskit import QiController_backend
backend = QiController_backend(qic, sample, gates)
```

All the remaining Qiskit code stays exactly the same, at least from the perspective of the user. The custom backend requires a reference to the QiController, as well as a QiCells sample object. The sample object needs to include all relevant pulse lengths and times, as these are required during the translation of circuits to QiCode. It is therefore necessary to first fully characterize the connected quantum processor chip and store the information inside the sample object.

Qiskit circuits are structured as sequence of gate operations, similar to the structure of QiGates. When defining appropriate QiGates, a one-to-one mapping between Qiskit circuits and QiCode can be performed. This showcases the great flexibility of the QiCode experiment description language. A predefined set of QiGates for all common single-qubit Qiskit gates is provided for the translation. It can be replaced by a custom dictionary when creating the custom backend, see above, if the user wants to provide different gates. This can also be used to supply additional gates, for example a two-qubit gate. As these are highly experiment- and sample-specific, no predefined two-qubit gate is provided by default.

To translate the Qiskit circuit, each Qiskit gate will be replaced by the equivalent QiGate which is then added to a QiJob instance. Qiskit also has a concept of individual timelines for single qubits, and barriers can be used to synchronize them again. These will be translated to Sync commands in QiCode which essentially fulfill the same purpose. Measurement operations are handled the same way, and replaced by a Measurement QiGate as defined in Code 4.6. A resulting QiJob corresponding to the example in Code 4.21 will then look like this:

Code 4.23: Resulting QiJob from the translation of the Qiskit circuit in Code 4.21.

```
with QiJob() as job:
    q = QiCells(3)
    Hadamard(q[0])
    CNOT(q[0], q[1])
    CNOT(q[0], q[2])
    Sync(q[0], q[1], q[2])
    Measurement(q[0], save_to="result")
    Measurement(q[1], save_to="result")
    Measurement(q[2], save_to="result")
```

`Hadamard` and `CNOT` are QiGates which need to be defined inside the dictionary of gates. Qiskit also offers a transpile method which allows a generic Qiskit circuit to be optimized for a specific backend. Even if not all possible gates are implemented in the dictionary, it is sufficient to have a universal set of gates. Then, Qiskit can decompose all other gates to this basis set within the transpile method.

After the translation from a Qiskit circuit to a QiJob, this job is executed on the QiController with the predefined `"counts"` data collection mode. It will aggregate the measured qubit states directly on the Taskrunner by interpreting the qubit measurements as single classical bits of a register. The task then counts how often which register value is measured. This aggregation is reported back to the client and can be accessed the same way as for other Qiskit backends using the `get_counts()` method. Therefore, there is no difference from the view of the user between using an IBM or a QiController backend.

The Qiskit integration opens up tremendous possibilities to further employ the QiController in the field of quantum computing. Qiskit provides a rich ecosystem to interact with quantum processors, including a library of high-level quantum algorithms which can now all be readily used and accessed with the QiController. Qiskit also has an active community one can benefit from and which now can utilize the QiController as interface between Qiskit and actual qubit chips.

4.7 Summary

The client software provided with the QiController is an essential component to leverage its capabilities. It is bundled as a Python package, called qiclib, and provides means to access and control the platform with different layers of abstraction. This ranges from low-level access via remote procedure calls, provided by the ServiceHub, up to the high-level experiment description language QiCode.

QiCode enables users to functionally and intuitively describe experiments performed with the QiController. It is designed from ground up for experiments and applications with superconducting qubits and in close exchange with experimental physicists. Experiment descriptions are called QiJobs and can act on one or multiple qubits which are represented as abstract QiCells object. Commands exist to perform operations on single cells, i.e. qubits, like playing a pulse or performing a recording. These can be combined into modular, reusable building blocks, called QiGates. QiCode also supports advanced control logic, like variables, conditional clauses and loops. Each cell has its own timeline where commands are executed sequentially but independent of the other cells, if not explicitly or implicitly synchronized. The description is compiled into a platform configuration which is then loaded onto the QiController. With the Taskrunner, multiple repetitions are performed and the result data is collected. A user-defined task can be specified for custom data processing.

Integrations into the quantum measurement framework Qkit and the IBM quantum development kit Qiskit further extend the usability of the QiController. With a custom Qiskit backend, users can conveniently run quantum algorithms written in Qiskit on actual quantum processors controlled by the QiController. The Qkit integration simplifies the recording of the experiments shown in the next chapter. There, the QiController and QiCode are employed to perform exemplary experiments showcasing the capabilities and strengths of the platform.

5 Experimental Applications

This chapter provides exemplary experiments where the QiController is successfully utilized. Some of these experiments are either impossible to implement using common laboratory equipment due to limitations of the setup, or require much more complex control from the user control computer. For each experiment, both the physics and the platform perspective are covered. From a physics perspective, the necessary background is provided to understand and motivate the application. Furthermore, the exemplary results are discussed, thereby also demonstrating the functional capability of the platform. In each case, the physics view is complemented by the QiController setup, as well as a discussion about the provided functionality and benefits of the platform. Apart from the specific applications introduced in the following, the QiController is also actively used on a daily basis in research with superconducting quantum bits (qubits) at Karlsruhe Institute of Technology (KIT) and University of Glasgow.

5.1 Fast Feedback Control for State Preparation

5.1.1 Motivation and Physics Background

Quantum computing relies on the ability to efficiently initialize the state of the qubits. This is also formulated as the second condition of the DiVincenzo criteria. The most simple operation lets the qubit state reach thermal equilibrium by waiting five to ten times the energy relaxation time T_1 of the qubit. With improving qubit properties, this duration becomes more and more substantial and significantly slows down the execution of quantum algorithms. As example, waiting times of at least 0.5 ms are necessary for qubits with $T_1 \approx 100\,\mu\text{s}$, which is much longer than typical gate times of tens of nanosecond. Furthermore, the thermal initialization leads to a non-negligible population of the excited state $|1\rangle$. Therefore, a faster, and more effective approach to initialize the qubit state is required.

One option is to measure the state and perform a correction pulse, if necessary, to reliably initialize the qubit state in $|0\rangle$. Quantum non-demolition (QND) measurements, as introduced in Section 2.2.4, project any superposition onto the two discrete basis states $|0\rangle$ and $|1\rangle$. However, prior to the measurement, one does not know the resulting state. Thus, if the qubit is projected to the $|1\rangle$ state, one needs to perform an additional π-pulse to rotate the state into the ground state $|0\rangle$. This way, one can reliably initialize the qubit, if measurement and π-pulse are accurate and the latency is much smaller than the T_1 time of the qubit. Also some quantum error correction schemes depend on such a feedback mechanism. This cannot be realized with common laboratory equipment as the communication delays between the different systems are much larger. In contrast, the QiController supports this operation on the sequencer level. It is only limited by the delay of the readout signal through the experiment and the processing latency within the platform.

5.1.2 Experiment Setup

For the experiment, a superconducting Fluxonium qubit [38] is used. It has a T_1 time of 80 µs, therefore state initialization by thermalization would require 0.4 to 0.8 ms. The transition frequency of the qubit is $f_{01} = 1.26\,\text{GHz}$ and the frequency of the readout resonator $f_\text{r} = \omega_r/2\pi = 7.244\,\text{GHz}$. An analog radio frequency (RF) frontend is utilized for both signals to reach these frequencies. It is connected to the QiController which generates readout and manipulation pulses and records the response of the sample. To obtain single-shot readout fidelity, a quantum-limited parametric amplifier [112] is used. Relatively long readout pulses of 800 ns further enhance the signal quality. The π-pulse is calibrated to be a 160 ns long square pulse. More details on the experiment setup can be found in [46].

5.1.3 QiController Setup

The QiController can be programmed using the following QiCode job when reusing the QiGates defined in Section 4.3:

Code 5.1: Execution of feedback operations including control measurements using QiCode.

```python
with QiJob() as state_init:
    q = QiCells(1)
    state = QiStateVariable()

    # Conditional pi pulse depending on qubit state
    Measurement(q[0], save_to=state)
    with If(state == 1):
        PiPulse(q[0])

    # Verify the state in which the qubit ended up
    Measurement(q[0], save_to="result")
    Thermalize(q[0])
```

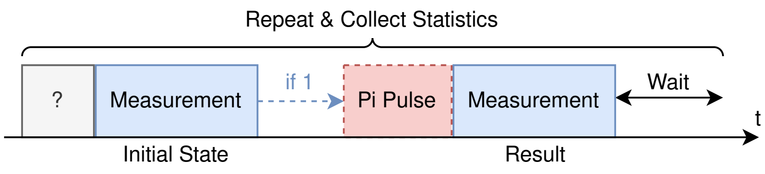


Figure 5.1: Sequence to initialize an unknown initial qubit state to $|0\rangle$ with a conditional π pulse.

```
    # Repeat with thermal |1> state as starting point
    PiPulse(q[0])
    Measurement(q[0], save_to=state)
    with If(state == 1):
        PiPulse(q[0])
    Measurement(q[0], save_to="result")
    Thermalize(q[0])

state_init.run(qic, sample, averages=1000000, data_collection="iqcloud")
```

In this QiJob, the qubit state is once reset from thermal equilibrium $|0\rangle_{\text{th}}$. Afterwards, it is reset a second time but from the reversed thermal equilibrium $|1\rangle_{\text{th}}$ prepared by a π-pulse. This way, one can compare the efficiency of the state preparation in both cases. The schematic pulse sequence for both cases is depicted in Figure 5.1.

Prior to the experiment, the state discrimination needs to be configured. For this, two histograms in the in-phase and quadrature (I/Q) plane are created with the measurement result of the qubit in $|0\rangle_{\text{th}}$ and $|1\rangle_{\text{th}}$, see Figure 5.2. With single-shot readout, two separate regions are visible, one representing $|0\rangle$ and one $|1\rangle$. The line separating the plane between these regions is calculated and loaded into the signal recorder, compare Equation 3.2. Then, the qubit state will be correctly estimated and reported back to the sequencer which can react based on it.

5.1.4 Results and Discussion

The results of the fast feedback state initialization are shown in Figure 5.2. When starting in the state of thermal equilibrium $|0\rangle_{\text{th}}$, the qubit's excited state $|1\rangle$ is still populated non-negligible with 11.7 % probability. This corresponds to an effective qubit temperature of 30 mK as can be inferred by using the Maxwell Boltzmann distribution. It is slightly higher than the cryostat base temperature of roughly 20 mK. The difference can be explained by limited thermal coupling of the chip to the environment and energy deposition by readout and manipulation pulses. After

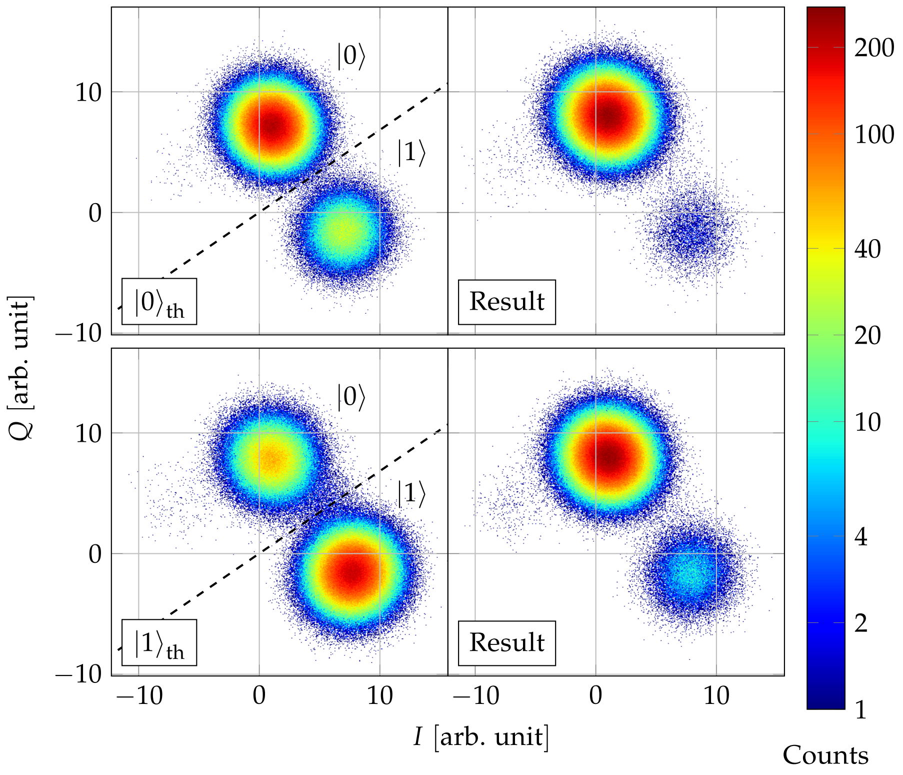


Figure 5.2: Demonstration of the state preparation by fast feedback using the QiController. The plots show histograms of one million single-shot measurements each. On the left, the initial state is displayed. The dashed line indicates the decision border programmed on the QiController to distinguish the qubit states. On the right, the corresponding result after the reset operation is shown. Note that the color scale is logarithmic.

the state initialization procedure, the probability of the qubit to be in the excited state is reduced to only 0.6 %, i.e. the qubit is cooled to 11.9 mK.

When the the qubit is first excited into $|1\rangle_{\text{th}}$, the probability of the qubit being in $|1\rangle$ is 77.2 %. Therefore, contrary to the other situation, a π-pulse needs to be performed in most cases. After the initialization, the population in the $|1\rangle$ state is reduced to 3.1 %. This is slightly higher than if the qubit is already in the ground state but still lower than thermal equilibrium. The effective qubit temperature corresponds to 17.5 mK.

The achieved fidelity of the initialization thus varies between 99.4 % and 96.9 %. It is limited by the experiment setup and the duration of the pulses, as well as the fidelity of the π-pulse and the readout, including the state discrimination. The

Bayesian error of the latter alone places an upper bound of 99.5 % fidelity due to a slight overlap of the Gaussian-like noise visible in Figure 5.2. The increased infidelity of a reset from $|1\rangle_{\mathrm{th}}$ can be partially attributed to a limited fidelity of the π-pulse which is implemented as square pulse and performed much more often than in the other case. The energy relaxation from the excited state back to the ground state, characterized by T_1, also contributes to the error. The whole sequence of operation with electrical delays, platform latency, readout pulse length etc. takes roughly $t = 1.5\,\mu\mathrm{s}$. During this time, a 100 % $|1\rangle$ population would decay to a $e^{-t/T_1} \approx 98.1\,\%$. However, a state change between the first and the second measurement will lead to an unexpected result and thus to a decreased fidelity. To improve the fidelity, the feedback latency, the measurement and π-pulse times can be reduced, and the T_1 time of future qubits increased.

Concerning the QiController, the feedback latency could already be further reduced from 428 ns to 352 ns after the experiment took place. This is mainly attributed to a change of the design clock frequency from 125 MHz to 250 MHz. At the same time, the cell multiplexer has been added after this measurement which leads to an additional delay to route the signals between unit cells and converter channels. Furthermore, additional pipelining has been introduced at critical signal paths to ease timing requirements for the field-programmable gate array (FPGA) when multiple digital unit cells are implemented. The platform latency is also covered in Section 3.7.3.

In conclusion, the state of a Fluxonium qubit was successfully initialized with a fidelity exceeding 96.9 %. This demonstrates that the QiController supports fast feedback control, thereby making high-precision online state preparation possible and opening up possibilities for advanced control schemes in superconducting qubit research and quantum computing.

5.2 Quantum Jumps Measurement

5.2.1 Motivation and Physics Background

When measuring a qubit, it will always either be in $|0\rangle$ or $|1\rangle$. When one constantly monitors the qubit state, one will experience so-called quantum jumps. These are discrete jumps between the two states which become visible due to the continuous measurement. The time of a jump is completely random, but guided by statistical principles laid out by the energy relaxation time T_1 due to unwanted interaction with the environment.

Monitoring these jumps can yield interesting insights into the dynamics of the qubit. However, this requires to continuously process an input data stream from the analog-to-digital converters (ADCs) of 12 GB/s per qubit, consisting of the two 12 bit ADC inputs for I/Q signals sampled at 4 GHz. Thus, online data processing is key in order to reduce the data rate to an acceptable amount which can be handled and stored in software. For quantum jumps, this reduction can be as far as one bit per measurement to represent the obtained qubit state. Depending on the length of an individual recording, this results in millions of qubit states being measured each second.

5.2.2 Experiment Setup

To measure these qubit states, a similar setup as in the previous experiment is required. Instead of a Fluxonium qubit, the superconducting concentric Transmon qubit presented in Section 2.2.6 is used in this case. Due to aging of the materials, the properties of the qubit changed since the initial characterizations. The updated qubit transition frequency is $f_{01} = 6.334\,\text{GHz}$ and the readout resonator is located at $f_\text{r} = 8.576\,\text{GHz}$. The T_1 time of the qubit was determined to be roughly 20 µs. To be able to detect the state with single measurements, special amplification is needed. For this, a traveling wave parametric amplifier (TWPA) [110] is used. A 400 ns readout duration per qubit state is chosen which is sufficient to distinguish the qubit states but resulted in a small overlap of the two measurement regions reducing the readout fidelity. However, longer durations have not been feasible as the T_1 time is limited. Instead, the readout power is significantly increased above single-photon level for sufficient signal-to-noise ratio (SNR) which might adversely contribute to local heating on the qubit chip. Special care has to be taken that the constant power input to the chip does not shift the qubit or the resonator.

5.2.3 QiController Setup

The QiController is set up to perform a continuous, repetitive recording each 400 ns on the FPGA, as depicted in Figure 5.3. The experiment can be defined using the following code:

Code 5.2: QiJob for the continuous quantum jumps measurement.

```
shots = 32000000 # 32 mio. consecutive qubit state measurements
with QiJob() as quantum_jumps:
    q = QiCells(1)
    # Turns on continuous readout tone
    PlayReadout(q[0], QiPulse("cw", frequency=q[0]["rec_frequency"]))
    Wait(q[0], 100e-9) # Wait for resonator to stabilize
```

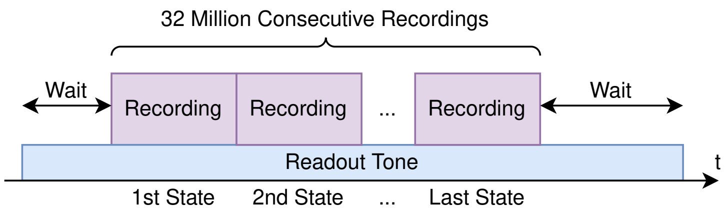


Figure 5.3: Sequence to perform 32 million continuous and consecutive qubit state measurements.

```
# Turn on continuous, repetitive recordings each 400ns
Recording(q[0], 400e-9, q[0]["rec_offset"], "result", continuous=True)
# Wait for the number of measurements, then turn off continuous mode again
Wait(q[0], (shots - 1) * 400e-9)
Recording(q[0], 400e-9, q[0]["rec_offset"], continuous=False)
Wait(q[0], 10e-6)  # Wait until last recording has finished
PlayReadout(q[0], QiPulse("off"))
```

To continuously read out the resonator, a steady tone is applied. After some settling time, the recording module is started in the continuous mode where it seamlessly aligns consecutive measurements. One qubit state is extracted every 400 ns resulting in a timing precision for a quantum jump of ±200 ns. After waiting for the desired number of recordings, the recording module is stopped again.

The data storage memory will be configured as ring buffer and store the qubit states as individual bits inside 32 bit values until the Taskrunner collects and transfers them to the user client. Collecting the qubit states online during the experiment execution is necessary as the data storage memory inside the FPGA can only hold up to 32 768 boolean state values. This corresponds to 13 ms of recording, and thus is not suited for longer measurements. The nearly deterministic and stable execution of the Taskrunner without unpredictable interference by other processes is a great advantage. It enables continuous data fetching and processing without any data loss from the ring buffer due to fluctuating execution time. The Taskrunner itself can store 3.84 billion states in the random access memory (RAM) without transferring them to the user, corresponding to 25.6 min of data recording. The task for this experiment copies the packed states from the data storage and forwards them to the user in the same format. Its source code is given in Appendix C.3. The task can be attached to the QiJob as custom data processing:

Code 5.3: Custom data processing for the continuous quantum jumps measurement.

```
# Data converter to unpack the states stored as single bits in 32bit unsigned integers
def data_converter(databoxes):
    db = databoxes[0] # Task returns one databox per qubit -> in this case only 1
    return [(packed >> i) & 0x1 for packed in db for i in reversed(range(32))]
```

```
# Attach custom data processing with Taskrunner to job
quantum_jumps.set_custom_data_processing(
    "quantum_jumps_multi.c", converter=data_converter, mode="uint32"
)

# Run the experiment with custom data processing (default if attached)
quantum_jumps.run(qic, sample, averages=shots)
states = quantum_jumps.cells[0].data("result")[0]
```

The provided data converter processes the obtained data boxes and extracts the single bits of the values to a list of zeros and ones. In principle, data transfer to the user can also happen simultaneous to the experiment execution, thereby making continuous recordings over very long timescales feasible.

5.2.4 Results and Discussion

For this demonstration, 32 million states are recorded. Of these, the qubit was detected roughly 5.6 million times in state $|1\rangle$, accounting to a population of 17.4 %. This is quite a high value, indicating that either the state separation was not very good and a significant overlap resulted in many wrongly detected $|1\rangle$ states, or the temperature of the qubit chip was substantially increased. If one assumes the probability is correct and neglects higher states, one obtains an effective qubit temperature of around 200 mK. While this is significantly higher than for the Fluxonium qubit, it is comparable to previous experiments with the same qubit [43].

Figure 5.4 shows a section of the quantum jumps measurement, as well as an analysis of the full data regarding the durations the qubit state remained unchanged. From the histograms, transition rates can be inferred leading to an energy relaxation time $T_1 \approx 7\,\mu s$. This is substantially smaller than the measurement obtained using pulsed measurements. However, this is not unexpected as the qubit properties will be affected by the constant population inside the resonator. The continuous readout tone can furthermore induce local heating onto the chip which would also reduce the T_1 time. This also seems plausible due to the high observed effective qubit temperature and the strong readout tone required to distinguish the qubit states.

For the exponential fits visible in Figure 5.4, the lowest bin has been excluded. It is significantly larger than the remaining data and does not fit to the exponential decay otherwise performed. It is likely that this is caused by the non-perfect discrimination of the qubit state and thus a non-negligible overlap in the I/Q plane. In this case, due to the overlap and noise, one would expect to see frequent changes of the qubit state on a fast time scale, and thus an increase in the first bin of the histogram. For state $|0\rangle$, the first bin includes all durations up to 14.4 µs (36 consecutive state recordings). For state $|1\rangle$, it includes durations up to 2 µs (5 consecutive state recordings).

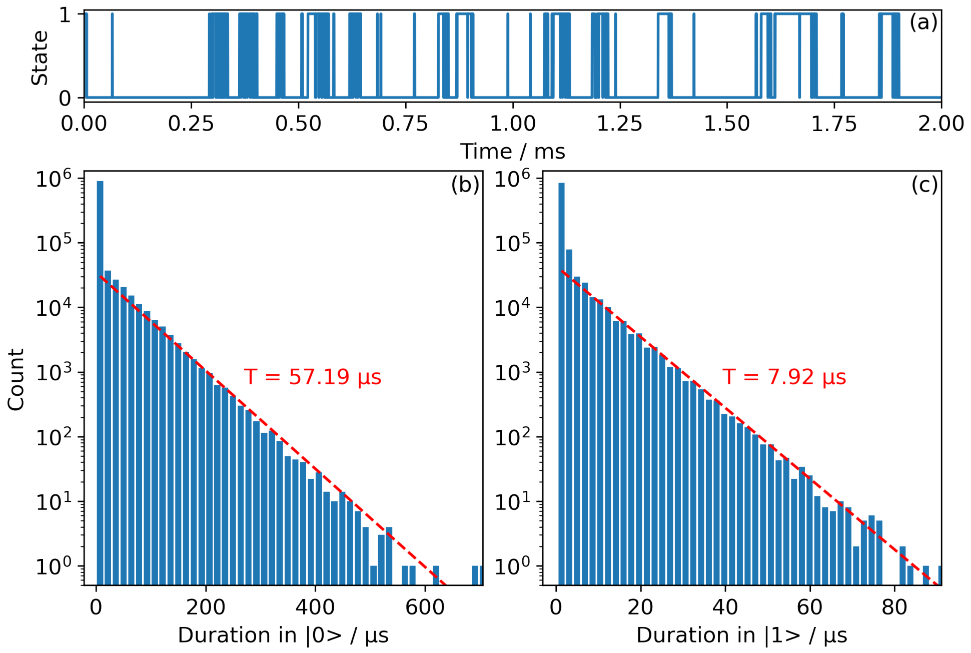


Figure 5.4: Quantum jumps measurement with 32 million states. (a) 2 ms cutout of the measured states. (b) Histogram over the duration which the qubit stayed in state $|0\rangle$. The red dashed line is an exponential fit to obtain the characteristic time scale. (c) Same histogram, but over the duration which the qubit stayed in state $|1\rangle$.

From a technical perspective regarding the QiController, recording 32 million consecutive qubit states resulted in a total acquisition time of 12.8 s. During this time, the ADC input data stream of 12 GB/s is reduced by the QiController to 313 kB/s. The accumulated input data at the ADCs of 153.6 GB during this time is processed to only 4 MB of result data. Thereby, the QiController performed a tremendous data reduction by a factor of nearly 40 000.

Quantum jump measurements can also be performed with multiple qubits. The provided user task for the Taskrunner already supports multi-qubit operation. With five qubits, for example, the QiController can measure a qubit state as fast as every 68 ns while still correctly fetching and forwarding the data using the Taskrunner. For lower recording durations, the signal recorders fill the memory inside the data storage modules faster than the Taskrunner can collect the data and data loss will inevitably occur. However, with typical recording durations being multiple hundreds of nanoseconds, this is more than enough for nearly all applications.

To conclude, the QiController supports enormous data reductions by reducing hundreds of nanoseconds of raw ADC data down to one bit of information. Data reductions by a factor of nearly 40 000 have been experimentally verified. The flexible architecture of the QiController does not only enable the user to perform this reduction in a seamless and continuous manner, it also provides the means to further collect and send this data to the user in parallel to the experiment execution. Thus, quantum jumps can be recorded continuously for very long periods. While not shown in this demonstration, quantum jumps have already been successfully recorded with the QiController over multiple hours.

5.3 Detecting and Changing the Speed of Light

5.3.1 Motivation and Physics Background

Building a quantum computer not only requires high-quality qubits. Quantum information also has to be stored during and between calculations. Therefore, an efficient and universal quantum memory is needed. One approach is to significantly reduce the speed of light of propagating microwave signals carrying this information in a controlled manner [149]. This effect is called slow light and significant progress has already been made with atoms at optical frequencies, now achieving coherent quantum storage times of over an hour [150]. For superconducting circuits, however, comparable devices are still lacking. By combining multiple qubits in a so-called qubit metamaterial, the dispersion relation of this material can be engineered and slow light can be achieved [151]. Moreover, by actively controlling the qubits, in-situ control of the speed of light is possible. While this was already theoretically described by Leung and Sanders [151], so far, an experimental verification was still missing. Using the QiController and an eight Transmon qubit metamaterial, a first realization of slow light in superconducting circuits could be successfully demonstrated [152]. It thereby prepares the ground for future applications in quantum computing, e.g. as on demand storage-and-retrieval memory.

Slowing down the speed of light can be achieved in numerous ways. One of the most prominent techniques is a nonlinear optical phenomenon based on quantum interference called electromagnetically induced transparency (EIT) [153]. It is observed in atoms with three energy levels $|0\rangle$, $|1\rangle$, and $|2\rangle$, where the $|0\rangle - |2\rangle$ transition is suppressed. For perfect EIT, these need to form a Λ-type energy structure, see Figure 5.5a. A signal which is applied to the $|0\rangle - |1\rangle$ transition will be absorbed by the system. EIT is created by applying a constant control tone to the $|1\rangle - |2\rangle$ transition leading to Rabi oscillations between these two states.

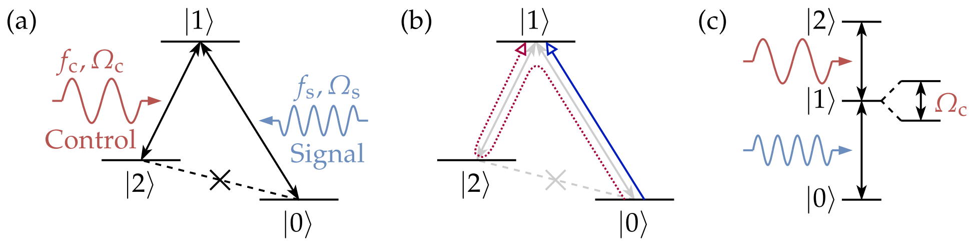


Figure 5.5: Structure of a three energy level atom. (a) Λ-type energy structure with a control tone and a signal tone applied. (b) The same structure indicating the two absorption paths which are destructively interfering in EIT. (c) Ladder-type energy structure with the same tones applied and Autler-Townes splitting indicated.

Now, two possible paths exist for the signal to excite the $|0\rangle - |1\rangle$ transition. This can either happen directly or via the driven oscillation of the control tone, i.e. $|0\rangle - |1\rangle - |2\rangle - |1\rangle$, compare Figure 5.5b. Both paths interfere destructively, thereby suppressing the absorption and leading to a transparency window. Hence, applying the control tone induces transparency in the system, as the name EIT suggests [153].

The transparency window exhibits a steep dispersion profile. As the group velocity v_g is inversely proportional to the slope of the dispersion relation $\partial k/\partial\omega$, this leads to a low v_g and thus slow light [151]. The speed reduction in this case can be described by the group index n_g. It is defined as $n_g = c/v_g$ with c being the speed of light in vacuum. A group index of 100 thus indicates that the speed through the material is 100 times slower than in vacuum.

Superconducting Transmon qubits are artificial atoms and are also described by the same quantum optical formalism. Yet, they do not have a Λ-type energy level structure and thus perfect EIT is not possible [152]. But also with the ladder-type structure of Transmons, EIT can still be observed using the $|0\rangle - |1\rangle$ transition for the signal and the $|1\rangle - |2\rangle$ one for the control [154]. By combining multiple superconducting artificial atoms as linear array, this effect can be utilized to create a coherent signal storage [151]. When driving the $|1\rangle - |2\rangle$ transition using a control tone with amplitude proportional to Ω_c, these levels hybridize and Autler-Townes splitting [155] can be observed as depicted in Figure 5.5c. The $|0\rangle - |1\rangle$ transition splits up in two separate band gaps with increasing Ω_c leading to the emergence of a transparency window in between with group velocity v_g being proportional to Ω_c^2 [152]. Therefore, low control amplitudes Ω_c are desired to obtain the highest possible signal retardation. At the same time, high amplitudes Ω_c are beneficial as the splitting is proportional to Ω_c and it thereby determines the bandwidth of the transparency window.

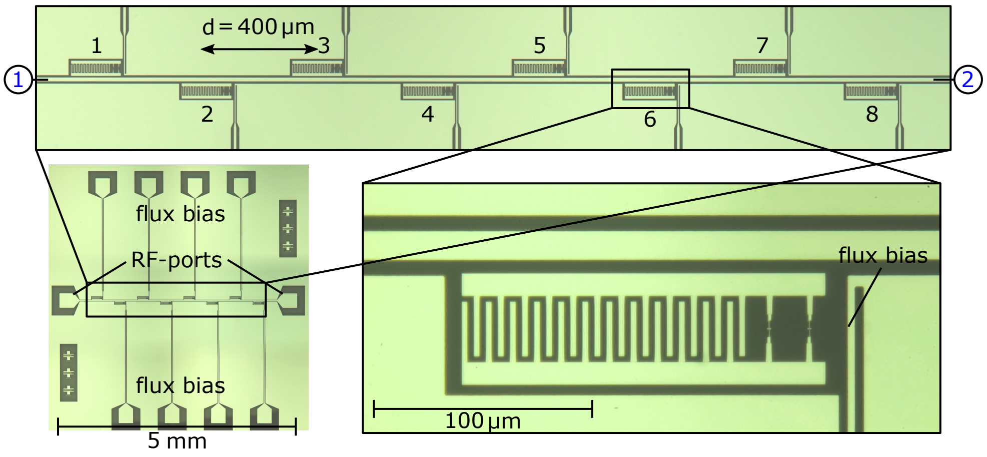


Figure 5.6: Microscopic pictures of the eight qubit metamaterial. Light green regions are the superconducting aluminum film while the dark areas correspond to the underlying sapphire wafer where the superconductor has been removed. The Transmon qubits are capacitively coupled to a central waveguide used for both control and signal tones entering at port 1 and leaving at port 2. Each qubit has a separate local flux bias line to tune its transition frequency. The spacing between neighboring qubits is 400 µm. The sample was fabricated by Jan Brehm and the pictures are taken from his work [152].

5.3.2 Experiment Setup

The metamaterial used in this experiment is an eight superconducting Transmon qubit chip which is capacitively coupled to a coplanar transmission line waveguide. The chip is fabricated and measured by Jan Brehm (KIT) [152]. A picture of the sample is shown in Figure 5.6. For the qubits to behave as a single metamaterial, the qubits need to be densely spaced, closer than the utilized signal wavelengths. The transition frequency f_{01} of the qubits can be individually tuned using flux bias lines. These introduce a local magnetic flux at the qubits which have a superconducting quantum interference device (SQUID) loop integrated. Seven of the eight qubits are tuned to a common transition frequency of $f_{01} = 7.812\,\mathrm{GHz}$ by external DC current sources. Then, a continuous control tone with a frequency of $f_c = f_{12} = 7.533\,\mathrm{GHz}$ and varying amplitude Ω_c is applied to resonantly drive the $|1\rangle - |2\rangle$ transition and create the Autler-Townes splitting.

Having prepared the metamaterial, the speed of light through it is measured by generating and recording signal pulses. Due to the limited bandwidth, Gaussian-shaped pulses with $\sigma = 50\,\mathrm{ns}$ at frequency f_{01} are used. The amplitude of the pulses are adjusted to the single photon level so they do not saturate the qubits. From the delay τ of the pulses through the metamaterial, the group velocity v_g

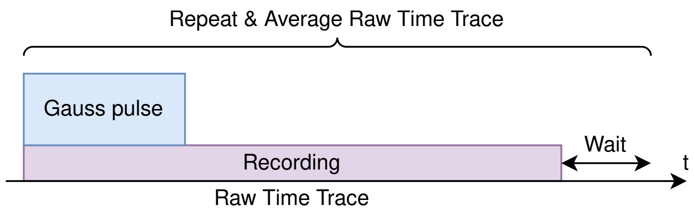


Figure 5.7: Sequence to measure the raw response of the system after a Gaussian pulse.

can be calculated. It is given by $v_g = d/\tau$, with $d = 2.4\,\text{mm}$ being the length of the metamaterial.

5.3.3 QiController Setup

For determining the delay τ of the signal through the metamaterial, the raw measurement capability of the QiController is used. The Gaussian-shaped pulses are created by the signal generator. The returning signals are then recorded by the signal recorder and subsequently averaged within the Taskrunner. Afterwards, they are transmitted to the user and further demodulated to obtain the Gaussian envelope within Python. The schematic pulse sequence of this experiment is given in Figure 5.7, corresponding to the following QiJob:

Code 5.4: QiJob to measure the signal delay through the metamaterial.

```python
with QiJob() as signal_delay:
    q = QiCells(1)
    # Create Gauss pulse, shape is between +- 3 sigma (sigma = 50ns)
    PlayReadout(q[0],
        QiPulse(6*50e-9, shape=ShapeLib.gauss, frequency=q[0]["rec_frequency"]))
    Recording(q[0], length=1000e-9, offset=0, save_to="result")
    Wait(q[0], 2e-6) # Give system some time to recover
signal_delay.run(qic, sample, averages=10000, data_collection="raw")
# [...] Process obtained raw data to determine retardation
```

Depending on the delay through the metamaterial, the returned pulse signal will be shifted in time relative to the generation of the pulse. By comparing the position to a reference which was recorded when the qubits transitions are all far detuned from the signal frequency, the pulse delay τ can be extracted and the group velocity v_g calculated.

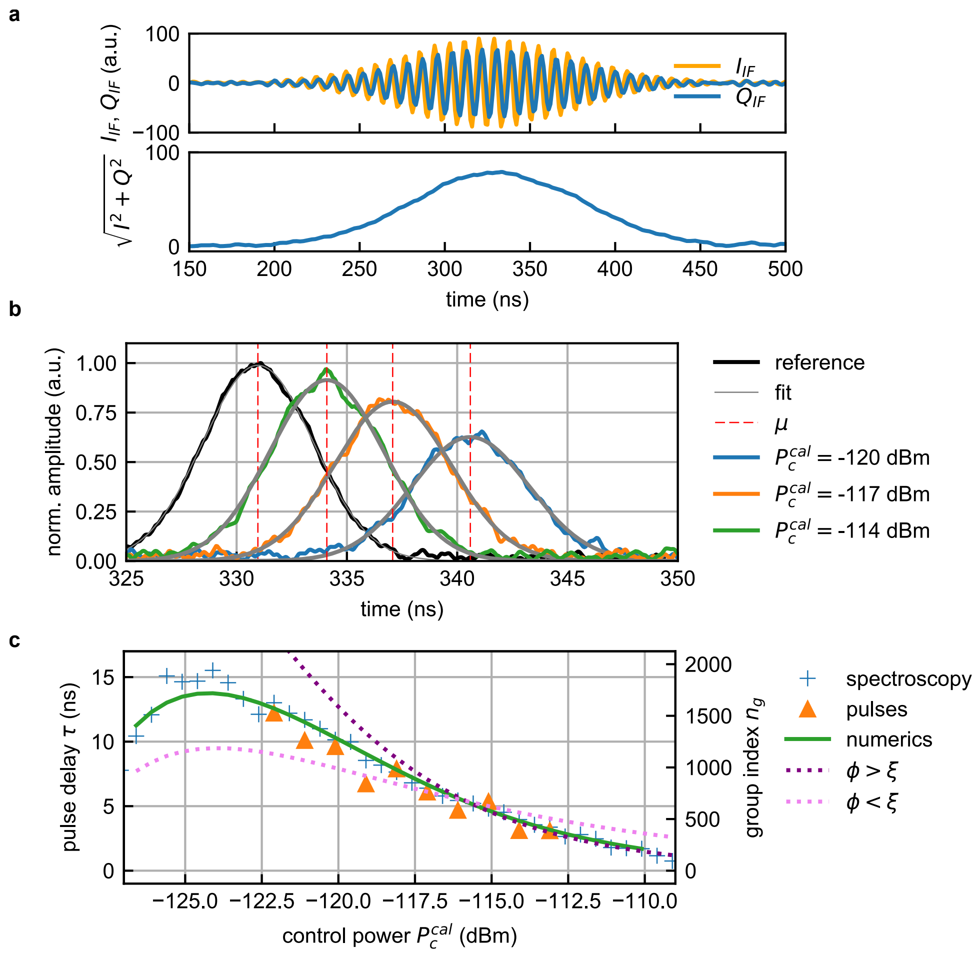


Figure 5.8: (a) Raw time trace of a Gaussian pulse recorded by the QiController and its extracted envelope amplitude after digital down-conversion (DDC) and low-pass filtering. (b) Envelopes of the detected Gaussian pulses for different control powers P_c. To better distinguish the pulses, their length is shortened by a factor of 20, while their fitted maximum was kept at the original position. (c) Pulse delays τ through the metamaterial for different control powers. The delays are both extracted experimentally from spectroscopic and pulsed measurements, as well as verified theoretically by a numerical simulation. [152]

5.3.4 Results and Discussion

The results for different control amplitudes Ω_c and thus control powers P_c are shown in Figure 5.8. A clear retardation is visible with decreasing control power. At the same time also the amplitude of the transmitted signal decreases as the bandwidth of the transparency window gets smaller and an increasing part of the signal is reflected. The time-domain measurement results with the QiController are also confirmed by numerical simulations and spectroscopic measurements (not shown here, see [152]) which are in good quantitative agreement.

The speed of light could be reduced by up to a factor of 1500 in pulsed measurements, corresponding to a delay of 12 ns. Larger delays are not possible with pulsed measurements as then, the bandwidth of the transparency window becomes smaller than the bandwidth of the Gaussian pulse being $1/\sigma = 20\,\mathrm{MHz}$. In spectroscopic measurements, higher delays of up to 15 ns could be observed, corresponding to a group index of $n_g = 1900$. Substantially larger group indices can be achieved when reducing the decoherence rate of the second excited state $|2\rangle$ of the qubits. This can be implemented by using another type of qubits instead of Transmons, like Fluxonium qubits which have a better suited, Λ-type energy level structure. [152]

To summarize, the flexibility of the QiController made it possible to use it as experiment instrument to observe slow light in an eight Transmon qubit metamaterial. Due to its versatile usability, the reduction of the speed of light could be measured. The validity of the obtained values could be verified as these are in quantitative agreement with spectroscopic measurement results. The QiController thereby supported the research for future applications in quantum computing where an improved version of the metamaterial might be used as on demand storage-and-retrieval memory.

5.4 Simultaneous, Time-Resolved Multi-Qubit Characterizations

5.4.1 Motivation and Physics Background

When building a quantum processor, the properties of all qubits need to be stable and well known. Noise sources inside the chip can cause undesired parameter fluctuations leading to a decreased fidelity in both control and readout. Fluctuations occurring on time scales of hours and days can be mitigated by frequent recalibration of the parameter values but this becomes increasingly difficult and time consuming

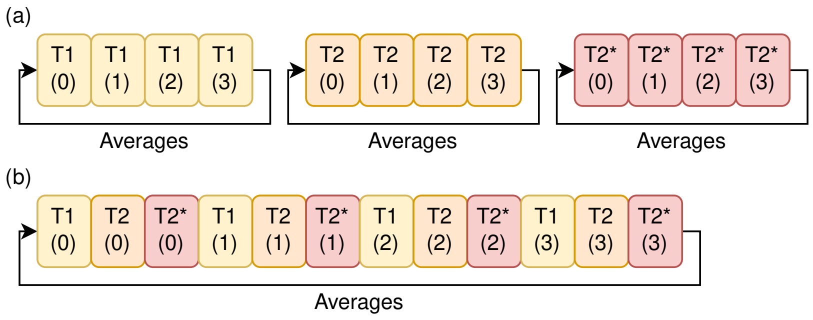


Figure 5.9: Concept of interleaving multiple experiments, in this case a T1 measurement, a Ramsey experiment (T2), and a spin echo experiment (T2*). Each block stands for a single qubit measurement with fixed parameters. The number in brackets indicates the parameter variations. (a) Typical measurement procedure where the experiments are executed one after each other and separately averaged. (b) Interleaved execution of the three experiments where single measurements of the experiments happen one after each other. Repetitions and averaging are only performed after all parameter points for all experiments are measured.

with growing qubit count [156]. Furthermore, to build a large scale quantum computer, it is important to analyze if noise and decoherence mechanisms are only local to single qubits, or if there is correlated noise between multiple qubits. Time-resolved long-term characterization measurements of the qubits can be performed in order to better understand the extent and mechanisms of such noise. By simultaneously measuring all relevant parameters of all qubits, they can be further correlated to extract information about the origin of the noise source and hence qubit decoherence.

Performing the different qubit characterization experiments after one another would make the results susceptible to slow-fluctuating noise. If such noise is dominant in the system, consecutive measurements of different qubit properties are not comparable anymore. Instead, the qubit parameters should be effectively measured at the same time. A way to achieve this is to interleave the different characterization measurements in a single experiment. Each characterization measurement consists of a one-dimensional parameter sweep where delays between pulses are varied. Instead of performing the complete sweep of one experiment and averaging it, followed by another experiment, the individual measurements within the sweeps of multiple experiments can be interleaved. This interleaving can be pictured as a time-division multiplexed scheme were single parameter points of the different characterization measurements alternate, as depicted in Figure 5.9.

For a single superconducting Transmon qubit, such an experiment was already performed by Schlör *et al.* [36]. The results of this experiment indicated that a small number of microscopic two-level systems located at the edges of the superconducting film are responsible for low-frequency burst noise in the qubit. Interleaving the experiment sequences is a complicated task and cumbersome to implement using common laboratory equipment like arbitrary waveform generators (AWGs) and digitizer cards. Therefore, follow-up measurements by the same authors, although not the data presented in the paper above, were taken with the QiController. This simplified the whole measurement process, from experiment creation to data collection.

To determine if the properties between pairs of qubits are correlated and thus influenced by a global effect, the correlation coefficient can be calculated. For two random number variables X and Y, the correlation coefficient is defined using their variance Var and covariance Cov as:

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)}\sqrt{\mathrm{Var}(Y)}} \quad . \tag{5.1}$$

The correlation coefficient can be estimated based on a sample of n values each, i.e. X_i and Y_i, $i \in 1, 2, \ldots, n$, by:

$$\mathrm{Corr}(X,Y) \approx \frac{\sum_{i=1}^{n}(X_i - \langle X\rangle)(Y_i - \langle Y\rangle)}{\sqrt{\sum_{i=1}^{n}(X_i - \langle X\rangle)^2 \sum_{i=1}^{n}(Y_i - \langle Y\rangle)^2}} \tag{5.2}$$

where $\langle X\rangle$ and $\langle Y\rangle$ are the expectation values of the random number variables X and Y, respectively. The value of the correlation coefficient can be between -1 and 1. Complete correlation is indicated by an absolute value of 1. If two variables are uncorrelated, the correlation coefficient will be zero. It thus gives an estimate how much two variables are correlated. In this experiment, the random variables are the properties of the different qubits.

5.4.2 Experiment Setup

The experiment is performed with a superconducting chip comprising six concentric Transmon qubits which was designed and fabricated at the National Institute of Standards and Technology (NIST) in Boulder, Colorado [134]. The qubits are not coupled among each other but only via separate readout resonators to a common microwave transmission line. Pictures of the sample are shown in Figure 5.10. Of the six qubits fabricated on the chip, five are fully functional and can be used for multi-qubit experiments. The qubits have no dedicated manipulation port. Therefore, manipulation pulses are fed into the chip using the same transmission line that is also utilized for the readout pulses. The whole chip is thus only

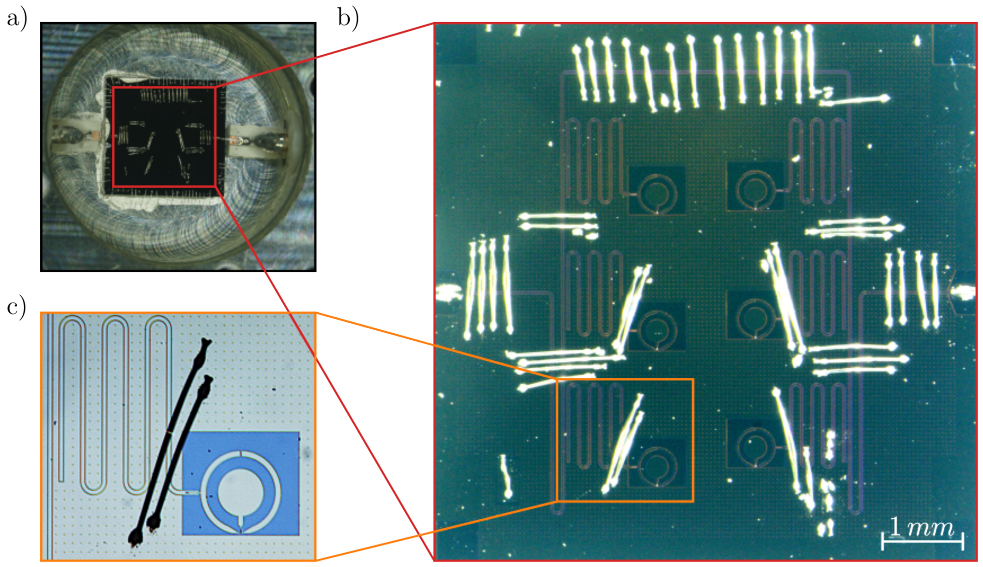


Figure 5.10: Photographs of the six qubit NIST sample. (a) Chip mounted inside a sample box with connected SMA pins. (b) Photograph of the whole chip. Additional bonds over the co-planar waveguide structures are visible to suppress parasitic modes. (c) Microscopic picture of a single concentric Transmon qubit with belonging readout resonator that is strongly capacitively coupled to the qubit. The pictures are taken from the master thesis of Maximilian Kristen [134].

connected to the room temperature electronics by two microwave lines: one input line for readout and manipulation pulses, and one output line for the returned readout signal. The baseband signals are directly combined by frequency-division multiplexing (FDM) within the QiController. After digital-to-analog conversion, they are separately up-converted using two analog unit cells before being merged into a single microwave line and fed into the cryostat. The local oscillator within the analog unit cell is located at 6.85 GHz for the readout signals and at 4.97 GHz for the control pulses. This is the combined value of the two-stage frequency conversion process described in Section 3.3, i.e. where a DC signal from the platform would hypothetically be shifted to in the frequency domain.

The properties of the five functioning qubits are given in Table 5.1. It also contains the intermediate frequencies of the complex-valued baseband generated within the digital unit cells of the QiController. The π-pulse durations t_π have been calibrated to a whole number of clock cycles by adjusting the relative amplitude of the pulses A_{01}. As the signals are created in the digital domain, special care has to be taken that the multiplexing of the signals does not lead to a value overflow. Therefore, all scaling factors summed up result in 100 % while individual ones are significantly smaller. Unfortunately, the qubit properties are not very competitive which makes it

Table 5.1: Properties of the five functioning Transmon qubits on the NIST chip.

	Qubit 1	Qubit 2	Qubit 3	Qubit 4	Qubit 5
f_r / GHz	6.877	6.930	6.979	7.028	7.080
f_{01} / GHz	4.813	5.084	4.710	4.933	4.901
$f_{\mathrm{IF,r}}$ / MHz	26.96	79.65	129.14	177.65	230.20
$f_{\mathrm{IF,01}}$ / MHz	−156.78	114.42	−259.73	−37.26	−69.49
t_π / ns	120	80	104	80	80
A_{01} / %	22.5	16.1	21.6	20.9	18.9
T_1 / µs	1.57	0.69	0.58	0.98	0.22
T_2 / µs	3.08	1.02	1.53	1.50	0.41

difficult to perform meaningful experiments, especially for the fifth qubit where T_1 is on the same order as the π-pulse time. The low qubit times can be attributed to parasitic slot line modes in the resonators which would require more and accurately placed on-chip bonds to suppress them [134].

The recording duration is chosen to be 400 ns with a readout pulse of 416 ns for each of the qubits. The SNR of the amplification chain was not good enough to reach single-shot readout, as the utilized TWPA, for unknown reasons, did not operate according to its specifications. Instead, the experiment executions needed to be averaged 4000 times to obtain results which could be later fitted to extract the qubit properties.

5.4.3 QiController Setup

Interleaved T_1, Ramsey and spin echo experiments are executed to obtain T_1, T_2, T_2^* and the qubit frequency f_{01}. The control pulses in the Ramsey experiment are deliberately 5 MHz detuned from the qubit transition frequency f_{01} to obtain Ramsey fringes. The frequency of the obtained oscillation matches the detuning and can thus be used to precisely determine f_{01}. 22 delay values are measured for the T_1 experiment which are logarithmically spaced between 8 ns and 20 µs. The same values are also used for the spin echo experiment, as both are expected to show an exponential decay with respect to the delay. For the Ramsey experiment, oscillations need to be detected. Therefore, substantially more points are used: 75 delay values are linearly spaced between zero and 3 µs.

The Taskrunner is leveraged to perform the parameter variations and interleave the experiments. External variables are defined to select the experiment to perform and

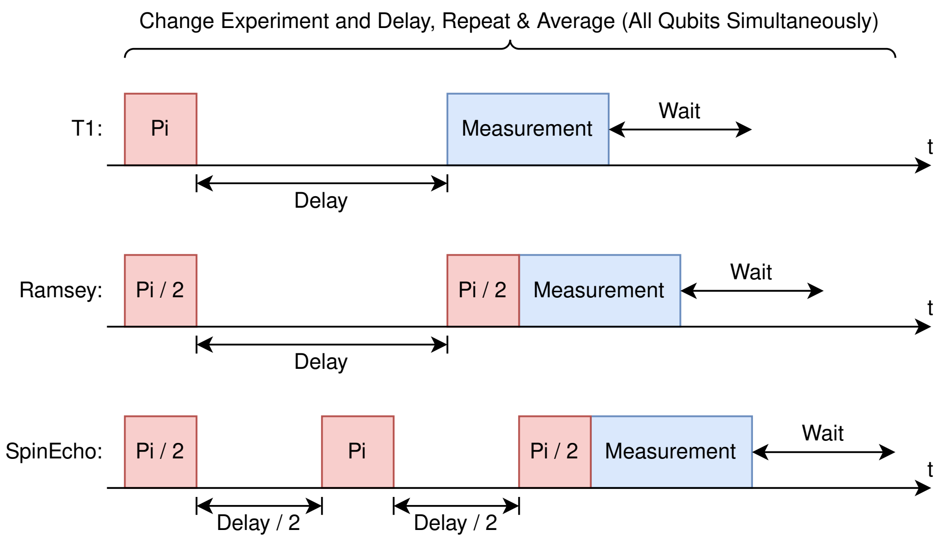


Figure 5.11: Sequences for T_1, T_2 (Ramsey), and T_2^* (spin echo) measurements. By changing between the different experiment sequences before varying the delays for individual ones, the measurements can be performed interleaved.

the delay to use. They are updated in the sequencer registers before the Taskrunner starts the execution. The QiJob describing all three experiments looks like:

Code 5.5: Interleaved experiment execution together with special Taskrunner application.

```
with QiJob() as interleaved:
    delay = QiVariable(name="delay")
    expsel = QiVariable(name="experiment")
    delay2 = QiVariable()
    Assign(delay2, delay >> 1)
    for q in QiCells(len(sample)):
        with If(expsel == 0): # T1 Experiment
            PiPulse(q)
            Wait(q, delay)
            Measurement(q)
        with If(expsel == 1): # Ramsey
            PiHalfPulse(q, detuning=-5e6)
            Wait(q, delay)
            PiHalfPulse(q, detuning=-5e6)
            Measurement(q)
        with If(expsel == 2): # SpinEcho
            PiHalfPulse(q)
            Wait(q, delay2)
            PiPulse(q, phase=numpy.pi/2)
            Wait(q, delay2)
            PiHalfPulse(q)
            Measurement(q)
        Thermalize(q)
```

It is also depicted in Figure 5.11. As only a single measurement is performed during the sequencer run, the averaging capability of sequencer and signal recorder are used to efficiently repeat the same sequence multiple times. The obtained averaged I and Q value is fetched by the task on the Taskrunner which also sorts the results by experiment and delay. The task furthermore changes the delay parameter and switches between the different experiments. The source code of the task is shown in Appendix C.2. The order in which the different experiments are performed and the delay values to use are passed as parameters to the task:

Code 5.6: Parameters and data converter for the Taskrunner application.

```python
def data_converter(databoxes):
    # Convert a single data box with alternating I/Q to amplitude and phase values
    def to_amp_pha(data):
        tmp = numpy.array(data[::2]) + 1j * numpy.array(data[1::2])
        return numpy.abs(tmp), numpy.angle(tmp)
    # Each qubit returns 3 databoxes, for t1, ramsey, and spinecho
    return [
        {
            "t1": to_amp_pha(data[3 * i + 0]),
            "ramsey": to_amp_pha(data[3 * i + 1]),
            "echo": to_amp_pha(data[3 * i + 2]),
        }
        for i in range(len(databoxes) // 3) # one dictionary per qubit
    ]

# Experiment parameters
delays_t1 = 8*np.logspace(-9, -5.6, 22) ## 22 logarithmically spaced values
delays_echo = delays_t1
delays_ramsey = np.arange(0, 3e-6, 40e-9) # 75 linearly spaced values
averages = 4000 # how many averages to perform per point
order = [1, 0, 1, 2, 1] # ramsey, t1, ramsey, echo, ramsey

# Prepare the parameters that are passed to the task
# (flatten converts the nested list to a 1D list)
task_parameters = flatten(
    [
        [3], # number of experiments (t1, ramsey, and echo)
        [len(order)], # how many measurements per loop
        [len(sample)], # number of qubits/cells
        _cell_map, # Which digital unit cells to use
        order, # In which order to interleave the experiments
        [len(delays_t1), len(delays_ramsey), len(delays_echo)],
        _nco_frequencies, # values to use as NCO frequency per qubit and experiment
        np.concatenate((delays_t1, delays_ramsey, delays_echo)),
    ]
)
# Variables starting with underscore need to be extracted from QiJob (not shown here)
```

Due to its complexity, the execution of this experiment is wrapped inside its own class, derived from the generic experiment class which is generated after compilation of the QiJob. The above code snippet only presents a condensed extract of the class content. Further values, like the cell map and the frequencies for the control signal generators which are adapted between the experiments, are additionally extracted from the compiled QiJob which is not shown here.

To make the task operation on the Taskrunner more tangible, a simplified pseudo-code for one qubit would look like this:

Code 5.7: Pseudo-code to emphasize the operation of the Taskrunner for one qubit.

```
result_databoxes = [[], [], []] # create data boxes for t1, ramsey, echo
number_of_executions = len(delays_t1) + len(delays_ramsey) + len(delays_echo)
for i in range(number_of_executions):
    # Select experiment and parameters
    exp = select_experiment_based_on_order_and_remaining_points(i)
    delay = get_experiment_delay_from_position(exp, i)
    frequency = get_nco_frequency_value_for_experiment(exp)
    # Update modules within FPGA
    set_nco_frequency_of_control_signal_generator(frequency)
    set_sequencer_register("experiment", exp)
    set_sequencer_register("delay", delay)
    # Start execution and persist result
    start_sequencer()
    wait_while_sequencer_is_busy()
    result_databoxes[exp].append(get_averaged_result())
    update_progress_value_for_user(i)
finish_databoxes_and_send_to_user(result_databoxes)
```

For the sake of completeness, interleaved measurements can also be performed without a special Taskrunner execution. However, this requires that the delays for all measurements are the same or can be easily calculated from within the sequencer. In this case, the time-division multiplexing can be achieved with a single for loop within the sequencer which then takes care of all parameter variations internally:

Code 5.8: Interleaved experiment execution with same delays for T_1, Ramsey, and spin echo.

```
with QiJob() as job:
    delay = QiVariable()
    delay2 = QiVariable()
    with ForRange(delay, 0, 4e-6, 200e-9):
        Assign(delay2, delay >> 1) # Divide by two
        for q in QiCells(len(sample)):
            # T1 Experiment
            PiPulse(q)
            Wait(q, delay)
            Measurement(q, save_to="t1")
            Thermalize(q)
            # Ramsey
            PiHalfPulse(q, detuning=-5e6)
            Wait(q, delay)
            PiHalfPulse(q, detuning=-5e6)
            Measurement(q, save_to="ramsey")
            Thermalize(q)
            # SpinEcho
            PiHalfPulse(q)
            Wait(q, delay2)
            PiPulse(q, phase=numpy.pi/2)
            Wait(q, delay2)
            PiHalfPulse(q)
        Measurement(q, save_to="spinecho")
        Thermalize(q)
```

In this case, the data is sequentially written into the data storage memory and collected by the default Taskrunner task without any adaptations needed. Yet, in most cases, the different decay times require different time scales to be recorded and thus the more flexible, but also more complex QiJob of Code 5.5 including custom Taskrunner operation is used for this experiment and the results discussed in the following.

5.4.4 Results and Discussion

Repetitive multi-qubit characterizations have been performed for more than 25 h. Each repetition took (5.15 ± 0.03) s and 17 600 repetitions have been performed during this time. Hence, each single experiment point execution took on average 10.8 µs, dominated by the energy relaxation time of the first qubit which is waited five times (7.85 µs) and the pulse sequence itself. As the standard deviations of the obtained qubit properties have been too large to extract any meaningful correlations, the raw data of five repetitions each has been additionally averaged. This reduced the number of time points to 3520 with (25.73 ± 0.10) s measurement duration each.

The time-resolved properties of the first qubit, as resulting from the fits to the measured data, are shown in Figure 5.12. Similar measurement results have also been extracted for the other qubits. The time-resolved T_2 values of all qubits are shown in Figure 5.13. While the experiment data shows no systematic deviations within the first 22 hours, a clear drop in qubit properties and shift of its resonance frequency is visible near the end of the measurement. The shift is clearly correlated between the different characterization measurements and thus confirms that the measurement scheme is working as expected. The change was induced by unintended, external heating inside the cryostat which substantially worsened the qubit properties and also led to a reduced SNR of the readout signal. It is therefore excluded from the following evaluations and analysis, as indicated by the red dashed line in Figure 5.12.

From the remaining data, the mean characteristic times for the first qubit can be determined as $T_1 = (1.49 \pm 0.07)$ µs, $T_2 = (2.91 \pm 0.17)$ µs and $T_2^* = (2.9 \pm 0.4)$ µs. As T_2 and T_2^* are quite similar, this suggests that phase noise on the timescale of T_2, i.e. on the order of megahertz and slower, does not play an important role in the qubit decoherence. Indeed, the pure dephasing time can be calculated as $T_\varphi = 113$ µs using Equation 2.4. This is two orders of magnitude larger than the energy relaxation time T_1 indicating that phase noise is not dominant in the decoherence dynamics of

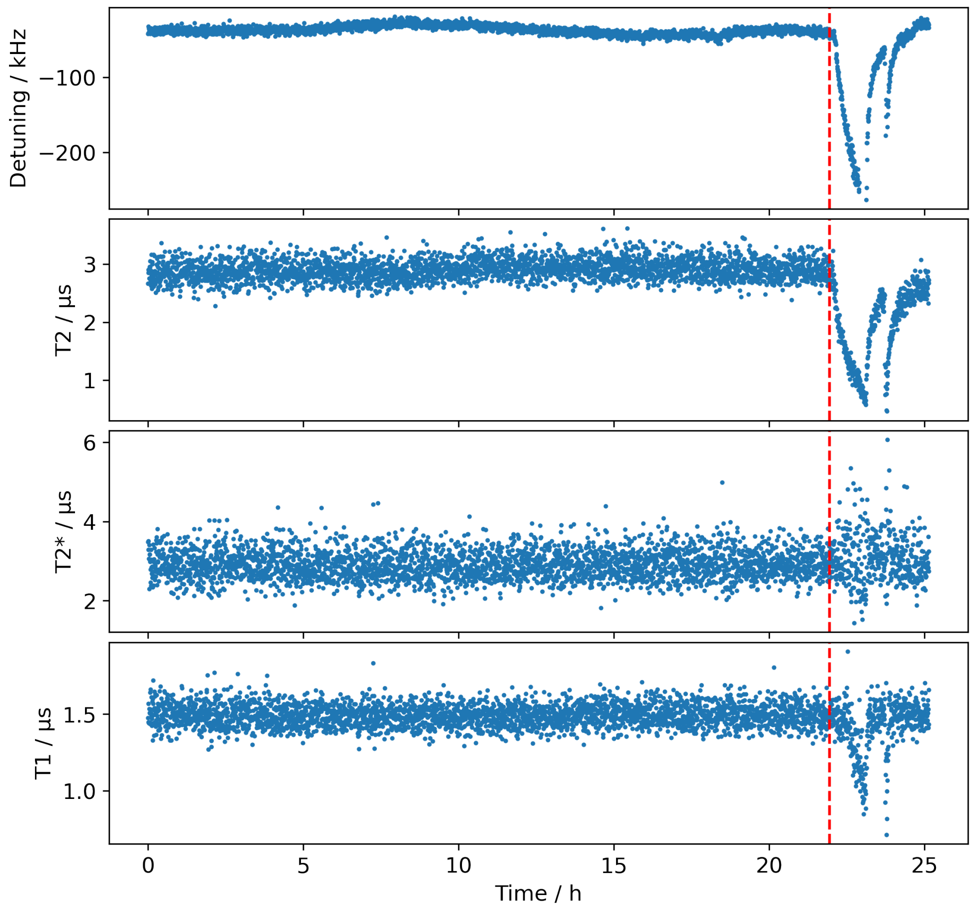


Figure 5.12: Time-resolved properties of the first qubit over more than 25 h. The red dashed line separates the initial normal phase from a phase where unintended heating inside the cryostat occurred.

the first qubit at all. For the other qubits, the situation looks different, as can be seen in Table 5.2. While still being larger than T_1, the pure dephasing is within the same order of magnitude for these qubits. Although not during this experiment, it even happened that qubit 2 and 5 could not be resolved at all using pulsed measurements, indicating that these are also more prone to fluctuating phase noise. During the interleaved measurement, the values also fluctuated as is indicated by the standard deviations of the qubit properties.

When correlating the qubit properties over time between the individual qubits, no clear correlation is visible. The correlations between the decoherence time T_2 of pairs of qubits are shown in Figure 5.14. However, time correlations happening on

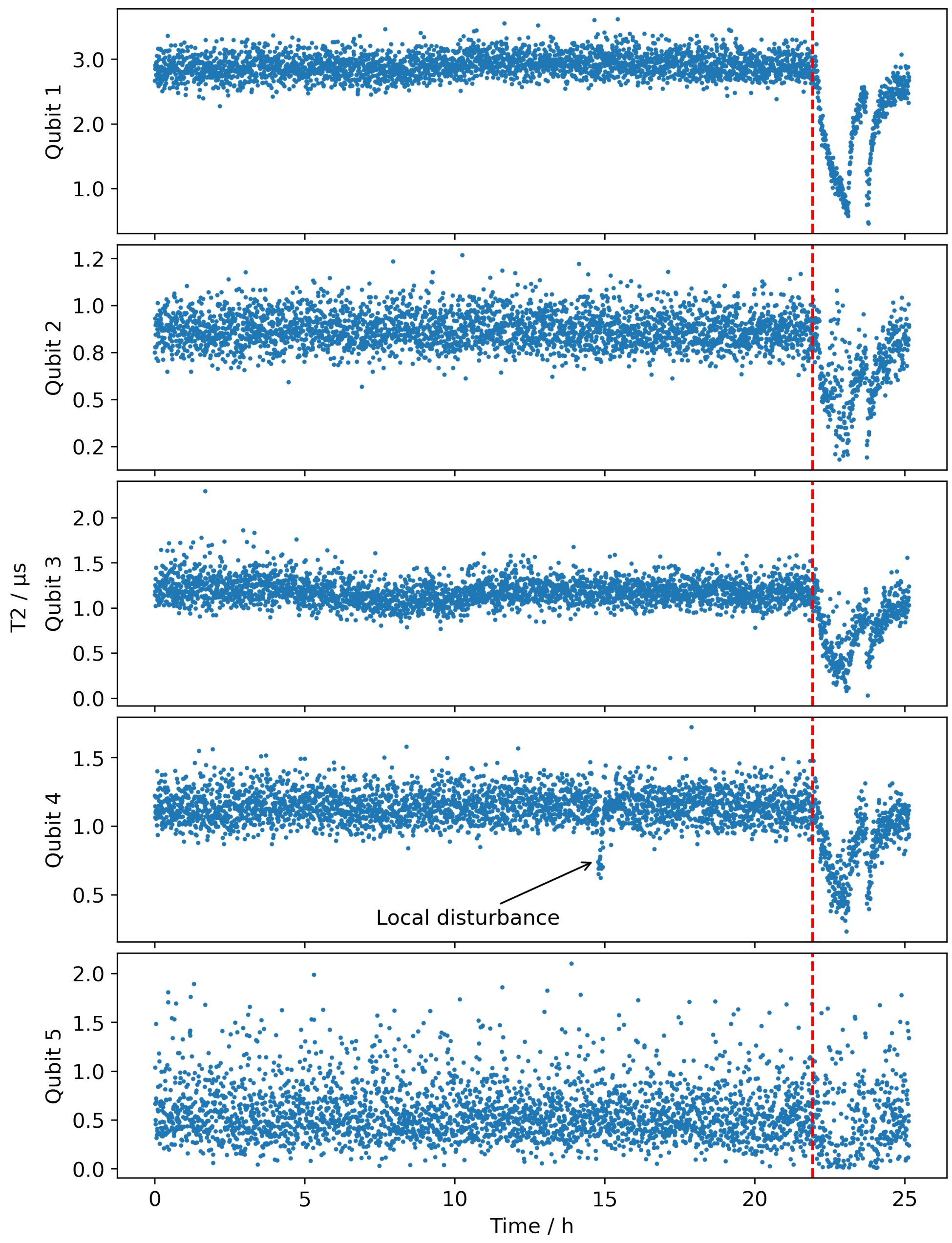


Figure 5.13: Time-resolved T_2 values of all qubits over more than 25 h. The red dashed line separates the initial normal phase from a phase where unintended heating inside the cryostat occurred. Qubit 4 furthermore experienced a sudden and short drop in its coherence which is also annotated in the plot.

Table 5.2: Mean properties of the five functioning Transmon qubits on the NIST chip during the interleaved measurement, excluding the phase of accidental heating.

	Qubit 1	Qubit 2	Qubit 3	Qubit 4	Qubit 5
T_1 / µs	1.49 ± 0.07	0.63 ± 0.05	0.73 ± 0.06	0.78 ± 0.08	0.48 ± 0.16
T_2 / µs	2.91 ± 0.17	0.88 ± 0.09	1.18 ± 0.14	1.15 ± 0.12	0.6 ± 0.3
T_2^* / µs	2.9 ± 0.4	1.2 ± 0.2	1.2 ± 0.2	1.5 ± 0.4	0.7 ± 0.4
T_φ / µs	114	2.90	6.3	4.35	1.47

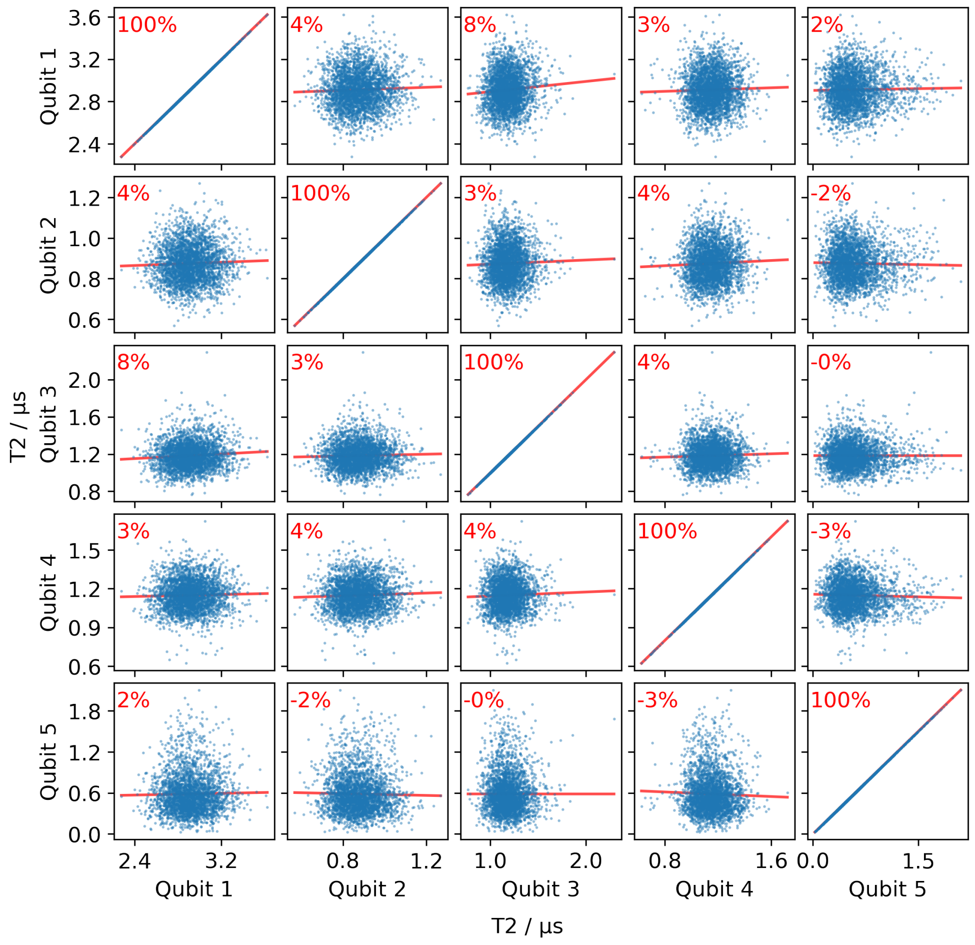


Figure 5.14: Inter-qubit correlations of the time-resolved decoherence time T_2. The red value indicates the correlation coefficient for the data of these two qubits. The red line in the background is a linear fit through the data to visualize potential correlations.

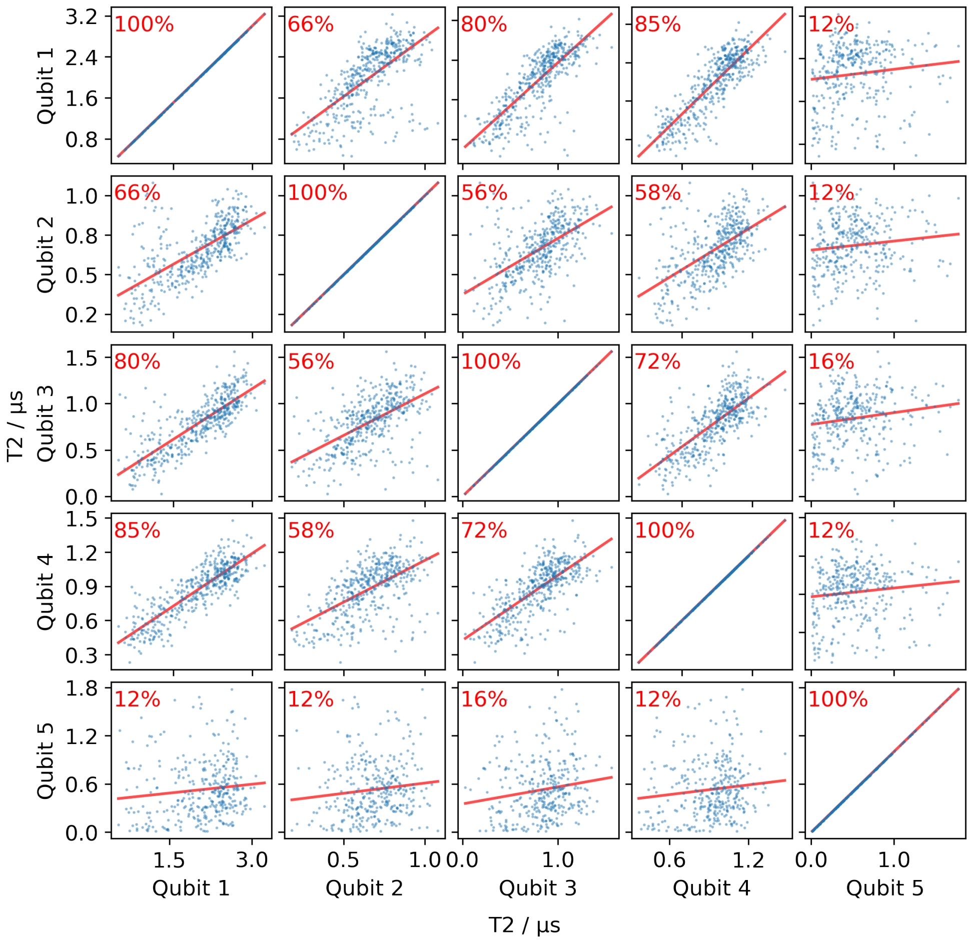


Figure 5.15: Inter-qubit correlations of the time-resolved decoherence time T_2 during the accidental heating period. The red value indicates the correlation coefficient for the data of these two qubits. The red line in the background is a linear fit through the data to visualize the correlations.

a time scale shorter than 25 s could not be resolved due to the lack in SNR of the experiment setup.

To verify that correlations on larger time scales are detected by the platform, the time window of the accidental heating can be inspected. The results are shown in Figure 5.15 for the decoherence time T_2. A clear correlation between all properties, with expect of the fifth qubit, is visible. This is intuitively clear as the heating is a global disturbance which will decrease the quality of all qubits. The decoherence time of the fifth qubit only shows a weak correlation to the other ones. Judging from the data, this is due to the already low decoherence time of the qubit even without the additional disturbance.

The absence of clear correlations in the normal data suggests that no global disturbance substantially affects the qubit properties. Instead, it seems that the decoherence properties of the qubit are dominated by local effects, at least on time scales longer than 25 s. This hypothesis is also supported by the data of qubit 4 which shows a sudden and short drop in qubit frequency and decoherence times after around 15 h which could not be observed in the other qubits (compare Figure 5.13). Further effects might become visible on a shorter timescale. This would require optimizations of the SNR of the amplifier chain, as well as an improved qubit chip with better decoherence properties.

Concerning the QiController, such experiments with time resolutions below one second would be possible without any adaptations needed to the platform, except for adapting the experiment parameters. However, the obtained data already verifies that the operation is working as expected and meaningful data can be recorded with the QiController. Experiments performed with other qubit samples and setups furthermore corroborate that the SNR is not limited by the platform, but by the experiment setup within the cryostat.

To conclude, multi-qubit characterization measurements can be successfully performed using the QiController in a simultaneous and time-resolved manner with large flexibility. While single pulse sequences have been on the order of microseconds, the time-resolved characterizations were executed for over 25 h. Data processing, aggregation and transfer to the client was completely handled online during the experiment execution on the QiController. The experiment furthermore demonstrated correct operation of a frequency-division multiplexed control and readout scheme with the platform, and scalable operation with multiple digital unit cells involved.

5.5 Summary

The QiController can be used for a wide variety of experiments. It supports fast feedback operations with intrinsic latency of 352 ns. Qubit states can be recorded continuously and extracted online, corresponding to tremendous data reductions. Using the Taskrunner, results can be collected and sent to the user in parallel to the recording, enabling continuous measurements over very long time periods. It also enables parameter variations and online data processing on the platform, decoupled from the user client. Due to the large flexibility of the QiController, not only gate-based operations can be performed, but also more fundamental research can be conducted. As an example, Gaussian-shaped pulses can be generated and recorded in order to extract their propagation speed through a test sample.

Due to its flexibility, the QiController supports a large array of superconducting circuits. Each of the four exemplary experiments in this chapter uses a different qubit chip. Efficient and reliable qubit state initialization is demonstrated with a fidelity exceeding 96.9 %, an essential requirement for quantum computing. Quantum jumps are observed by continuously measuring the qubit state. 32 million states are recorded in only 12.8 s. Time-resolved long-term characterization measurements with a five qubit chip are performed for more than a day. The measurements indicate that the dominant noise sources are local to the individual qubits, as no correlations between them could be observed. Heating the sample leads to a correlated reduction of the qubit coherences as expected, thereby verifying the measurement scheme. A quantum metamaterial is studied as potential candidate for a universal quantum memory, based on a controlled reduction of the speed of light through this material.

All presented experiments are facilitated by the flexible and modular architecture of the QiController, as well as its unique features and the versatile QiCode experiment description language. Thereby, experiments which are unfeasible or even impossible to implement with commercial laboratory equipment can be performed with the platform. This is corroborated by the fact that the QiController is also actively used by other researchers at Karlsruhe Institute of Technology and abroad.

6 Conclusion

Quantum computing is a groundbreaking technology. It promises to drastically outperform conventional computers for a variety of computational problems, including optimization problems, as well as quantum simulations and encryption. To build a universal quantum computer, the full hardware and software stack need to be considered. Sophisticated control electronics is a crucial component of a quantum computer to bridge the gap between quantum processor and quantum algorithm. For superconducting quantum bits (qubits), the presented QiController provides a versatile, scalable, and integrated control electronics platform. It thereby constitutes an important step towards a universal quantum computer.

The platform enables users to perform experiments which are unfeasible or even impossible to handle using generic laboratory equipment. It consists of hardware, firmware, and software. The hardware is based on the radio frequency system-on-chip (RFSoC) architecture which combines eight 14 bit DACs and 12 bit ADCs, all operating at 4 GSPS, with a field-programmable gate array (FPGA) and multiple processors. The hardware also contains a custom-developed RF electronics to reach the required frequency domain for superconducting qubits of up to 9 GHz. All components are closely interconnected leading to a tight integration and data exchange. On the application processor of the RFSoC, a modular communication server, called ServiceHub, provides access to all components of the system via Ethernet. The real-time processor hosts the presented Taskrunner framework to execute user-created tasks facilitating complex control schemes and online data processing with minimal overhead directly on the platform. The FPGA firmware consists of reusable, so-called digital unit cells. They contain all logic to control a single qubit, including signal generation and recording, as well as cycle-accurate sequencing of operations. Limited by the resources available in the FPGA, up to 15 qubits can be controlled by a single RFSoC. For larger qubit numbers, a scheme to connect and synchronize multiple platforms has been conceived.

A high-level experiment description language called QiCode was designed and implemented. It enables users to functionally and intuitively describe the behavior of the QiController using a simple yet powerful Python-based language. Thereby, it drastically reduces the entry barrier to utilize the unique capabilities of the platform as no expert knowledge of the system is required. An integration into IBM Qiskit is provided to leverage its rich ecosystem. With it, quantum algorithms written in Qiskit can be executed on actual quantum processors controlled by the platform.

The QiController is actively used by other researchers. Currently, multiple platforms are in operation at the Institute of Physics and the Institute for Quantum Materials and Technologies at the Karlsruhe Institute of Technology, as well as at the James Watt School of Engineering at the University of Glasgow. The platform was extensively tested in various experiments to demonstrate and verify its capabilities.

High-precision online state preparation has been performed, demonstrating that advanced control schemes required in quantum computing can be executed. Quantum jumps of qubits have been continuously recorded with an enormous online data reduction by factors of roughly 40 000. The speed of light inside a quantum metamaterial has been observed, yielding potential applications as quantum memory. Performing time-resolved multi-qubit characterizations demonstrated the scalability of the QiController and successful operation of a frequency-division-multiplexed control and readout scheme. While single pulse sequences are on the order of microseconds, the time-resolved characterizations were executed for more than a day. Data processing, aggregation and transfer to the client is handled online during the experiment execution. This is only possible due to the flexible and modular architecture of the platform, especially the digital unit cells, Taskrunner, ServiceHub, and Python client.

With its unique features, the QiController fulfills all prerequisites to be quickly distributed and employed in research groups around the world working with superconducting qubits. Future improvements of the platform could extend the system for even larger qubit numbers by fully integrating multi-platform synchronization and data exchange. Moreover, due to its high flexibility, the platform is perfectly suited as test bed to evaluate requirements for future, large-scale quantum processors and their control electronics.

Bibliography

[1] J. Preskill, “Quantum Computing in the NISQ era and beyond”, *Quantum*, vol. 2, no. 79, 2018.

[2] B. Schumacher, “Quantum coding”, *Phys. Rev. A*, vol. 51, pp. 2738–2747, 4 Apr. 1995.

[3] D. R. Simon, “On the power of quantum computation”, in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 116–123.

[4] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”, *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[5] E. Gerjuoy, “Shor’s factoring algorithm and modern cryptography. an illustration of the capabilities inherent in quantum computers”, *American Journal of Physics*, vol. 73, no. 6, pp. 521–540, 2005.

[6] H.-L. Huang, Y.-W. Zhao, T. Li, F.-G. Li, Y.-T. Du, X.-Q. Fu, S. Zhang, X. Wang, and W.-S. Bao, “Homomorphic encryption experiments on ibm’s cloud quantum computing platform”, *Frontiers of Physics*, vol. 12, no. 1, p. 120 305, Dec. 2016, ISSN: 2095-0470.

[7] L. K. Grover, “A fast quantum mechanical algorithm for database search”, in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC ’96, Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 212–219.

[8] R. P. Feynman, “Simulating Physics with Computers”, *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, Jun. 1982.

[9] F. Gaitan, “Finding flows of a navier–stokes fluid through quantum computing”, *npj Quantum Information*, vol. 6, no. 1, p. 61, Jul. 2020, ISSN: 2056-6387.

[10] A. Perdomo-Ortiz, N. Dickson, M. Drew-Brook, G. Rose, and A. Aspuru-Guzik, “Finding low-energy conformations of lattice protein models by quantum annealing”, *Scientific Reports*, vol. 2, no. 1, p. 571, Aug. 2012, ISSN: 2045-2322.

[11] Y. Cao, J. Romero, and A. Aspuru-Guzik, "Potential of quantum computing for drug discovery", *IBM Journal of Research and Development*, vol. 62, no. 6, 6:1–6:20, 2018.

[12] B. Bauer, S. Bravyi, M. Motta, and G. Kin-Lic Chan, "Quantum algorithms for quantum chemistry and quantum materials science", *Chemical Reviews*, vol. 120, no. 22, pp. 12 685–12 717, Nov. 2020, ISSN: 0009-2665.

[13] R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. L. Heras, R. Babbush, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, D. Sank, A. Vainsencher, J. Wenner, T. C. White, E. Solano, H. Neven, and J. M. Martinis, "Digitized adiabatic quantum computing with a superconducting circuit", *Nature*, vol. 534, no. 7606, pp. 222–226, Jun. 2016, ISSN: 1476-4687.

[14] V. Dunjko and H. J. Briegel, "Machine learning & artificial intelligence in the quantum domain: A review of recent progress", *Reports on Progress in Physics*, vol. 81, no. 7, p. 074 001, Jun. 2018.

[15] A. Steane, "The ion trap quantum information processor", *Applied Physics B*, vol. 64, no. 6, pp. 623–643, Jun. 1997.

[16] T. Monz, P. Schindler, J. T. Barreiro, M. Chwalla, D. Nigg, W. A. Coish, M. Harlander, W. Hänsel, M. Hennrich, and R. Blatt, "14-qubit entanglement: Creation and coherence", *Phys. Rev. Lett.*, vol. 106, p. 130 506, 13 Mar. 2011.

[17] G. J. Milburn, "Quantum optical fredkin gate", *Phys. Rev. Lett.*, vol. 62, pp. 2124–2127, 18 May 1989.

[18] ——, "Photons as qubits", *Physica Scripta*, vol. 2009, no. T137, p. 014 003, 2009.

[19] J. L. O'Brien, "Optical quantum computing", *Science*, vol. 318, no. 5856, pp. 1567–1570, 2007.

[20] D. Loss and D. P. DiVincenzo, "Quantum computation with quantum dots", *Phys. Rev. A*, vol. 57, pp. 120–126, 1 Jan. 1998.

[21] S. Li, J. Xia, J. Liu, F. Yang, Z. Niu, S. Feng, and H. Zheng, "Inas/gaas single-electron quantum dot qubit", *Journal of Applied Physics*, vol. 90, no. 12, pp. 6151–6155, 2001.

[22] M. Veldhorst, J. C. C. Hwang, C. H. Yang, A. W. Leenstra, B. de Ronde, J. P. Dehollain, J. T. Muhonen, F. E. Hudson, K. M. Itoh, A. Morello, and A. S. Dzurak, "An addressable quantum dot qubit with fault-tolerant control-fidelity", *Nature Nanotechnology*, vol. 9, pp. 981–985, 2014.

[23] V. Bouchiat, D. Vion, P. Joyez, D. Esteve, and M. H. Devoret, “Quantum coherence with a single cooper pair”, *Physica Scripta*, vol. 1998, no. T76, p. 165, 1998.

[24] J. Koch, T. M. Yu, J. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, “Charge-insensitive qubit design derived from the cooper pair box”, *Phys. Rev. A*, vol. 76, p. 042 319, 4 Oct. 2007.

[25] V. E. Manucharyan, J. Koch, L. I. Glazman, and M. H. Devoret, “Fluxonium: Single cooper-pair circuit free of charge offsets”, *Science*, vol. 326, no. 5949, pp. 113–116, 2009.

[26] C. Rigetti, J. M. Gambetta, S. Poletto, B. L. T. Plourde, J. M. Chow, A. D. Córcoles, J. A. Smolin, S. T. Merkel, J. R. Rozen, G. A. Keefe, M. B. Rothwell, M. B. Ketchen, and M. Steffen, “Superconducting qubit in a waveguide cavity with a coherence time approaching 0.1 ms”, *Phys. Rev. B*, vol. 86, p. 100 506, 10 Sep. 2012.

[27] D. Castelvecchi, “Quantum computers ready to leap out of the lab in 2017”, *Nature News*, vol. 541, pp. 9–10, 2017.

[28] J. Kelly. “A preview of bristlecone, google’s new quantum processor”, Google. (Mar. 5, 2018), [Online]. Available: `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html` (visited on 11/29/2021).

[29] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, “Quantum supremacy using a programmable superconducting processor”, *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019, ISSN: 1476-4687.

[30] M. H. Devoret and R. J. Schoelkopf, “Superconducting circuits for quantum information: An outlook”, *Science*, vol. 339, no. 6124, pp. 1169–1174, 2013.

[31] M. H. Devoret, A. Wallraff, and J. M. Martinis, “Superconducting qubits: A short review”, Dec. 2004.

[32] G. Catelani, S. E. Nigg, S. M. Girvin, R. J. Schoelkopf, and L. I. Glazman, “Decoherence of superconducting qubits caused by quasiparticle tunneling”, *Phys. Rev. B*, vol. 86, p. 184 514, 18 Nov. 2012.

[33] E. Knill, “Quantum computing with realistically noisy devices”, *Nature*, vol. 434, pp. 39–44, Apr. 2005.

[34] I. Pogorelov, T. Feldker, C. D. Marciniak, L. Postler, G. Jacob, O. Krieglsteiner, V. Podlesnic, M. Meth, V. Negnevitsky, M. Stadler, B. Höfer, C. Wächter, K. Lakhmanskiy, R. Blatt, P. Schindler, and T. Monz, “Compact ion-trap quantum computing demonstrator”, *PRX Quantum*, vol. 2, p. 020 343, 2 Jun. 2021.

[35] L. Grünhaupt, M. Spiecker, D. Gusenkova, N. Maleeva, S. T. Skacel, I. Takmakov, F. Valenti, P. Winkel, H. Rotzinger, W. Wernsdorfer, A. V. Ustinov, and I. M. Pop, “Granular aluminium as a superconducting material for high-impedance quantum circuits”, *Nature Materials*, p. 1, 2019.

[36] S. Schlör, J. Lisenfeld, C. Müller, A. Bilmes, A. Schneider, D. P. Pappas, A. V. Ustinov, and M. Weides, “Correlating decoherence in transmon qubits: Low frequency noise by single fluctuators”, *Phys. Rev. Lett.*, vol. 123, p. 190 502, 19 Nov. 2019.

[37] A. Blais, R. Huang, A. Wallraff, S. M. Girvin, and R. J. Schoelkopf, “Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation”, *Phys. Rev. A*, vol. 69, p. 062 320, 6 Jun. 2004.

[38] D. Gusenkova, M. Spiecker, R. Gebauer, M. Willsch, D. Willsch, F. Valenti, N. Karcher, L. Grünhaupt, I. Takmakov, P. Winkel, D. Rieger, A. V. Ustinov, N. Roch, W. Wernsdorfer, K. Michielsen, O. Sander, and I. M. Pop, “Quantum nondemolition dispersive readout of a superconducting artificial atom using large photon numbers”, *Phys. Rev. Applied*, vol. 15, p. 064 030, 6 Jun. 2021.

[39] M. Werninghaus, D. J. Egger, F. Roy, S. Machnes, F. K. Wilhelm, and S. Filipp, “Leakage reduction in fast superconducting qubit gates via optimal control”, *npj Quantum Information*, vol. 7, no. 1, p. 14, Jan. 2021.

[40] S. Kono, K. Koshino, D. Lachance-Quirion, A. F. van Loo, Y. Tabuchi, A. Noguchi, and Y. Nakamura, “Breaking the trade-off between fast control and long lifetime of a superconducting qubit”, *Nature Communications*, vol. 11, no. 1, p. 3683, Jul. 2020, ISSN: 2041-1723.

[41] J. Braumüller, M. Sandberg, M. R. Vissers, A. Schneider, S. Schlör, L. Grünhaupt, H. Rotzinger, M. Marthaler, A. Lukashenko, A. Dieter, A. V. Ustinov, M. Weides, and D. P. Pappas, "Concentric transmon qubit featuring fast tunability and an anisotropic magnetic dipole moment", *Applied Physics Letters*, vol. 108, no. 3, p. 032 601, 2016.

[42] L. DiCarlo, J. M. Chow, J. M. Gambetta, L. S. Bishop, B. R. Johnson, D. I. Schuster, J. Majer, A. Blais, L. Frunzio, S. M. Girvin, and R. J. Schoelkopf, "Demonstration of two-qubit algorithms with a superconducting quantum processor", *Nature*, vol. 460, no. 7252, pp. 240–244, Jul. 2009, ISSN: 1476-4687.

[43] R. Gebauer, "FPGA-based quantum feedback for superconducting qubits", M.S. thesis, Karlsruhe Institute of Technology, Karlsruhe, 2018.

[44] M. Pfirrmann, I. Boventer, A. Schneider, T. Wolz, M. Kläui, A. V. Ustinov, and M. Weides, "Magnons at low excitations: Observation of incoherent coupling to a bath of two-level systems", *Phys. Rev. Research*, vol. 1, p. 032 023, 3 Nov. 2019.

[45] C. Lang, C. Eichler, L. Steffen, J. M. Fink, M. J. Woolley, A. Blais, and A. Wallraff, "Correlations, indistinguishability and entanglement in hong–ou–mandel experiments at microwave frequencies", *Nature Physics*, vol. 9, no. 6, pp. 345–348, Jun. 2013, ISSN: 1745-2481.

[46] R. Gebauer, N. Karcher, D. Gusenkova, M. Spiecker, L. Grünhaupt, I. Takmakov, P. Winkel, L. Planat, N. Roch, W. Wernsdorfer, A. V. Ustinov, M. Weber, M. Weides, I. M. Pop, and O. Sander, "State preparation of a fluxonium qubit with feedback from a custom FPGA-based platform", *AIP Conference Proceedings*, vol. 2241, no. 1, p. 020 015, 2020.

[47] R. Gebauer, N. Karcher, J. Hurst, M. Weber, and O. Sander, "Taskrunner: A flexible framework optimized for low latency quantum computing experiments", in *2021 IEEE 34th International System-on-Chip Conference (SOCC)*, 2021.

[48] D. Ristè, C. C. Bultink, K. W. Lehnert, and L. DiCarlo, "Feedback control of a solid-state qubit using high-fidelity projective measurement", *Phys. Rev. Lett.*, vol. 109, p. 240 502, 24 Dec. 2012.

[49] N. Ofek, A. Petrenko, R. Heeres, P. Reinhold, Z. Leghtas, B. Vlastakis, Y. Liu, L. Frunzio, S. M. Girvin, L. Jiang, M. Mirrahimi, M. H. Devoret, and R. J. Schoelkopf, "Extending the lifetime of a quantum bit with error correction in superconducting circuits", *Nature*, vol. 536, no. 7617, pp. 441–445, Aug. 2016, ISSN: 1476-4687.

[50] C. K. Andersen, A. Remm, S. Lazar, S. Krinner, J. Heinsoo, J.-C. Besse, M. Gabureac, A. Wallraff, and C. Eichler, “Entanglement stabilization using ancilla-based parity detection and real-time feedback in superconducting circuits”, *npj Quantum Information*, vol. 5, no. 1, pp. 1–7, 2019.

[51] M. Möller and C. Vuik, “On the impact of quantum computing technology on future developments in high-performance scientific computing”, *Ethics and Information Technology*, vol. 19, no. 4, pp. 253–269, Dec. 2017, ISSN: 1572-8439.

[52] K. A. Britt, F. A. Mohiyaddin, and T. S. Humble, “Quantum accelerators for high-performance computing systems”, in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 2017, pp. 1–7.

[53] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782.

[54] D. P. DiVincenzo, “The Physical Implementation of Quantum Computation”, *Fortschritte der Physik*, vol. 48, no. 9-11, pp. 771–783, Jan. 2000.

[55] C. P. Williams, *Explorations in Quantum Computing*, 2nd. London: Springer-Verlag London Limited, 2011.

[56] J. A. Bergou and M. Hillery, *Introduction to the theory of quantum information processing*, ser. Graduate texts in physics. New York: Springer, 2013.

[57] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, “Elementary gates for quantum computation”, *Phys. Rev. A*, vol. 52, pp. 3457–3467, 5 Nov. 1995.

[58] J.-M. Normand, *A Lie group : rotations in quantum mechanics*. Amsterdam [u.a.]: North-Holland Publ. Co., 1980, Literaturangaben S. 417-473, ISBN: 0444861254.

[59] P. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan, “A new universal and fault-tolerant quantum basis”, *Information Processing Letters*, vol. 75, no. 3, pp. 101–107, 2000, ISSN: 0020-0190.

[60] N. Schuch and J. Siewert, “Natural two-qubit gate for quantum computation using the XY interaction”, *Phys. Rev. A*, vol. 67, p. 032 301, 3 Mar. 2003.

[61] D. C. McKay, S. Filipp, A. Mezzacapo, E. Magesan, J. M. Chow, and J. M. Gambetta, “Universal gate for fixed-frequency qubits via a tunable bus”, *Phys. Rev. Applied*, vol. 6, p. 064 007, 6 Dec. 2016.

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

[63] P. Wang, C.-Y. Luan, M. Qiao, M. Um, J. Zhang, Y. Wang, X. Yuan, M. Gu, J. Zhang, and K. Kim, "Single ion qubit with estimated coherence time exceeding one hour", *Nature Communications*, vol. 12, no. 1, p. 233, Jan. 2021, ISSN: 2041-1723.

[64] C. J. Ballance, T. P. Harty, N. M. Linke, M. A. Sepiol, and D. M. Lucas, "High-fidelity quantum logic gates using trapped-ion hyperfine qubits", *Phys. Rev. Lett.*, vol. 117, p. 060 504, 6 Aug. 2016.

[65] A. Laing, A. Peruzzo, A. Politi, M. R. Verde, M. Halder, T. C. Ralph, M. G. Thompson, and J. L. O'Brien, "High-fidelity operation of quantum photonic circuits", *Applied Physics Letters*, vol. 97, no. 21, p. 211 109, 2010.

[66] F. Flamini, N. Spagnolo, and F. Sciarrino, "Photonic quantum information processing: A review", *Reports on Progress in Physics*, vol. 82, no. 1, p. 016 001, Nov. 2018.

[67] The Quantum Daily, Ed. "A detailed review of qubit implementations for quantum computing". (May 21, 2020), [Online]. Available: `https://thequantumdaily.com/2020/05/21/tqd-exclusive-a-detailed-review-of-qubit-implementations-for-quantum-computing/` (visited on 10/07/2021).

[68] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned", *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982, ISSN: 1476-4687.

[69] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation", *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 439, no. 1907, pp. 553–558, 1992.

[70] "Deutsch-jozsa algorithm", Qiskit. (), [Online]. Available: `https://qiskit.org/textbook/ch-algorithms/deutsch-jozsa.html` (visited on 06/21/2021).

[71] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited", *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 339–354, 1998.

[72] G. Brassard, P. Høyer, M. Mosca, and A. Tapp, "Quantum amplitude amplification and estimation", in *Quantum computation and information (Washington, DC, 2000)*, ser. Contemp. Math. Vol. 305, Amer. Math. Soc., Providence, RI, 2002, pp. 53–74.

[73] M. Homeister, "Klassische Verschlüsselungen knacken: Primfaktorzerlegung", in *Quantum Computing verstehen: Grundlagen – Anwendungen – Perspektiven*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, pp. 193–236, ISBN: 978-3-658-22884-2.

[74] W. C. Huffman and V. Pless, *Fundamentals of error-correcting codes*. Cambridge University Press, 2003.

[75] A. Peres, "Reversible logic and quantum computers", *Phys. Rev. A*, vol. 32, pp. 3266–3276, 6 Dec. 1985.

[76] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory", *Phys. Rev. A*, vol. 52, R2493–R2496, 4 Oct. 1995.

[77] A. M. Steane, "Error correcting codes in quantum theory", *Phys. Rev. Lett.*, vol. 77, pp. 793–797, 5 Jul. 1996.

[78] D. Gottesman, "Stabilizer codes and quantum error correction", Ph.D. dissertation, California Institute of Technology, 1997.

[79] A. Kitaev, "Fault-tolerant quantum computation by anyons", *Annals of Physics*, vol. 303, no. 1, pp. 2–30, 2003, ISSN: 0003-4916.

[80] J. Ghosh, A. G. Fowler, and M. R. Geller, "Surface code with decoherence: An analysis of three superconducting architectures", *Phys. Rev. A*, vol. 86, p. 062 318, 6 Dec. 2012.

[81] H. Kamerlingh Onnes, "The resistance of pure mercury at helium temperatures", *Communications from the physical laboratory of the University of Leiden*, vol. 12, no. 120, p. 1, 1911.

[82] W. Meissner and R. Ochsenfeld, "Ein neuer effekt bei eintritt der supraleitfähigkeit", *The Science of Nature - Naturwissenschaften*, vol. 21, no. 44, pp. 787–788, 1933.

[83] J. Bardeen, L. N. Cooper, and J. R. Schrieffer, "Theory of superconductivity", *Phys. Rev.*, vol. 108, pp. 1175–1204, 5 Dec. 1957.

[84] V. V. Schmidt, *The Physics of Superconductors: Introduction to Fundamentals and Applications*, P. Müller and A. V. Ustinov, Eds. Berlin: Springer, 1997.

[85] B. D. Josephson, "Possible new effects in superconductive tunnelling", *Physics Letters*, vol. 1, pp. 251–253, Jul. 1962.

[86] M. D. Lukin, M. Fleischhauer, R. Cote, L. M. Duan, D. Jaksch, J. I. Cirac, and P. Zoller, "Dipole blockade and quantum information processing in mesoscopic atomic ensembles", *Phys. Rev. Lett.*, vol. 87, p. 037 901, 3 Jun. 2001.

[87] M. Saffman, T. G. Walker, and K. Mølmer, "Quantum information with rydberg atoms", *Rev. Mod. Phys.*, vol. 82, pp. 2313–2363, 3 Aug. 2010.

[88] O. Astafiev, Y. A. Pashkin, Y. Nakamura, T. Yamamoto, and J. S. Tsai, "Quantum noise in the josephson charge qubit", *Phys. Rev. Lett.*, vol. 93, p. 267 007, 26 Dec. 2004.

[89] N. Earnest, S. Chakram, Y. Lu, N. Irons, R. K. Naik, N. Leung, L. Ocola, D. A. Czaplewski, B. Baker, J. Lawrence, J. Koch, and D. I. Schuster, "Realization of a Λ system with metastable states of a capacitively shunted fluxonium", *Phys. Rev. Lett.*, vol. 120, p. 150 504, 15 Apr. 2018.

[90] A. Somoroff, Q. Ficheux, R. A. Mencia, H. Xiong, R. V. Kuzmin, and V. E. Manucharyan, *Millisecond coherence in a superconducting qubit*, 2021.

[91] I. M. Pop, K. Geerlings, G. Catelani, R. J. Schoelkopf, L. I. Glazman, and M. H. Devoret, "Coherent suppression of electromagnetic dissipation due to superconducting quasiparticles", *Nature*, vol. 508, no. 7496, pp. 369–372, Apr. 2014, ISSN: 1476-4687.

[92] B. W. Shore and P. L. Knight, "The Jaynes-Cummings model", *Journal of Modern Optics*, vol. 40, no. 7, pp. 1195–1238, 1993.

[93] M. L. Bellac, *A Short Introduction to Quantum Information and Quantum Computation*. Cambridge University Press, 2006.

[94] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, "Efficient Z gates for quantum computing", *Phys. Rev. A*, vol. 96, p. 022 330, 2 Aug. 2017.

[95] W. E. Lamb, "Theory of an optical maser", *Phys. Rev.*, vol. 134, A1429–A1450, 6A Jun. 1964.

[96] A. Lupaşcu, S. Saito, T. Picot, P. C. de Groot, C. J. P. M. Harmans, and J. E. Mooij, "Quantum non-demolition measurement of a superconducting two-level system", *Nature Physics*, vol. 3, no. 2, pp. 119–123, Feb. 2007, ISSN: 1745-2481.

[97] J. Majer, J. M. Chow, J. M. Gambetta, J. Koch, B. R. Johnson, J. A. Schreier, L. Frunzio, D. I. Schuster, A. A. Houck, A. Wallraff, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, "Coupling superconducting qubits via a cavity bus", *Nature*, vol. 449, no. 7161, pp. 443–447, Sep. 2007, ISSN: 1476-4687.

[98] A. Schneider, J. Braumüller, L. Guo, P. Stehle, H. Rotzinger, M. Marthaler, A. V. Ustinov, and M. Weides, "Local sensing with the multilevel ac stark effect", *Phys. Rev. A*, vol. 97, p. 062 334, 6 Jun. 2018.

[99] S. Schlör, "Temperature dependence of coherence in transmon qubits", Master thesis, Karlsruhe Institute of Technology (KIT), 2015.

[100] A. Wallraff, D. I. Schuster, A. Blais, L. Frunzio, J. Majer, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, "Approaching unit visibility for control of a superconducting qubit with dispersive readout", *Phys. Rev. Lett.*, vol. 95, p. 060 501, 6 Aug. 2005.

[101] J. M. Gambetta, "Control of superconducting qubits", in *Quantum Information Processing*, (Jülich), D. DiVincenzo, Ed., ser. Schriften des Forschungszentrums Jülich. Reihe Schlüsseltechnologien / Key Technologies, vol. 52, Jülich: Forschungszentrum Jülich GmbH Zentralbibliothek, 2013, B4.1–B4.50, ISBN: 978-3-89336-833-4.

[102] A. Dewes, F. R. Ong, V. Schmitt, R. Lauro, N. Boulant, P. Bertet, D. Vion, and D. Esteve, "Characterization of a two-transmon processor with individual single-shot qubit readout", *Phys. Rev. Lett.*, vol. 108, p. 057 002, 5 Feb. 2012.

[103] G. S. Paraoanu, "Microwave-induced coupling of superconducting qubits", *Phys. Rev. B*, vol. 74, p. 140 504, 14 Oct. 2006.

[104] C. Rigetti and M. Devoret, "Fully microwave-tunable universal gates in superconducting qubits with linear couplings and fixed transition frequencies", *Phys. Rev. B*, vol. 81, p. 134 507, 13 Apr. 2010.

[105] S. Sheldon, E. Magesan, J. M. Chow, and J. M. Gambetta, "Procedure for systematically tuning up cross-talk in the cross-resonance gate", *Phys. Rev. A*, vol. 93, p. 060 302, 6 Jun. 2016.

[106] Qkitgroup. "Qkit - a quantum measurement suite in python". (2021), [Online]. Available: `https://github.com/qkitgroup/qkit` (visited on 08/08/2021).

[107] IBM. "Qiskit - Open-Source Quantum Development". (2021), [Online]. Available: `https://qiskit.org/` (visited on 09/20/2021).

[108] J. F. Cochran and D. E. Mapother, "Superconducting transition in aluminum", *Phys. Rev.*, vol. 111, pp. 132–142, 1 Jul. 1958.

[109] A. B. Zorin, "Quantum-limited electrometer based on single cooper pair tunneling", *Phys. Rev. Lett.*, vol. 76, pp. 4408–4411, 23 Jun. 1996.

[110] C. Macklin, K. O'Brien, D. Hover, M. E. Schwartz, V. Bolkhovsky, X. Zhang, W. D. Oliver, and I. Siddiqi, "A near&#x2013;quantum-limited josephson traveling-wave parametric amplifier", *Science*, vol. 350, no. 6258, pp. 307–310, 2015.

[111] B. Yurke, L. R. Corruccini, P. G. Kaminsky, L. W. Rupp, A. D. Smith, A. H. Silver, R. W. Simon, and E. A. Whittaker, "Observation of parametric amplification and deamplification in a josephson parametric amplifier", *Phys. Rev. A*, vol. 39, pp. 2519–2533, 5 Mar. 1989.

[112] P. Winkel, I. Takmakov, D. Rieger, L. Planat, W. Hasch-Guichard, L. Grünhaupt, N. Maleeva, F. Foroughi, F. Henriques, K. Borisov, J. Ferrero, A. V. Ustinov, W. Wernsdorfer, N. Roch, and I. M. Pop, "Nondegenerate parametric amplifiers based on dispersion-engineered josephson-junction arrays", *Phys. Rev. Applied*, vol. 13, p. 024 015, 2 Feb. 2020.

[113] A. M. Korolev, V. I. Shnyrkov, and V. M. Shulga, “Note: Ultra-high frequency ultra-low dc power consumption hemt amplifier for quantum measurements in millikelvin temperature range”, *Review of Scientific Instruments*, vol. 82, no. 1, p. 016 101, 2011.

[114] Project Jupyter. “Jupyter”. (2021), [Online]. Available: `https://jupyter.org/` (visited on 10/05/2021).

[115] K. Technologies. “Labber”. (), [Online]. Available: `https://www.keysight.com/de/de/products/software/application-sw/labber-software.html` (visited on 06/22/2021).

[116] K. I. of Technology. “Qkit - a quantum measurement suite in python”. (), [Online]. Available: `https://github.com/qkitgroup/qkit` (visited on 06/22/2021).

[117] U. Tietze, C. Schenk, and E. Gamm, *Halbleiter-Schaltungstechnik*, 16., erweiterte und aktualisierte Auflage. Berlin: Springer Vieweg, 2019, ISBN: 3662485532; 9783662485538.

[118] Xilinx, Inc. “Zynq UltraScale+ MPSoC, Heterogeneous multiprocessing platform for broad range of embedded applications”. (), [Online]. Available: `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html` (visited on 08/04/2021).

[119] ——, *UltraScale architecture and product data sheet: Overview*, DS890, version 4.0, Mar. 16, 2021.

[120] ——, *UltraScale architecture configurable logic block, User guide*, UG574, version 1.5, Feb. 28, 2017.

[121] ——, *UltraScale architecture DSP slice, User guide*, UG579, version 1.10, Sep. 22, 2020.

[122] ——, *UltraScale architecture memory resources, User guide*, UG573, version 1.12, Mar. 17, 2021.

[123] D. Lewis, “DesignCon 2004 SerDes architectures and applications”, 2004.

[124] Xilinx, Inc., *UltraScale architecture SelectIO resources, User guide*, UG571, version 1.12, Sep. 28, 2019.

[125] ——, “Zynq UltraScale+ RFSoC”. (), [Online]. Available: `https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html` (visited on 08/04/2021).

[126] ——, *Zynq UltraScale+ RFSoC RF data converter v2.4 gen 1/2/3, LogiCORE IP product guide*, PG269, version 2.4, Nov. 30, 2020.

[127] A. P. M. Place, L. V. H. Rodgers, P. Mundada, B. M. Smitham, M. Fitzpatrick, Z. Leng, A. Premkumar, J. Bryon, A. Vrajitoarea, S. Sussman, G. Cheng, T. Madhavan, H. K. Babla, X. H. Le, Y. Gang, B. Jäck, A. Gyenis, N. Yao, R. J. Cava, N. P. de Leon, and A. A. Houck, "New material platform for superconducting transmon qubits with coherence times exceeding 0.3 milliseconds", *Nature Communications*, vol. 12, no. 1, p. 1779, Mar. 2021, ISSN: 2041-1723.

[128] D. Rosenberg, D. Kim, R. Das, D. Yost, S. Gustavsson, D. Hover, P. Krantz, A. Melville, L. Racz, G. O. Samach, S. J. Weber, F. Yan, J. L. Yoder, A. J. Kerman, and W. D. Oliver, "3d integrated superconducting qubits", *npj Quantum Information*, vol. 3, no. 1, p. 42, Oct. 2017, ISSN: 2056-6387.

[129] F. Motzoi, J. M. Gambetta, P. Rebentrost, and F. K. Wilhelm, "Simple pulses for elimination of leakage in weakly nonlinear qubits", *Phys. Rev. Lett.*, vol. 103, p. 110 501, 11 Sep. 2009.

[130] T. Walter, P. Kurpiers, S. Gasparinetti, P. Magnard, A. Poto čnik, Y. Salathé, M. Pechal, M. Mondal, M. Oppliger, C. Eichler, and A. Wallraff, "Rapid high-fidelity single-shot dispersive readout of superconducting qubits", *Phys. Rev. Applied*, vol. 7, p. 054 020, 5 May 2017.

[131] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, "A quantum engineer's guide to superconducting qubits", *Applied Physics Reviews*, vol. 6, no. 2, p. 021 318, 2019.

[132] A. D. Córcoles, E. Magesan, S. J. Srinivasan, A. W. Cross, M. Steffen, J. M. Gambetta, and J. M. Chow, "Demonstration of a quantum error detection code using a square lattice of four superconducting qubits", *Nature Communications*, vol. 6, no. 1, p. 6979, Apr. 2015, ISSN: 2041-1723.

[133] J. Heinsoo, C. K. Andersen, A. Remm, S. Krinner, T. Walter, Y. Salathé, S. Gasparinetti, J.-C. Besse, A. Poto čnik, A. Wallraff, and C. Eichler, "Rapid high-fidelity multiplexed readout of superconducting qubits", *Phys. Rev. Applied*, vol. 10, p. 034 040, 3 Sep. 2018.

[134] M. Kristen, "Time resolved quantum sensing of microwave fields with a transmon qubit", M.S. thesis, Karlsruhe Institute of Technology, 2019.

[135] N. Karcher, R. Gebauer, R. Bauknecht, R. Illichmann, and O. Sander, "Versatile configuration and control framework for real time data acquisition systems", *IEEE Transactions on Nuclear Science*, pp. 1–1, 2021.

[136] D. M. Pozar, *Microwave engineering*. John wiley & sons, 2011.

[137] R. Gartmann, "Digitally controllable integrated RF electronics for manipulation and readout of qubits", M.S. thesis, Karlsruhe Institute of Technology, 2019.

[138] *Wishbone B4 - WISHBONE System-on-Chip (SoC) interconnection architecture for portable IP cores*, OpenCores, 2010.

[139] A. Waterman and K. Asanović, Eds., *The RISC-V instruction set manual, Volume i: Unprivileged ISA*, Dec. 13, 2019.

[140] R. Illichmann, "High level description of experiments with superconducting quantum bits", M.S. thesis, Karlsruhe Institute of Technology, 2021.

[141] Cloud Native Computing Foundation. "gRPC - a high-performance, open source universal RPC framework". (2020), [Online]. Available: `https://grpc.io/` (visited on 04/08/2020).

[142] PetrLukas. "Open-amp". (2018), [Online]. Available: `https://github.com/OpenAMP/open-amp` (visited on 02/29/2020).

[143] R. Bauknecht, "Reliability of embedded FPGA-SoC systems in long-term physics experiments", M.S. thesis, Karlsruhe Institute of Technology, 2020.

[144] J. Serrano, M. Lipinski, T. Wlostowski, E. Gousiou, E. van der Bij, M. Cattin, and G. Daniluk, "The white rabbit project", *Proceedings of IBIC2013*, 2013.

[145] Zurich Instruments. "HDAWG - 750 MHz Arbitrary Waveform Generator". (2021), [Online]. Available: `https://www.zhinst.com/others/en/products/hdawg-arbitrary-waveform-generator#specifications` (visited on 09/18/2021).

[146] Tektronix. "Arbitrary Waveform Generators - AWG5200 Series Datasheet". (2020), [Online]. Available: `https://download.tek.com/datasheet/AWG5200-Series-Arbitrary-Waveform-Generator-Datasheet-76W608488.pdf` (visited on 09/18/2021).

[147] Active Technologies. "ARB Rider 5062(D)/5064(D)/5068(D), Technical datasheet". (2021), [Online]. Available: `https://www.activetechnologies.it/wp-content/uploads/2021/05/AWG5000Series_datasheet_210520.pdf` (visited on 10/13/2021).

[148] E. Gamess and R. Surós, "An upper bound model for TCP and UDP throughput in IPv4 and IPv6", *J Netw Comput Appl*, vol. 31, no. 4, pp. 585–602, 2008, ISSN: 1084-8045.

[149] A. I. Lvovsky, B. C. Sanders, and W. Tittel, "Optical quantum memory", *Nature Photonics*, vol. 3, no. 12, pp. 706–714, Dec. 2009, ISSN: 1749-4893.

[150] Y. Ma, Y.-Z. Ma, Z.-Q. Zhou, C.-F. Li, and G.-C. Guo, "One-hour coherent optical storage in an atomic frequency comb memory", *Nature Communications*, vol. 12, no. 1, p. 2381, Apr. 2021, ISSN: 2041-1723.

[151] P. M. Leung and B. C. Sanders, "Coherent control of microwave pulse storage in superconducting circuits", *Phys. Rev. Lett.*, vol. 109, p. 253 603, 25 Dec. 2012.

[152] J. D. Brehm, "Superconducting quantum metamaterials", Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2021, 132 pp.

[153] M. Fleischhauer, A. Imamoglu, and J. P. Marangos, "Electromagnetically induced transparency: Optics in coherent media", *Rev. Mod. Phys.*, vol. 77, pp. 633–673, 2 Jul. 2005.

[154] A. A. Abdumalikov, O. Astafiev, A. M. Zagoskin, Y. A. Pashkin, Y. Nakamura, and J. S. Tsai, "Electromagnetically induced transparency on a single artificial atom", *Phys. Rev. Lett.*, vol. 104, p. 193 601, 19 May 2010.

[155] S. H. Autler and C. H. Townes, "Stark effect in rapidly varying fields", *Phys. Rev.*, vol. 100, pp. 703–722, 2 Oct. 1955.

[156] J. J. Burnett, A. Bengtsson, M. Scigliuzzo, D. Niepce, M. Kudra, P. Delsing, and J. Bylander, "Decoherence benchmarking of superconducting qubits", *npj Quantum Information*, vol. 5, no. 1, p. 54, Jun. 2019, ISSN: 2056-6387.

[157] F.-L. Luo, Ed., *Digital Front-End in Wireless Communications and Broadcasting: Circuits and Signal Processing*. Cambridge University Press, 2011.

[158] J. Proakis and M. Salehi, *Communication Systems Engineering*, 2nd ed., ser. Pearson Education. Prentice Hall, 2002, ISBN: 9780130950079.

List of Publications

1. R. Gebauer, N. Karcher, and O. Sander, “A modular RFSoC-based approach to interface superconducting quantum bits”, in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–9.
2. R. Gartmann, N. Karcher, R. Gebauer, O. Krömer, and O. Sander, “Progress of the ECHo SDR readout hardware for multiplexed MMCs”, *Journal of Low Temperature Physics*, submitted for publication.
3. R. Gebauer, N. Karcher, J. Hurst, M. Weber, and O. Sander, “Taskrunner: A flexible framework optimized for low latency quantum computing experiments”, in *2021 IEEE 34th International System-on-Chip Conference (SOCC)*, 2021.
4. N. Karcher, R. Gebauer, R. Bauknecht, R. Illichmann, and O. Sander, “Versatile configuration and control framework for real time data acquisition systems”, *IEEE Transactions on Nuclear Science*, pp. 1–1, 2021.
5. D. Gusenkova, M. Spiecker, R. Gebauer, M. Willsch, D. Willsch, F. Valenti, N. Karcher, L. Grünhaupt, I. Takmakov, P. Winkel, D. Rieger, A. V. Ustinov, N. Roch, W. Wernsdorfer, K. Michielsen, O. Sander, and I. M. Pop, “Quantum nondemolition dispersive readout of a superconducting artificial atom using large photon numbers”, *Phys. Rev. Applied*, vol. 15, p. 064 030, 6 Jun. 2021.
6. R. Gebauer, N. Karcher, D. Gusenkova, M. Spiecker, L. Grünhaupt, I. Takmakov, P. Winkel, L. Planat, N. Roch, W. Wernsdorfer, A. V. Ustinov, M. Weber, M. Weides, I. M. Pop, and O. Sander, “State preparation of a fluxonium qubit with feedback from a custom FPGA-based platform”, *AIP Conference Proceedings*, vol. 2241, no. 1, p. 020 015, 2020.
7. R. Gebauer, N. Karcher, O. Sander, M. Weides, and O. Krömer, “Elektronische Anordnung zum Erzeugen und Auswerten von Mikrowellensignalen und Verfahren zum Betreiben einer solchen”, patent pending, Nov. 28, 2019.
8. R. Gebauer, “FPGA-based quantum feedback for superconducting qubits”, M.S. thesis, Karlsruhe Institute of Technology, Karlsruhe, 2018.

List of Abbreviations

ADC analog-to-digital converter

APU application processing unit

ASIC application-specific integrated circuit

AST abstract syntax tree

AWG arbitrary waveform generator

BCS Bardeen-Cooper-Schrieffer

BRAM block RAM

CFG control flow graph

CIC cascaded integrator-comb

CLB configurable logic block

CSV comma-separated values

CW continuous wave

DAC digital-to-analog converter

DC direct current

DDC digital down-conversion

DJJAA dimer Josephson junction array amplifier

DRAG derivative removal by adiabatic gate

DSP digital signal processing

DUC digital up-conversion

EIT electromagnetically induced transparency

FDM frequency-division multiplexing

FIR finite impulse response

FPGA	field-programmable gate array
FSBL	first stage boot loader
HEMT	high-electron-mobility transistor
I/O	input/output
I/Q	in-phase and quadrature
IF	intermediate frequency
IOB	I/O block
IPE	Institute for Data Processing and Electronics
ISA	instruction set architecture
JPA	Josephson parametric amplifier
KIT	Karlsruhe Institute of Technology
LUT	look-up table
MPSoC	multi-processor system-on-chip
NCO	numerically controlled oscillator
NIST	National Institute of Standards and Technology
OCM	on-chip memory
OpenAMP	Open Asymmetric Multi Processing
PCB	printed circuit board
PHI	Institute of Physics
PIMC	platform information and management core
PL	programmable logic
PLD	programmable logic device
PMCCNTR	performance monitors cycle count register
PS	processing system
QMC	quadrature modulator correction
QND	quantum non-demolition
qubit	quantum bit

RAM	random access memory
RF	radio frequency
RFSoC	radio frequency system-on-chip
RMS	root mean square
RPC	remote procedure calls
RPU	real-time processing unit
RX	receiver
SerDes	serializer and deserializer
SFDR	spurious free dynamic range
SNR	signal-to-noise ratio
SoC	system-on-chip
SQUID	superconducting quantum interference device
SRAM	static RAM
TCM	tightly-coupled memory
TCXO	temperature-compensated crystal oscillator
TWPA	traveling wave parametric amplifier
TX	transmitter
VNA	vector network analyzer
WB	Wishbone

Appendix

A Signal Processing

A.1 Quadrature Frequency Mixing

Interfacing superconducting quantum bits (qubits) requires microwave signals in the frequency range of up to ten gigahertz with shapes defined on a nanosecond scale. To generate these signals, a carrier frequency is modulated by the output of high-resolution, nanosecond-accurate digital-to-analog converters (DACs) [117, p. 1387 ff.]. The output of the converters at an intermediate frequency ω_{IF} is shifted to a target frequency band ω_{RF} by so-called heterodyning. The frequency difference $\omega_{\mathrm{LO}} = \omega_{\mathrm{RF}} - \omega_{\mathrm{IF}}$ is given by a local oscillator. Intermediate frequency signal and local oscillator signal are then combined using a single mixer. To obtain both phase and sideband control, quadrature frequency mixing with an IQ mixer is employed [117, p. 1207 ff.]. Using this technique, the signal's phase information is preserved during the mixing process.

From a mathematical perspective, an IQ mixer multiplies the in-phase and quadrature (I/Q) components of a signal with intermediate frequency, $S_{\mathrm{I}}(t)$ and $S_{\mathrm{Q}}(t)$, with the signal of a local oscillator, $S_{\mathrm{LO}}(t) = \cos\omega_{\mathrm{LO}}t$, to obtain the resulting signal:

$$S_{\mathrm{RF}}(t) = S_{\mathrm{I}}(t)\cos\omega_{\mathrm{LO}}t + S_{\mathrm{Q}}(t)(-\sin\omega_{\mathrm{LO}}t) \ . \tag{1}$$

Without loss of generality, the phase of the local oscillator signal can be assumed to be zero as a choice of reference. To represent a signal with arbitrary amplitude $A(t)$ and phase $\varphi(t)$, one chooses $S_{\mathrm{I}}(t) = A(t)\cos(\omega_{\mathrm{IF}}t + \varphi(t))$ and $S_{\mathrm{Q}}(t) = A(t)\sin(\omega_{\mathrm{IF}}t + \varphi(t))$. It then holds:

$$\begin{aligned} S_{\mathrm{RF}}(t) &= A(t)\left[\cos(\omega_{\mathrm{IF}}t + \varphi(t))\cos\omega_{\mathrm{LO}}t - \sin(\omega_{\mathrm{IF}}t + \varphi(t))\sin\omega_{\mathrm{LO}}t\right] \\ &= A(t)\cos\left(\omega_{\mathrm{RF}}t + \varphi(t)\right) \ . \end{aligned} \tag{2}$$

One obtains a signal where the phase and amplitude modulation on the intermediate frequency IQ signal is directly transferred to a carrier frequency $\omega_{\mathrm{RF}} = \omega_{\mathrm{LO}} + \omega_{\mathrm{IF}}$.

The intermediate frequency ω_{IF} can also be chosen negative which would result in the same output as if S_{I} and S_{Q} are swapped at the input of the mixer.

If the mixer is not operating perfectly, the amplitudes of S_{I} and S_{Q} are not identical, or the phase difference between them is not exactly 90°, the mirror frequency $\omega_{\mathrm{LO}} - \omega_{\mathrm{IF}}$ will also be present in the resulting signal. If this is the case, one can calibrate the mixer by varying the relative amplitude of the inputs and their phase difference in order to reduce the mirror signal.

To convert a radio frequency (RF) signal $S'_{\mathrm{RF}}(t)$ back to the intermediate frequency, another IQ mixer is used. When operated with RF signal

$$S'_{\mathrm{RF}}(t) = A'(t)\cos\left(\omega_{\mathrm{RF}}t + \varphi'(t)\right) \tag{3}$$

as input, the mixer multiplies it with the local oscillator and outputs the following components:

$$\begin{aligned} S'_{\mathrm{I}}(t) &= S'_{\mathrm{RF}}(t)\cos\omega_{\mathrm{LO}}t = \frac{A'(t)}{2}\left[\cos(\omega_{\mathrm{IF}}t + \varphi'(t)) + \cos(\omega_{\mathrm{P}}t + \varphi'(t))\right] \quad \text{and} \\ S'_{\mathrm{Q}}(t) &= S'_{\mathrm{RF}}(t)(-\sin\omega_{\mathrm{LO}}t) = \frac{A'(t)}{2}\left[\sin(\omega_{\mathrm{IF}}t + \varphi'(t)) - \sin(\omega_{\mathrm{P}}t + \varphi'(t))\right] \quad . \end{aligned} \tag{4}$$

The second terms with $\omega_{\mathrm{P}} = \omega_{\mathrm{RF}} + \omega_{\mathrm{LO}}$ can be suppressed when using an appropriate low-pass filter behind the mixer. It thereby reconstructs the desired I/Q component with amplitude $A'(t)/2$ and phase $\varphi'(t)$. In order to obtain a stable phase relation and being able to recover the phase, the same local oscillator signal should be used for both frequency up- and down-conversion.

A.2 Digital Down Conversion

While quadrature frequency mixing performs a frequency translation in the analog domain, a digital down-conversion (DDC) is used for similar signal processing in the digital domain. It converts a digitized signal at a limited frequency band to a lower frequency signal at lower sampling rate. This simplifies subsequent digital signal processing (DSP) steps. DDCs can work both with real signals or with complex I/Q ones. As part of an analog-to-digital converter (ADC), they often accept a real input signal at high sampling rate and output a complex baseband signal at a lower sampling rate for further processing in an FPGA, see also Section 2.4. But also a complex input signal can be used and shifted in frequency [157, p. 333 ff.].

A DDC consists of three stages: a mixer, a low-pass filter, and a downsampler. In the mixer, the input signal is multiplied with a (complex) reference signal generated by a numerically controlled oscillator (NCO). In the complex case, the

I/Q components can be represented in the complex plane as $S_{\text{in}} = A(t)e^{i\omega_{\text{in}}t}$ and the reference as $S_{\text{ref}} = e^{-i\omega_{\text{ref}}t}$. For the reference, the frequency was negated to perform a down-conversion during the mixing process, which can be formulated as complex multiplication:

$$S_{\text{in}} \cdot S_{\text{ref}} = A(t)e^{i\omega_{\text{in}}t} \cdot e^{-i\omega_{\text{ref}}t} = A(t)e^{i(\omega_{\text{in}}-\omega_{\text{ref}})t} \quad . \tag{5}$$

Using the mixer, one thus shifts the frequency of the input signal by $-\omega_{\text{ref}}$. For real input signals, one will also obtain a component with frequency $\omega_{\text{in}} + \omega_{\text{ref}}$. To reject this component, a consecutive low-pass filter is used. It also limits the band of the resulting signal and thus suppresses noise or other signals from outside this band. Then, also a lower sample rate can be used without loosing any information. This is handled by the downsampler.

Depending on the implementation of the low-pass filter, the downsampler can also be integrated in this filter. An example would be a boxcar integrator that simply sums up all incoming values over a specified time window and outputs one average value per time window. Other commonly used low-pass filters include finite impulse response (FIR) or cascaded integrator-comb (CIC) filters.

One special case of a DDC is to choose $\omega_{\text{ref}} = \omega_{\text{in}}$. In this case, the signal of the carrier frequency will be down-converted to a DC signal. The resulting I and Q components are then equivalent to the amplitude A and phase φ information of the signal which thus can be extracted:

$$I + iQ = Ae^{i\varphi} \quad . \tag{6}$$

A.3 Frequency-Division Multiplexing

Frequency-division multiplexing (FDM) is a technique to combine multiple signals onto a single transmission medium. To achieve this, the bandwidth of the medium is divided in multiple, non-overlapping frequency bands, so-called frequency channels. Each band contains a separate signal [117, p. 1222 f.]. In this scheme, multiple baseband signals $s_i(t)$ are each transmitted by an individual carrier with frequency f_i and combined into a single, FDM signal:

$$S_{\text{FDM}}(t) = s_0(t)\cos(2\pi f_0 t) + s_1(t)\cos(2\pi f_1 t) + \ldots + s_n(t)\cos(2\pi f_n t) \tag{7}$$

The distance between the carrier frequencies f_i has to be larger than the bandwidth of the baseband signals $s_i(t)$ so that they do not overlap in the frequency domain. Otherwise, accurate reconstruction of the individual signals is not possible anymore. In general, reconstruction of a single baseband signal can be performed using a DDC with a suitable low-pass filter that selects a single frequency channel and rejects all others [158, p. 94 f.].

B Programmable Logic

B.1 Sequencer Commands

The following commands are supported by the RISC-V based sequencer:

- `ADDI`: add sign-extended immediate to register
- `ANDI, ORI, XORI`: bitwise logical AND, OR, and XOR on sign-extended immediate and register
- `SLLI, SRLI, SRAI`: logical left shift, logical right shift, and arithmetic right shift, respectively, of a register by shift amount as immediate
- `LUI`: load upper immediate
- `ADD, SUB`: add or subtract two registers, respectively
- `MUL, MULH`: multiply two registers and save the lower or upper part of the result, respectively
- `AND, OR, XOR`: bitwise logical AND, OR, and XOR on two registers
- `SLL, SRL, SRA`: logical left shift, logical right shift, and arithmetic right shift, respectively, of a register by shift amount in another register
- `BEQ, BNE`: compare two register and take the branch if they are equal or unequal, respectively
- `BLT, BLTU`: compare two register and take the branch if the first is less than the second using signed or unsigned comparison, respectively
- `BGE, BGEU`: compare two register and take the branch if the first is greater than or equal the second using signed or unsigned comparison, respectively
- `JAL`: perform an unconditional jump to the target address defined by an program counter offset as immediate
- `LOAD`: copy a value obtained via the Wishbone (WB) bus to a register
- `STORE`: write a value from a register to an address on the WB bus

More details on these 27 commands from the base integer and multiplication set can be found in the RISC-V instruction set architecture (ISA) specification [139]. Besides them, six additional commands are implemented as special-purpose set. These are introduced in Section 3.4.3 and used for sequencing of operations.

B.2 Signal Generator Timing

The signal generator needs in total 13 cycles from receiving a trigger signal at the WB bus until outputting it to the AXI-stream interface. The delay stems from the following process inside the module:

1. The WB register interface detects the write access and sets the trigger code at its output.
2. The signal generator detects the trigger and loads the appropriate trigger set configuration. It also triggers the sample players.
3. The sample players detect the trigger and update their internal state. They also apply the address for the first envelope samples to the block RAM (BRAM).
4. The BRAM returns the first envelope samples to the sample players.
5. The sample player contains an output multiplexer to hold the last sample. It is now forwarding the samples from the BRAM.
6. An output delay queue is added to match the delay between envelope samples and NCO samples of the mixer. The signal needs to be delayed by three cycles.
7. The second delay cycle.
8. The last delay cycle. The envelope samples are applied to the output of the sample player.
9. A pipelining stage delays the data for one cycle to relax the timing.
10. The complex multiplier performs the multiplication in two cycles.
11. The second cycle of the multiplication. The result is applied to the output of the multiplier.
12. The data is calibrated by multiplying it with a scaling factor.
13. A pipelining stage delays the data for one cycle to relax the timing. The data is applied to the AXI-stream interface.

B.3 Measurement-based Feedback Timing

The digital unit cell provides feedback with 352 ns intrinsic platform latency, of which 72 ns, i.e. 18 logic cycles on the FPGA, are due to processing of the result in the programmable logic (PL). This can be further detailed as follows:

- 4 cycles to transfer the trigger from the sequencer via the WB bus to the recording module's state machine.
- 2 cycles to process the trigger and start the recording.
- 280 ns (70 cycles) trigger delay to compensate for the delays through the cell multiplexer, the converters, as well as the electrical delay through the analog signal paths in loop-back mode. This does not account for additional delay through an experiment setup as this is experiment-specific.
- Length of the recording, also experiment-specific and thus not considered by the intrinsic platform latency.
- 7 cycles to process the last incoming data and finalize the accumulation.
- 1 cycle to set the result of the measurement.
- 2 cycles to calculate the estimated qubit state from the result.
- 2 cycles to write the state into a sequencer register and perform a branch operation without a jump (as a consecutive conditional pulse should not be skipped).

The pulse generation part does not have to be covered as it is identical for the readout and the conditional pulse. It thus does not contribute to the time difference between both pulses. The same applies for the length of the recording, as it is chosen to be identical to the readout pulse length, and the latency is measured from the end of the readout pulse. Thereby, it is already subtracted from the latency making it as independent as possible from varying experiment parameters.

C Taskrunner

C.1 Interface to the Taskrunner Firmware

The Taskrunner firmware provides the following methods for user tasks to interact with firmware and user client:

- `void rtos_EnterCriticalSection()`
  starts a critical section where context switches are prohibited.
- `void rtos_ExitCriticalSection()`
  ends a critical section and allows context switches to happen.
- `void rtos_RestartTimer()`
  resets the internal performance monitors cycle count register (PMCCNTR).
- `uint32_t rtos_GetCycleCountTimer()`
  reads the cycle count in the PMCCNTR (32bit register counting at 500MHz).
- `uint32_t rtos_GetNsTimer()`
  reads the PMCCNTR and converts its value to a nanosecond delay, overflowing after 4.2 s.
- `void rtos_SetProgress(uint32_t progress)`
  updates the progress visible to the client to the given value.
- `void* rtos_GetParameters()`
  returns the pointer to the parameter memory region in the shared DDR4 memory.
- `uint32_t rtos_GetParametersSize()`
  returns the valid size of the parameter memory region in bytes.
- `void* rtos_GetDataBox(uint32_t size)`
  creates and allocates a new data box with the specified size in the shared DDR4 memory.
- `void rtos_FinishDataBox(void* databox)`
  finalizes the given data box so it can be accessed by the client.
- `void rtos_DiscardDataBox(void* databox)`
  frees the given data box without sending the data to the client.
- `void rtos_printf(const char *format, ...)`
  prints a message to the UART channel. Uses the same syntax as a regular `printf`.
- `void rtos_ReportError(const char *error_msg)`
  adds a raw string to the error message queue that can be read out by the client.
- `void rtos_PrintfError(const char *format, ...)`
  adds a formatted string to the error message queue which can also include the values of internal variables etc.

C.2 Interleaved Experiment Task for Multiple Qubits

The following script is executed on the Taskrunner together with the QiCode defined in Code 5.5. In this version of the script, the detuning of the Ramsey pulses is still handled by the Taskrunner and not by the Sequencer itself.

```c
#include <string.h>
#include "task.h"
#include "cells.h"

int task_entry()
{
  rtos_printf("\r\nStart Multi Interleaved Qubit Experiments\r\n");
  uint32_t *param_list = (uint32_t *)rtos_GetParameters();
  uint32_t param_count = rtos_GetParametersSize() / sizeof(uint32_t);
  if (param_count < 3)
  {
    rtos_PrintfError("Not enough parameters provided (%d given).", param_count);
    return -1;
  }
  uint32_t num_of_experiments = (param_list++)[0];
  uint32_t experiments_per_loop = (param_list++)[0];
  uint32_t cell_num = (param_list++)[0];

  uint32_t param_count_expected = 3  // General parameters
    + experiments_per_loop  // Experiment order
    + cell_num  // Cell map (which cells to address)
    + num_of_experiments  // Experiment-specific parameters
    + num_of_experiments * cell_num;  // Cell and Exp.-specific parameters
  if (param_count < param_count_expected)
  {
    rtos_PrintfError(
      "Not enough parameters provided (needed atleast %d, but %d given).",
      param_count_expected, param_count);
    return -1;
  }
  uint32_t *cell_list_param = param_list;
  param_list += cell_num;
  uint8_t cell_list[cell_num];
  uint8_t cell_count = cells_get_count();
  for (int c = 0; c < cell_num; c++)
  {
    // Check if the cell index is valid (within range of available cells)
    if (cell_list_param[c] >= cell_count)
    {
      rtos_PrintfError(
        "Requested cell %d, but only 0 to %d available.",
        cell_list_param[c], cell_count - 1);
      return 1;
    }
    // Copy from 32bit to 8bit unsigned
    cell_list[c] = (uint8_t)cell_list_param[c];
  }

  uint32_t *experiment_order = param_list;
  param_list += experiments_per_loop;
  uint32_t *experiment_executions = param_list;
```

```
param_list += num_of_experiments;
uint32_t *experiment_nco_freq[cell_num];
for (uint8_t c = 0; c < cell_num; c++)
{
   experiment_nco_freq[c] = param_list;
   param_list += num_of_experiments;
}

for (uint32_t iexp = 0; iexp < num_of_experiments; iexp++)
   param_count_expected += experiment_executions[iexp];
if (param_count != param_count_expected)
{
   rtos_PrintfError(
      "Not enough parameters provided (needed %d, but %d given).",
      param_count_expected, param_count);
   return -1;
}
uint32_t *experiment_delays[num_of_experiments];
for (uint32_t iexp = 0; iexp < num_of_experiments; iexp++)
{
   experiment_delays[iexp] = param_list;
   param_list += experiment_executions[iexp];
}

// Fetch cell pointers from platform (do not forget to free at the end!)
Cell *cells = cells_create();
Cell cell[cell_num];
for (uint32_t c = 0; c < cell_num; c++)
{
   cell[c] = cells[cell_list[c]];

   // Set start address to 0 (exp select via register)
   seq_set_start_address(cell[c].sequencer, 0);
}

// Data structures
iq_pair *data[cell_num][num_of_experiments];
uint32_t pos[num_of_experiments];
uint32_t sum_of_executions = 0;
for (uint32_t iexp = 0; iexp < num_of_experiments; iexp++)
{
   for (uint32_t c = 0; c < cell_num; c++)
   {
      data[c][iexp] = rtos_GetDataBox(experiment_executions[iexp] * sizeof(iq_pair));
   }
   pos[iexp] = 0;
   sum_of_executions += experiment_executions[iexp];
}

rtos_printf(
   "In total, perform %d experiment executions with %d cells.\r\n",
   sum_of_executions, cell_num);

// At the beginning we once wait until controller has finished possible previous task
cells_wait_while_busy();

// Initialize with last experiment in row, so by incrementing we start with the first
uint32_t iorder = experiments_per_loop - 1;
uint8_t exp = 0;
for (uint32_t i = 0; i < sum_of_executions; i++)
```

```
  {
    // Select next experiment:
    // Check if next experiment still needs to be executed
    // or if it should be skipped because no delays remain
    do
    {
      iorder = (iorder + 1) % experiments_per_loop;
      exp = experiment_order[iorder];
    } while (pos[exp] >= experiment_executions[exp]);

    for (uint8_t c = 0; c < cell_num; c++)
    {
      // Set the NCO Frequency of the manipulation pulsegen
      pg_set_internal_frequency_reg(cell[c].manipulation, experiment_nco_freq[c][exp]);
      // Write the delay register
      seq_set_register(cell[c].sequencer, 1, experiment_delays[exp][pos[exp]]);
      // Write the experiment select register
      seq_set_register(cell[c].sequencer, 2, exp);
    }

    // Start the sequencer experiment execution
    cells_start(cell_list, cell_num);

    // Wait until execution finished
    cells_wait_while_busy();

    for (uint8_t c = 0; c < cell_num; c++)
    {
      // Store result in the appropriate location in the right data box
      rec_get_averaged_result(cell[c].recording, &data[c][exp][pos[exp]]);
    }

    rtos_SetProgress(i + 1);

    // Current execution finished so increment the counter
    pos[exp]++;
  }

  for (uint32_t c = 0; c < cell_num; c++)
  {
    for (uint32_t iexp = 0; iexp < num_of_experiments; iexp++)
    {
      rtos_FinishDataBox(data[c][iexp]);
    }
  }

  // Important to free the cells at the end to not generate a memory leak!
  cells_free(cells);
  rtos_printf("Task finished.\r\n");
  return 0;
}
```

The parameters are generated within a custom Python experiment class where one can pass the delays for the different experiments, the order in which the experiments should be interleaved and on which cells they should be executed. One can additionally define a per experiment detuning of the pulses which will be set by the task.

C.3 Quantum Jumps Collection Task for Multiple Qubits

The following C code is executed on the Taskrunner to collect the states measured by the QiCode in Code 5.2. While the QiCode was only written for one cell, the task also supports multiple qubits simultaneously and the QiCode could be easily expanded.

```c
#include "task.h"
#include "cells.h"

#define MAX_ADDR 1024
#define STATES_PER_REG 32

int task_entry()
{
  uint32_t *param_list = rtos_GetParameters();
  uint32_t param_count = rtos_GetParametersSize() / sizeof(uint32_t);
  if (param_count < 4)
  {
    rtos_PrintfError(
      "This task needs atleast 4 parameter values (only %d given).",
      param_count);
    return -1;
  }
  uint32_t repetitions = param_list[0]; // How many repetitions to perform
  if (repetitions % STATES_PER_REG != 0)
  {
    rtos_PrintfError(
      "This task can only perform a multiple of %d repetitions (%d requested).",
      STATES_PER_REG, repetitions);
    return -1;
  }
  uint32_t cell_num = param_list[1]; // How many cells need to be addressed
  if (param_count != 2 + 2 * cell_num)
  {
    rtos_PrintfError(
      "This task needs exactly %d parameter values (%d given).",
      2 + 2 * cell_num, param_count);
    return -1;
  }
  uint32_t *cell_list_param = &(param_list[2]); // Indices of the cells to use
  // Recording counts would follow next, but not used here

  // Verify the parameters
  uint8_t cell_list[cell_num];
  uint8_t cell_count = cells_get_count();
  for (int c = 0; c < cell_num; c++)
  {
    // Check if the cell index is valid (within range of available cells)
    if (cell_list_param[c] >= cell_count)
    {
      rtos_PrintfError(
        "Requested cell %d, but only 0 to %d available.",
        cell_list_param[c],
        cell_count - 1);
      return 1;
    }
    // Copy from 32bit to 8bit unsigned
    cell_list[c] = (uint8_t)cell_list_param[c];
  }

  // Fetch cell pointers from platform (do not forget to free at the end!)
  Cell *cells = cells_create();
  Cell cell[cell_num];
  for (uint32_t c = 0; c < cell_num; c++)
  {
    cell[c] = cells[cell_list[c]];

    // For each cell, initialize the storage module
    // Reset BRAM 0 and activate wrapping
    stg_set_bram_control(cell[c].storage, 0, true, true);
    // Record state in BRAM 0 and accumulate in dense mode
    stg_set_state_config(cell[c].storage, 0, true, true, true);
  }

  // Initialize the databoxes
  uint32_t *states[cell_num];
  uint32_t last_addr[cell_num];
  uint32_t count[cell_num];
  uint32_t *bram[cell_num];
  for (uint32_t c = 0; c < cell_num; c++)
  {
    states[c] = rtos_GetDataBox(sizeof(uint32_t) * (repetitions / STATES_PER_REG));
    for (uint32_t i = 0; i < repetitions / STATES_PER_REG; i++)
      states[c][i] = 0;
    last_addr[c] = 0;
    count[c] = 0;
    bram[c] = stg_get_bram_pointer(cell[c].storage, 0);
  }
  // Wait for potential previous task
  cells_wait_while_busy();

  uint32_t next_addr, i, c;

  // Synchronously start all relevant cells
  cells_start(cell_list, cell_num);
  bool busy = true;
  while (busy)
  {
    // Check busy here at beginning to ensure we do this loop once again if
    // sequencers finishe in the meantime (to collect remaining data)
    busy = cells_is_any_busy();

    for (c = 0; c < cell_num; c++)
    {
      next_addr = stg_get_next_address(cell[c].storage, 0);
      if (next_addr < last_addr[c])
      {
        // Address wrapped -> collect remaining ones
        for (i = last_addr[c]; i < MAX_ADDR; ++i)
        {
          states[c][count[c]++] = *(bram[c] + i);
        }
        last_addr[c] = 0; // Continue at beginning
      }
      if (next_addr > last_addr[c])
      {
            // More states present
            for (i = last_addr[c]; i < next_addr; ++i)
            {
               states[c][count[c]++] = *(bram[c] + i);
            }
            last_addr[c] = next_addr;
         }
      }
      // Take count from first cell (fetched first -> the least progressed one)
      rtos_SetProgress(count[0] * STATES_PER_REG);
   }

   for (c = 0; c < cell_num; c++)
   {
      if (count[c] * STATES_PER_REG < repetitions)
      {
         rtos_PrintfError(
            "Expected %d states, but only collected %d for cell %d!\n\r"
            "The remaining states could not been catched in time...",
            repetitions, count[c] * STATES_PER_REG, c);
      }

      rtos_FinishDataBox(states[c]);
   }

   // Important to free the cells at the end to not generate a memory leak!
   cells_free(cells);
   return 0;
}
```

The parameters for the task are the standard set of parameters generated by QiCode when the QiJob is compiled. As these are sufficient for this case, they haven't been changed in Code 5.3 when attaching the custom task to the QiJob.

D Characterization

D.1 Phase Noise of Analog Output Signal

The following table provides phase noise values in dBc/Hz for different carrier frequencies (rows) and at different offsets (columns). The data is extracted from the same measurement as Figure 3.24.

Carrier / Offset	100 Hz	1 kHz	10 kHz	100 kHz	1 MHz	10 MHz
10 MHz	−121	−139	−159	−165	−148	−34
50 MHz	−107	−127	−146	−152	−166	−174
100 MHz	−102	−121	−140	−146	−161	−172
150 MHz	−99	−118	−138	−142	−158	−170
200 MHz	−96	−116	−135	−140	−155	−169
250 MHz	−93	−113	−134	−138	−153	−168
300 MHz	−92	−112	−132	−136	−152	−167
350 MHz	−90	−111	−131	−135	−150	−166
400 MHz	−89	−109	−130	−134	−149	−165
450 MHz	−88	−108	−128	−133	−148	−164
500 MHz	−88	−107	−128	−132	−147	−163

Acknowledgments

I would like to express my great appreciation to my supervisor Prof. Dr. Marc Weber for the opportunity to work on this exciting topic and for his guidance and feedback along the way. Additionally, I am grateful for his support to broaden my horizon by enabling me to participate in various conferences and pursue an international MBA program.

Furthermore, I want to thank Prof. Dr. Martin Weides for being the second reviewer of this thesis, for the inspiring discussions we had, and for the fruitful collaboration with his group at the University of Glasgow.

Special thanks go to PD Dr. Oliver Sander for his excellent supervision. Whenever needed, I could be sure to have an open ear and get valuable feedback which also contributed to my personal development. We had many productive discussions that significantly improved this thesis. Thank you for all the possibilities you provided and for the freedoms I had to manage and advance this thesis' project in your research group.

I also want to express my gratitude to Prof. Dr. Alexey Ustinov for his input to this thesis and the great cooperation with his group and laboratory at the Institute of Physics at KIT. The same holds true for Dr. Ioan Pop and group at the Institute of Physics and the Institute for Quantum Materials and Technologies at KIT. Thank you for the productive collaboration and the many valuable scientific discussions during this time.

This thesis would not have been possible without the help of many amazing colleagues both at the Institute for Data Processing and Electronics and the Institute of Physics. Special thanks go to Nick Karcher for numerous vivid discussions and his assistance with countless technical challenges. Our joint efforts to establish a common infrastructure, be it our IT systems, our build environment, or the ServiceHub, contributed to the success of this thesis. Moreover, thanks go to Robert Gartmann for his contribution regarding the analog RF frontend and his continued support with all kinds of challenges. Among others, I want to especially thank Dr. Hannes Rotzinger, Dr. Andre Schneider, Dr. Jan Brehm, Dr. Steffen Schlör, Dr. Daria Gusenkova, and Ilya Besedin for many valuable discussions and their support to perform my experiments. Your precious input and assistance significantly improved the results of this thesis and made it possible to design the QiController and QiCode according to the requirements of the final user.

A big thank you goes to the countless students I had the pleasure to supervise and work with during my thesis. You had a major impact on the final platform and its capabilities, be it QiCode, the ServiceHub, the Taskrunner, the FPGA modules, or the analog frontend PCB.

Furthermore, I am very thankful for the generous financial support by the Landesgraduiertenförderung and the Helmholtz International Research School for Teratronics (HIRST) at KIT. For support during my studies, I am grateful to the Deutschlandstipendium and the Netzwerk Deutschlandstipendium Karlsruhe e.V. which significantly contributed to my personal advancement.

Last but not least, I would like to express my deep gratefulness to my family and my partner Lena. Thank you for your never-ending support on my journey and for the great food along the way.

Karlsruhe, December 2021 Richard Gebauer