

QuMAsim: A Quantum Architecture Simulation and Verification Platform

Mengyu Zhang

<CE-MS-2018-023>

Abstract

Quantum microarchitecture is a key component in bridging the gap between quantum software and quantum hardware of a fully programmable quantum computer. Confronting the control problem of superconducting quantum processors, an experimental microarchitecture (QuMA) has been proposed in previous work. As the size of the target quantum chip continues to evolve, the complexity of QuMA scales up accordingly. This increase in complexity has led to a growing challenge in QuMA's design, development, and verification. To solve these problems, we build a QuMA simulator which can automate the current QuMA verification process and accelerate the design phase of QuMA. We called this simulator QuMAsim. The first version of QuMAsim is based on CC-Light, an instance of QuMA for controlling a surface-7 superconducting qubit chip. Then this simulator is extended to be self-configurable for different quantum chips. Several applications are built based on this simulator. The verification platform consists of the simulator, the VHDL implementation of QuMA and a validator is designed to automate the verification procedure. In addition, we built a quantum virtual machine based on QuMAsim, which includes other quantum software and simulators to simulate the execution of quantum algorithms on each layer of the quantum computer. We demonstrate the potential of QuMAsim by performing some experiments with it and its applications, including the gearbox circuit simulation.

QUMASIM: A QUANTUM ARCHITECTURE SIMULATION AND VERIFICATION PLATFORM

by

Mengyu Zhang

in partial fulfillment of the requirements for the degree of

Master of Science
in Microelectronics

at the Delft University of Technology,
to be defended publicly on Friday August 24, 2018 at 14:00 AM.

Supervisor:	Prof. dr. K. L. M. Bertels	
Thesis committee:	Prof. dr. K. L. M. Bertels	TU Delft
	Prof. dr. L. Dicarolo	TU Delft
	Dr. C. G. Almudever	TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Dedicated to my family and friends

CONTENTS

List of Figures	vii
List of Tables	ix
List of Acronyms	xi
Acknowledgements	xiii
1 Introduction	1
1.1 Context	1
1.2 Research motivation	2
1.3 Research approach	2
1.4 Organization	3
2 Background	5
2.1 Quantum Computing Basics	5
2.1.1 Qubits	5
2.1.2 Quantum Gates	6
2.1.3 Universal Quantum Computation	6
2.2 Full Stack Quantum Computer	6
2.3 Hardware	8
2.3.1 Superconducting Qubits	8
2.3.2 Quantum chip topology	9
2.3.3 Implication on the Control.	10
2.3.4 Quantum Microarchitecture	10
2.4 Quantum Instruction Set Architecture	12
2.4.1 QISA overview	12
2.4.2 eQASM.	12
2.5 Compiler and Language.	14
3 Simulator implementation	15
3.1 QuMA Structure.	15
3.1.1 Overview.	15
3.1.2 Quantum Pipeline	17
3.1.3 Classical Pipeline	18
3.1.4 Comprehensive Feedback Control	19
3.2 Simulator Implementation	20
3.2.1 Goals and Requirements	20
3.2.2 SystemC	21
3.2.3 QuMASim overview	21
3.2.4 Design Space Exploration	21
3.3 Self-configurable Simulator.	22
3.3.1 Quantum Layout Information	22
3.3.2 Instruction Set Architecture Modifications.	24
3.3.3 Final Implementation	25
3.4 Simulator Output	25
3.4.1 Triggered output	26
3.4.2 Register values	26
3.4.3 Measurement results.	26
3.4.4 Instruction under execution	27

3.5	Conclusion	27
4	Applications	29
4.1	Verification Platform	29
4.1.1	Overview and approach selection	29
4.1.2	Implementation	30
4.2	Quantum Virtual Machine	32
4.2.1	Overview.	32
4.2.2	OpenQL Framework	33
4.2.3	Simulation Platform	33
4.2.4	A Real Implementation	35
4.3	Conclusion	39
5	Experiments	41
5.1	Verification Experiment.	41
5.1.1	Testbench	41
5.1.2	Verification Environment	41
5.1.3	Result	42
5.2	Full Stack Simulation	42
5.2.1	Simulation Environment.	42
5.2.2	CNOT-measurement test.	43
5.2.3	Gearbox Circuit Simulation	45
5.3	Microarchitecture Design Experiment	49
5.3.1	Different Microwave Setup.	49
5.3.2	Implementation	49
6	Conclusion and Future Work	51
6.1	Conclusion	51
6.2	Future Work.	52
	Bibliography	53

LIST OF FIGURES

2.1	Overview of quantum computer system stack	8
2.2	Images at various scales of implementation of surface-code fabric. (a)Starmon qubit. (b) Transmission-line crossover. (c) Vertical I/O.	9
2.3	Layout of a surface code using 7 qubits	10
2.4	The relationship between control unit and quantum processor	10
2.5	Schematic of how to control a Surface-17 chip	11
3.1	Overview of the structure of QuMA_v1	15
3.2	Overview of the structure of QuMA_v2, it is more complex compared to QuMA_v1 since it needs to control more qubits	16
3.3	Quantum control unit structure	18
3.4	Comprehensive feedback control logic	19
3.5	The overall structure of QuMASim	21
3.6	The quantum processor architecture for surface-17	23
3.7	Modifications on microarchitecture for SC17	24
3.8	The modifications for SMIT instruction. (a) The SMIT binary format for SC7 chip. (b) The SMIT binary format for SC17 chip. (c) T-mask register structure for SC17 chip.	24
4.1	Structure of the verification platform, which consists of QuMASim, the HDL implementation and a validator	30
4.2	Structure of the real-world implementation and structure of the full stack simulator	33
4.3	Control electronics between central controller and superconducting qubits	33
4.4	OpenQL framework structure	34
4.5	Structure of electronics simulator in the full stack quantum simulator	35
4.6	Structure of the microwave operation generator	36
4.7	Structure of the flux operation generator	36
4.8	Simulation interface which connects QuMASim and electronics simulator to QuantumSim. This interface is derived from a generic simulator interface for all simulation back-ends.	37
4.9	Detailed functions for controlling QuantumSim in the interface	38
5.1	Recording modules for tracing the DIO outputs of VHDL implementation of QuMA_v3. These modules are instantiated from the verification library	42
5.2	An example of the CNOT-measurement circuit	43
5.3	The 150 measurement results of qubit 0 generated by the quantum virtual machine	45
5.4	A special case of gearbox circuit	45
5.5	Level-1 gearbox circuit implemented by recycling the same qubit as the ancilla qubit. The sequence of operations to the case of a single failure followed by correction and success	46
5.6	The experimental implementation of a RUS block	46
5.7	(a) The microwave setup used in CC-Light. (b) Proposed new microwave setup	50

LIST OF TABLES

2.1	Quantum circuit of some commonly-used quantum gates.	7
2.2	Overview of eQASM Instructions. The operator :: concatenates the two bit strings.	13
5.1	Measurement probabilities of the the theoretically achieved quantum state	47
5.2	Simulation result without errors	48
5.3	Simulation result with errors	48
5.4	Simulation time with and without comprehensive feedback control.	48

LIST OF ACRONYMS

QuMA	Quantum MicroArchitecture
QISA	Quantum Instruction Set Architecture
QEC	Quantum Error Correction
NISQ	Noisy Intermediate-Scale Quantum
QASM	Quantum Assembly Languages
NN	Nearest-Neighbour
QCU	Quantum Control Unit
FPGA	Field-Programmable Gate Array
SC7	Surface Code 7
SC17	Surface Code 17
CPU	Central Processing Unit
RTL	Register Transfer Level
VLIW	Very Long Instruction Word
SIMD	Single Instruction Multiple Data
SOMQ	Single Operation Multiple Qubit
VSM	Vector Switch Matrix
UVM	Universal Verification Methodology

ACKNOWLEDGEMENTS

First of all, I would like to express my appreciation to my thesis supervisor Prof. Koen Bertels for granting me the opportunity to do my master project in the field of quantum computing. He also gave me the opportunity to involve myself in a number of academic activities and discussions of the entire Quantum and Computer Engineering (Q&CE) department. It is my honour and luck to be able to complete my master project under his supervision.

I would like to thank my colleagues from the Quantum Computer Architecture Lab for all the discussions and fun moments. Special thanks to Xiang Fu for the great support and the detailed guidance on my master project. He offered me a lot of help at the beginning and inspired me at every stage of my work to help me complete this project in a field I am not familiar with. I also want to express my thanks to Leon Risebos, Nader Khammassi, Lingling Lao and Imran Ashraf for all the spontaneous discussions that help me better understand quantum computing to motivate the research. As a master student, I am able to participate in all academic discussions and meetings held in the lab, as well as those entertainment hours. I truly felt part of the team.

I would also like to thank Jeroen van Straten for providing me with guidance on VHDL advanced coding techniques. The DiCarlo group provided a very good working environment for my project, because my work is based on the real hardware. I am especially grateful to Brian Tarasinski for helping me understand the areas of research I have rarely understood before. Without the help from them, my project will be difficult to carry out.

I would like to take this opportunity to thank my fellow batchmates at TU Delft, Jian Zheng, Mingfeng Li, Saiyi Wang, Ye Li, Xiaoming Wen, for those cherished moments in the 10th floor of the faculty of EEMCS.

Finally, I would like to thank my beloved family, my girlfriend, and all the friends who have given me unconditionally love and support over the last two years. During my time studying abroad, I can always rely on them to overcome the bad times in this period.

Thank you all very much!

Mengyu Zhang
Delft, August 2018

1

INTRODUCTION

1.1. CONTEXT

Quantum computing has the potential to solve several computational problems that are intractable for classical computers. A quantum algorithm exploits quantum phenomena, namely superposition and entanglement, to achieve speedup over the best classical algorithm solving the same problem. The most famous examples are factoring large numbers using Shor's algorithm [1] and searching an unordered database using Grover's search [2]. These algorithms may require millions or even billions of physical qubits which is far from current technology. This difference causes a significant resource gap between practical quantum algorithms and real quantum computers.

Useful quantum computation would require a full stack architecture. To construct a universal and fully programmable quantum computer, clarifying the required component of a quantum computer is essential. In the QuTech quantum computer system stack [3], multiple layers from top level quantum algorithms to underlying quantum chips are included in a full stack quantum computer. In the middle of this system view, the Quantum Instruction Set Architecture (QISA) and the Quantum Microarchitecture layers play a key role in bridging the gap between quantum software and quantum hardware.

Nowadays, research in quantum computer engineering has focused primarily at devising high-level programming languages and compilers, and building reliable low-level quantum hardware. Relatively little studies have focused on how to control operations on experimental quantum processors. However, with Noisy Intermediate-Scale Quantum (NISQ) cite preskill2018quantum devices with 50 to hundreds of qubits coming soon, we need to start thinking about how to use the compiler output to fully control the quantum chip. To solve this problem, an experimental microarchitecture (QuMA) is proposed in [3] for controlling a superconducting quantum processor.

In our lab, two instances of QuMA are implemented using Field-Programmable Gate Array (FPGA). QuTech Control Box (CBox) targets controlling one or two qubits. The quantum control microarchitecture in CBox is QuMA, which can execute QuMIS instructions. Since there are only a few QuMIS instructions which are low-level and tightly bound to the hardware, the structure of QuMA is relatively simple. Addressing controlling more qubits and the inherent quantum operation issue rate problem of QuMA, we designed an executable QISA, eQASM, which is instantiated to 32-bit instruction set to control seven qubits and implemented by an updated version of the control microarchitecture, Central Controller-Light (CC-Light).

Compared to QuMA, the updated QuMA microarchitecture QuMA_v2 outputs a larger number of signals to orchestrate the behavior of underlying analog devices to control the seven qubits. The VLIW and SOMQ execution, and the selective broadcasting supported by Vector Switch Matrix (VSM), ask for more complex addressing logic and dynamic checking to avoid conflict operations. Comprehensive feedback control requires interaction between the classical pipeline and the quantum pipeline, and between the non-deterministic timing domain and the deterministic timing domain, which can stall the pipeline for an indefinite number of cycles depending on the quantum program. All these contribute to a higher complexity of control microarchitecture, which pose a big challenge in its design, development, and verification.

1.2. RESEARCH MOTIVATION

The current development flow of QuMA suffers from several drawbacks. At the design stage, we can only do a limited design space exploration (DSE) at abstraction level since we lack information of the inner structure of QuMA by using the diagram-based method. After design phase, we need to directly write HDL to describe the entire system which is very rigid since it has no flexibility. Then at the verification stage, it requires the hardware programmer to check the correctness of multiple signals in simulation for every eQASM program in the benchmark. All these drawbacks lead to a low-efficient and error-prone development flow.

Based on the experience with the design of classical processors, all the three steps (DSE, development, and verification) can be boosted via automation. Therefore, a new approach has been proposed to automate this procedure as much as possible. In this new method, we will perform a high-level modelling at first which can explore design possibilities at high level. Then a platform is used to do automated simulation and verification with low-level details. After we make sure this design can fulfill the requirements, it is finally implemented on hardware like FPGA.

The DSE can be performed via a configurable simulator for the quantum control microarchitecture, like simulators for classical processors such as gem5 [4]. Besides, the verification of the HDL implementation can be automated using the universal verification methodology (UVM) [5], which would require a simulator to reference values for the checking points for each input program. Thus, a dedicated simulator for QuMA is required to achieve the automation of the QuMA development flow.

1.3. RESEARCH APPROACH

A simulator that describe the target system at a relatively low level (preferably at register-transfer level) can help the automation of the DSE, development, and verification of QuMA. During the research period of the thesis, we designed a simulator called QuMASim to fulfill these requirements. We also built several applications based on QuMASim to exploit its potential to help the design and verification of QuMA. Finally, we did some experiments to prove the capability of QuMASim and demonstrate its use cases.

To achieve the goal of an automated development process, QuMASim needs to be implemented as a cycle-accurate and hardware-based simulator. There are several challenges in implementing such a simulator:

1. We want this simulator contains all low-level constraints of quantum microarchitecture that are exposed in the discussion of [3], and we are targeting verification over the VHDL implementation. Ensuring that the simulator can demonstrate all of the functions of proposed QuMA is challenging.
2. Since we need to implement this simulator using a low-level description, its flexibility is bounded to be limited. In order to explore the design space, we hope that the simulator can be automated for different configurations. How to make this simulator as configurable as possible will be a problem.
3. The simulation environment would also be a problem. First, we want all user could do easy simulation regardless of their operating systems. Second, there should be a method to communicate QuMASim simulation and HDL simulation, since one of our targets is to implement a verification platform.

QuMASim is built with respect to these challenges. In this project, we take CC-Light as a starting point to build the first version of QuMASim. QuMASim also takes the eQASM program as input and outputs quantum operations with their respective timing information. We have tested multiple eQASM programs and compare its results to CC-Light to check its correctness. The first version is also supposed to control a 7-qubit quantum chip, just like CC-Light. QuMASim can be used for design space exploration. Before implementing the VHDL description of a new QuMA, the simulator could easily test the new design and help us to compare different possibilities of the microarchitecture. Besides, QuMASim is developed to be self-configurable for different quantum processors, such as a Surface Code 17 (SC17) quantum chip. Inversely, now QuMASim can be used to verify the correctness of VHDL implementation by comparing the signals of interest from both simulations. In this work, an application called verification platform is built based on this purpose.

In addition, a quantum virtual machine is implemented on the basis of QuMASim and other quantum software/emulators. OpenQL[6] is a high-level quantum programming framework which has an eQASM back-end and generates executable code for the CC-Light. So the user can write his quantum algorithm in the OpenQL high-level language then compile it to generate the eQASM code. There are also some qubit state simulator like QX [7] and QuantumSim [8] which can perform the quantum simulation. In this work, we connect the QuMASim to QuantumSim so the generated operations could be executed on the error model

provided by QuantumSim. The measurement results could be sent back to QuMASim to realize feedback control. In this way, the correctness of quantum programs can be checked at both architecture level and qubit state level.

1.4. ORGANIZATION

This thesis is structured as follows:

Chapter 2 introduces the relevant background information for this thesis. It provides some basic knowledge of quantum computing and gives a broad view about a fully programmable quantum computer.

Chapter 3 discusses the implementation of QuMASim. In this chapter, we introduce the basic structure of this simulator and the main function of each module. We also show how to make this simulator work in a full stack system view, including assembler and instruction set architecture.

In chapter 4, the applications that are built based on QuMASim are introduced. We first discuss the modifications that we made for a self-configurable simulator. Then the working principles of the verification platform and quantum virtual machine are explained in detail.

Some experiments are performed on QuMASim and the results are reported in Chapter 5.

Finally, we conclude and give recommendations for future work in Chapter 6.

2

BACKGROUND

2.1. QUANTUM COMPUTING BASICS

2.1.1. QUBITS

Qubits are the fundamental elements of quantum computers just like classical bits in a physical system where classical bit has two exclusive states, 0 or 1 and can only be in one of them at a certain point in time. These two states are represented by $|0\rangle$ and $|1\rangle$. Qubits can be in a linear combination of these two states which can be represented by

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad (2.1)$$

where

$$\alpha, \beta \in \mathbb{C} \quad \text{and} \quad |\alpha|^2 + |\beta|^2 = 1. \quad (2.2)$$

Using the logical states as a basis, this superposition state can be represented by a complex-valued two-vector

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \quad (2.3)$$

The probabilistic nature of qubits presents themselves when being measured. The measurement probability of each state is the square of its amplitudes. Observing a qubit collapses its state to one of the possible measurement outcomes and destroys the information in the probability amplitudes. As a result, α and β are not directly observable.

In classical computers, n bits can only be in one of the 2^n possible states at a time. As a consequence, the entire system can only process this solo state read from the n bits at one time. The second feature of qubits extends the capabilities of classical bits and it is entanglement. Qubits can be entangled with each other which means that the state of the entire system cannot be described by the tensor product of each qubit state. An example of a non-entangled state is

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad (2.4)$$

this state can still be written as the tensor product of two individual states and is therefore not entangled. The state

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \quad (2.5)$$

is entangled since it can not be written as two individual states.

2.1.2. QUANTUM GATES

Quantum gates are used to manipulate the quantum state. These gates can be expressed as matrices and of which the effect can be modeled into matrix-vector multiplication. All quantum gates are arbitrary unitary matrices which means that quantum gates are always reversible. These operators can be simultaneously applied to the entire superposition which gives rise to quantum parallelism. On the other hand, superposition and entanglement provide an exponential state space as discussed before. Both facts will contribute to the speedup of quantum computers compared to classical computers.

Quantum gates could be divided into two types: single-qubit gates and multiple-qubit gates. A single-qubit gate can be represented as a 2×2 unitary matrix and is a rotation on the Bloch sphere. A simple example of single qubit gate is Pauli- X gate which rotate the qubit by π along the x axis. A few other basic single qubit gates are the Pauli- Y , Pauli- Z and Hadamard gate. Especially the Hadamard gate is interesting since it maps a computational basis state to a superposition state.

Multiple-qubit gates are essential for creating entanglement and a well-known example is controlled-NOT (CNOT) gate. CNOT transfers the basis state $|a\rangle \otimes |b\rangle$ to another basis state $|a\rangle \otimes |b \oplus a\rangle$. There are more two-qubit gates like Controlled-phase (CPhase) gate. Three-qubit gates also exist and one commonly-used example is the Toffoli gate, also known as the controlled-controlled NOT or CNOT gate. It is similar to CNOT gate but has two control qubits. Table 2.1 presents some of the useful quantum gates for this thesis.

Measurement of a state is inherently probabilistic depending on the amplitude of the vector in measurement basis. The quantum state collapses the wave-function to one of its classical eigenvalues. This is called projective measurement and is a significant difference from classical theories where measurement does not destroy the original state tested. Thus multiple measurements are required to reconstruct the probability distribution of the state. The exact complex amplitude cannot be measured, thus allowing a degree of freedom. This arbitrary phase factor can be ignored with no effect of the solution probability.

2.1.3. UNIVERSAL QUANTUM COMPUTATION

Like *NAND*, *NOR* gate sets in classical logic, there exists a universal set of gates on quantum logic that allows any arbitrary quantum gate to be decomposed with members of the set. An example of a universal set of quantum gates is *H*, *T*, *CNOT*. Any universal gate set should at least contain a multiple-qubit gate and a non-Clifford gate.

A universal quantum gate set could help realize universal quantum computation. Moving towards a broader view of constructing a useful quantum computer, several requirements need to be fulfilled given by DiVincenzo's criteria [9]:

1. A scalable physical system with well characterized qubits
2. The ability to initialize the state of the qubits to a simple fiducial state.
3. Long relevant decoherence times.
4. A universal set of quantum gates.
5. A qubit-specific measurement capability.
6. The ability to inter-convert stationary and flying qubits.
7. The ability to faithfully transmit flying qubits between specified locations.

2.2. FULL STACK QUANTUM COMPUTER

Figure 2.1 provides a high-level view of the quantum system stack. This stack consisting multiple layers. The bottom layers are quantum chip together with quantum to classical interface. These two layers are technology dependent and many implementations focus on improving the fragility of qubits. Different technologies have been investigated like superconductors [10, 11], trapped ions [12] and nitrogen-vacancy centers [13]. Among these technologies, superconducting qubits give the biggest potential for scalability so the rest of paper only focus on this technology. Besides physical qubits, logical qubits also draw attention with performance improved by quantum error correction. But this is out of the scope of NISQ to be discussed, so we will only pay attention to the noisy qubits in this thesis.

The middle layers are the control microarchitecture and Quantum ISA. QISA provides a interface to communicate software and hardware just like in classical architectures. QISA usually includes both quantum

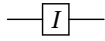
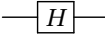
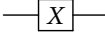
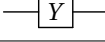
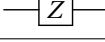
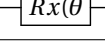
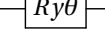
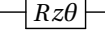
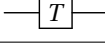
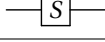


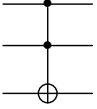
Gate Name	Gate Circuit	Gate Unitary
Identity		$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Hadamard		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Pauli-X		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Rotation-X		$\begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$
Rotation-Y		$\begin{bmatrix} \cos \frac{\theta}{2} & \sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}$
Rotation-Z		$\begin{bmatrix} e^{-i \frac{\theta}{2}} & 0 \\ 0 & e^{i \frac{\theta}{2}} \end{bmatrix}$
T		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i \frac{\pi}{4}} \end{bmatrix}$
Phase		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
CNOT		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
CPhase		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
Toffoli		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 2.1: Quantum circuit of some commonly-used quantum gates.

instructions and classical instructions since a quantum computer will always consist of both computing components [14]. These instructions are fed into a control microarchitecture and decoded into required control signals with precise timing. These control signals are processed by quantum to classical interface based on the specific quantum technology. These signals are translated into required pulses and sent to quantum chip.

The top layer represents quantum algorithms that are described by high-level quantum programming languages like Scaffold [15] and LIQUi|> [16]. These algorithms lack the low-level hardware information and assumes that all operations can be executed perfectly. Compilers help to provide these information so the quantum algorithms can be executed by the potential quantum hardware. In compilation layer, these algorithms are compiled into a series of instructions that defined by QISA.

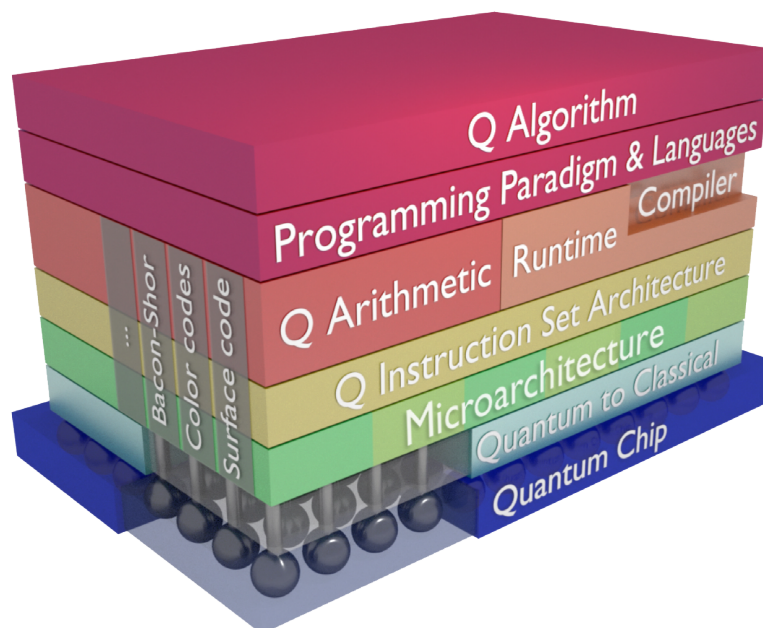


Figure 2.1: Overview of quantum computer system stack

2.3. HARDWARE

2.3.1. SUPERCONDUCTING QUBITS

Nowadays, there are various potential quantum technologies being pursued to implement qubits, such as trapped ions [12, 17], quantum state in superconducting circuits [10, 11, 18], nuclear spins or electron spins in silicon [19–21], and nitrogen-vacancy centers [13, 22]. Other candidates like Majorana fermion based topological qubits (not yet built) are actively being researched as well. The currently most promising technologies are trapped ions and superconducting qubits, both of which have demonstrated to satisfy the DiVincenzo criteria [10, 11, 17, 23].

As stated before, this thesis focus on superconducting qubits. Superconducting integrated circuits are Josephson junction based harmonic oscillators coherently controllable and measurable by magnetic flux pulses and microwaves. The performance of superconducting qubits benefits from the non-linearity of Josephson junctions and surrounding microwave circuitry. These systems have potentially excellent scalability since this technology have not encountered any hard physical limits. Superconducting qubits are usually fabricated with well-established fabrication techniques such as photo and electron-beam lithography. However, superconducting qubits also suffer from drawbacks like low coherence times. This also makes quantum error correction very important for this type of system [24–28].

Transmon [29] is a promising technology which achieves error rates lower than the fault-tolerance threshold for surface code. A universal gate set which comprised of single-qubit gates (mainly x and y rotations) and the CZ gate is used at here. The transmon is a lumped-element nonlinear LC resonator. The first-excited state is used as the qubit $|0\rangle$. The transition frequency between these state can be tuned by controlling the flux through the loop between the two Josephson junctions.

Figure 2.2 shows a prototype seven-port transmon developed in Dicarlo lab, Delft University of Technology. The vertical I/O will be realized either using through-silicon through-silicon-vias or bump bonding in a flip-chip arrangement.

For transmon qubits, single-qubit gates are executed by applying calibrated microwave pulses (typically 20 ns) through microwave-drive lines. These pulses are commonly generated by single-modulation of a carrier using functions generated by an Arbitrary Waveform Generator (AWG). The phase of the carrier determine the axis of rotation operations and the amplitude decides the rotation angle. Some predefined calibrated microwaves are stored and employed for specific pulses (e.g., π and $\pi/2$). Arbitrary single-qubit gates can be decomposed into x - and y -axis rotations so all quantum operations could be applied by this methodology although it will cost longer operation sequences.

The most frequently used two-qubit gate is CZ gate. Such a gate can be performed between qubits coupled

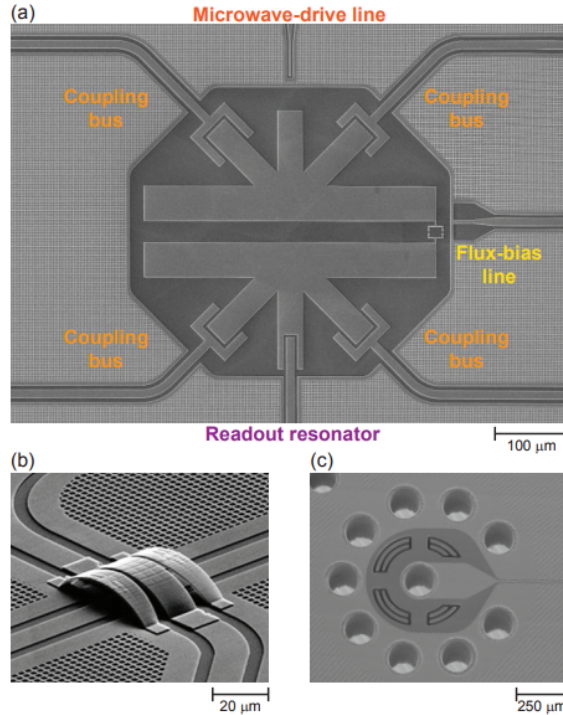


Figure 2.2: Images at various scales of implementation of surface-code fabric. (a) Starmon qubit. (b) Transmission-line crossover. (c) Vertical I/O.

to a common resonator or capacitor. It is realized by applying calibrated flux pulses (usually 40 ns) to the flux-bias line. If the frequency of these two qubits are calibrated suitably, they could be coupled together for the CZ gate.

Qubit measurement exploits qubit-state dependent fundamental frequency of a resonator which is coupled both to the qubit and to a feedline. Feedlines consist of readout resonators which are simultaneously interrogated using frequency-division multiplexing. A pulsed measurement usually lasts for 300 ns - 2 μ s and is transferred through the feedline to project the qubit state to $|0\rangle$ or $|1\rangle$. Demodulation, integration, and discrimination of the transmitted signal is used to generate the measurement result.

2.3.2. QUANTUM CHIP TOPOLOGY

The surface code is a promising topological QEC code and attracts a lot of experimental research [30]. Although we do not consider about error corrections in NISQ era, the fabricated quantum chip is targeting to the future technology. A surface code using 7 qubits which is also known as Surface Code 7 (SC7) can form a logical qubit and is used for testing in this thesis. The example Figure 2.3 shows the topology of a quantum chip of SC7. As we can see from this layout, the SC7 processor uses four data qubits (in red) and three ancilla qubits (in green and blue). The ancilla qubits are used to check X/Z parity between interact to the data qubits.

The surface code has convenient two-dimensional layout with only nearest-neighbor qubit interactions. For example, qubit 2 and 3 can not be applied a two-qubit gate directly since they are not connected. This problem could be solved by using qubit 0 or 5 to swap the information. This is related to the mapping of quantum circuits and will not be further discussed in this thesis since it is not part of our work.

For control simplicity, we named each qubit in the processor and each direct edge that connects nearest qubit pairs. For example, edge number 0 represents the edge which connects qubit pair 2 - 0 while qubit 2 acts as left qubit (control qubit) and qubit 0 is right qubit (target qubit). There are 7 qubits and 16 edges for this SC7 chip and these information should be stored in the controller of this chip. These numbers will scale up as the number of qubits increases, and the method to solve this problem will be discussed later in section 4.

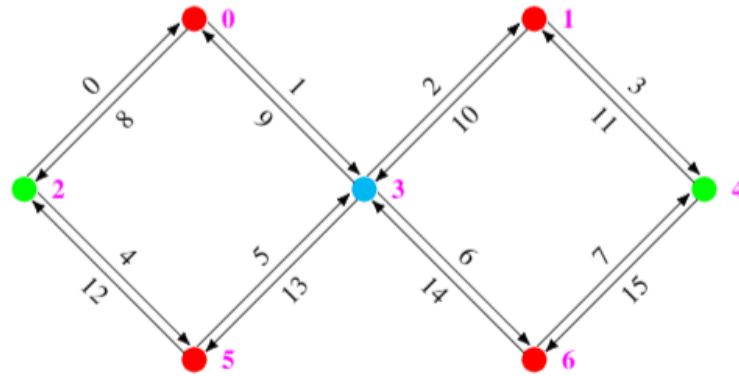


Figure 2.3: Layout of a surface code using 7 qubits

2.3.3. IMPLICATION ON THE CONTROL

In quantum computing, the data is stored and processed in qubits. To perform quantum operations, analog pulses are generated by an external controller and then applied on qubits. In contrast to classical computing, the operations are transferring around while the data is stationary on computing bits. In classical processors, data is stored and processed in binary format and the digital system share the homogeneity among different subsystems. But data in quantum computing could be entangled in qubits so the data evolves in a format encoded in superposed states. Thus, the difference between quantum data and control signals leads to the separation of quantum controller and quantum processor. A quantum controller usually use AWGs to generate the required pulses for the targeting qubits. Figure 2.4 displays how the control unit communicates with quantum chip.

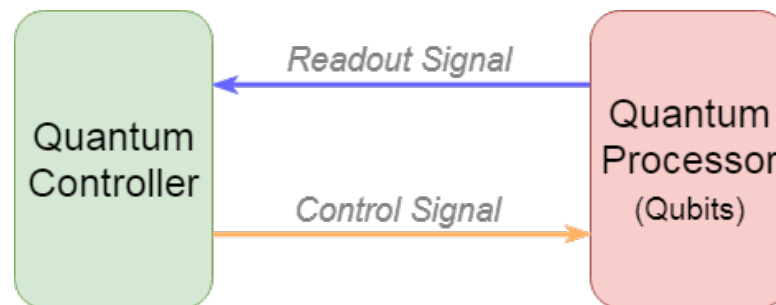


Figure 2.4: The relationship between control unit and quantum processor

A more detailed implementation of quantum controller which targets to a SC17 chip consisting of transmon qubits that we have introduced before is shown in Figure 2.5 [31]. Each qubit has a dedicated flux line, a microwave-drive line and a readout resonator for readout. These 17 dispersively-coupled resonators are grouped onto three different feedlines and two-qubit operations are applied through bus resonators by opting for coupling nearest-neighbor data and ancilla transmons. The microwave-drive lines and flux-bias lines are driven by AWGs and the feedlines are driven by UHFQC. The information about these control electronics will not be elaborated at here but we do notice that there are some limitations brought by them. How to handle these control electronic specifications will be discussed in later sections.

2.3.4. QUANTUM MICROARCHITECTURE

In our discussion, quantum computer architectures comprise two parts: Quantum Instruction Set Architecture (QISA) and Quantum Microarchitecture (QuMA). These two layers bridge the gap between software and hardware in quantum computing. Currently, quantum processors are controlled by well-defined electrical signals like microwave pulses with precise timing. To generate these signals, some dedicated electronics devices are used to control the quantum chips. However, this control methodology suffers from high resource consumption, control complexity and low scalability. Therefore, it is necessary to design a microarchitecture to improve the performance of controller.

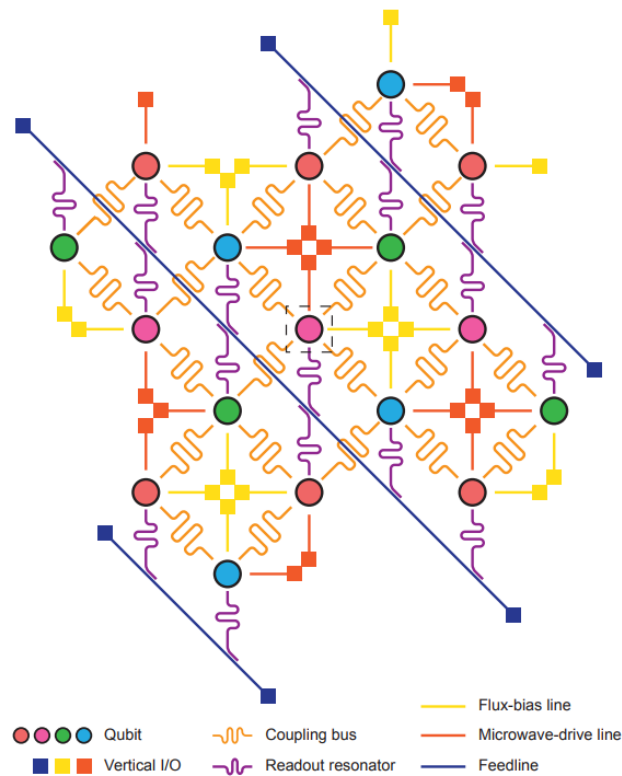


Figure 2.5: Schematic of how to control a Surface-17 chip

The main challenges in designing such a control microarchitecture are listed below:

- **Complex analog waveform control.** The input signal of quantum processors are complex analog signals since quantum operations are performed by sending well-defined analog pulses. The measurement outcome of qubits also resides in the output analog signals which means the output signal of quantum processors are also analog signals. A commonly used method to produce the required pulses uses arbitrary waveform generators (AWGs). The well-defined analog pulses are calibrated and placed in the memory of these devices before executing quantum algorithms.

Measurement results are contained in an analog signal and a discriminator is required to perform integration and discrimination on this signal. The software-based discriminator suffers from problems like long latency which makes real-time feedback control impossible. Thus, a scalable measurement discrimination method with short latency constitutes a challenge.

- **Instruction definition.** The quantum instruction set architecture (QISA) is the dividing line between quantum hardware and quantum software. It is challenging to design an instruction set that considering the low-level constraints of the interface to the quantum processor, e.g., timing information. It is difficult for quantum compilers to generate technology-dependent instructions for the following reasons:

1. The constraints of quantum processor will affect the layout of qubits used in the algorithm, e.g., the nearest-neighbour constraint.
2. Quantum technology is rapidly evolving and new methods of implementing quantum gates are continuously being introduced. It is important to find a way to introduce these changes without affecting the rest of architecture.
3. It is difficult to support a set of technology independent quantum instructions and their current state of the art.

A quantum microarchitecture, QuMA [3], for a superconducting quantum processor is presented to solve these problems. The input of QuMA is a binary file generated by a compiler where classical and quantum code are combined. The entire architecture uses a heterogeneous model and classical code is executed by the classical host CPU. The quantum code contains auxiliary classical instructions and quantum instructions, and is executed by the quantum co-processor. More detailed introduction about it will be displayed in Section 3.1.

2.4. QUANTUM INSTRUCTION SET ARCHITECTURE

2.4.1. QISA OVERVIEW

QISA is an essential component of a fully stack quantum computer since it is the dividing line between hardware and software. To achieve that, QISA target to be mathematically equivalent to the circuit model in an efficient way, with or without interacting with classical computing resources. Most proposed quantum assembly language like f-QASM [32]. However, they do not fully take into account the low-level constraints to interface with NISQ processors. For example, a generally neglected but crucial constraint in the NISQ era is the timing of operations, which may ruin the computation process when not effectively scheduled. Another example of a low-level constraint is that only a limited number of different operations can be applied on qubits at the same time with some control electronics. Besides, quantum-classical heterogeneous computing is required by some quantum algorithms, such as variational quantum eigensolver. But existing instruction sets do not either define an heterogeneous programming model or reveal enough information about the underlying programming model. Therefore, an executable QASM-based QISA, named eQASM, is proposed in [33]. We will use this instruction set in the rest of this thesis.

2.4.2. EQASM

The design of eQASM is guide by several main principles. eQASM should contain well-defined methods to specify the timing of quantum operations. It should be simple to allow a straightforward implementation by the microarchitecture. The quantum operation issue rate is a potential bottleneck of the quantum microarchitecture and should be addressed, e.g. by densely encoding the instructions such as done with SIMD and VLIW for classical architectures. In addition, low-level hardware information should be included in a configuration file and exposed in eQASM to enable platform-specific optimization.

Based on the the principles described above, the characteristics of eQASM are listed below:

- **Efficient timing specification:** eQASM defines a more efficient method to explicitly specify the timing of quantum operations at the instruction level.
- **SOMQ execution:** Addressing the quantum operation issue rate problem, eQASM introduces Single-Operation-Multiple-Qubit (SOMQ) execution, which can considerably increase quantum code density.
- **VLIW architecture:** eQASM adopts a Very-Long-Instruction-Word (VLIW) architecture to further increase the quantum operation issue rate; a single quantum instruction can contain multiple quantum operations.
- **Runtime feedback:** eQASM defines two kinds of feedback: fast conditional execution for simple but fast feedback, and comprehensive feedback control for arbitrary user-definable feedback.
- **User-configurable quantum operations.** eQASM postpones the specification of available quantum operations in the instruction set from QISA design time to compile time. The operations can be configured by the programmer.
- **Supporting quantum experiments.** Tailored for the NISQ technology, eQASM can be used to efficiently describe quantum algorithms as well as quantum experiments to calibrate qubits and quantum operations.

Table 2.2 gives an overview of eQASM instructions.

The functions of these instructions are introduced as following:

- **Control instructions.** Compare and branch instructions will consist of control logic of eQASM. CMP is used to compare the values of two registers. The result is stored in the branch register which is represent as B . The status of condition is represented by three flag bits, and the computation of these three flag bits are listed below:

Table 2.2: Overview of eQASM Instructions. The operator `::` concatenates the two bit strings.

Type	Syntax	Description
Control	<code>CMP</code> <i>Rs</i> , <i>Rt</i>	Compare GPR <i>Rs</i> and <i>Rt</i> and store the result into the comparison flags.
	<code>BR</code> <code><Comp. Flag></code> , <code>Offset</code>	Jump to <code>PC + Offset</code> if the specified comparison flag is '1'.
Data Transfer	<code>FBR</code> <code><Comp. Flag></code> , <i>Rd</i>	Fetch the specified comparison flag into GPR <i>Rd</i> .
	<code>LDI</code> <i>Rd</i> , <i>Imm</i>	$Rd = \text{sign_ext}(\text{Imm}[19..0], 32)$.
	<code>LDUI</code> <i>Rd</i> , <i>Imm</i> , <i>Rs</i>	$Rd = \text{Imm}[14..0]::\text{Rs}[16..0]$.
	<code>LD</code> <i>Rd</i> , <i>Rt</i> (<i>Imm</i>)	Load data from memory address $Rt + \text{Imm}$ into GPR <i>Rd</i> .
	<code>ST</code> <i>Rs</i> , <i>Rt</i> (<i>Imm</i>)	Store the value of GPR <i>Rs</i> in memory address $Rt + \text{Imm}$.
	<code>FMR</code> <i>Rd</i> , <i>Qi</i>	Fetch the result of the last measurement instruction on qubit <i>i</i> into GPR <i>Rd</i> .
Logical	<code>AND/OR/XOR</code> <i>Rd</i> , <i>Rs</i> , <i>Rt</i>	Logical and, or, exclusive or, not.
	<code>NOT</code> <i>Rd</i> , <i>Rt</i>	
Arithmetic	<code>ADD/SUB</code> <i>Rd</i> , <i>Rs</i> , <i>Rt</i>	Addition and subtraction.
Waiting	<code>QWAIT</code> <i>Imm</i>	Specify a timing point by waiting for the number of cycles indicated by the immediate value <i>Imm</i> or the value of GPR <i>Rs</i> .
	<code>QWAITR</code> <i>Rs</i>	
Target Specify	<code>SMIS</code> <i>Sd</i> , <code><Qubit List></code>	Update the single- (two-)qubit operation target register <i>Sd</i> (<i>Td</i>).
	<code>SMIT</code> <i>Td</i> , <code><Qubit Pair List></code>	
Q. Bundle	<code>[PI,] Q_Op [Q_Op]*</code>	Applying operations on qubits after waiting for a small number of cycles indicated by <i>PI</i> .

```

1 X = unsigned(Rs) >= unsigned(Rt)
2 Y = unsigned(Rs) > unsigned(Rt)
3 Z = sign(Rs) ^ sign(Rt)

```

We can obtain the condition by implementing logic operation on the flag bits. For example, the values in two registers are equal when

```

1 X * ~Y = 1

```

Branch is used to determine whether the classical controller should execute instructions in sequence or jump to a specific point. The program counter will load the address of a specific instruction if the condition matches branch register. Otherwise it will increment itself to load the next instruction. A mathematical description is given below:

```

1 PC = branch register matches condition ? specific address : PC + 1

```

- Data transfer instructions. These instructions are used for common-purpose load operation. `LDI` is used for signed numbers and `LDUI` for the unsigned. For the signed operation, the immediate number will be loaded to the target register. Notice that the value of register needs to be sign-extended to ensure the correct result. For the unsigned operation, the immediate number is loaded into the upper bits of target register while its lower bits is determined by the resource register. Fetch branch register (`FBR`) is used for load operation with specific purpose. The `FBR` instruction will set the target register to 1 or 0 based on the branch condition. Fetch measurement result (`FMR`) is more complex and is used to fetch the result of the last measurement operation. This instruction helps realize comprehensive feedback control logic which is introduced in section 3.1.4.
- Logical and arithmetic instructions. These instructions are simple and straightforward: they are used for logical and arithmetical operations. All these instructions have two operands which are stored in *Rs* and *Rt*, and the result will be put into destination register (*Rd*).
- Quantum waiting instructions. These two instructions are used to specify the interval between previous timing point and the following timing point. The supported waiting time range is from 0 to $2^{20} - 1$. In the `QWait` instruction, you can use the 20-bit immediate value to specify the waiting time. In the `QWaitR` instruction, you can specify the general purpose register *Rs*, of which the low 20 bits are used as the waiting time. Zero waiting time is also supported in the hardware, of which the effect is nothing.
- Target specify instructions. In `QuMA`, the target of a quantum gate is specified by a mask. A mask is firstly stored in a register and then readout by the quantum operation instruction. There are two

register files used to store the masks for single-qubit gates and two-qubit gates, which are labeled with "S" and "T", separately. The instruction SMIS and SMIT are used to set the mask for single-qubit gates and two-qubit gates, respectively.

- Quantum bundle instructions. Bundle is used for specifying quantum instructions which contain quantum operations. In current QuMA, each bundle contains two VLIW instructions and can be processed by VLIW pipelines in quantum pipeline. The definition of quantum operations is flexible since we want this instruction set as technology independent as possible.

2.5. COMPILER AND LANGUAGE

Quantum algorithms should be described by specific languages and compiled by quantum compiler. To enable quantum programmers to express complex algorithmic constructs, a high-level programming language is needed. A number of programming languages exist, such as Scaffold [15], Quipper [34] and LIQUi| [16]. All these programming languages mostly target large-scale quantum computation, with little consideration in low-level constraints required by nowadays and near-term quantum devices. Microsoft Quantum Development Kit is using the recently proposed Q# [35], a quantum-focused domain-specific programming language, which highlights a heterogeneous quantum computing model which can safely interleave classical and quantum computations. These high-level quantum programming languages with compiler support have been proposed to efficiently describe quantum applications.

The compiler bridges the semantic gap between the quantum algorithm and the quantum hardware. The high-level quantum algorithms are translated into quantum operations and produce executable machine code for the target quantum platform. Quantum compilers usually provide several functionality such as synthesis of quantum circuits. Quantum circuit are reversible and has to be optimally decomposed in a series of quantum gates which belonging to a universal gate set. Compilers also need to consider quantum error correction. Logical qubits are encoded into several physical qubits and QASM instruction for performing operations on such encoded qubits are generated. Additionally, the compiler also maps the quantum circuit to the topology of the real quantum processor. In high-level algorithms, the circuit description does not usually consider physical limitations such as nearest-neighbour constraint. It is therefore important to optimize the mapping process. The QASM based quantum instructions are then translated into eQASM which also include the timing and physical qubit identifiers. These instructions are sent to the microarchitecture for execution on the quantum processor.

In this project, we use the OpenQL framework as the compiler to convert the quantum algorithms to microarchitecture executable QASM programs. The OpenQL framework is under development at QuTech, which allows hybrid quantum-classical coding in high-level programming languages like Python or C++. A more detailed description of OpenQL can be found in section 4.2.2.

3

SIMULATOR IMPLEMENTATION

In the previous chapter, we saw that a controller microarchitecture is required to solve the control challenges in full stack quantum computer. In this chapter, we first discuss the structure of an instance of QuMA which is named CC-Light, then we present the QuMA simulator, QuMAsim. The first version of QuMAsim is based on CC-Light, then further improvements are made.

3.1. QUMA STRUCTURE

3.1.1. OVERVIEW

As stated in Section 2.3, there are many control difficulties need to be solved for quantum computer. The first version of QuMA is proposed in [3] and the diagram is showed in Figure 3.1. QuMA is a heterogeneous architecture which includes a classical CPU as a host and a quantum co-processor as an accelerator. QuMA accepts a binary file generated by the OpenQL compiler infrastructure [6]. These instructions are processed and generate micro operations at a deterministic timing. The analog-digital interface converts digital signals into corresponding analog pulses which will perform quantum operations on qubits.

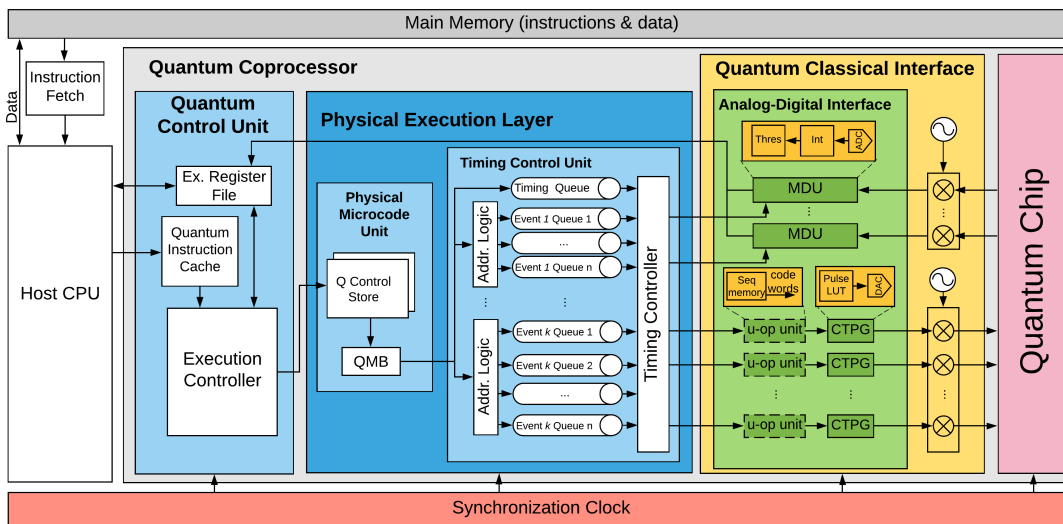


Figure 3.1: Overview of the structure of QuMA_v1

The main mechanisms used in this architecture are:

1. Codeword-based event control. The analog-digital interface divides the QuMA into digital signals and analog signals. From left to right, the codeword-based triggers are translated into pulses representing quantum operations on the qubits. From right to left, readout waveform is digitized and processed to discriminate the measurement outcomes as binary results by the measurement discrimination unit.

Therefore, this mechanism allows controlling analog pulse generator using instructions. Fast and flexible feedback control is enabled since the waveform are not required to be uploaded during runtime.

2. Queue-based event timing control. The timing control unit implements queue-based event timing control in QuMA. It divides QuMA into two parts: on the left side is the non-deterministic timing domain and on the right side is the deterministic timing domain. In the non-deterministic timing domain, all control operations are executed in an as-fast-as-possible style. In the deterministic timing domain, quantum operations are stored in the queues and are emitted with specific time stamp. So the output of the timing control unit have very precise timing information.

The timing control unit consists of a timing queue and multiple event queues. The time stamps are stored in timing queue and a clock is used to control the read request of this queue. The counter value is the intervals between two consecutive timing points and a time stamp is read out when the counter countdown to zero. At that time, this time point is broadcast to all event queues and every buffered events with corresponding time stamp is emitted to the analog-digital interface.

3. Multilevel instruction decoding. QuMA also focus on flexibility of instruction decoding. In quantum pipeline, the initial binary instructions are decoded into codeword-based events before the timing control unit. By enabling multilevel instruction decoding, the quantum instruction can fill the queues as fast as possible without worrying about complex analog waveform control with rigid timing constraints. This scheme is implemented through multiple modules and will be introduced in later stage.

These mechanisms help to realize the quantum operation execution. Besides it, there is also a quantum control unit in this QuMA which is dedicated to the classical control of the entire quantum program. This unit consists of classical pipeline, measurement register file and instruction cache. All binary instructions are stored in the instruction cache before execution and then are fetched by classical pipeline. The classical instructions are processed inside the classical pipeline and the quantum instructions are sent into quantum pipeline. The measurement results are stored in the measurement register file which is used to achieve comprehensive feedback control. The detailed functions of quantum pipeline and the classical control unit will be explained in the following sections.

To control a larger number of qubits, multiple new techniques are used in the second version of QuMA which is showed in Figure 3.2. Very-Long-Instruction-Word (VLIW) architecture and Single-Operation-Multi-Qubit (SOMQ) execution are used to address the quantum operation issue problem. Besides, the comprehensive feedback control logic is implemented so we use the measurement results to do some conditional operation. We will discuss the details of this architecture in the following sections.

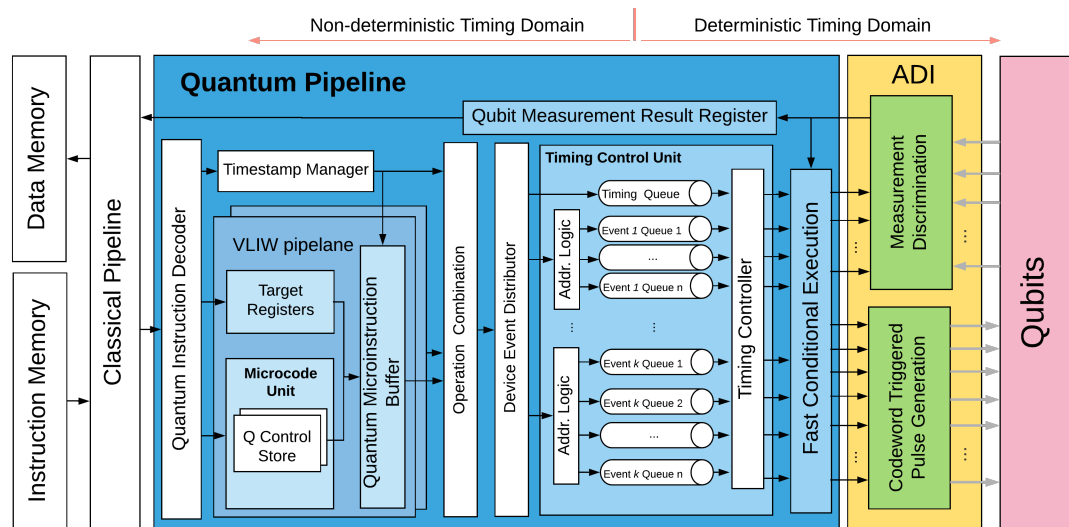


Figure 3.2: Overview of the structure of QuMA_v2, it is more complex compared to QuMA_v1 since it needs to control more qubits

3.1.2. QUANTUM PIPELINE

The quantum pipeline consists of multiple modules that help to realize the above schemes. Generally, the quantum instructions are processed in this pipeline and generate quantum operations with precise timing information for the analog-digital interface. It also registers the measurement results sent back from analog-digital interface and report them back to classical pipeline. In the following paragraphs we discuss the main function of each module inside quantum pipeline:

- **Quantum Instruction Decoder:** this module decodes the binary instructions. There are two kinds of quantum instructions: mask instructions, operation instructions and wait instructions. Mask instructions are used to define the mask registers which consist of single-qubit mask and two-qubit mask. Operation instructions contain the quantum operations and their targeting qubits. The targeting qubits information is stored in corresponding mask registers and will be read out in the following stage. Every operation instruction contains timing information as well as wait instructions. As long as an interval time is defined in these instruction, a valid signal will be set for the following module to generate a time stamp. For operation instructions, the opcode is also decoded from it binary and will be used as index of control store which will be explained later.
- **Mask Register File:** for mask instructions, the mask register will write the value decoded from binary instruction. For operation instructions, the value of targeting register will be read out. There are two kinds of register in this module, single-qubit register and two-qubit register. For single-qubit register, the mask value is encoded as the qubit number while the edge number information is encoded in two-qubit register.
- **Timestamp Manager:** as mentioned in quantum instruction decoder, a time stamp is generated for every instruction as long as a timing difference is defined. Notice that this manager is outside the VLIW pipeline since the VLIW instructions from one single operation instruction share the same time stamp. Besides the time stamp for operations, there are also some timing points are generated. These timing points are directly sent to the timing queue in the timing control unit while the time stamps are grouped with corresponding quantum micro-operations in operation combiner.
- **Microcode Unit:** this microcode method is introduced in [36] and it enables flexible complex instruction definition using the same hardware implementation. In microcode unit, the quantum instructions are translated into a sequence of microinstructions based on the microprograms uploaded into the Q control store. The micro operations are codeword-based as stated before so the content of the control store is based on the opcode and left/right codewords.

There is an address decoder submodule which works in parallel with control store. The mask value is fed into this decoder to generate the operation selection signal. This signal is used to select the left/right operation of the two-qubit pair and is sent to quantum microinstruction buffer together with the left and right microinstructions.

- **Quantum Microinstruction Buffer.** In this module, quantum microinstructions for quantum gates are decomposed into separate micro-operations. The input microinstructions are multiplexed based on the selection signal. Together with microcode unit, the multilevel instruction decoding scheme is realized through a opcode - microinstruction - micro-operation path.
- **Operation Combination.** After VLIW pipeline, the micro-operations from different VLIW instructions should be combined. In this module, a conflict check is applied at first to see if any operation violation on qubits. These operations are aligned together after passing this conflict check. Notice that the pre-interval of zero is supported at here, which means the operations will pend on this module until the next instruction arrived. And these two consecutive instruction will align again if the pre-interval of the second instruction is zero. This module is also the place that the time stamps are grouped together with micro-operations.
- **Device Event Distributor.** The output of the previous module are the quantum operations for each qubit. But in real control these operations should be grouped into different events for different control electronics as mentioned in Section 2.2.4. All quantum operations are divided into three types at here: microwave, flux and measurement. All input operations are translated into different events based on the information of control electronics like AWG and UHFQC. Then these events are pushed to the

event queues in the timing control unit. Device event distributor divides the entire QuMA into control electronics independent domain and control electronics dependent domain.

- Timing control unit. Timing control unit is used to realize the queue-based event timing control. The function of this module is described above.

3.1.3. CLASSICAL PIPELINE

The above section introduced the function of quantum pipeline and explained how quantum instruction is executed. Now we will discuss about the classical control a little bit. The structure of the quantum control unit is shown in Figure 3.3. The functions of these units are listed below.

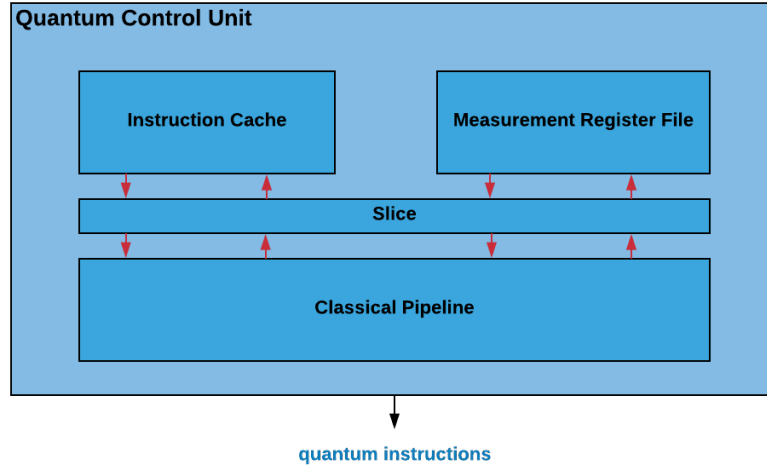


Figure 3.3: Quantum control unit structure

- Instruction Cache: in current CC-Light design, the instruction cache is not really a cache yet. It is just a memory where all binary instructions generated from the quantum program are stored. But in future design where a heterogeneous architecture will be implemented, this memory is supposed to be implemented as a cache between the classical pipeline and host CPU. There is a Program Counter (PC) which indicates the position of current instruction. The PC increments itself during usual time unless a branch takes place. When the branching signal from classical pipeline is set, the PC value changes to the targeting address and the corresponding signal is read out. When branching finished, there is also a branch done signal to indicate the classical pipeline recover from the stall and continue to process.
- Classical Pipeline: classical pipeline is implemented as a four-stage pipeline where the stages are: instruction fetch, decode, execute and write back. The main function of instruction fetch stage is to control the branch signal. The pipeline stop running when the previous instruction was a taken branch and start running when the instruction cache reports that a branch command was serviced. The branch target address will also be calculated and sent to instruction cache at here.

In the decoding stage, the binary instruction is decoded. We know that eQASM supports some auxiliary classical instructions like compare and logical operations and these instructions are used for classical control. If the instruction is a quantum instruction, it will be forwarded to quantum pipeline directly. The stalling logic also locates in decode stage. There are three stalling situations: stall when quantum measurement registers are not ready for more measurement instructions pending in the queue, stall when the quantum pipeline is not ready for more instructions, stall when we are executing an FMR instruction and the ready signal is low. The first situations are easy to understand and the last one is due to the feedback control logic and will be explained later.

After the decoding stage, the source (S) and target (T) operands and operation type are generated for the execution stage. The basic operations (AND, OR, ADD, etc) and compare operations (CMP, TEST)

are executed in this stage. Then the results are sent to write back stage if the result needs to be write to a classical register. Thus, the execution of classical pipeline is finished.

- Measurement Register File: this file contains the registers which are used to store the measurement results which are sent back from analog-digital interface. These values can be fetched by FMR instruction and used for conditional execution. The detailed logic is much more complex and is explained in the following section.
- Slice: the slice is a configurable register slice that can be placed between sub modules to improve FPGA timing. Any number of these units can be inserted. There are two kinds of register slice that can be inserted: ready slice and data slice.

3.1.4. COMPREHENSIVE FEEDBACK CONTROL

Figure 3.4 shows the logic of comprehensive feedback control in QuMA_v2. This logic starts from classical pipeline when a measurement instruction is processed. The measurement instruction is sent into quantum pipeline and finally generates a measure trigger for the measurement device. Inside the quantum pipeline, a measurement issue signal "Measissue" and a measured qubits signal "MeasuredQubits" are generated. These signals indicate that there are some measurement operations applied on certain qubits during the execution and are sent to the measurement register file.

When the measurement triggers the measurement device, it would generate corresponding analog waveform to apply measurement operations on quantum processor, which usually take $300ns$ to $2\mu s$. The measurement device collects the results and return them to QuMA. The results are sent into the measurement analysis module where the measurement result and valid signals for each qubit are sorted. Then these two signals will be sent into measurement register file and can be fetched by classical pipeline.

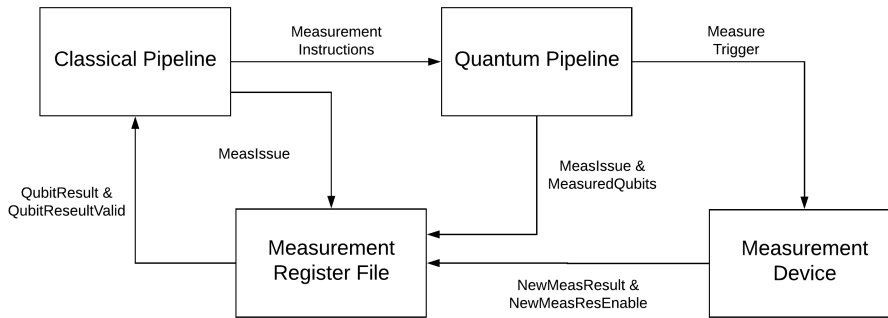


Figure 3.4: Comprehensive feedback control logic

In eQASM, a fetch measurement result (FMR) instruction is used to fetch the measurement result from measurement register file to a general purpose register file. Then this result can be used for conditional operation like compare and branch. The measurement instruction does not stall the microarchitecture but this instruction will.

The measurement result of a measurement instruction can only be returned after a period of time since it is issued from the classical pipeline to the quantum pipeline. This latency can range from hundreds of nanosecond to several microseconds due to three factors:

- Quantum pipeline requires several processing stages to translate the measurement instruction into corresponding measurement events. It requires around 7 cycles.
- Measurement events are buffered in an event queue in the timing control unit before its execution. It can introduce an indefinite latency depending on previous quantum instructions, which latency can range from tens of nanosecond to several microseconds.
- After the measurement events are fired from the timing control unit and sent to the measurement device, e.g., UHFQC, it takes a period of time for the measurement device and the qubits to accomplish the entire measurement process. It can introduce a latency from 400 ns to 3 us.

The existence of this latency means that, the `FMR` instruction cannot directly get the measurement result of the last measurement instruction for the target qubit in many cases. So, the microarchitecture must handle the latency introduced by the measurement process, by stalling the execution of `FMR` instruction when required.

To support comprehensive feedback control, a counter C_i is attached to each qubit measurement result register Q_i , with an initial value of 0. Once a measurement instruction acting on qubit i is issued from the classical pipeline to the quantum pipeline, C_i increments by 1. If the measurement discrimination unit writes back a measurement result for qubit i , C_i decrements by 1. Q_i is valid only when C_i is 0. If C_i is not 0 when the instruction `FMR Rd, Qi` is issued, the pipeline is stalled until C_i is 0. In this way, it is ensured that the instruction `FMR Rd, Qi` always fetches the result of the last measurement instruction acting on qubit i .

3.2. SIMULATOR IMPLEMENTATION

As stated in research motivation, we need a QuMA simulator to leverage the complexity of VHDL debugging. In this section, we first explain the goals and requirements of the simulator, and then we describe the implementation details.

3.2.1. GOALS AND REQUIREMENTS

To get a clearer view of the motivation of QuMASim, we need to understand the QuMA development flow at first. Whenever the underlying analog devices and quantum processor evolve, the design of the quantum control microarchitecture can be modified accordingly. The development flow of QuMA can be summarized in three steps:

1. Design: We need to propose an overall design idea for QuMA after realizing its requirements. These requirements can be summarized by instantiating QISA restrictions and specifying the control electronics setup for the target quantum chip. This design should be specified as a high level block diagram to demonstrate its functioning.
2. Development: During this stage, we explain the details of each module and describe the designed microarchitecture at Register-Transfer Level (RTL) using a hardware description language (HDL), such as VHDL.
3. Verification: The goal of verification is to make sure the HDL implementation of QuMA has no errors before the FPGA synthesis stage. The current methodology consists in running a set of eQASM programs in simulation tools like Questasim [37]. The waveform of all signals of interest will be displayed by this software. Then we need to check the correctness of this waveform and find out the potential errors from it. The design can then be synthesized to the FPGA when the verification is done.

As mentioned earlier, this development flow is an error-prone and inefficient process. In order to solve this problem, we hope to automate this process as much as possible. In our lab, a system-level microarchitecture analyzer tool has been developed by L. Risebos. This Quantum μ Architecture Performance Analyzer (QUAPA) models QuMA at the transaction level to enable flexible design space exploration to optimize objective functions such as quantum operation issue rate. It focuses on the relationship among different modules without regarding to the internal structure of each module which means no low-level constraints considered in this tool. So for now, QUAPA's design space exploration cannot help HDL implementation directly since they are not communicating at the same level. Therefore, we propose QuMASim here to bridge the gap between them and implement the automation design approach.

The goals and requirements of QuMASim can be divided into two main parts:

1. QuMASim should be able to help the automation of DSE and development. To achieve that, this simulator should be able to accept configuration from users so some basic parameters can be defined for this simulator easily. It should be self-configurable to some extent so we do not need to change its inner structure each time for different configuration. Besides, the simulation should be easy for all users disregarding their working environment so everyone can run easy simulation with the help of QuMASim.
2. In addition to simulation, this simulator is also required to be able to assist the verification procedure. Since we are targeting the VHDL implementation at here, this simulator should have a low-level modelling of modules of interest. The simulation should be cycle-accurate so the behavior of VHDL

implementation can be checked. And we want to be able to trace any signals of interest in a certain format which can help the later verification.

These are the overall goals and requirements for QuMASim. In the following sections we discuss our method to implement QuMASim under the guidance of these limitations.

3.2.2. SYSTEMC

In order to implement QuMASim in a flexible manner while maintaining a low-level hardware-oriented description, we chose SystemC as the language. SystemC is a system-level language, actually a C++ library. This language enables the software programming language (C++) to have hardware design capabilities to take advantage of the knowledge of these programming languages. Since QuMASim is required to be a cycle-accurate hardware-based simulator, the core of QuMASim needs to be implemented in a low-level description. However, when the future design reaches the system level, which means that the heterogeneous architecture will be applied, the advantages of SystemC will play an important role. In addition, the flexibility offered by high-level programming languages is enormous, which is critical to implementing configurable simulators. Since SystemC is a C++ library, the simulator can use all C++ features, such as embedding other modules written in the high-level programming language (C++/Python).

3.2.3. QUMASIM OVERVIEW

QuMASim is built based on the structure of CC-Light which is targeting to a SC7 chip. The structure diagram is shown in Figure 3.5.

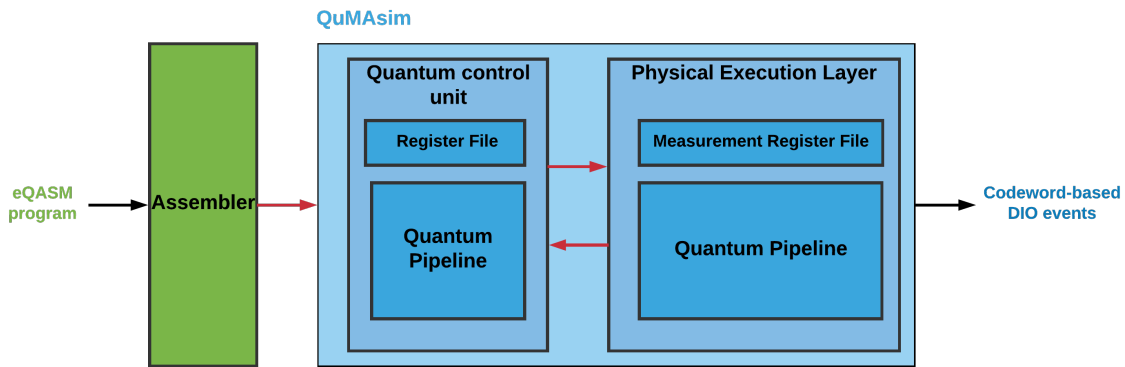


Figure 3.5: The overall structure of QuMASim

As shown in the figure, the input to the simulator is a QISA program written in eQASM. The program is sent to the assembler to generate binary instructions. These binary instructions are then loaded into the instruction cache before the simulation begins. At the beginning of the simulation, the instructions are read into the classic pipeline and processed there. The classic pipeline will read and write register files during execution and send quantum instructions to the quantum pipeline. The quantum instructions are then processed, and the final output is a quantum operation with timing information. Since one of the initial goals was to verify the correctness of QuMA's VHDL implementation, the detailed logic in these modules is almost identical to what we described in the previous section. Therefore, we can use this simulator to achieve cycle-accurate simulation.

3.2.4. DESIGN SPACE EXPLORATION

The simulator can be used to explore the design space of quantum microarchitectures, and there are many different ways to do this. One of the potential goals is a different quantum processor architecture. To do this, the simulator can be configured for different quantum processor architectures. The simulator can adjust different quantum chips by providing quantum layout information to it, requiring information such as quantum bits, edge address information, and electronic control. Therefore, QuMASim can be tested on different quantum chips in addition to the DiCarlo Group's SC7/SC17 chips.

Another goal is to connect to the qubit state simulator. QuMASim can be used to simulate the instructions

generated for QuMA and generate an output of the operation of each qubit with timing information. These operations can be fed to qubit state simulators such as QX and QuantumSim to evolve qubit states. Therefore, the simulator can assist in quantum program verification at the architectural level. If the result of the simulator is a qubit state rather than a timing operation, the output will be more explicit. In fact, QuMAsim may have two outputs: one is a DIO event generated by a device event dispatcher, and the other is a quantum operation of each qubit generated by bypassing the device event dispatcher. Both outputs have their own advantages and there is no conflict.

Supporting different quantum microarchitectures is also very important. Our quantum co-processor uses the VLIW architecture. These VLIW channels consist of several units, namely the address manager, the microcode unit, and the quantum microinstruction unit. If we want to apply different quantum microarchitectures on the simulator, we must modify the VLIW width and unit mentioned above. It would be interesting if we can simulate different quantum microarchitectures on the simulator and might help with future research on QuMA. If we want to test different structures on these modules, we need to manually modify the internal structure of the modules in the quantum pipe. However, it is very difficult to automate this process. But at least this can help the QuMA implementation during the design phase.

The most ambitious goal is to support different quantum technologies. The current simulator only targets superconducting qubits, but it is possible to adapt it to different quantum techniques, such as spin qubits and ion-traps. This requires us to modify different quantum technologies. For example, we need to modify the microcode unit, the number and width of queues, and the quantum classic interface. Since we do not know which quantum technology to use in the future, it would be interesting if the simulator could adapt to different quantum techniques. It is also challenging due to the need to re-implement many physical and control schemes.

3.3. SELF-CONFIGURABLE SIMULATOR

We will discuss in detail how to further develop QuMAsim into a self-configurable version. After the first version of QuMAsim was implemented, the first thing we thought about was: How about another quantum chip? Next, we modified QuMAsim to support a SC17 quantum chip, as shown in Figure 3.6. In this process, we found that the simulator can be configurable for different quantum chips.

3.3.1. QUANTUM LAYOUT INFORMATION

The first step is to figure out the differences between different quantum chips. The layout information in the quantum processor architecture are:

- The number of qubits. This information is straightforward and it is used almost everywhere in the quantum pipeline. Many modules and signals are instantiated based on the number of qubits. Some parameters are derived from this information, like the number of flux events.
- Direct edge address. The two-qubit gate can only apply to an allowed direct edge. For instance, qubit 7 and qubit 10 are connected by edge 12/36 and a two-qubit gate can be applied to these two qubits. While qubit 7 and qubit 11 are not allowed to take such interaction. This information is used in the address mask decoder where the mask information of instruction is decoded. If the edge 0 or edge 25 is set to true, then qubit 0 would be chosen as right(target) qubit.
- Number of feedlines and related qubits. The number of measurement events depends on the number of feedlines used in the quantum processor. For surface-17 the feedline number is three. Each UHFQC can support 10 qubits maximally, so we need to break long feedline into smaller parts if its related qubits number is larger than 10. The simulator can generate measurement events distributor and corresponding timing queue automatically with this information
- Number of different frequency groups. The number of microwave events depends on how many different frequencies are needed. Each AWG-8 has two channels which means it can provide two different frequencies. The method to divide frequency group requires further discussion.

The last two differences can be described as controlling electronic specifications as they are all related to control electronics. This information will be sent to the appropriate module for modification, as shown in Figure 3.7. Multiple modules use the number of qubits information to perform operations such as defining the width of the signal. The edge address information is used in an address decoder module that generates an

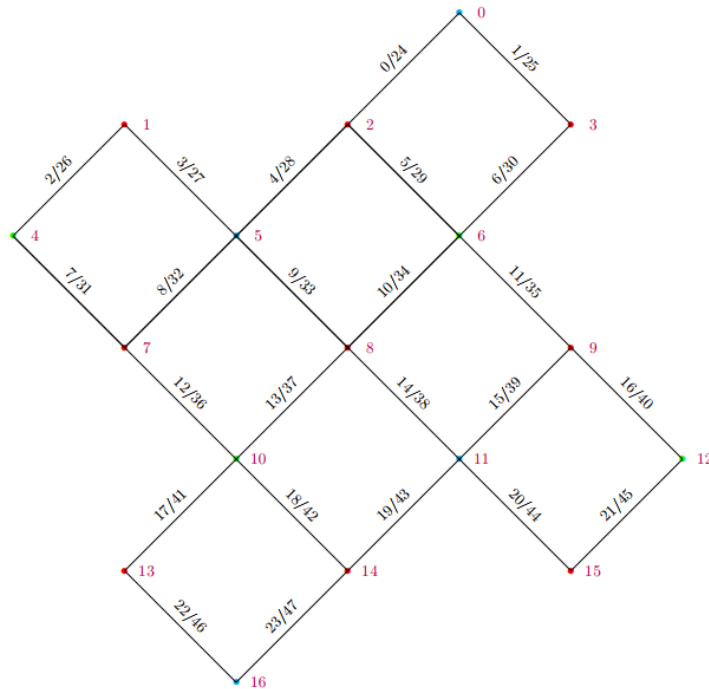


Figure 3.6: The quantum processor architecture for surface-17

operation selection signal. For example, edge nine is decoded in the mask of the quantum instruction, which means that the double qubit gate selects the qubit pair $\langle 5,8 \rangle$. The address decoder will decode the mask, so qubit five will choose the left operation, and qubit eight will choose the correct operation.

Besides that, the control electronics specification is sent into device event distributor which divides the quantum pipeline into control electronics independent domain and control electronics dependent domain. This specification contains information like the number of microwave AWG-8, Flux AWG-8, and UHFQC. The quantum operations for each qubit is then classified into different events based on this information. An example of microwave AWG specification for SC7 is shown below:

```

1 "electronic_setup": {
2     "microwave": {
3         "awg_devices": [
4             {
5                 "name": mw0,
6                 "channels": [
7                     {
8                         "channel_number": 0,
9                         "target_qubits": [0, 1]
10                    },
11                    {
12                        "channel_number": 1,
13                        "target_qubits": [2, 3, 4]
14                    }
15                ]
16            },
17            {
18                "name": mw1,
19                "channels": [
20                    {
21                        "channel_number": 0,
22                        "target_qubits": [5, 6]
23                    }
24                ]
25            }
26        ]
    }
}

```

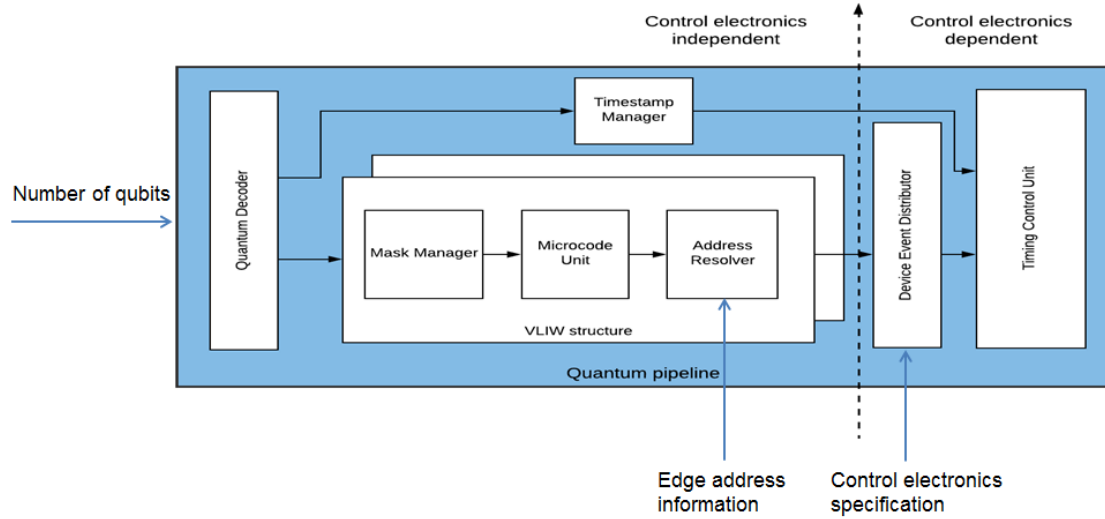


Figure 3.7: Modifications on microarchitecture for SC17

27
28

```

    },
}
    
```

This example means that two AWG-8 are used for microwave control. Each AWG-8 has two channels but may not be fully occupied. The first device uses two channels, and their target qubits are 0, 1 and 2, 3, 4 respectively. This division is related to the frequency groups defined for quantum chip based on the data/ancilla qubits. So for this example, two microwave events are generated and pushed to the event queues in the timing control unit.

3.3.2. INSTRUCTION SET ARCHITECTURE MODIFICATIONS

The change of quantum processor architecture also implies the instruction set architecture side. The mask instructions need to be modified which are used to encode mask information. More precisely, SMIS instruction is used to encode qubit information into S-mask, and SMIT instruction is used to encode edge address information into T-mask. As discussed before, both qubit information and edge address information are changed for different quantum chips, so these two instructions need modification. An example of modifying SMIT instruction from SC7 to SC17 chip is shown in Figure 3.8.

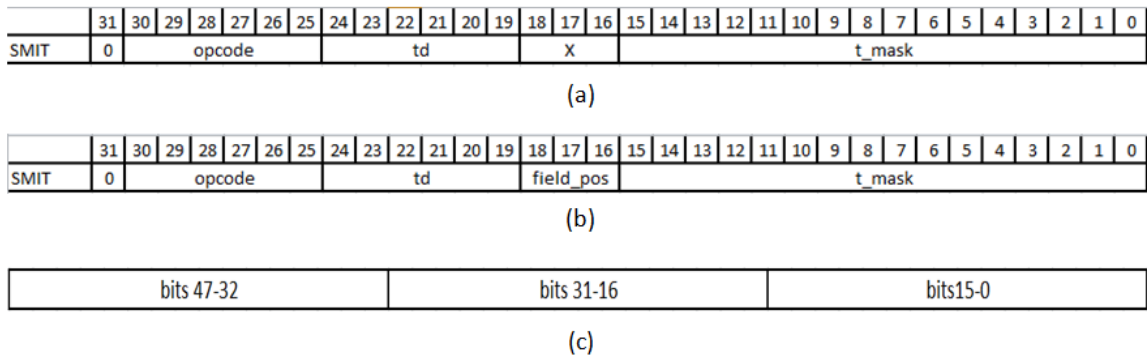


Figure 3.8: The modifications for SMIT instruction. (a) The SMIT binary format for SC7 chip. (b) The SMIT binary format for SC17 chip. (c) T-mask register structure for SC17 chip.

We can see that the SMIT binary format of the SC7 chip can support the maximum 16 bits of the edge address, which is enough for the SC7 chip. But for the SC17 chip, there are a total of 48 direct edges, and

this format needs to be updated. We can place all 48 bits directly like the old format, but this method is inefficient. Therefore, we use 16 18 bits in the binary instruction as the field position to determine the position of the 16-bit information in the T-mask register. This SMIT format can support up to 128 bits of edge address information. To take this change in the instruction set architecture, you also need to modify the T mask register in the mask manager. For SC17, these T-mask registers with a width of 48 bits are divided into three fields for different field locations in the SMIT binary instructions. This means that each SMIT instruction in eQASM will generate three binary instructions, each with a different field position to fill in the information of the different fields in the T-mask register.

3.3.3. FINAL IMPLEMENTATION

To make QuMASim self-configurable for different quantum processor architectures, we need to provide the layout information of quantum chip to the simulator. In QuMASim, the modifications for microarchitecture are easy to be made self-configurable thanks to the flexibility brought by the high-level programming language. For the assembler, which is also implemented in C++, the generation of SMIS/SMIT instructions also require the layout information. Thus, the assembler is also modified for taking the quantum layout information as input, while the simulator takes three input at now: a file contains binary instructions, a file contains quantum layout information, and a file contains the control store content.

3.4. SIMULATOR OUTPUT

This section will introduce a text format which we defined for recording the signals of interest. We need a standard format so we can trace and record the simulation result in a unified way which is beneficial for later use. This is also important for the following comparison and the cooperation between VHDL and QuMA simulator. With a standard text format, VHDL and SystemC can be programmed independently. And if the text format is predefined, the verification platform can also become a standard which can be used for later development such as surface-17/49 and different microarchitecture design.

This format is named as Timing Event Logging Format (TELF) since timing is always crucial for signals inside QuMA. This format is based on the Comma-Separated Values (CSV) format and is easy to process. The timing information for each signal of interest is required to be recorded. Thus, we propose the following file structure as the standard verification interface for QuMA:

```

1 #OutputType
2 clock cycle,          <key0>,      <key1>, ...
3 <number of cycle 0>, <value>,    <value>, ...
4 <number of cycle 1>, <value>,    <value>, ...
5 ...

```

The TELF files are case insensitive.

There might be multiple text files to record a single type of signal of interest. For example, the triggered output has five signals for surface-7 and has eight signals for surface-17. Thus, we do not limit the number of text files that are generated from both simulations. The verification platform will simply parse all the CSV files and classify all the signals based on their type. With this methodology, a scalable verification platform could be built.

The first line is the output type. The second line in the file defines the content to be recorded. A 'key' can be a signal name or a property of some signal or module of interest. It is required that the first key always to be 'clock cycle'. From the second line on, every line should contain the value of each key at the time specified by <number of cycles>.

Multiple value types are supported. Prefix 0b and 0h are required to be added for binary values and hexadecimal values respectively. If no prefix is added, the value is recognized as a decimal value. Besides that, the particular value format of VHDL should also be supported. Sometimes the VHDL result could be undefined('X') or high impedance('Z'), and these values can be recognized by the later comparison stage.

In order to thoroughly verify the correctness of the microarchitecture, we should consider recording multiple signals of interest. Different type of signals should be recorded in different text files with the extension .csv. The following sections will describe these different signals with details.

3.4.1. TRIGGERED OUTPUT

The final triggered event output contains information of event value, target qubit, and timing information. The correctness of these signals can prove the correctness of quantum pipeline. The recommended output format is:

```

1 #TriggeredOutput
2 Clock cycle,  MW_DI00,  MW_DI01,  MSMT_DI00,  MSMT_DI01,  FLUX_DI00
3 611,          0x00000,  0x00000,  0x23,      0x00,      0x00000
4 ...

```

3.4.2. REGISTER VALUES

There are two kinds of registers should be recorded: general purpose registers and mask registers.

GENERAL PURPOSE REGISTER

The general purpose registers are accessed by classical instructions like ADD, SUB and also by special instructions like FMR. We should trace the register value once it is changed. The suggested output format is:

```

1 #RegValue
2 Clock cycle,  R register number,  register value
3 12,          3                    8
4 ...

```

There are at most one register value would be changed every clock cycle. Several advanced features could be added in the future:

- Record the previous value as well. Sometimes this would be required to help debugging.
- Trace the instruction leads to the event writing a register.

MASK REGISTER

There are 32 S-registers and 64 T-registers in current CC-Light. The status of quantum operation target registers are checked if we compare the value of these registers. The output format is almost the same as GPR:

```

1 #RegValue
2 Clock cycle,  S register number,  register value
3 20,          1,                    7
4 ...

```

```

1 #RegValue
2 Clock cycle,  T register number,  register value
3 20,          3,                    1
4 ...

```

Notice that a mask register can be given multiple values during one time, the text parser should take care of it. In this example, the register value 7 represent that qubit 0, 1 and 2 are selected. This information can be translated in the verification platform for the programming simplicity.

3.4.3. MEASUREMENT RESULTS

The measurement result registers are updated after measurement instructions. The correctness of values of these results can prove the correctness of comprehensive feedback control logic, including measurement result analysis module.

```

1 #MSMTResult
2 Clock cycle,  qubit result,  qubit valid
3 32,          0010110,      0000010
4 ...

```

The qubit result is the measurement result, and the qubit valid indicates which qubit is valid.

3.4.4. INSTRUCTION UNDER EXECUTION

Some instructions would not write the registers. To make sure these instructions are working correctly, we also need to trace the instructions that are under execution. The PC value that sends into the memory indicates which instructions would be sent into the classical pipeline. A special occasion is a branching condition. When a branch instruction is executed in the classical pipeline, the pipeline will not process the following instructions until the branch done signal from instruction cache is set high. Thus, the branch done signal should also be recorded.

```
1 #Insn
2 Clock cycle,    PC value,    Branch done,    Instruction
3 32,             10,             1,             0x00000100
4
5 ...
```

3.5. CONCLUSION

In this chapter, we introduced the structure of QuMA and how we implemented QuMASim based on it. We discussed the potential of this simulator, such as what we can do with it and how to develop it further. We propose a way to self-configure for different quantum processor architectures and clarify the impact of this change on the microarchitecture and instruction set architecture. Finally, we introduced the standard text format for recording the simulator information. In the next chapter, we will present two applications built on QuMASim: the verification platform and the quantum virtual machine.

4

APPLICATIONS

4.1. VERIFICATION PLATFORM

This verification platform is designed to verify the correctness of QuMA's VHDL implementation. We will first discuss the potential design approach for this application, and the implementation of this platform will be described later.

4.1.1. OVERVIEW AND APPROACH SELECTION

The goal of the verification platform is to verify the correctness of the VHDL implementation of QuMA. The basic idea is to run both SystemC and VHDL simulations and then compare the signals of interest. Since we already have a simulator and we can trace any of its signals, the remaining question is how to run the VHDL simulation and which exact signals need to be compared.

SystemC is widely adopted in the system level design methodology, which is basically a higher level of abstraction. SystemC and VHDL co-simulation can also be enabled. A new approach to system-level design using SystemC to simulate VHDL and SystemC hardware modules is ideal. This co-simulation is interesting if our QuMA simulator can be used in some high-level system designs in the future, i.e., heterogeneous computing systems. But now we have implemented all RTL modules in the QuMA simulator which can be used to verify the VHDL implementation. Currently, there is no need to consider co-simulation methods.

The basic approach is to run VHDL and SystemC simulation in parallel. Since the SystemC simulator has implemented all core modules of QuMA, the output of these two simulations are supposed to be identical. By comparing these signals of interest, we can obtain the debug information for the VHDL implementation.

The simulator only implements classical modules and quantum pipeline modules, so we can only test the same module in VHDL. Modules like the AXI interface can be neglected. Then we can write a testbench to simulate these modules. The VHDL top-level module should take the same input files as the simulator: the control store initialization file and the binary instruction file. Then we should be able to trace the signals of interest and compare them in an appropriate way. The signals in the QuMA simulator are easy to obtain, and all we need to do is find a way to read the VHDL signal.

One way is to write a simple VHDL module that reads the most important signals, such as output DIO events and classic registers. By comparing these key signals, we can determine if this microarchitecture is working properly. However, this method does not explain any further details of each particular module.

Another way is to compare VCD files. The VCD file contains information about the value changes of the variables in the design. Basically, if we use a VCD file, we can get a cycle-accurate comparison. By enabling cycle-accurate comparisons, the verification platform can inform detailed debugging information so that the user can easily identify the module and clock cycle that caused the error. VCD files are not readable, so a VCD parser is required. I found some python VCD parser libraries, but they have not been tested. This method can take advantage of the RTL feature of the QuMA emulator, but it can be a bit difficult to implement. A good approach might be to try to compare the general signals I mentioned in the first method and then look at the VCD method if necessary.

The signals of interest should be able to reflect the correctness of VHDL in multiple aspects. The signals we want to monitor in this verification platform are listed as follows:

- Output operation with timing information: these operations should have the same type of operation and codewords. The timing difference between every two operations should also be the same.
- Classical signals: Since the quantum control unit is independent of the changes in the quantum chip, it is not necessary to track every signal in the classic pipeline. The value of the register file still needs to be compared. If the values stored in the registers are the same, we can basically know that every classic operation works.
- If the output operations are different, there may be an error inside the quantum pipeline. We want to trace the signals generated by each quantum instruction in the quantum pipeline, and then we can know which module caused the error.
- Signals related to the comprehensive feedback control: the logic of the comprehensive feedback control is somewhat independent of the quantum pipeline and is performed by the FMR instruction in the classical pipeline. The result is stored in the register file of the quantum control unit. We need to record the measurement result and valid signals that are sent into the measurement register file unit, and then we can check if the function of the comprehensive feedback control logic is correct.

4.1.2. IMPLEMENTATION

The overall approach to implementing this verification platform is simple: run the VHDL and SystemC simulations with the same inputs, and then compare the signals of interest in the comparison unit to generate the final debug information. The diagram for this verification platform is displayed in 4.1. Both simulations will use the same input, binary instructions and microcode unit content. The configuration file for QuMASim is not counting as one of input at here since QuMASim should be configured to implement the same structure as the VHDL implementation. Since QuMASim is cycle-accurate and has the same behaviour as the VHDL implementation, the output of both simulations is supposed to be identical. Therefore, we can compare the signals of interest from these two simulations to find out the VHDL implementation errors.

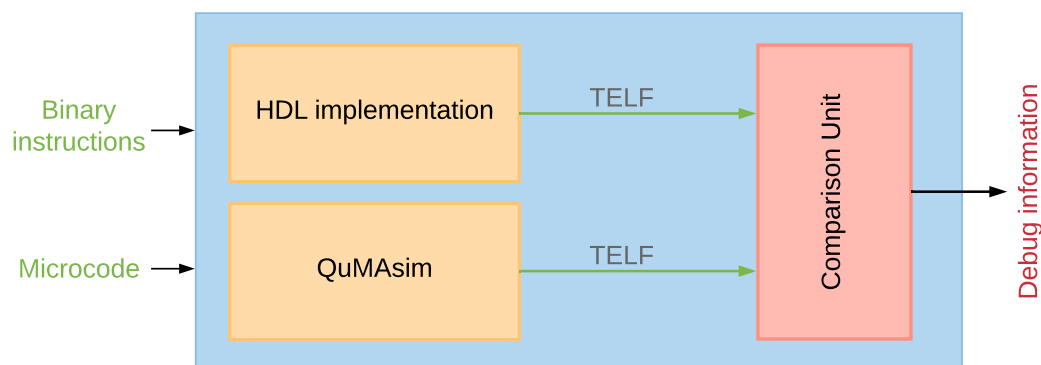


Figure 4.1: Structure of the verification platform, which consists of QuMASim, the HDL implementation and a validator

The timing event logging format (TELF) is used as the communication method between two different simulation environment. Both simulated signals of interest are recorded in this format and can then be easily compared. For QuMASim, the record of TELF is simple due to the high level of programming language features. For VHDL, it is quite strict, so we provide a TELF record library for easy signal recording. Currently, the library contains two modules: one for recording vector variables and the other for recording bit variables. This library is sufficient for current use, as the signals of interest listed in Section 3.4 fall into only these two types. In the verification phase, the user simply logs the signal to the corresponding recording module and records it in the TELF. The user also needs to set some generics for this module, such as signal width and signal name. For different VHDL designs, this method is simple and extensible because only some connection code needs to be added.

We have implemented a validator to process the generated TELF file. The validator can be divided into two parts: a text parser and a comparison unit. For simplicity, the validator will be written in python.

TEXT PARSER

As mentioned earlier, the TELF format is based on a comma-separated value (CSV) format and always records timing information. Therefore, the TELF parser is first required to process these TELF files. When calling the validator, the user needs to provide a directory of these TELF files, and then the text parser will automatically read all files with a .csv suffix. These files are then processed separately according to different simulation environments.

In the TELF file, the first line defines the type of signal to be recorded, and all signals will be grouped into different groups according to their signal type. The rest of the file contains signal information, including the signal name and its value each time it is changed. The first content is always the clock period that represents the timing information. The rest is the so-called "key", which represents the signal name, certain signals or attributes of the module of interest. The following line contains the value of each key. Since this format is based on CSV, a generic CSV parser is required to extract useful information. The output of this parser should have the following format:

```

1 {'clock cycle': [46, 47, 53],
2  'mw_dio0': [0, 3, 0],
3  'msmt_dio0': [0, 0, 4],
4  'flux_dio0': [22, 0, 0],
5  ...
6 }
```

The parser supports multiple digital formats: binary, decimal, hexadecimal and special values in VHDL. Therefore, it is necessary to check the value format before comparing. For comparison purposes, all values should be converted to decimal. For example, if the parser reads a value start with $0b'$, the rest of the value will be converted from binary to decimal and then stored in the corresponding list.

The implementation of the text parser needs to pay attention to the following difficulties:

- Due to the VLIW structure in QuMA, the signal that you want to record may be written multiple times. When multiple writes are detected, we need to remove the extra content. This only happens to signals in the quantum pipeline, and we know in advance which signals need to be processed. Therefore, we will pre-process these signals before the formal comparison begins.
- For the triggered output, we need to reorder the sequences of different results. After the simulation is over, different event signals are recorded in different files, and each file has its own timeline. However, we need to combine these signals into one timeline because we need to check the timing information for all output events. Therefore, we need to merge the lists and rearrange their positions and correspond to their clock cycle order. After rearranging, you need to merge the different events that occur in the same clock cycle and generate a dictionary.
- The testbench of verification is used to test the limit of the microarchitecture. Therefore, eQASM programs of the testbench usually contain infinity loops. Due to the different simulation preparation times of VHDL and SystemC, the generated TELF files usually have different lengths. Thus, when dealing with TELF files with different lengths, we only compare the overlaps by removing the extra content of the larger files.

In conclusion, the text parser will classify all the input signals and then process them to extract useful information. This information is stored in different groups and will be compared in the comparison unit.

COMPARISON UNIT

The processor is divided into two parts: deterministic timing domain and non-deterministic timing domain. We need to compare the timing information of the signal in the deterministic timing domain. For non-deterministic timing domain, we only care about the sequence of events, not the specific timing information. More specifically, only the timing information of the trigger output needs to be compared here. After parsing all the CSV texts, we can start comparing. Unlike the text parser that can be used in a uniform manner, the results of different signals of interest should be compared differently. The error message should also be generated separately. The methods for comparing various signals of interest are as follows:

- For triggered output, there are three kinds of output events: microwave, measurement and flux. The general comparison should be simple: just compare the value and the clock cycle difference. An error report is required if the value is not correct. For microwave events, the event value contains valid

information and qubit codewords. For measurement events, the event value contains qubit valid information and measurement condition information. For flux events, the generic codewords and condition bits are contained in the event value. All of this information could be decoded during the comparison stage to obtain detailed error information.

- For register values, the information in the general purpose register is just a numeric value. But the mask register contains more information. The single-qubit mask contains qubit information and the two-qubit mask contains edge address information. If the mask register value is incorrect, this information should be decoded.
- Measurement results. The measurement results indicate the result of each qubit and the measurement valid signal indicates which qubit result is valid. If any of these two signals is wrong, this two information should be combined in the error report.
- Instruction under execution. The PC value actually represents the instruction position in the memory. If an error occurred at here, the error report should be able to find which part of the instruction flow caused the error. The branch done signal represents whether the branch condition is finished or not and is used to verify the correctness of branch logic. Sometimes the instructions are sent to classical pipeline but are not executed due to the pipeline stall.

4.2. QUANTUM VIRTUAL MACHINE

In this section, we will propose the design of the quantum virtual machine which simulates the execution of quantum programs in the real world. A general overview will be introduced first and followed by a detailed implementation. We chose quantum as the bottom qubit state simulator and OpenQL as the top-level compiler.

4.2.1. OVERVIEW

A full stack simulator is a quantum virtual machine which can run a quantum program on all layers of a quantum computer. The program should be written in a high-level language and then some executable assembly instructions (eQASM) are generated through the compiler. The assembly program can be processed by the quantum microarchitecture simulator (QuMASim) and the output would be the quantum operations with timing information. Finally, these operations are applied to a quantum state simulator (QuantumSim) to obtain the final result.

There are two goals can be achieved with such a full stack quantum simulator:

1. Quantum program verification. In NISQ era, the errors are crucial to quantum programs. We can check the fidelity of an algorithm while investigating the error impact and architecture restrictions.
2. With a qubit state simulator, the measurement result can be reported back to QuMASim. This is especially useful for quantum programs which take advantage of feedback control.

Figure 4.2 shows the workflow of the real-world implementation and the simulation flow of the supposed full stack simulation. QuMASim is a cycle-accurate microarchitecture simulator which can fully reflect the behaviour of CCLight. The output is the quantum operation with timing information. These operations are encoded in the codeword and are processed by the control electronics like AWG and UHFQC in the real-world implementation. Thus, we need a similar layer to do the same thing in the simulation, and this is the quantum classical interface. The QCI will translate the output of QuMASim into some real operations that can be applied on a quantum state simulator. The QuantumSim is chosen for this full stack simulator because of its precise error model.

The quantum classical interface is the counterpart of control electronics in real-world implementation. Figure 4.3 shows an example of this layer which is used to control a SC7 superconducting quantum processor. The quantum classical interface should be able to simulate the function of these control electronics. But the simulation could have different levels. For example, if we want to simulate the real physical effect on qubits, we need to be able to simulate the microwave in this layer. But if we are using qubit state simulator like QuantumSim, we just need to translate the quantum operation at here.

At the top level, the compiler should be able to generate an eQASM program which is executable on QuMASim. And OpenQL already has a back-end for eQASM so we do not need to worry about this part. This is the overview of the structure of the quantum virtual machine. We will introduce a real implementation in the next section and the main focus is about how to control QuantumSim from QuMASim.

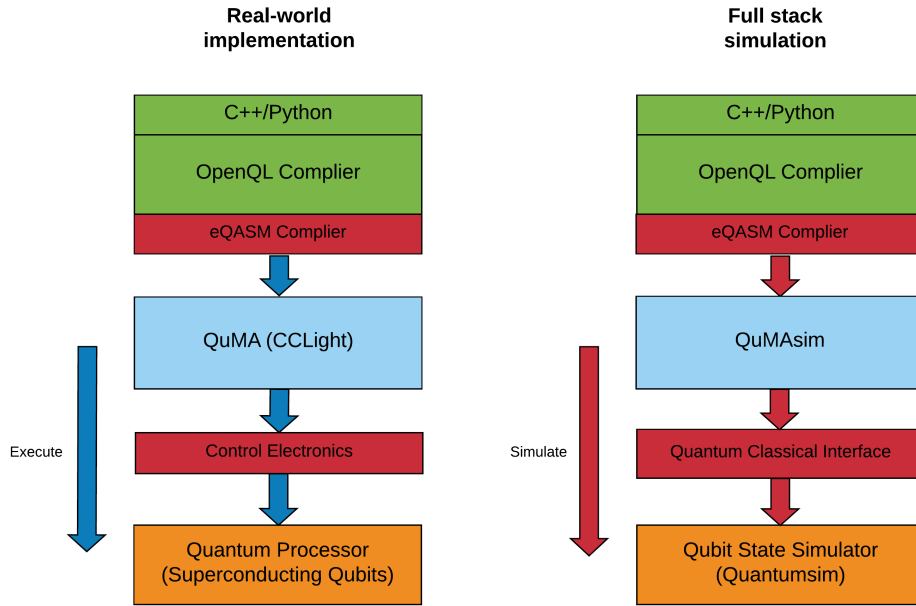


Figure 4.2: Structure of the real-world implementation and structure of the full stack simulator

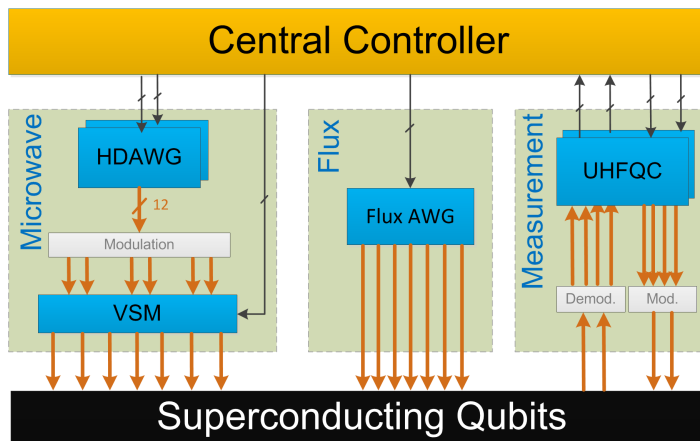


Figure 4.3: Control electronics between central controller and superconducting qubits

4.2.2. OPENQL FRAMEWORK

OpenCL is an open industry standard for cross-platform and classical heterogeneous parallel computing which served as the basis for the OpenQL language. OpenQL is inspired by OpenCL and is based on a heterogeneous programming model. The architecture of this framework is shown in Figure 4.4. The OpenQL framework provides a compiler which can generate the intermediate Common QASM and the compiled eQASM for various target platforms. OpenQL compiler back-end envisions various features like decomposition, optimization, scheduling, mapping and interfacing with the QASM compiler. Thus, we will continue to use OpenQL in this quantum virtual machine implementation.

4.2.3. SIMULATION PLATFORM

Quantum behaviour can be simulated on classical computers using quantum state simulator. With these simulators, we can experiment with algorithms in the absence of a large physical quantum computer. There are different ways to simulate qubits, and every type of simulation has its advantages and disadvantages. We will discuss two simulators which are related to this thesis in this section: QX simulator and QuantumSim.

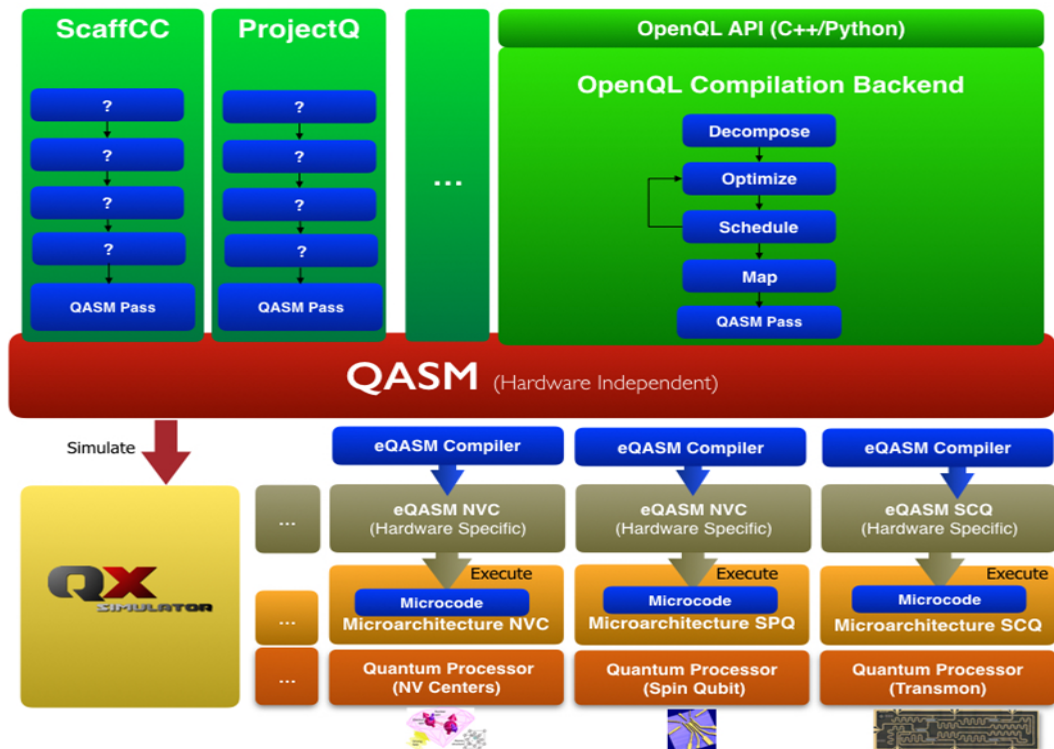


Figure 4.4: OpenQL framework structure

QX SIMULATOR

The QX simulator [7] is a high-performance universal quantum simulator developed at the Quantum and Computer Engineering lab of the Delft University of Technology. A universal quantum simulator can simulate arbitrary quantum gates using their unitary matrix representations. The number of simulated qubits depends on the available memory resources, typically up to 35 qubits can be simulated on a server with 500GB of memory.

The QX simulator takes as input a quantum assembly language named Common QASM (cQASM)[38] and provides high simulation speeds for qubit state evolution. The QASM input describes the quantum circuits using a set of quantum assembly instructions. The QX simulator can read the QASM file and simulate the execution of the described circuit on either a perfect or a realistic error-prone quantum computer. QX simulates noisy execution using different error models such as the depolarizing channel or the pauli-twirling channel.

QUANTUMSIM

QuantumSim [8] is a GPU-accelerated full density matrix simulator of quantum circuits. It can calculate the density matrix of a composite quantum system after applying local processes. These processes are described by process matrices from models with experimental parameters. The state of the quantum system is represented by the density matrix in QuantumSim. The density matrix evolves with the application of some local quantum processes. These processes are described by the Pauli transfer matrix. All of these processes should be performed flawlessly during QuantumSim's simulation. The measurement operation allows a probability projection of the quantum state, and then we can obtain a conditional density matrix from the result.

The original goal of quantumsim was to perform simulations to study quantum error correction on Surface-17. Quantumsim uses an accurate error model to achieve this goal, and this is the most attractive point for us. Quantumsim has used a well-established theoretical model as the basis and uses experimental tomography to provide fixed parameters for observed noise outside of these models. Details and parameters of these error models can be found in [10].

4.2.4. A REAL IMPLEMENTATION

QuMASim takes an eQASM program as input and generates quantum operations with timing information. The input instructions are decoded by multiple layers include assembler, microcode unit and micro-operation unit. Besides these layers, the codeword based operations should be processed by the quantum classical interface to generate the quantum operations which can be applied to the density matrix of QuantumSim. All these layers should accept a unified configuration before the simulation starts.

As introduced above, QuantumSim has well-defined and precise error models. Quantum algorithms can take advantage of these error models and the simulation results would be more precise compared to using other simulators. Thus, we take QuantumSim as the bottom qubit state simulator for implementing our quantum virtual machine in this thesis.

ELECTRONICS SIMULATOR

The main challenge of design this quantum virtual machine is how to connect QuMASim and QuantumSim appropriately. An electronics simulator is implemented to simulate the behaviour of control electronics and its structure has been shown in Figure 4.5. As we can see from this figure, the outputs of QuMASim are codeword-based DIO events for different operation types. These events should be translated into quantum operation for each qubit that can be used to control the QuantumSim. Thus, a module named electronics simulator is implemented which takes the DIO events as input. Inside the electronics simulator, there is a generator for microwave, flux and measurement type of operation, respectively.

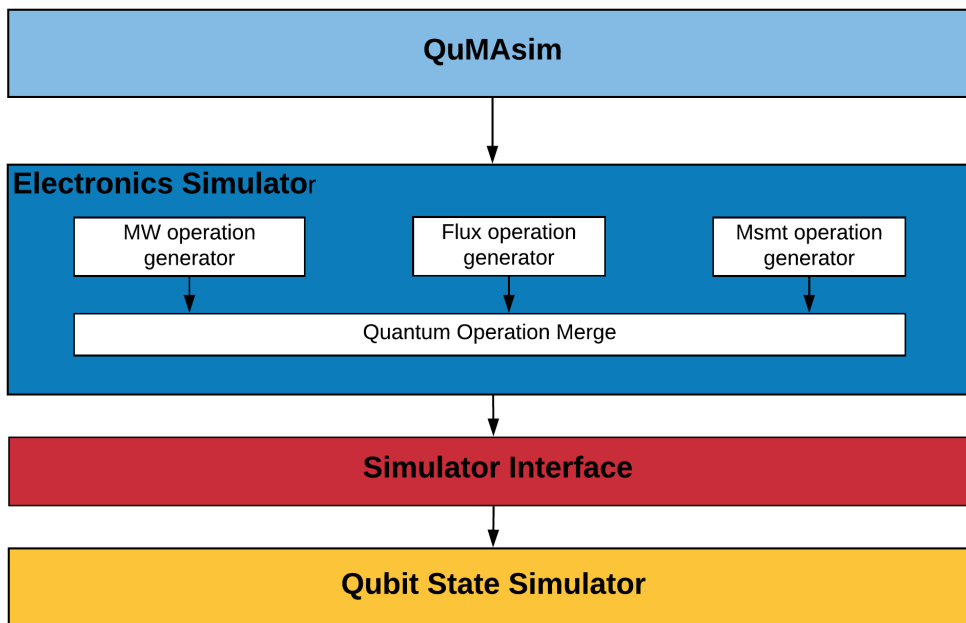


Figure 4.5: Structure of electronics simulator in the full stack quantum simulator

For both microwave and flux operations, the generators are based on look-up table (LUT) structure. The input of LUT is codeword and it will find the corresponding operation for this codeword. Each qubit has its own LUT and the structure of microwave generator is as shown in Figure 4.6. The contents of LUT are provided by a configuration file which is defined by users. It should contain the relationship between codewords and quantum gates. These gates could be universal gates which are predefined in the design. The user could also define their own gates at here and they need to provide the Kraus and gate time information in the configuration file. The outputs of these LUTs are so-called waveforms. These signals are not true analog waveforms but we are trying to include information about analog waveforms in the simulation environment. These signals are then fed into a configurable buffer which is used to simulate the delay of these control electronics. The delay time is also provided by users. The final outputs of this generator are quantum operations of microwave type.

The structure of the flux generator is as shown in Figure 4.7. The basic structure is the same as microwave generator but we need to consider about frequency for flux operations. The qubit-pair information for the two-qubit gate is lost at the output of QuMASim so we need to reconstruct it in this module. Thus, we need

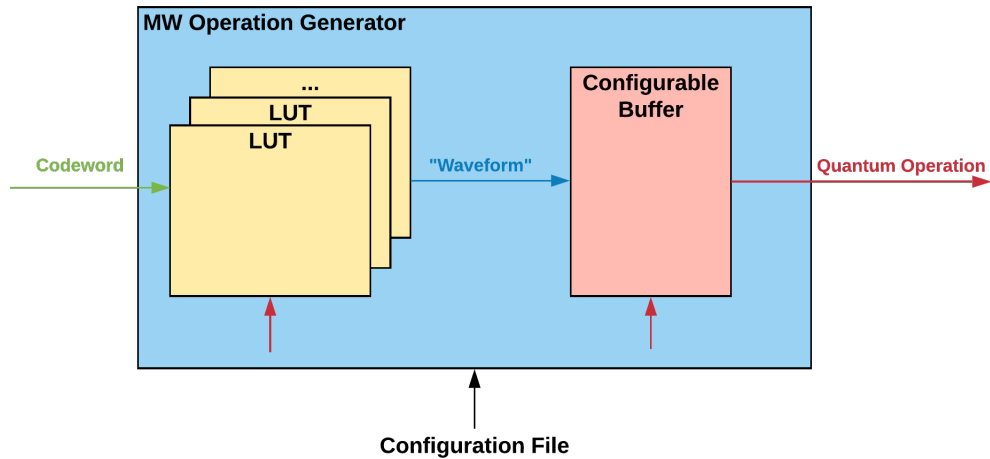


Figure 4.6: Structure of the microwave operation generator

to calculate the frequency for each qubit after applying the corresponding flux operation in this generator to find out the qubit-pair for each two-qubit gate. The frequency model for the qubits in the quantum chip is predefined and the calculation is based on this model. Finally, the quantum operations of flux type are generated at the output of this generator. The structure of measurement generator is not shown here since it has a simple and similar structure. The measurement codewords are valid signals for each qubit so it is easy to generate measurement operations.

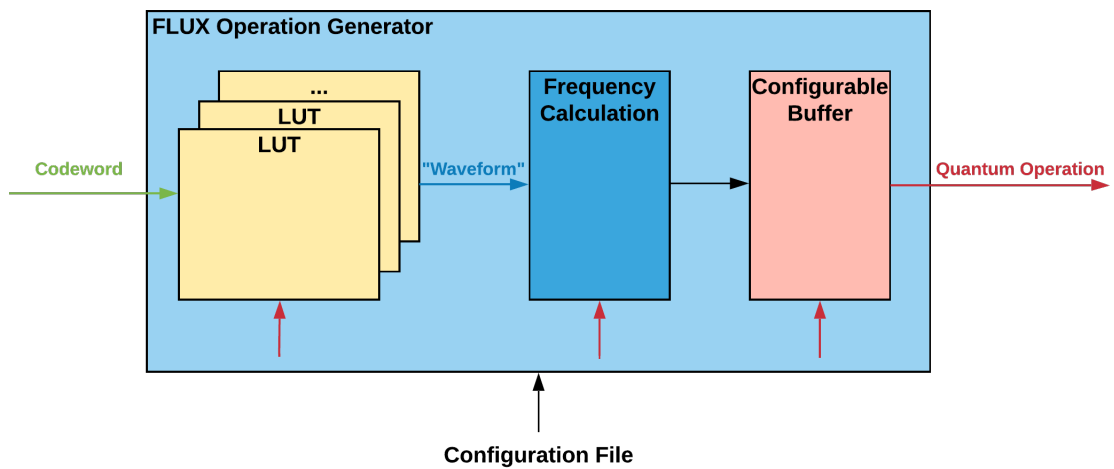


Figure 4.7: Structure of the flux operation generator

An example of the generator configuration file is shown below:

```

1 "electronic_content": {
2   "microwave": {
3     "DelayTime": 10,
4     "LUTContent": [
5       {
6         "0": NULL,
7         "1": Hadamard,
8         "2": Pauli-Y,
9         "3": [0, 1, 1, 0],

```

```

10     ...
11     },
12   ],
13 },
14 "flux": {
15   "DelayTime": 10,
16   "LUTContent": [
17     {
18       "0": NULL,
19       "1": CZ,
20     },
21   ],
22 },
23 },
24 }

```

The contents of LUT are the relationship between codewords and practical quantum operations. These operations can be represented by either string or unitary. The delay time is used to configure the buffer since real electronic devices have delay time to generate analog waveforms.

INTERFACE TO QUANTUMSIM

The outputs of these three generators are fed into an align unit where the operation for each qubit is checked and aligned. The output of this unit is a vector of quantum operations and the vector width equal to the number of qubits. These quantum operations are finally sent into an interface which also takes some signals from QuMAsim as input, e.g., simulation initialization signal and stop signal. This interface will call corresponding functions based on its input and these functions will operate directly on simulation back-end which is QuantumSim in this design. The diagram of this connection with quantum is as shown in Figure 4.9.

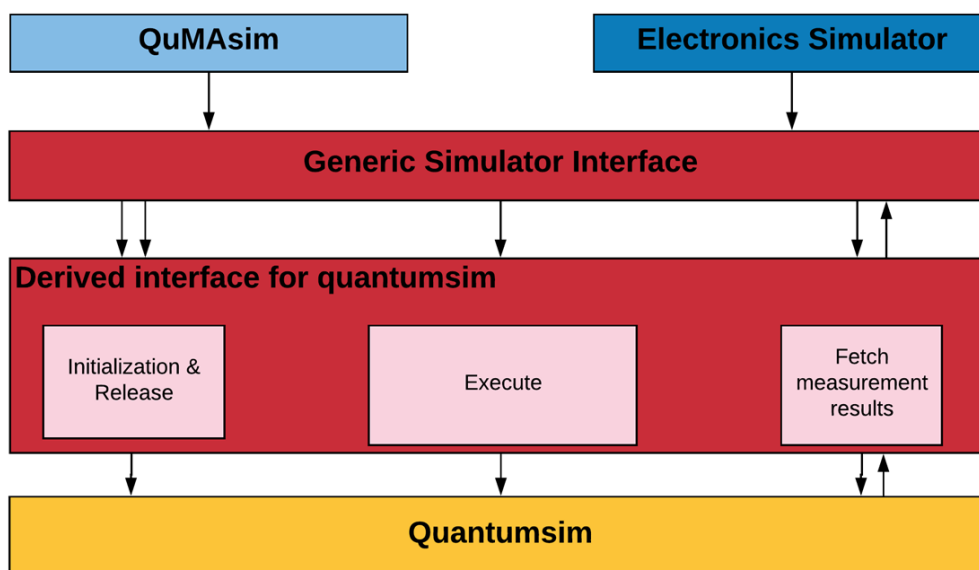


Figure 4.8: Simulation interface which connects QuMAsim and electronics simulator to QuantumSim. This interface is derived from a generic simulator interface for all simulation back-ends.

This interface is derived from a generic simulator interface which is designed to be the interface of all simulation back-ends. This interface contains function like *create_qubits*, *remove_qubits* and *execute*. But the implementation of these functions may change for different simulation back-ends. At here we only discuss the derived interface for QuantumSim which is shown in Figure 4.9. Notice that the above layers of this interface are written in SystemC (C++) and QuantumSim is written in python. To embed QuantumSim in this C++ environment, the C++/python API is used here. This API can help C++ programmers to embed python modules.

We first need to understand the basic working principle for QuantumSim in order to implement these functions in this interface. In QuantumSim, the errors are introduced by sandwiching with idling operation.

Derived interface for QuMASim and quantumsim

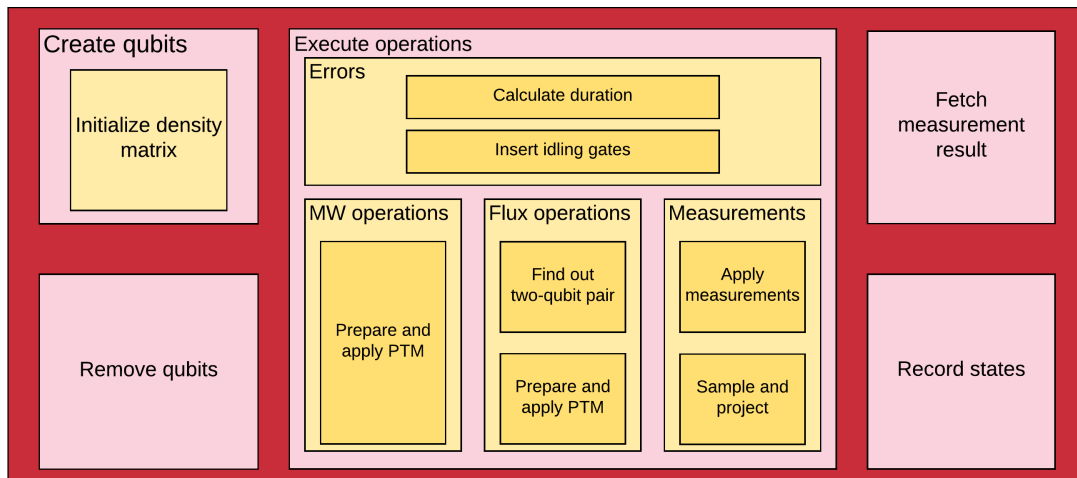


Figure 4.9: Detailed functions for controlling QuantumSim in the interface

An idle operation is amplitude-phase damping with t_1 and t_2 being specified. The timing information is determined by the duration of the idle operation. For example, A Hadamard gate is added to qubit A at time 0ns and another Hadamard gate is added at 40ns. To specify this 40ns timing difference, an idle operation with 40ns duration is added between these two operations.

Another important operation is measurements. The measurement operation will return the state probabilities to the user. Then, we need to sample from these probabilities. The density matrix is required to be projected for each bit to measure with the sampled state. Finally, the density matrix should be renormalized after measurement and projection.

The functions of this interface are as follows:

- Initialize the density matrix. At the beginning of QuMASim simulation, the simulator should call this function to create a sparse density matrix instance. The simulator should indicate the number of qubits of targeting quantum chip.
- Inserting idling gates. The idling operation is amplitude-phase damping which would introduce damping error in the circuit. To ensure the timing of circuit is correct, we need to insert idling gate before each active operation. The timing difference between two consecutive active operations is the duration of this idling gate. With this duration value, we can calculate gamma and lambda to make the PTM for this damping gate and apply it to the corresponding qubit.
- Prepare the Pauli transfer matrices. We need to make a PTM for each quantum operation. The PTM for Hadamard gate is predefined in QuantumSim while the rotation operations required to calculate the angle at first. For two-qubit operation, only the cphase gate is valid for superconducting qubits. The PTM for cphase is also predefined.
- Apply the Pauli transfer matrices. Apply the PTM to corresponding qubits. All single-qubit operations are pending on the qubit until a two-qubit operation or measurement is applied.
- Measurement. The measurement operation would return the possibilities to obtain a certain state. So we also need to sample from these possibilities in the interface. After obtaining the result, we need to project the measurement to reduce the size of the density matrix.
- Return measurement result. The interface can return the current measurement result to QuMASim. This result would be stored in a register file and can be fetched back to the classical pipeline by FMR instruction. QuMASim can achieve real-time feedback control with this function.

- Print final result. After the SystemC simulation, the interface would execute all the pending operations and print the final density matrix.

The general workflow of this simulation is as follows:

1. When the simulation starts, an initialization signal is sent to this interface and the *create_qubits* function is called and the argument of this function is the number of qubits of targeting quantum chip. These qubits will be named with their number and a density matrix is initialized. All following quantum operations are applied to this density matrix.
2. The *execute* function is called when a quantum operation is triggered and all valid operations are passed to the interface. Before applying these quantum operations, we need to insert idling gates which can specify timing and introduce errors. This idling gate is actually a phase-amplitude-damp gate which is determined by amplitude damping time T1 and phase damping time T2. These two time are measured based on some quantum experiments and are predefined in this interface. Then we need to calculate the acting time and duration of this idling gate. The acting time is the middle value of the action time of the previous gate and current gate while the duration is the difference between these two acting time. After this calculation, the idling gate can be inserted correctly.
3. The second step of the *execute* function is to apply quantum operations. For different microwave operations, the corresponding PTM is generated and applied. The angle in radian is passed as the argument for rotation operation. For flux operation, the qubit pair is found at first since the frequency relationship is calculated in the codeword decoder. Then, the cphase PTM is applied to this qubit pair in density matrix. For measurements, we just need to tell QuantumSim which bit to measure and it would return a dictionary with probabilities. Then we need to sample the result from these two probabilities and project the density matrix with the sampled state. Notice that all these quantum operations need to be sandwiched to introduce error. For example, a single-qubit gate usually takes 20ns to occur and we will take the middle point as the acting time of this gate while the rest parts are implemented as idling gates. There are debates whether sandwiching should be symmetric but we will use the symmetric scheme for now. For measurement operation, there are some additional declaration errors and they are also predefined before the simulation starts. Usually speaking, we will renormalize the density matrix after each measurement and projection.
4. After each measurement operation, a *fetch_result* function will be called which will return the measurement results back to QuMASim and the comprehensive feedback control will make use of them. These results are also recorded in a text file for the user to observe after simulation. When simulation ends, a stop signal is sent into this interface and the *remove_qubits* function is called to clear the memory of the density matrix. Then, the entire simulation procedure is finished.

This is the implementation of the quantum virtual machine which takes QuantumSim as the bottom simulator. Other simulators like QX is also available and is not difficult to implement since only a derived interface is required.

4.3. CONCLUSION

We introduced two applications in this chapter. The verification platform can be used for the verification of VHDL implementation of QuMA and is beneficial for the developer who is interested in microarchitecture design. The quantum virtual machine has a more general use case because it helps designer know if a quantum program can actually execute on real hardware: QuMASim brought the hardware restriction into the simulation. In the next chapter, we will do experiments to exploit the potential of QuMASim and will show the results.

5

EXPERIMENTS

5.1. VERIFICATION EXPERIMENT

Besides QuMA_v1 and QuMA_v2 that have been described in section 3.1, QuMA_v3 has also been developed which is targeting to a surface-17 quantum processor. This version is based on QuMA_v2 and has the same overall structure. Now we want to verify this QuMA using the verification platform that we have discussed in section 4.1 so that developers can avoid unnecessary work. We will discuss how to do the automated verification in the rest of this section.

5.1.1. TESTBENCH

The testbench consists of multiple eQASM programs, each of which is used to test one aspect of QuMA. These programs may not have any practical meaning, but they can test the microarchitecture comprehensively. This testbench includes following kinds of test:

- Branch test. This program will test the branch function in the classical pipeline. Comparison and branch instructions are a highlight of eQASM, they belong to the auxiliary classic control logic. With these two instructions, we can add for-loops and if-else logic in the high-level programming and it is valuable for describing quantum algorithms.
- Quantum operations test. This test contains multiple programs. We knew that for QuMA, there are four output types: microwave, vsm, flux and measurement. Each of these operations should be tested comprehensively to cover a variety of event divisions and values. We also need to test the persistence of the timing queue because it can not be empty during execution, which is something we need to take care of. In addition, the VLIW and SOMQ structure also need to be tested.
- Comprehensive feedback control test. As stated in section 3.1.4, the feedback control is pretty complex in QuMA since the entire logic covers various modules. Since we are only testing microarchitecture, we need to set some predefined measurement results inside QuMA. These results would be fetched by FMR instruction and processed in the classical pipeline. The delay of comprehensive feedback control is a crucial point of testing.

5.1.2. VERIFICATION ENVIRONMENT

To make sure that QuMA_{sim} and QuMA_v3 have the same internal structure, we first need to define the layout and electronics setup of the quantum processor. The quantum processor topology is shown in Figure 3.6. We need to use two microwave AWGs, three flux AWGs and three UHFQCs to control this processor. The detailed setup could be found in appendix. QuMA_{sim} can configure for this Surface-17 quantum processor automatically after determining the configuration. For QuMA_v3, these related modules that need to be redesigned are manually modified by V. Mattei from the Quantum Computer Architecture Lab.

As mentioned in section 4.1.2, The next step is to add a TELF file recording scheme to track the signals of interest. We have implemented a VHDL verification library which can be used directly to record signals in a TELF file. In Figure 5.1, we have shown the connection between DIO event outputs and the vector tracking module in HDL Designer. When these modules are added, the signal values are automatically recorded during

the simulation. Then these files can be processed by our verification platform. For QuMA_{sim}, the recording scheme is rather simple since we can use the high-level programming language to do this.

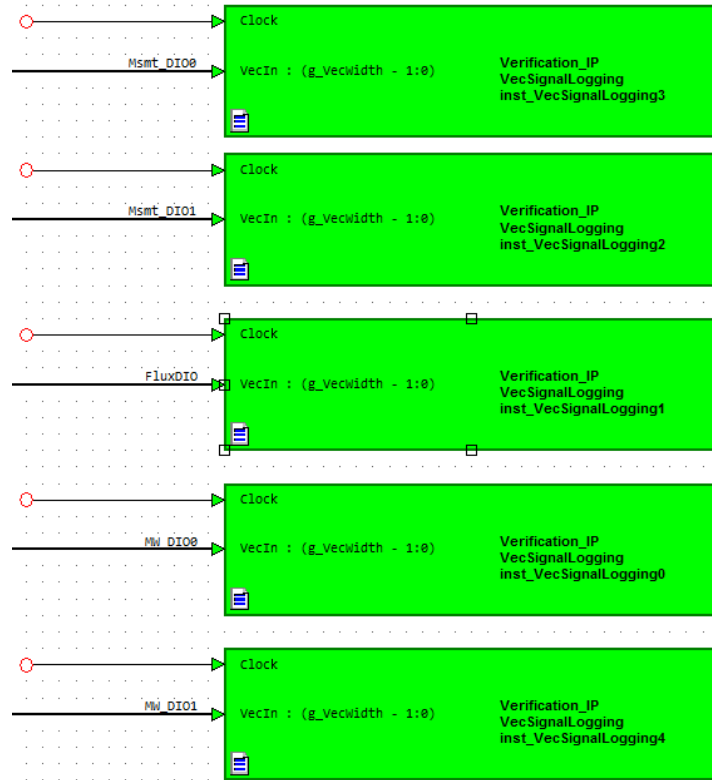


Figure 5.1: Recording modules for tracing the DIO outputs of VHDL implementation of QuMA_{v3}. These modules are instantiated from the verification library

5.1.3. RESULT

We have verified QuMA_{v3} using this verification platform. We performed the verification process in accordance with the above experimental steps and tested several eQASM programs. The results of running the verification platform indicate that QuMA_{v3} can run correctly as QuMA_{sim} because there are no error reports. This result proves that the QuMA_{v3} has zero simulation error and can be synthesized into the FPGA in the next step.

In short, the verification platform can help users easily get debugging information. The user can omit the error-prone and time-consuming process of checking the VHDL simulation waveform. In this experiment that verified QuMA_{v3}, we got optimistic results, indicating that this microarchitecture is error free. The main reason for this result is that this QuMA_{v3} is based entirely on the structure of QuMA_{v2}, and the changes we made are concentrated on the device event dispatcher and timing control unit. Modules like the classic control unit have not changed at all. If we implement a more radically modified microarchitecture, the verification platform should be able to generate more useful feedback. However, how to quickly implement VHDL code is still a problem for an ideal automated microarchitecture development process.

5.2. FULL STACK SIMULATION

5.2.1. SIMULATION ENVIRONMENT

After building quantum virtual machine described in section 4.2, we can achieve quantum algorithm simulation on each level of full stack quantum computer. This simulation procedure can be divided into the following parts:

1. Specify the configuration file. This file will contain information about quantum chip layout and control electronics setup. This configuration file will be used by the OpenQL compiler and QuMA_{sim}.

2. Describe the targeting quantum algorithm using the OpenQL framework. We will choose Python as the high-level programming language, and this algorithm should be defined as a function. This function should contain an OpenQL program which accepts the configuration file as input. This program could be divided into several kernels, each of them represents a sub-circuit of the entire algorithm. Finally, this program can be compiled by OpenQL to generate an eQASM program. Both configuration file and OpenQL program should be ready before the simulation starts.
3. Initialize OpenQL. To use OpenQL, we first need to do some initialization work like set the output directory.
4. Run the described experiment. The generated eQASM program will be put in the output directory.
5. Generate binary instructions from assembler. We have an assembler which can read an eQASM program and generate binary instructions. These binary instructions then can be loaded into QuMASim.
6. Generate hardware configuration for QuMASim. We need to generate the microcode unit content before the simulation starts. The microcode unit contains a decoding scheme from the opcode to the codeword. This content can be automatically generated by PycQED which is a Python library which is developed by DiCarlo Lab.
7. Execute QuMASim simulation. To run QuMASim simulation, we need to provide the binary instruction file, the microcode content file and the configuration file mentioned in step 1. Then QuMASim can execute the eQASM program properly and generate control signals for analog devices. Since we have connected QuMASim and QuantumSim here, these signals are converted to quantum operations and act on QuantumSim in real time. We also need to provide a configuration file to the electronics simulator as mentioned in Section 4.2.4. Finally, some simulation results, such as measurement results and density matrix values, can be recorded.

In this simulation procedure, step 1 and 2 need to be done before the simulation starts. Step 3 through 7 can be performed automatically through a Python script. If the user wants to do some tricks with this simulation, such as logging some intermediate quantum states, the user can also add changes to this script by themselves.

5.2.2. CNOT-MEASUREMENT TEST

A simple CNOT-measurement test is implemented to test the functionality of the quantum virtual machine. In this test, we will initialize a qubit to state 1, and apply a CNOT gate to this qubit (as the control bit) and another qubit (as the target qubit). Then, the state of this target qubit will be measured and recorded. Figure 5.2 shows an example of this CNOT-measurement circuit. This circuit will be repeated for 150 times in this test and the measurement result for each time will be recorded.

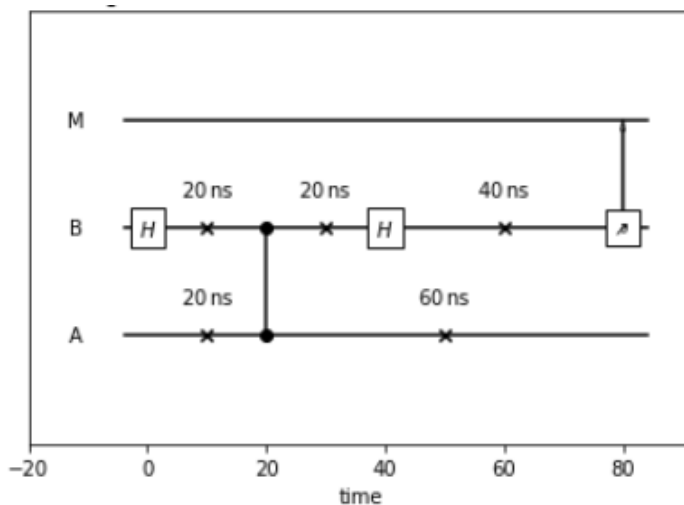


Figure 5.2: An example of the CNOT-measurement circuit

To simulate this test, we first use the OpenQL framework to describe the program, as shown in the Listing 5.1. Using the compiler provided in the OpenQL framework, we can generate an eQASM program that is displayed in Listing 5.2. Finally, this eQASM program will be executed by QuMASim to generate the corresponding quantum operations. These operations can be applied to QuantumSim to obtain measurement results, and the final result is shown in Figure 5.3.

As we can see, the qubit 0 toggles each round as qubit 2 is the $|1\rangle$ state, but after a while qubit 2 decays and the flipping stops. Once during the flipping phase, the qubit decays to $|0\rangle$ while in the $|1\rangle$ state, and then is flipped to $|1\rangle$ again, resulting in two subsequent 1 measurements. After flipping phase, an occasional declaration error could be observed and we can see the result of a continuous measurement of 1.

```

1 # The program is defined as a Python function
2 def msmt_test():
3     sweep_points = [1,2]
4     p.set_sweep_points(sweep_points, len(sweep_points))
5
6     # Set program with information about name, platform and number of qubits
7     p = ql.Program('msmt_test', platform, platform.get_qubit_number())
8
9     # Prepare two kernels of this program
10    k0 = ql.Kernel('aKernel0', platform, platform.get_qubit_number())
11    k1 = ql.Kernel('aKernel1', platform, platform.get_qubit_number())
12
13    # Prepare qubit 2, this operation is only performed once
14    k0.gate('X', [2])
15
16    # Apply the CNOT gate and then measure qubit 0. These operations are performed 150
17    # times
18    k1.gate('H', [0])
19    k1.gate('CZ', [0, 2])
20    k1.gate('H', [0])
21    k1.gate('measure', [0])
22
23    # Add kernels to the program and compile it
24    p.add_kernel(k0)
25    p.add_for(k1, 150)
26    p.compile()

```

Listing 5.1: The OpenQL program which describes this CNOT-measurement experiment.

```

1 smis s0, {0}
2 smis s2, {2}
3 smit t0, {(0, 2)}
4 start:
5
6 # Kernel 0
7 aKernel0:
8     1    x s2
9     qwait 2
10
11 aKernel1_for2_start:
12     ldi r29, 150
13     ldi r30, 1
14     ldi r31, 0
15
16 # Kernel 1
17 aKernel1:
18     1    h s0
19     2    cz t0
20     4    h s0
21     2    measz s0
22     qwait 2
23
24 # Kernel 1 is executed for 150 times
25 aKernel1_for2_end:
26     add r31, r31, r30
27     cmp r31, r29
28     nop
29     br lt, aKernel1

```

30
31
32

```
nop
nop
```

Listing 5.2: The eQASM program which automatically generated by the OpenQL compiler.

```
The measurement results of qubit 0 are:
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[]
```

Figure 5.3: The 150 measurement results of qubit 0 generated by the quantum virtual machine

This test demonstrates that the quantum virtual machine can work properly from the top compiler layer the bottom qubit state simulator. The results of this experiment and the theoretical results are somewhat different due to the error model in Quanticsim. Therefore, we can do some valuable simulations using this quantum virtual machine, which is closer to real-world quantum experiments.

The next experiment is based on a meaningful circuit that will take advantage of the most attractive point of this quantum virtual machine: feedback control.

5.2.3. GEARBOX CIRCUIT SIMULATION

IMPLEMENTATION VIA ACTIVE FEEDBACK

The previous experiment has shown how the quantum virtual machine implements a full stack simulation. Now we will implement another simulation with a more complex circuit while taking advantage of comprehensive feedback control. This experiment is based on gearbox circuit as proposed in [39].

Consider a quantum device in which the only available rotation is of angle $\pm\phi$. The gearbox circuit shows that we can achieve rotations of angle $\theta = q(\phi)$ with $q(*)$ being a non-linear function as shown in follow equation:

$$q(x) = \arctan(\tan(x)^2). \tag{5.1}$$

The circuit is reproduced in Figure 5.4. An ancilla qubit is prepared in state $|0\rangle$. The data qubits is initialized in an arbitrary state $|\psi\rangle$, and the rotation around the x-axis of angle θ will be performed on this qubit. As we can see, the basic gearbox circuit is rather simple: two single qubit rotations, a controlled-NOT gate and a hermitian conjugate of the phase gate.

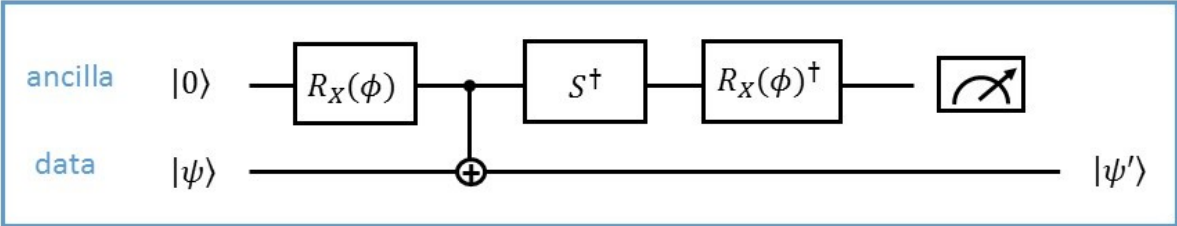


Figure 5.4: A special case of gearbox circuit

Then the ancilla qubit is measured in the computational bases. If the ancilla qubit is in state $|0\rangle$, then the data qubit successfully performs the hypothetical rotation, and its new state is shown in Equation 5.2. If the ancilla qubit is in state $|1\rangle$, the circuit was not successful and the output state of data qubit is shown in Equation 5.3.

$$|\psi'\rangle = R_x(\theta) |\psi\rangle \tag{5.2}$$

$$|\psi'\rangle = R_x(-\frac{\pi}{4}) |\psi\rangle \tag{5.3}$$

If the circuit is failed, one can easily correct the failure and restore the initial situation. Therefore, it is possible to implement the gearbox circuit as many time as needed to achieve success, a procedure that is called repeat-until-success (RUS) [40].

The failure of RUS does not affect the information contained in the data qubit. The initial arbitrary state $|\psi\rangle$ can be fully recovered after the correction procedure. The probability of success is given by the formula:

$$p(\phi) = \sin(\phi)^4 + \cos(\phi)^4 \quad (5.4)$$

This probability is larger than $\frac{1}{2}$ and it makes this gearbox circuit a high-efficiency circuit.

As mentioned above, in the implementation of this gearbox circuit, one needs to be able to measure the ancilla qubit to determine if the circuit was successful, and complete it by correcting and repeating the RUS procedure. Thus, we can use the comprehensive feedback control, which is introduced in section 3.1.4, to implement the gearbox circuit in a faithful way.

Figure 5.5 shows the quantum circuit corresponding to a RUS block producing a failure outcome. After feedback, the data qubit is corrected back to state $|\psi\rangle$ while the ancilla qubit is set to state $|0\rangle$. The second RUS block is successful, as indicated by the measurement outcome +1. The active feedback at this point may indicate to perform the state tomography of the data qubit.

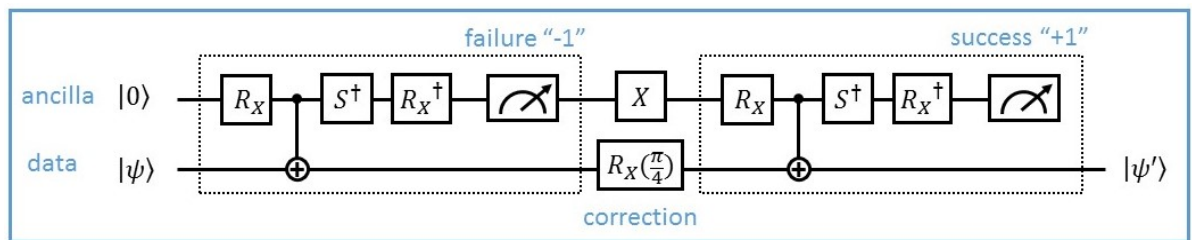


Figure 5.5: Level-1 gearbox circuit implemented by recycling the same qubit as the ancilla qubit. The sequence of operations to the case of a single failure followed by correction and success

EXPERIMENTAL IMPLEMENTATION

In preparation to the experimental implementation, we have wrote an eQASM program for this circuit. Figure 5.6 shows the circuit that we used to implement a RUS block as described above. In this special case, we chose a rotation around the x-axis of angle $\frac{\pi}{2}$ as the preparation of data qubit. We also chose the θ to be $\frac{\pi}{3}$ for this experiment. The CNOT gate is decomposed into two single qubit rotations around y axis and a CPhase gate.

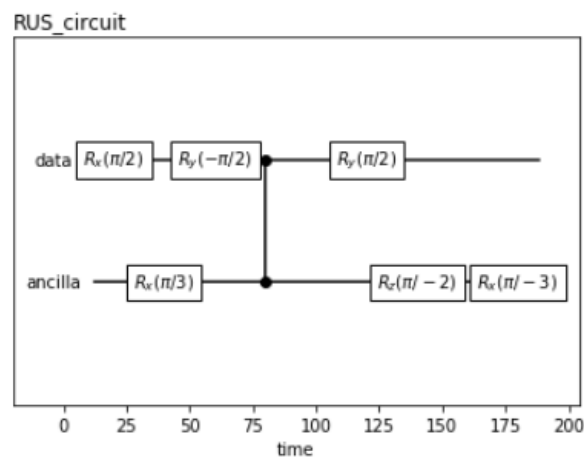


Figure 5.6: The experimental implementation of a RUS block

After implementing this RUS block, we need to measure the ancilla qubit and fetch the result back to microarchitecture. A fetch measurement result (FMR) instruction will help the realization with this feedback

control. If this circuit is not successful, a correction procedure for both data qubit and ancilla qubit is required. The ancilla qubit will in state $|1\rangle$, and we can use a Pauli-X gate to make it back to $|0\rangle$. For data qubit, there is a trick to implement this correction. As discussed above, the state of the data qubit at that time is as shown in Equation 5.3. However, the unitary of rotation operations in [39] is supposed to be like 5.5 while in real physics, the unitary of rotation is demonstrated as 5.6.

$$R_x(\phi) = \exp(-iX\phi) \quad (5.5)$$

$$R_x(\phi) = \exp(-iX\frac{\phi}{2}) \quad (5.6)$$

In the experimental implementation, rotating operations require twice the angle to realize this quantum circuit. Thus, we use a rotation around x axis of angle $\frac{\pi}{2}$ to be the correction of data qubit. After this correction procedure, both ancilla qubit and data qubit are in the state before the first RUS and are ready for another repetition until it successes.

After applying this gearbox circuit, we need to perform the state tomography for the data qubit. For simplicity, our tomography scheme is to measure the state $|\psi'\rangle$ for multiple times, say, 500 times. In our experimental implementation, we will measure the data qubit if the gearbox circuit succeed. If the data qubit is in $|1\rangle$, we will apply a Pauli-X gate to it. As long as its state is in $|0\rangle$, we can repeat the entire gearbox circuit again, including the preparation procedure. All measurement results will be recorded properly.

After each simulation, we will obtain 500 measurements of data qubit. Then we can estimate the state of data qubit by calculating the number of one measurements and zero measurements. We will compare this state to the state of $R_x(\theta)|\psi\rangle$. As discussed above, the achieved state should be equal to Equation 5.7. However, we know that we need to use twice the angle to realize the hypothetical rotation unitary from Equation 5.5 and Equation 5.6. Therefore, in the actual experimental implementation, the implementation angle of θ is Equation 5.8.

$$|\psi'\rangle = e^{\arctan(\tan(\phi)^2)} |\psi\rangle \quad (5.7)$$

$$\theta = (\arctan(\tan(\frac{\phi}{2})^2)) * 2 \quad (5.8)$$

EXPERIMENTAL RESULTS

In our experimental implementation, we will always choose $R_x(\frac{\pi}{2})$ as the preparation of state $|\psi\rangle$. Then we will choose several different angles for ϕ to implement to gearbox circuit. Before starting the quantum virtual machine simulation, we use Quantsim to calculate the measurement probability of state $R_x(\theta)|\psi\rangle$. The result is shown in Table 5.1. The first row is the selected basic rotation angle. The second row is the theoretically achieved angle of rotation. And P_0/P_1 are the measurement probabilities of the final state.

ϕ	15	30	45	60	75	90	180
θ	1.98	8.21	19.47	36.87	60.98	90	180
P_0/P_1	0.48/0.52	0.43/0.57	0.33/0.67	0.20/0.80	0.06/0.94	0/1	0.50/0.50

Table 5.1: Measurement probabilities of the the theoretically achieved quantum state

These results can be used as the standard measurement probabilities, which we can compare later with the simulation results. Then we start the simulation and turn the error model off so that we can verify the correctness of our simulation program. The results are shown in Table 5.2.

We can see that these results are very close to the results shown in the previous table. From these two probabilities, we can reconstruct the magnitude of the $|0\rangle/|1\rangle$ computational basis. We did not implement a comprehensive state tomography at here, but in fact we can know the density matrix of the final state $|\psi\rangle$ through QuantumSim. This density matrix is checked by comparing to the state $R_x(\theta)|\psi\rangle$ to ensure that our eQASM program is working correctly.

Then we repeat this simulation procedure, but this time we add the error to the circuit. The simulation results are shown in Table 5.3. We can see that these new measurement probabilities are different from the results of error-free simulations. Therefore, we obtain some rough error rates for the gearbox circuit by calculating the probability difference. As indicated by the last row of Table 5.3, these error rates vary from different angle ϕ and range from 1.8% to 31.6%.

ϕ	15	30	45	60	75	90	180
number of 0/1 measurements	239/261	214/286	178/322	110/390	38/462	0/500	264/236
P_0/P_1	0.48/0.52	0.43/0.57	0.35/0.65	0.22/0.78	0.07/0.93	0/1	0.53/0.47

Table 5.2: Simulation result without errors

ϕ	15	30	45	60	75	90	180
number of 0/1 measurements	290/210	253/247	244/256	198/302	177/323	158/342	275/225
P_0/P_1	0.58/0.42	0.50/0.50	0.48/0.52	0.39/0.61	0.35/0.65	0.31/0.69	0.55/0.45
error rate	10.2%	7.8%	13.2%	17.6%	27.8%	31.6%	1.8%

Table 5.3: Simulation result with errors

As stated before, one interesting aspect of this gearbox simulation is that it take advantage of feedback control. By utilizing feedback control, we can quickly respond to previous results and implement the next steps. Therefore, the execution time of the entire circuit is improved by enabling comprehensive feedback control. Table 5.4 shows some results about this problem. The number of RUS blocks increases linearly with angle ϕ , which make sense since the success probability of RUS is determined by Equation 5.4. T_1 is the entire simulation time for this experiment and it can be calculated by:

$$T_1 = (\#of RUS) \times (T_{RUS} + T_{fdbc}) + 500 \times T_{fdbc} + 500 \times 50 \quad (5.9)$$

T_{RUS} is the delay of a RUS block, which is 7 clock cycles in our implementation. T_{fdbc} is the delay of comprehensive feedback control in QuMA, and it costs 18 clock cycles. Before applying each gearbox circuit, we will use an additional 100 clock cycles as the relation time because we want to reduce the error rate caused by inappropriate correction of data qubit. The calculated simulation times are shown in the third row of Table 5.4.

If we do not use comprehensive feedback control in this experiment, we have to wait for a long relaxation time to ensure the state of data qubit back to $|0\rangle$ before initialization. This relaxation time is based on the amplitude damping time of the qubit. This time is around $3\mu s$ for the transmon qubit which we are using in DiCarlo lab. Therefore, we can calculate this T_2 which is the simulation time without feedback control. The corresponding speedup is also calculated for each angle ϕ as shown in Table 5.4. Notice that all these time results are displayed in units of clock cycles.

ϕ	15	30	45	60	75	90	180
number of RUS blocks	518	579	677	795	903	1052	517
T_1	46950	48475	50925	53875	56575	60300	46925
T_2	87950	89475	91925	94875	97575	101300	87925
speedup	87.3%	84.6%	80.5%	76.1%	72.5%	68.0%	87.3%

Table 5.4: Simulation time with and without comprehensive feedback control.

The gearbox circuit is one of the most interesting things that can be done with 2-3 qubits and feedback control. In this experiment, we have proven that the simulation time for this experiment was generally im-

proved by about 80% due to the feedback control logic in QuMA. The gearbox circuit also suffers from quantum errors as shown in Table 5.3. In our speculation, the main cause of these errors may be the delay of feedback control. For example, we need to wait about 18 clock cycles before applying conditioned operations based on feedback. This time is quite long, and the quantum state is no longer the same as the state before measurement. If we can implement a faster feedback control, we can further improve the behavior of the circuit.

5.3. MICROARCHITECTURE DESIGN EXPERIMENT

5.3.1. DIFFERENT MICROWAVE SETUP

As mentioned before, the QuMA used in CC-Light is constrained to a specific hardware configuration. For CC-Light, each microwave AWG is divided into two channels. Each channel has four analog output, and these signals are feed into the VSM module. VSM will mask the input signals and then apply them on corresponding qubits. As you can see in Figure 5.7(a), this setup will use three frequencies to control seven qubits, and it takes advantage of frequency reuse. This control scheme is targeting QuSurf architecture which is developed in the DiCarlo lab.

However, many other experiments do not need to be restricted by this constraint. A new scheme is as shown in Figure 5.7(b). In this setup, each AWG is divided into four channels and each channel has two analog output. These two analog signals will be mixed by IQ-Mixer and finally applied to the qubits. Since the target quantum processor has seven qubits, two AWGs are sufficient for microwave control in the absence of VSM. This scheme will use seven different frequencies, one for each qubit.

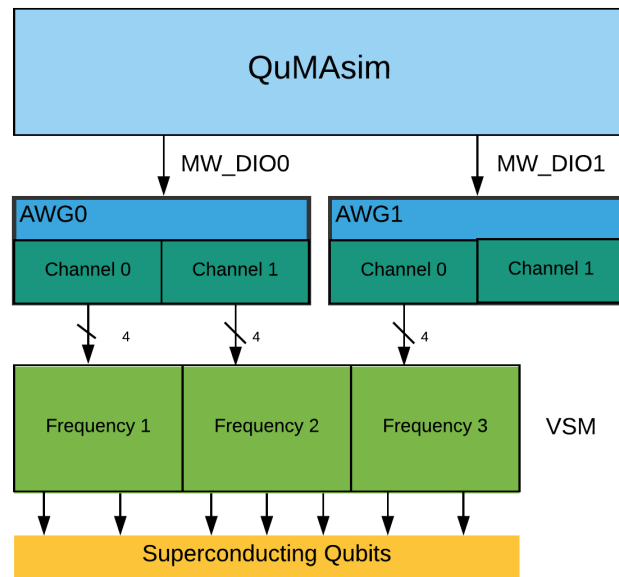
We can imagine that it would be great if this microarchitecture can operation relatively independent of the underlying setup for the universal solution. However, this information is hard-coded in the VHDL of CC-Light, and it is difficult to make it fully configurable as the high-level simulator. Thus, we choose a compromise option at here: implement the microarchitecture for both setups as described above, and allow the user to select which setup to use.

5.3.2. IMPLEMENTATION

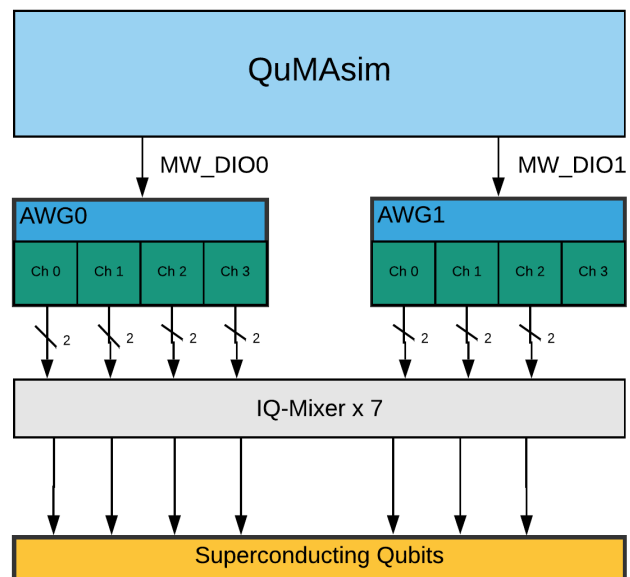
Different microwave setups require corresponding changes to the device event distributor and timing control unit of microarchitecture. Device event distributor will convert micro-operations for each qubit into event signals which are used to control analog devices. Since we have changed the basic settings of the underlying microwave device, we also need to change the structure that generates microwave event signals, as shown in Figure ??.

The VSM event generator is no longer needed in the proposed new structure. And we do not need to mix these micro-operations before the mapping module, because each micro-operation has its own channel in the new microwave setup. Since we are still using two AWG8s for microwave control, the new mapping module will also generate two event signals. As long as each signal contains at most four micro-operations, the specific division of the qubits is flexible. Every event signal will be stored in a corresponding event queue. So for the timing control unit, we just need to remove the event queue of the VSM signal.

These are changes in the microarchitecture. As mentioned earlier, we want the user to choose which settings to use before executing. Therefore, we have also implemented a multiplexer in this microarchitecture that takes a user-definable signal as input.



(a)



(b)

Figure 5.7: (a) The microwave setup used in CC-Light. (b) Proposed new microwave setup

6

CONCLUSION AND FUTURE WORK

6.1. CONCLUSION

The motivation for this paper is the design and verification challenges faced by quantum microarchitecture. To solve these problems, we have implemented a QuMA simulator to automate the development procedure of QuMA. In the beginning, QuMASim can be understood as the simulator of CC-Light, which is based on QuMA_v2 and is used to control a SC-7 quantum processor. QuMASim can accept the same input as CC-Light, i.e., the binary instructions, and simulate the behavior of CC-Light and output digital signals which are used to control the underlying analog device. As a step forward, we made this simulator self-configurable for different configurations, e.g., different electronics setups and different quantum processor typologies.

We have built some QuMASim-based applications, such as the verification platform and the quantum virtual machine. The verification platform uses QuMASim to verify the correctness of VHDL implementation, which simplifies this procedure by automatically generating debug information for microarchitecture developers. To use this tool, we run both QuMASim and VHDL simulation with the same inputs and use a validator to compare the results of these two simulations. With this platform, developers could avoid an error-prone and low-efficient microarchitecture development flow.

The quantum virtual machine can simulate the execution process of a quantum program on all layers of a quantum computer. With QuMASim in this full stack simulator, we can validate quantum programs at both the algorithm level and the architecture level. In other words, QuMASim takes low-level restrictions into the consideration for executable quantum algorithms. In addition, QuMASim has comprehensive feedback control logic in its design, so this tool is especially useful for quantum programs which take advantage of feedback control.

Besides these applications, QuMASim itself can be used to explore the design possibilities of quantum microarchitecture. If we want to update the microarchitecture, e.g., to implement a new microcode unit that can implement gate decomposition, we can test these new structures with QuMASim and find out which one is the best before hardware implementation. Generally speaking, QuMASim can be used as a simulation platform to facilitate the design space exploration of quantum microarchitecture.

We have done several experiments to demonstrate the potential of QuMASim. We have used the verification platform to verify the QuMA_v3 which is targeting a SC-17 superconducting quantum chip, which gave very optimistic results. We have implemented several experiments for the quantum virtual machine that tested the capabilities of full stack simulation. Of particular note is the gearbox circuit, which is one of the most interesting experiments that can be done at the current stage because it has a meaningful target and it utilizes feedback control. We can also use QuMASim to test the new structure of QuMA because it contains all low-level details for hardware-based simulation.

Finally, we can conclude the contributions of the thesis are:

- Implementation of a quantum microarchitecture simulator, which is self-configurable for different configurations.
- Development of the verification platform which can verify the correctness of VHDL implementation of QuMA.

- Connection with QuantumSim and implementation of the quantum virtual machine which can be used as a full stack quantum simulator.
- The quantum virtual machine is used to perform some meaningful quantum circuits that take into account the constraints of hardware and errors of superconducting qubits.
- QuMASim can be used to help the automation of design space exploration and development of QuMA.

6.2. FUTURE WORK

The work in this thesis could be extended in different directions. Now, QuMASim is already configurable for different quantum chip layouts, but it would be very interesting to extend the configurable parameters, e.g., different VLIW widths and a different number of timing queues. In principle, QuMA itself is technology independent, but we can include technology-specific constraints in QuMASim. It would be interesting to let QuMASim support other quantum technologies other than superconducting qubits, such as spin qubits and ion-traps.

These two applications built during this project, the verification platform and the quantum virtual machine, can be further developed. For the verification platform, it is possible to use it with the Universal Verification Methodology (UVM) for more comprehensive verification. For the quantum virtual machine, we can use other qubit state simulators, such as QX, which have the potential to simulate a larger number of qubits. The quantum virtual machine can help validate quantum programs at the architecture level, and it would be interesting to report on the impact of the architecture based on simulation results.

Finally, we would like to use QuMASim as a part of the microarchitecture development flow. Interesting work could be done by interacting with other tools in our lab, such as μ Architecture Performance Analyzer (QUAPA). On this basis, we can achieve true automated development of microarchitecture from design to implementation. QuMASim is able to perform cycle-accurate simulations to bridge the gap between high-level modeling and low-level hardware.

BIBLIOGRAPHY

- [1] P. W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on* (1994) pp. 124–134.
- [2] L. K. Grover, *A fast quantum mechanical algorithm for database search*, in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)* (ACM, 1996) pp. 212–219.
- [3] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, *An experimental microarchitecture for a superconducting quantum processor*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)* (IEEE, 2017) pp. 813–825.
- [4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, *The gem5 simulator*, ACM SIGARCH Computer Architecture News **39**, 1 (2011).
- [5] S. Rosenberg and K. Meade, *A practical guide to adopting the Universal Verification Methodology (UVM)* (Cadence Design Systems, 2013).
- [6] n. Khammassi and I. Ashraf, *OpenQL Quantum Programming Framework*, <https://github.com/QE-Lab/OpenQL> (2018).
- [7] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, *Qx: A high-performance quantum computer simulation platform*, in *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, 2017) pp. 464–469.
- [8] T. E. O’Brien, B. Tarasinski, and L. DiCarlo, *Density-matrix simulation of small surface codes under current and projected experimental noise*, npj Quantum Information **3**, 39 (2017).
- [9] D. P. DiVincenzo, *The physical implementation of quantum computation*, arXiv:quant-ph/0002077 (2000).
- [10] D. Ristè, S. Poletto, M.-Z. Huang, A. Bruno, V. Vesterinen, O.-P. Saira, and L. DiCarlo, *Detecting bit-flip errors in a logical qubit using stabilizer measurements*, Nature Communications **6**, 6983 (2015).
- [11] J. Kelly, R. Barends, A. G. Fowler, A. Megrant, E. Jeffrey, T. C. White, D. Sank, J. Y. Mutus, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, I. C. Hoi, C. Neill, P. J. J. O’Malley, C. Quintana, P. Roushan, A. Vainsencher, J. Wenner, A. N. Cleland, and J. M. Martinis, *State preservation by repetitive error detection in a superconducting quantum circuit*, Nature **519**, 66 (2015).
- [12] C. Monroe, D. Meekhof, B. King, W. M. Itano, and D. J. Wineland, *Demonstration of a fundamental quantum logic gate*, Physical Review Letters **75**, 4714 (1995).
- [13] J. Cramer, N. Kalb, M. A. Rol, B. Hensen, M. S. Blok, M. Markham, D. J. Twitchen, R. Hanson, and T. H. Taminiau, *Repeated quantum error correction on a continuously encoded qubit by real-time feedback*, Nature Communications **7** (2016).
- [14] X. Fu, L. Rieseboos, L. Lao, C. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, *A heterogeneous quantum computer architecture*, in *Proceedings of the ACM International Conference on Computing Frontiers* (ACM, 2016) pp. 323–330.
- [15] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, *Scaffold: Quantum programming language*, Technical Report (Princeton University, 2012).
- [16] D. Wecker and K. M. Svore, *LIQ|i⟩: A software design architecture and domain-specific language for quantum computing*, arXiv:1402.4467 (2014).

- [17] S. Debnath, N. Linke, C. Figgatt, K. Landsman, K. Wright, and C. Monroe, *Demonstration of a small programmable quantum computer with atomic qubits*, *Nature* **536**, 63 (2016).
- [18] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, J. M. Chow, and J. M. Gambetta, *Hardware-efficient quantum optimizer for small molecules and quantum magnets*, arXiv:1704.05018 (2017).
- [19] R. Hanson, L. P. Kouwenhoven, J. R. Petta, S. Tarucha, and L. M. K. Vandersypen, *Spins in few-electron quantum dots*, *Reviews of Modern Physics* **79**, 1217 (2007).
- [20] F. A. Zwanenburg, A. S. Dzurak, A. Morello, M. Y. Simmons, L. C. Hollenberg, G. Klimeck, S. Rogge, S. N. Coppersmith, and M. A. Eriksson, *Silicon quantum electronics*, *Reviews of Modern Physics* **85**, 961 (2013).
- [21] T. F. Watson, S. G. J. Philips, E. Kawakami, D. R. Ward, P. Scarlino, M. Veldhorst, D. E. Savage, M. G. Lagally, M. Friesen, S. N. Coppersmith, M. A. Eriksson, and L. M. K. Vandersypen, *A programmable two-qubit quantum processor in silicon*, *Nature* **555**, 633 (2018).
- [22] G. De Lange, Z. Wang, D. Riste, V. Dobrovitski, and R. Hanson, *Universal dynamical decoupling of a single solid-state spin from a spin bath*, *Science* **330**, 60 (2010).
- [23] 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* **6**, 6979 (2015).
- [24] P. W. Shor, *Scheme for reducing decoherence in quantum computer memory*, *Physical Review A* **52**, R2493 (1995).
- [25] A. Steane, *Multiple-particle interference and quantum error correction*, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **452**, 2551 (1996).
- [26] A. R. Calderbank and P. W. Shor, *Good quantum error-correcting codes exist*, *Physical Review A* **54**, 1098 (1996).
- [27] D. Gottesman, *Class of quantum error-correcting codes saturating the quantum hamming bound*, *Physical Review A* **54**, 1862 (1996).
- [28] H. Bombin and M. A. Martin-Delgado, *Topological quantum distillation*, *Physical Review Letters* **97**, 180501 (2006).
- [29] J. Koch, M. Y. Terri, 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*, *Physical Review A* **76**, 042319 (2007).
- [30] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, *Surface codes: Towards practical large-scale quantum computation*, *Physical Review A* **86**, 032324 (2012).
- [31] R. Versluis, S. Poletto, N. Khammassi, B. Tarasinski, N. Haider, D. J. Michalak, A. Bruno, K. Bertels, and L. DiCarlo, *Scalable quantum circuit and control for a superconducting surface code*, *Physical Review Applied* **8**, 034021 (2017).
- [32] S. Liu, X. Wang, L. Zhou, J. Guan, Y. Li, Y. He, R. Duan, and M. Ying, *Q|SI>: A quantum programming environment*, arXiv:1710.09500 (2017).
- [33] X. Fu, L. Rieseboos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsom, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, *eQASM: An executable quantum instruction set architecture*, Submitted (2018).
- [34] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, *Quipper: a scalable quantum programming language*, in *ACM SIGPLAN Notices*, Vol. 48 (ACM, 2013) pp. 333–342.
- [35] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, *Q#: Enabling scalable quantum computing and development with a high-level DSL*, in *Proceedings of the Real World Domain Specific Languages Workshop 2018* (ACM, 2018) p. 7.

-
- [36] M. V. Wilkes, *The best way to design an automatic calculating machine*, in *The early British computer conferences* (MIT Press, 1989) pp. 182–184.
- [37] Mentor Graphics, *Questa Advanced Simulator*, <https://www.mentor.com/products/fv/questa> (2018).
- [38] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels, *cqasm v1.0: Towards a common quantum assembly language*, (2018), arXiv:1805.09607 .
- [39] N. Wiebe and V. Kliuchnikov, *Floating point representations in quantum circuit synthesis*, arXiv:1305.5528 (2017).
- [40] M. R. Alex Bocharov and K. M. Svore, *Efficient synthesis of universal repeat-until-success circuits*, arXiv:1404.5320 (2017).

