Delft University of Technology

Quantum Control Architecture
Bridging the Gap between Quantum Software and Hardware

Fu, Xiang

DOI
10.4233/uuid:8205cc34-30df-45f0-b6eb-8081bdb765b8

Publication date
2018

Document Version
Final published version

Citation (APA)
Fu, X. (2018). *Quantum Control Architecture: Bridging the Gap between Quantum Software and Hardware*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:8205cc34-30df-45f0-b6eb-8081bdb765b8

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Quantum Control Architecture: Bridging the Gap between Quantum Software and Hardware

---

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof.dr.ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Tuesday 11 December 2018 at 15:00 o'clock

by

Xiang FU

Master of Engineering in Computer Science and Technology,
National University of Defense Technology, China
born in Yueyang, China

This dissertation has been approved by the promotors:
Prof.dr.ir. K. L. M. Bertels
Prof.dr. L. DiCarlo

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof.dr.ir. K. L. M. Bertels	Delft University of Technology, promotor
Prof.dr. L. DiCarlo	Delft University of Technology, promotor

Independent members:

Dr. M. Luján	University of Manchester, United Kingdom
Prof.dr. R. Wille	Johannes Kepler University Linz, Austria
Prof.dr. L. M. K. Vandersypen	Delft University of Technology
Dr. F. Sebastiano	Delft University of Technology
Prof.dr. S. Hamdioui	Delft University of Technology (reserve member)

Non-independent member:

Dr. C. G. Almudever	Delft University of Technology

Keywords: Quantum Instruction Set Architecture, Quantum Control Microarchitecture, Quantum Architecture Simulator

Printed by: Ipskamp Printing, Enschede

Front & Back: Designed with help from Dr. Yu Xin, Lin Chen, and Tiantian Du. The front cover is inspired by Michelangelo's painting *the Creation of Adam*: a fully programmable quantum computer requires collaboration of quantum software and hardware. The left hand is filled with the picture of the first waveforms measured from CBox_v3, which work forms the foundation of this thesis. The back cover is the picture (photographer: Jacob de Sterke) of one printed circuit board of QuTech Waveform Generator designed by Wouter J. Vlothuizen (TNO/QuTech), which implements the codeword-triggered pulse generation mechanism of QuMA.

Copyright © 2018 by X. Fu

ISBN 978-94-028-1305-0
An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

Dedicated to

Hua Xia Civilization
华夏文明

Acknowledgments

In the enormous space and long-lasting time, this thesis would be a trivial thing. However, treating this thesis as the most important fruit I harvested from my four-year work in TU Delft, I cannot easily get calm while seeing its birth. With great joy in my heart, I cannot forget you, who appear in my life and help me reach this achievement. At this moment, I realize how pale my words are when compared to my sincere gratitude to you.

My deepest thanks go to my promotors Prof. **Koen** Bertels and Prof. **Leonardo** DiCarlo. Thank you for providing me the golden opportunity to study quantum computer architecture in a "heterogeneous" environment composed of people from both the quantum computer architecture lab (QCAL) and the DiCarlo lab (DCL) with different working style, which enables me to grasp a lot of both hard and soft skills and produce considerable outcomes with the uncommon collaboration. Koen, thank you for letting me know how important it is to share a comfortable working environment and how to contribute to such an environment, for supporting me to develop my ideas, and for your delicious spaghetti dinners and a lot of fun parties at your place. Leo, thank you for teaching me to become a better collaborator, for teaching me to be strict in research, for your high standard which drove me to release my potential, and for your confidence in me and the support in turning my idea into real projects. Also, you set up a standard in my mind of a good teacher, like which I wish I could be in the future when teaching my students.

Carmina, thank you, thank you for your warmhearted greetings in every cold morning and your consideration in our daily life which made me feel QCAL was my second home outside China. I will never forget it. Thank you for your encouragement which strengthens my confidence in developing our architecture, and for allocating various resource required without condition in the CC-Light project which enables me focusing on the research work.

Grandpa **Jacob**, Mr. the Strong[1], you taught me so much, varying from the

---

[1] This is a joke by Leo: The surname of Jacob is *de Sterke*, which means *the Strong* in English.

basic VHDL programming, project quality control, project management, team dynamics, and so on. You are my mentor in engineering. Without you, I cannot develop the timing tape in CBox_v2 and later finish the QuMA processor in CBox_v3. Without you, the CC-Light project would be in a mess and cannot be controlled. I can still remember the moment that you are in such a hurry before the QuMA paper submission deadline to tell me that the description of the module frequencies is imprecise, which made me believe that our team together can do the best job in the world. Your jokes and 'dangerous' activities relaxed me so much in a high-stress working environment. I wish I could still play with you and hear you calling me "Processor Fu" even ten or twenty years later. But, could you please stop kicking my ass!?

Adriaan, you are more an engineer than a physicist to my understanding. You taught me how to write good code and ensure the code quality. You may not know what a treasure your analysis on my situation is when I was frustrated by other colleagues during the collaboration. You taught me always to think about things with a positive altitude, which helps me a lot. What enjoyable moments they were when we had beer together in the de Oude Jan after working for ten hours in the lab. I wish we could drink together more.

Leon, we have been working together for more than three years and produced interesting results. Thank you for teaching me various programming skills. We had so much fun together, and I wish we could still do it, no matter where you and I are. I wish you could enjoy your work in a satisfactory position in the future. **Imran**, you are always humble and productive. It is indeed a joy to work with you. Thank you for that. **Hans**, the discussion with you always challenged my understanding, which led us to the correct path to quantum computer architecture and quantum compiler. **Jeroen**, thank you for the classical pipeline of CC-Light you created in an amazing way. It was a supreme experience to work you and I learned a lot from it. Wish we could have more collaboration in the future. **Mengyu** （孟禹）, it was an enjoyable time that we created QuMAsim together. Thank you for providing me the opportunity to work with you. Wish you have a cozy life with **Xiaoxiao** （笑笑）in Shenzhen. **Miguel**, you are a gifted student. Without you, maybe we cannot finish the QCC project. Wish you can learn more and become an independent researcher in the future. My colleagues in QCAL, **Savvas**, **Dan**, **Andreas**, **Vieri**, **Nader**, **Aritra**, **Abid**, **Daniel**, **Amitab** thank you for your interesting discussions and a lot of fun we had together.

Niels, you always present a delighted face and cares about the feeling of colleagues in the lab. Thank you for bringing the relaxation in a relatively mentally-tense group. Please do not forget our mission to write the book *Niels and Xiang*[2]. **Ale**, can you be crazier regarding your ideas about the work we are doing? Thank you for inviting me to your parties. Thank you for making me feel being loved. Enjoy your happy moments with your family with Nolan! **Chris**, thank you for sharing the moment we imagine the future of quantum computing and discuss Go and *The Three-body Problem* together! Thank you for the music parties, where you make me envy you so much to play instruments wonderfully! **Ramiro**, it was a nice experience that we are getting more and more smooth and mature when working together. Thank you! **Stefano**, thank you for answering our entry-level questions in detail and patiently when we first stepped into the quantum computing field. It made us feel that people in the quantum computing field are always welcoming newcomers. **Flo, Brian, Tom, Thijs, Nadini, Xavi**, though I most of the time I cannot keep track of what you are talking about, I do enjoy seeing you treating the ideas seriously, which opens a window for me to see a new world. It is always a lot of fun to be with you. Thank you!

Raymond V. and **Raymond S.**, you are one of my most-important supporting force in my Ph.D. work. Without you, we cannot have the reliable hardware for CBox and CC-light, and I cannot learn so much about circuit design and test. Thank you for your generous contribution. **Wouter**, the discussion with you regarding distributed or centralized quantum architecture always remind me that my design might not be the right way. Without you, CC-Light can hardly have such a robust architecture and cannot work with other devices. Thank you for all of this. **Kelvin** and **Vincent**, it was an amazing experience that we work together in the CC-Light project. I wish we could have a chance to work together again. **Jules**, thank you for moment we imagine what electronics device should be in the future for quantum computing. **Gerco**, you maintains the embedded software and make it easy to use and stable in the DiCarlo lab. Thank you! **Stan**, thank you for making a fantastic theme (the cover of this book is developed based on this theme) and robust and flexible framework for Quantum Infinity, which we are all proud of. **Jordy**, you taught me how to use the Xilinx toolchain. Thank you. **Duije, Tessel, Jeremy, Garrelt, Nadia**, do you know how much I enjoy working with you? Thank you for all of this!

Lidwina and **Joyce**, every time I went to the sectary office, I can always get the required help from you. Your smiling faces form one of the most beautiful pictures of TU Delft. Thank you!

---

[2] Another joke by Leo: *Quantum Computation and Quantum Information* by Nielsen and Chuang is a Bible-like book in the quantum computing field. Leo said that: they have *Nielsen and Chuang*, but we have *Niels and Xiang*!

Anh, thank you for help revising my paper. More than ten pages of comments for the QuMA paper are what I will never forget. Without this, the QuMA paper can hardly reach such a neat structure and won the best paper award of MICRO-50. I learned so much about scientific writing from you. Wish you could get the expected experiment results and graduate soon. **Vlad**, **Mafalda**, **Giacomo**, thank you for inviting me to various activities (birthday party, Salsa party, Sushi dinner, etc.) when I first came to Delft. **Nelly** （惠莹）, **Hao** （浩哥）, **Jie** （美女师姐）, **Xiao** （薛潇）， **Guanzhong** （贯中）, **Xiaotong** （孝通）, **Roy**, it is so enjoyable that we could chat leisurely in Chinese in the lab. The QuTech Uitje organizers, Leon, Sophia, Daniel, Udit, Marta, Christian, Victoria, Maarten and Sjaak, thank you for creating such wonderful events. Though I only attend the one of 2018, it belongs to part of my best memory in the Netherlands.

Jian （总工） and **Yue** （越越）, thanks for your accompanying. We three came to TU Delft from China at the same time and enjoyed so much wonderful time. Cooking, skiing, digging oysters, swimming, jogging, and traveling, without you, my casual life could be a pale paper instead of a colorful page with a lot of memorable points. **Tiantian** （甜甜）, thank you for help me photoshop the system cube and improve the cover design. I wish you and Jian could lead a life with eternal happiness in the future. **Xing** （李星）, thank you for always counting me when cooking the rice when we stay in the same house. I wish you have a happy with Yue, and I am looking forward to seeing your shining moments at your wedding. **Xin** （郭昕）, thank you for your encouragements when sharing your experience with me. Our leisure chatting at the *party house* is one of my nice memory. Those words and encouragement helped me to better understand what I should puruse and how to puruse them. **Lingling** （玲玲）, being elegant, you are a rose in our lab. You are not only a colleague but also a best friend of mine. You are so clever and talented in cooking, which impressed me so much. Thank you for treating me with so much delicious food and sharing with me enjoyable moments. I wish we could continue our friendship and collaboration in the rest of our lives. **Zixuan** （子轩）， **Yu** （叔叔）, **Zhijie** （嫂嫂）, drinking, eating, and playing board games with the Werewolf group belongs to my best memory in Delft. I wish we could play these games and have more fun in China again in the future. Yu, I wish you could always enjoy more and more fun from painting!

Shanshan （珊珊姐）, thank you for your care during our time in Delft. **Jintao** （锦涛） and **Shi** （小胖）, thank you for your accompany with an optimistic attitude and humor. There has been a lot of fun with you. **Shengzhi** （小许）, my housemate, thank you for your delicious dishes. **Jiakun** （琨

哥）and **Xiaoyan**（赵晓燕，燕姐）, thank you for various help in the Warmoezierstraat house. 大佬们– **Qiang**（老刘）, **Xin**（小杜）, **Jian**（总工）, and **Yue**（越越）– thank you for the chatting and barbecue, which enriched my weekends time. 大佬们，华哥再也受不了你们的高级黑了，我先走一步了！Byebye. My friends, **Yande**（艳德）, Na（陈娜）, **He**（王贺）, **Lizhou**（利舟）, **Baozhou**（保周）, **Rong**（张荣）, **Shuai**（袁帅）, **YaZhou**（亚洲）, **Pengling**（鹏玲）, **Hai**（才子）, **Zhi**（洪智）, **Xiao**（林霄）, **Peiyao**（佩瑶）, **Yueting**（跃庭）, and my other Chinese friends, thanks to all of you, having dinners and chatting together exerts the loneliness from my life in Delft. **Zhijie**（嫂嫂）, **Xinyuan**（少将）, **Qile**（其乐）, **Dong**（李栋）, what a wonderful time we played and watched dota2 offline together! Thank you for that! My dota2 teammates, **Siyuan**（思远）, **Jin**（常进）, **Xueliang**（雪亮）, **Lin**（琳哥）, **Xiawei**（薇姐）, **Kefei**（克非）, thank you for your accompany helped me to spent countless boring nights. The hostess and host of Eastern Snack house, thank you for providing me so many delicious dinners. Wish your business gets larger and larger.

Xing（苏醒）, my faithful friend for more than ten years, thank you so much for helping me deal with various issues in China when I am in the Netherlands. Your integrity always reminds what a good man can be. I wish you can find a nice job in the future. I am also looking forward to the moment that we could work together again to create something changing the world. **Hongbo**（波波）, **Xiaoyan**（刘晓燕，燕姐）, and **Zhongyi**（大胡子）, you welcome me so warmly every time I returned to Changsha that I feel being missed by you. Thank you!

Yao（王耀）and **Sorin**, thank you for recommending me and enrolling me as a Ph.D. student to TU Delft. Without you, I will have no chance to study quantum computing in TU Delft. My supervisors and mentors in China, Dr. **Yue Wang**（王老师）, thank you for guiding me to lead a research career; Prof. **Zhengxiang Wei**（韦老师）, thank you for teaching me philosophy, which became the most powerful tool in my life; Prof. **Weixia Xu**（徐老师）and Prof. **Minxuan Zhang**（张老师）, thank you for supporting me all the time. Without you, I can hardly reach such a happy state by the end of my Ph.D.

“Tomato fried with onion can generate magical fragrance.” **Jiayi**（佳艺）, thank you for leaving the most colorful stroke in the paper of my life. You showed me the dimension of life about consideration and love with your touching activities, taught me how to respect and care about the feeling of others, and how to become a better man. Wish you could have a better life with a better partner.

Zhaokun（坤坤）, thank you for your accompany and bringing me so much happiness. Being as one of the three best chiefs in Delft (at least in my opinion), you cooked so much delicious food, which largely enriched my dining table. I wish you could have a happy life.

Finally, my deepest gratitude goes to my family. **Papa** and **Mama**, thank you for giving me so much. My sister, **Juan**（老姐）and my brother in law, **Liu**（姐夫）, thank you for taking care of our parents when I am not with them. You are the last pillars of my life in this world, anywhere anytime. I love you.

Xiang

Delft, November, 2018

Table of Contents

Acknowledgments i

Table of Contents vii

List of Tables xi

List of Figures xiii

List of Acronyms and Symbols xvii

1 Introduction 1
- 1.1 Computation Power Requirement 1
- 1.2 Towards Useful Quantum Computing 2
  - 1.2.1 Theoretical Challenge 2
  - 1.2.2 The Implementation Challenge 3
  - 1.2.3 Error Correction Challenge 3
  - 1.2.4 Control Challenge 4
- 1.3 Contribution and Thesis Organization 6

2 Background 9
- 2.1 Quantum Computing Basics 9
  - 2.1.1 Quantum Bit 9
  - 2.1.2 Quantum Gate 12
  - 2.1.3 Quantum Circuit 15
  - 2.1.4 Universal Quantum Gate Set 17
- 2.2 Quantum Algorithms 18
  - 2.2.1 Bernstein-Vazirani Problem 19
  - 2.2.2 Characteristics Analysis 21
- 2.3 Hardware Implementation 22
  - 2.3.1 Superconducting Qubits 23

2.3.2 Implication on the Control 24
2.4 Quantum Error Correction 28
2.5 Related Research 31
2.5.1 System Perspective 31
2.5.2 Quantum Programming Languages 32
2.5.3 Quantum System Structure 34
2.5.4 Efficiently Using Qubits 35
2.5.5 Quantum Control Microarchitecture 36
2.6 Quantum System Stack 37
2.7 Appendix 39

3 Experimental Quantum Microarchitecture 41
3.1 Introduction 41
3.2 Related Work 43
3.3 Microarchitectural Challenges 44
3.3.1 Motivational Example 44
3.3.2 Complex Analog Waveform Control 45
3.3.3 Instruction Definition 48
3.4 Quantum Microarchitecture 48
3.4.1 Codeword-Based Event Control 49
3.4.2 Queue-Based Event Timing Control 52
3.4.3 Multilevel Instruction Decoding 55
3.5 Evaluation 59
3.6 Implementation 60
3.6.1 Quantum Control Box 60
3.6.2 QuMA Implementation 62
3.7 Experimental Results 63
3.8 Potential Impact 67
3.9 Conclusion 68

4 eQASM: An Executable QISA 71
4.1 Introduction 71
4.1.1 Related Work and the Challenges 71
4.1.2 Contributions 72
4.2 eQASM Overview 73
4.2.1 Programming and Compilation Model 74
4.2.2 Design Guidelines 75
4.2.3 Architectural State 76

4.2.4 Instruction Overview 78
4.3 Architecture 81
4.3.1 Timing Model 81
4.3.2 Quantum Operation Definition & Decoding 83
4.3.3 Address Mechanism 83
4.3.4 Very Long Instruction Word 85
4.3.5 Fast Conditional Execution 87
4.3.6 Comprehensive Feedback Control 88
4.4 Instantiation & Implementation 88
4.4.1 Target Superconducting Quantum Chip 88
4.4.2 eQASM Instantiation Design Space Exploration . . . 89
4.4.3 Microarchitecture 93
4.4.4 Implementation 97
4.5 Experiment 98
4.6 Conclusion 101

5 Fault-Tolerant Quantum Microarchitecture 103
5.1 Introduction 103
5.2 Fault-Tolerant quantum computing 105
5.2.1 Quantum Error Correction 105
5.2.2 Fault-tolerant Logical Operations 106
5.3 Fault-Tolerant Control Microarchitecture 108
5.3.1 Qubit Addressing 109
5.3.2 Fault-Tolerant Logical Operations 111
5.3.3 Quantum Error Decoding 115
5.3.4 Measurement Result Unit 117
5.3.5 Pauli Frame 117
5.4 QUBE: The qubit plane 118
5.5 Conclusion 120

6 Quantum (Micro)architecture Simulator 121
6.1 Challenges in QuMA Development 121
6.2 Quantum Microarchitecture Development Flow 122
6.2.1 QuMA Development Flow 122
6.2.2 Drawbacks of the QuMA Development Flow 123
6.2.3 Potential Improvements 123
6.3 Design of QuMAsim 124
6.3.1 Requirement of QuMAsim 124

6.3.2 QuMAsim Implementation 125
6.4 Potential Applications 128
6.4.1 Design Space Exploration 128
6.4.2 RTL Verification 129
6.4.3 Full Stack Quantum Simulator 130
6.5 Conclusion 132

7 Conclusion & Outlook 133
7.1 Conclusion 133
7.2 Outlook 134

Bibliography 137

Summary 149

Samenvatting 151

Curriculum Vitae 153

List of Publications 155

List of Tables

2.1 Quantum circuit of some commonly-used multi-qubit gates. . 16

3.1 An example of the lookup table content of a codeword-triggered pulse generation unit for single-qubit gates. 51
3.2 Queue state of the *AllXY* experiment when $T_D = 0$. 54
3.3 Queue state of the *AllXY* experiment when $T_D = 40000$. . . . 54
3.4 Queue state of the *AllXY* experiment when $T_D = 40008$. . . . 54
3.5 The format of QIS instructions, quantum microinstructions. Taking the *AllXY* experiment as an example (Part I). 56
3.6 The format of micro-operations and codeword triggers. Taking the *AllXY* experiment as an example (Part II). 57
3.7 QuMIS instructions. 58

4.1 Overview of eQASM Instructions. 80
4.2 Definition of the micro-operation selection signal. 94

5.1 An example of the qubit symbol table, with the content recording the status of logical qubits in Fig. 5.5. 114

List of Figures

2.1 Geometrical representation of one qubit state using the Bloch sphere. 10

2.2 Quantum circuit of applying a single-qubit gate U_s on qubit q_0. 15

2.3 Quantum circuit of a multi-qubit gate U_m. 16

2.4 Example of a quantum circuit, which is the decomposition of the Toffoli gate as explained in Section 2.1.4. 17

2.5 Quantum algorithm for Bernstein-Vazirani problem. 19

2.6 Oracle used in the Bernstein-Vazirani algorithm. 20

2.7 Images at various scales of a transmon qubit coupled to a readout resonator in a planar circuit quantum electrodynamics chip. (a) Qubit (Q), resonator (R), flux-bias line (P_{F}), feedline input (P_{i}), and feedline output (P_{o}). (b) Zoom-in on the two Josephson junctions of the qubit. The magnetic flux threaded through the loop sets the qubit transition frequency f_{Q}. (c) Zoom-in on one of the two Josephson junctions. 23

2.8 In-phase and quadrature envelopes of X_π and Y_π pulses, including -50 MHz single-sideband modulation. 24

2.9 In classical execution, data travels undergoing operations one by one. In quantum execution, operations travel applying on qubits one by one. 25

2.10 Different to classical processors, a dedicated control unit is required to constructed which is separated from the quantum processor locating qubits. 26

2.11 Quantum controller based on AWGs and data collection cards. Colored lines denote analog signals and black lines denote digital signals. 26

2.12 All waveforms required by a T_1 experiment, which consists of multiple iteration with each iteration of a different idling time (τ) between the X_π pulse and the measurement. 27

2.13 Implementation of a distance-3 surface code, which comprises 17 qubits. 29

2.14 ESM circuit for the (a) $XXXX$ and (b) $ZZZZ$ stabilizer measurement, taking as example an X and Z ancilla as shown in Figure 2.13. 30

2.15 Overview of quantum computer system stack 37

2.16 Compiler infrastructure 38

3.1 Waveforms and timings for one round of the $AllXY$ experiment. 45

3.2 Overview of the Quantum MicroArchitecture (QuMA). 50

3.3 Operations of the *AllXY* experiment in the timeline. Measurement pulse generation and measurement result discrimination overlap in time and are shown using the same meter box. . . . 52

3.4 Schematic of the CBox firmware architecture. The QuMA core is implemented in the Master Controller. Dashed lines indicate functionality to be added in the future. 61

3.5 Experimental setup used for validation of the microarchitecture. 63

3.6 Schematic of the implemented QuMA. The thick gray lines are analog signals while the dark thin lines are digital signals. Dashed lines indicate functionality to be added in the future. . 64

3.7 The *AllXY* result of qubit 2. In the label, each *X*/*Y* (x/y) denotes a rotation by π ($\pi/2$) around the x/y axis of the Bloch sphere. 66

4.1 Heterogeneous quantum programming and compilation model. 74

4.2 Architectural state of eQASM. Arrows indicates the possible information flow. The thick arrows represent quantum operations, which reads information from the modules passed through. 76

4.3 Part of the code for a two-qubit *AllXY* experiment, which is used in validating eQASM in Section 4.5. 86

4.4 eQASM program for active qubit reset. This experimental result is shown in Section 4.5. 87

4.5 eQASM program using CFC. 89

4.6 Quantum chip topology of the target seven-qubit superconducting quantum chip. Numbers in red are the physical addresses of qubits. The numbers along the direct edges are addresses of the allowed qubit pairs. 90

4.7 Number of instructions for various architecture configurations for randomized benchmarking, Ising model, and square root. . 91

4.8 Format of the `SMIS` and `SMIT` (top two), `QWAIT` and `QWAITR` (middle two), and quantum bundle (bottom) instruction. 93

4.9 Quantum microarchitecture implementing the instantiated eQASM for the seven-qubit superconducting quantum processor. 95

4.10 Hardware structure implementing the instantiated eQASM for the seven-qubit superconducting quantum processor. Thin (thick) lines represent digital (analog) signals. 98

4.11 Two-qubit *AllXY* result, corrected for readout errors. 100

4.12 Single-qubit randomized benchmarking results for different intervals between gates. Dashed line indicates a 10% error rate for visual reference. 101

5.1 (a) The qubit layout for performing a CNOT gate between two logical qubits. Each logical qubit is encoded by a distance-3 rotated planar surface code where data qubits are on the vertices (solid circles) and ancilla qubits are on the plaquettes (open circles). The blue (red) squares and semi-circles represent stabilizers of the form $X(Z)^{\otimes 4}$ and $X(Z)^{\otimes 2}$, respectively. 'C' is the control qubit, 'T' is the target qubit, and 'A' is the ancilla. The joint M_{ZZ} on qubit 'A' and 'C' is realized by first performing ESM on the entire lattice in (b) for a merge and then performing ESM separately on the two lattices in (c) for splitting. 106

5.2 The ESM circuits (a) and (b) for measuring X- and Z-stabilizers ($X_{D5,D2,D4,D1}$ and $Z_{D6,D3,D5,D2}$). 106

5.3 The circuit to realize a measurement-based CNOT gate. 107

5.4 Overview of the Envisioned Fault-Tolerant Quantum Control Microarchitecture, FT_QuMA. 110

5.5 Virtual qubits mapped on physical qubits. Red is for the virtual address space and black for the physical address space. . . 111

5.6 Four flavours of a distance-3 surface code logical qubit using the same data qubits. It is assumed that redundant ancilla qubits are not used. The chirality of the logical qubit is left (right) when the physical qubit at $(x_{L,0}+1, y_{L,0}-1)$ $((x_{L,0}+3, y_{L,0}-1))$ is used as an ancilla qubit, where $(x_{L,0}+1, y_{L,0}-1)$ is the location of the logical qubit. The base ancilla type is X (Z) when the ancilla qubit at $(x_{L,0}+1, y_{L,0}-1)$ or $(x_{L,0}+1, y_{L,0}-1)$ is an X (Z) ancilla. 113

5.7 The number of instructions for performing one round of ESM on all alive logical qubits through compiler generation (top) and hardware generation (bottom). 116

5.8 Example QUBE architecture. 119

6.1 Quantum chip topology of the target 17-qubit superconducting quantum processor. 128

6.2 Format of the `SMIS` and `SMIT` for the 17-qubit quantum processor. 129

6.3 QuMAsim as used in a verification platform. 130

6.4 QVM: a full stack quantum simulator. 131

List of Acronyms and Symbols

2D	Two-dimensional
3D	Three-dimensional
ADI	Analog-Digital Interface
AWG	Arbitrary Wwaveform Generator
DDR	Double Data Rate
eQASM	executable Quantum Assembly
FPGA	Field Programmable Gate Array
FT	Fault Tolerant
GPU	Graphical Processing Unit
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IO	Input and/or Output
ISA	Instruction Set Architecture
NISQ	Noisy Intermediate-Scale Quantum
NN	Nearest-Neighbour
PF	Pauli Frame
QASM	Quantum Assembly
QEC	Quantum Error Correction
QECC	Quantum Error Correction Code
QED	Quantum Error Detection
QEX	Quantum Execution
QISA	Quantum Instruction Set Architecture
QuMIS	Quantum MicroInstruction Set
QuMA	Quantum MicroArchitecture
QuMAsim	Quantum MicroArchitecture Simulator
QVM	Quantum Virtual Machine
RTL	Register-Transfer Level
SIMD	Single Instruction Multiple Data
SOMQ	Single Operation Multiple Qubit
TELF	Timing Event Logging Format
VLIW	Very Long Instruction Word
VSM	Vector Switch Matrix

1

Introduction

1.1 Computation Power Requirement

There is a gap between understanding the principles of the world and full prediction of movement based on these principles, where computation plays an essential role. For example, Newton's laws describe the behavior of macroscopic objects but a significant amount of computation is required to predict the movement of them, for example in fluid dynamics. Besides, computation is also a crucial method for human beings to reform the world. Computing is the backbone of electronic devices in our daily life to provide various kinds of services, such as intelligent furniture, smartphones, and network-based services. New applications with higher computation power are continuously being developed to boost scientific research and improve our daily life, leading to a growing computation power requirement which has never been thoroughly satisfied.

Research on both software and hardware is being conducted to alleviate this problem. New algorithms and compilation techniques are being developed to reduce the computation required for given applications, and new architecture and new implementation technologies for computation and storage are being exploited to provide better performance. Domain-specific architectures to accelerate particular computing tasks [1] have gained increasing research attention since around 2000, including Graphical Processing Unit (GPU) [2] and Field Programmable Gate Array (FPGA) [3] for parallel computing or low power computing, Neural or Tensor Processing Unit (NPU/TPU) [4, 5] for machine learning, and so on. Novel device technologies are continuously being proposed to improve the performance of computation and storage, including memristors [6], carbon nanotubes [7], superconducting devices [8], phase change memory [9], etc.

However, these improvements are insufficient for particular types of problems such as simulating quantum chemical systems [10]. The limitation lies in the underlying computation model, the Turing machine [11] which is based on classical mechanics. Here, no efficient algorithms have been found to solve these problems. In this thesis, computers based on the Turing machine are called *classical computers* or *conventional computers*.

Addressing the inefficiency of using classical computers to simulate quantum systems, Richard Feynman proposed to utilize one quantum system to simulate another quantum systems in 1982 [10]. This idea was mathematically formulated by David Deutsch into the quantum Turing machine based on quantum mechanics in 1985 [12]. Later, the quantum Turing machine was theoretically demonstrated to be more efficient than the Turing machine [13]. Computers based on the quantum Turing machine are called *quantum computers*. Ever since then, research in quantum computing started to flourish.

1.2 Towards Useful Quantum Computing

1.2.1 Theoretical Challenge

The quantum Turing machine is a mathematical model. To make it more operable regarding algorithm design and physical implementation, David Deutsch proposed a mathematically equivalent model, called the circuit model [12]. A quantum computer built based on the circuit model is called the *standard quantum computer*, which is the focus of this thesis.

Since any quantum computer is difficult and expensive to build, a valid quantum algorithm is required to be faster (i.e., with a lower computational complexity) than any classical algorithm solving the same problem achieving the so-called *quantum speedup*. Ethan Bernstein and Umesh Vazirani demonstrated in theory that a quantum computer could be faster than any classical computer for some problem in 1996 [13]. Peter W. Shor proposed a quantum algorithm for prime factoring which can crack the widely-used Rivest-Shamir-Adleman (RSA) public key cryptosystem. Shor's factoring achieved exponential speedup over its best classical counterpart, firmly showing that quantum computers can solve real-world problems that are intractable by classical computers. Ever since then, more quantum algorithms have been designed for various applications, such as quantum simulation [14, 15], search [16], machine learning [17], and solving graph and algebraic problems [18, 19], etc.

1.2.2 The Implementation Challenge

Quantum computing should be performed on a physical system satisfying particular conditions. Some work has tried to summarize the required conditions for such physical systems, among them the most well-known is the DiVincenzo criteria [20], which are:

1. A scalable physical system with well-characterized qubits;
2. The ability to initialize the state of the qubits to a simple fiducial state, such as $|000\cdots\rangle$;
3. Long relevant coherence times, much longer than the gate operation time;
4. A "universal" set of quantum gates; and
5. A qubit-specific measurement capability.

Since Serge Haroche and David J. Wineland demonstrated the measurement and manipulation of individual quantum systems in 1980s [21], multiple quantum technologies have been investigated to implement qubits, such as nuclear magnetic resonance [22, 23], trapped ions [24, 25], quantum state in superconducting circuits [26–28], nuclear spins or electron spins in silicon [29–31], nitrogen-vacancy centers [32, 33], and photon polarization modes [34]. Some technologies have been demonstrated to have inherent scalability issues, such as nuclear magnetic resonance [35]. Some technologies are still in the early stage, such as spins in silicon and topological quantum computing based on Majorana. At the time of writing, the most promising technologies are trapped ions and superconducting qubits, both of which have demonstrated to satisfy the DiVincenzo criteria [25–27, 36].

Different from classical bits, qubits usually suffer from a short decoherence time. It implies that only a limited number of operations can be applied on the qubits before significant errors accumulate overriding the computation result. This forms a challenge for quantum computing since most quantum algorithms require an execution time much longer than the available qubit decoherence time.

1.2.3 Error Correction Challenge

In 1995, Peter W. Shor [37] proposed to encode quantum information into a *logical qubit* using a group of *physical qubits* according to some encoding scheme called *quantum error correction code* (QECC), and continuously correct errors that occur during computing. It enables storing information in the qubits for a time longer than individual qubit, which makes it possible to execute long quantum algorithms on imperfect qubits. This is the basic idea of quantum error correction (QEC) and fault-tolerant quantum computing, which forms an essential part of large-scale standard quantum computers.

Fault-tolerant quantum computing proposes much higher requirements on the physical implementation of a quantum computing system. First, multiple physical qubits are used to encode a logical qubit, which significantly increases the number of qubits required (usually by one to four orders of magnitude [38, 39]). Second, QECC can produce a logical qubit better than a physical qubit only if the physical qubit has an error rate lower than a certain threshold, called the *fault-tolerance threshold*. The threshold is usually very low and difficult to achieve with current physical systems, which ranges from 10^{-2} to 10^{-6} depending on the error correction scheme [38, 40].

QECC proposes a set of conditions on the interaction among qubits and the operation sequence applied on the qubits, and it can significantly affect the quantum processor structure and the quantum compiler design. A lot of works have provided proposals regarding how to organize qubits using given quantum technology targeting a particular QECC to support universal fault-tolerant quantum computing [41–46, 46–49]. Quantum compilers [50–55] are incorporating more routines not only to free the programmers from highly patterned tasks including reversible circuit synthesis, fault-tolerant implementation, qubit mapping and scheduling, but also to reduce the required number of physical qubits and execution time.

Since the requirement for fault-tolerant quantum computing is too strict to be satisfied in the near term, recent research suggests performing a computing task that goes beyond the capability of state-of-the-art classical computers with so-called near-term Noisy Intermediate-Scale Quantum (NISQ) technology [56, 57] without QEC. This is also termed quantum supremacy [58]. The promising application candidates for quantum supremacy include quantum sampling [56, 59] and quantum simulation [15], etc.

1.2.4 Control Challenge

A workable quantum computer should comprise both software and hardware working seamlessly. Quantum software describes the computation steps of applications, which is converted and used to control the quantum hardware to perform state evolution.

On the one hand, although high-level quantum programming languages have been developed to simplify the description of quantum algorithms, current quantum compiler output is not directly executable on real qubits since it considers little low-level constraints, such as precise timing control on nanosecond scale.

On the other hand, in current experiments, quantum processors are controlled with well-defined electrical signals, e.g., microwave-frequency and base-band pulses, which require accurate parameters and timing. To satisfy the strict requirements on control signals, dedicated electronic devices are typically used to interface with the quantum processor. However, existing control methods introduce high resource consumption, long configuration times, and control complexity, all of which scale poorly with the number of qubits [60]. It is another challenge to develop a control microarchitecture which can scale up to control tens of qubits in the near term.

Similar to a classical instruction set architecture being the interface of classical software and hardware, a quantum instruction set architecture (QISA) supported by a quantum control microarchitecture is required to bridge the gap between quantum software and hardware. Quantum computer architecture and control microarchitecture are the research focus of this thesis.

In response to the trend of building a NISQ quantum computer targeting quantum supremacy, the work of this thesis shifted the focus from supporting a QECC, such as surface code, to supporting current quantum experiments and small-scale quantum algorithms and to satisfying the requirement of near-term QECC implementation as well.

In this thesis, we focus on transmon qubits [61] in planar circuit quantum electrodynamics [62]. This is a promising architecture for solid-state quantum computing where qubit measurement and a universal gate set [20], comprised of single-qubit gates (mainly x and y rotations) and the Controlled-Z (CZ) gate, have already achieved error rates lower than the fault-tolerance threshold for surface code [39]. These physical error rates are limited by T_1, whose state-of-the-art in this architecture is 30 - 100 μs [26, 63, 64]. Recent experiments have demonstrated basic quantum error correction for this architecture, including the repetition code [26, 27] and elements of the surface code [65]. In addition, several cloud quantum computing platforms with tens of superconducting qubits are also available [64, 66, 67] at the time of writing. However, since it is still not clear which quantum technology would be the winner quantum technology, we also keep in mind trying to make the QISA and control microarchitecture capable of controlling various quantum technologies as much as possible.

1.3 Contribution and Thesis Organization

We start by introducing the background of this work and reviewing the related research in Chapter 2. To construct a fully programmable circuit-model-based quantum computer, our research started by clarifying all layers of a quantum computer from an architectural point of view as presented at the end of this chapter.

As explained in Section 1.2.4, existing control methods are insufficient regarding scalability and flexibility. To address this issue as well as to provide a programmable interface to quantum software, we propose an experimental control microarchitecture, QuMA, for superconducting quantum processors in Chapter 3. QuMA is highlighted by codeword-based event control, queue-based precise timing control, and multilevel instruction decoding. QuMA can execute a low-level quantum microinstruction set named QuMIS. We demonstrate QuMA and QuMIS by performing a standard gate-characterization experiment on a transmon qubit.

However, QuMIS suffers from no feedback, limited scalability due to low instruction information density, and limited flexibility due to being tightly bound to the electronic hardware implementation. To address these issues, we propose in Chapter 4 an executable QISA, eQASM (short for executable QASM). eQASM can be translated from the quantum assembly language (QASM), supports feedback, and is executed on an upgraded version of QuMA, QuMA_v2. eQASM alleviates the quantum operation issue rate problem by efficient timing specification, single-operation-multiple-qubit execution, and a very-long-instruction-word architecture. The definition of eQASM focuses on the assembly level to be expressive. Quantum operations are configured at compile time instead of being defined at QISA design time. We instantiate eQASM into a 32-bit instruction set targeting a seven-qubit superconducting quantum processor. We validate our design by performing several experiments on a two-qubit quantum processor.

Both QuMA and QuMA_v2 targets controlling NISQ devices without QEC. QEC is essential for large-scale quantum computing, which requires highly patterned control over a large number of qubits. Addressing this issue, we envisioned a heterogeneous microarchitecture, FT_QuMA, targeting quantum computing in the long term. Since QEC is essential for fault-tolerance implementation and surface code has a relatively high error threshold, we choose planar surface code as the underlying QECC. FT_QuMA supports runtime virtual-physical address translation, introduces a microarchitectural scheme for quantum error correction and detection, and a hardware mechanism that substantially reduces the codesize of the executable and reduces the execution overhead. This work is introduced in Chapter 5.

As the quantum control microarchitecture incorporates more functionality and controls more qubits, the hardware complexity grows correspondingly. It poses more challenge in the design, implementation, and verification of the control microarchitecture. To enable efficient design, development, and verification of the quantum (micro)architecture, we propose QuMAsim, a cycle-accurate (micro)architecture simulator based on SystemC, which can simulate the electronic part of the microarchitecture. By connecting QuMAsim with a qubit state simulator with a precise error model, such as QuantumSim [68], we can construct a full-stack simulator for NISQ technology, that we call *Quantum Virtual Machine* (QVM). Apart from the correctness of quantum program semantics at various levels, QVM can also verify low-level hardware constraints, including electronic control constraints and timing. This work is introduced in Chapter 6.

Chapter 7 is a conclusion and an outlook of the future work.

2

Background

This chapter starts with introducing quantum computing basics in Section 2.1. Section 2.2 performs an analysis of an example quantum algorithm, which reveals some features of quantum algorithms and gives an impression of the source of quantum speedup. In Section 2.3 we present the hardware implementation of superconducting qubits and the prevailing control methods, based on which our control architecture is built. Since quantum computing suffer from short coherence time of qubits and erroneous quantum operations, QECC is required for large-scale quantum computing, which is introduced in Section 2.4. Section 2.5 briefly reviews previous work related to this thesis. As a conclusion, Section 2.6 presents our proposed full stack of multiple layers for a quantum computer.

2.1 Quantum Computing Basics

2.1.1 Quantum Bit

Superposition

A classical bit has two exclusive states, 0 or 1, and can only be in one of them at any instant. In contrast, the elementary unit of quantum computing, the quantum bit or *qubit*, can exist in a superposition of its two basis states, $|0\rangle$ and $|1\rangle$, which is mathematically described by

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{2.1}$$

where

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

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

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \tag{2.3}$$

As constrained by Eq. 2.2, α and β can be rewritten as

$$\alpha = |\alpha|\, e^{i\gamma}, \quad \beta = |\beta|\, e^{i(\gamma+\varphi)}.$$

Let

$$\theta = 2 \arctan \left| \frac{\beta}{\alpha} \right|,$$

then we get

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

Since the global phase $e^{i\gamma}$ has no observable effects in quantum mechanics, it can be ignored and the qubit state can be further simplified into

$$|\psi\rangle = \cos\frac{\theta}{2} |0\rangle + e^{i\varphi} \sin\frac{\theta}{2} |1\rangle, \tag{2.4}$$

with φ, θ satisfying

$$\varphi \in [0, 2\pi) \quad \text{and} \quad \theta \in [0, \pi]. \tag{2.5}$$

The state of a single qubit can be intuitively visualized as a unique unit vector on the Bloch sphere as shown in Figure 2.1. The Bloch sphere is a unit sphere.

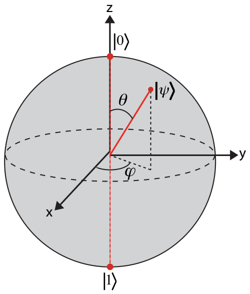


Figure 2.1: Geometrical representation of one qubit state using the Bloch sphere.

The basis state $|0\rangle$ ($|1\rangle$) corresponds to the intersection of the positive (negative) z-axis and the sphere, and any superposition state corresponds to a point on the sphere. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ are represented by the intersections of the positive and negative x-axis and the Bloch sphere, respectively. $\frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$ and $\frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$ are represented by the intersections of the positive and negative y-axis and the Bloch sphere, respectively.

In classical computers, the value read from a memory is identical to the information stored. This readout process is deterministic and no information is lost. In stark contrast, reading a qubit can only return one bit information even though the qubit can be in a superposed state where the coefficients can be continuous. This is done by a *measurement* of the qubit. During measurement, the qubit is projected onto $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively, and a corresponding bit of classical information is returned. Since the original state cannot be constructed based on the single measurement result, the information originally stored in the qubit is lost. Addressing destructive quantum measurement, quantum algorithms usually contain a stage before the final measurement to convert the computation result into a format represented in the basis state.

Entanglement

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 one state at one time. In a quantum system, each qubit of n independent qubits can be in superposition, with the entire system state described by the tensor product of each qubit state:

$$\begin{aligned} |\psi\rangle = &(a_{n-1}\,|0_{n-1}\rangle + b_{n-1}\,|1_{n-1}\rangle) \otimes \\ &(a_{n-2}\,|0_{n-2}\rangle + b_{n-2}\,|1_{n-2}\rangle) \otimes \cdots \otimes (a_0\,|0_0\rangle + b_0\,|1_0\rangle) \end{aligned} \tag{2.6}$$

This is also called product state. Expand this product state, we can get a superposition of 2^n states:

$$|\psi\rangle = \alpha_{1\cdots 11}\,|1\cdots 11\rangle + \alpha_{1\cdots 10}\,|1\cdots 10\rangle + \cdots + \alpha_{0\cdots 00}\,|0\cdots 00\rangle\,, \tag{2.7}$$

where

$$\alpha_{i_{n-1}\cdots i_1 i_0} = \prod_{k=0}^{n-1}\left[(1-i_k)\cdot a_k + i_k \cdot b_k\right], \quad i_k \in \{0,1\}, \tag{2.8}$$

and the subscription of the bits in all kets ($|\cdots\rangle$) is omitted with the convention that each bit from left to right corresponds to the state of one qubit from $n-1$ to 0.

It can be easily verified that the state $|\psi\rangle$ is also normalized:

$$\sum_{i_{n-1}=0}^{1} \cdots \sum_{i_1=0}^{1} \sum_{i_0=0}^{1} |\alpha_{i_{n-1}\cdots i_1 i_0}|^2 = 1. \tag{2.9}$$

Taking the logical states $\{|00\cdots0\rangle, |00\cdots1\rangle, \cdots, |11\cdots1\rangle\}$ as the basis, we can also express $|\psi\rangle$ using the vector

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{2^n-1} \end{bmatrix}. \tag{2.10}$$

The product state in Equation 2.6 can always be expanded to the addition format in Equation 2.7. But, the reverse is not always possible, especially when qubits are not independent of each other, which phenomenon is called *entanglement*. For example, any one of the four Bell states cannot be represented as the product of two individual qubit states:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle), \tag{2.11}$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}} (|00\rangle - |11\rangle), \tag{2.12}$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}} (|01\rangle + |10\rangle), \tag{2.13}$$

$$|\Psi^-\rangle = \frac{1}{\sqrt{2}} (|01\rangle - |10\rangle). \tag{2.14}$$

2.1.2 Quantum Gate

On the one hand, superposition and entanglement provide an exponential state space as shown in Section 2.1.1. On the other hand, quantum operations are intrinsic parallel by the linearity of quantum mechanics. Both facts form the foundation for quantum computers to achieve speedup over classical computers. We briefly introduce the notion of quantum gates.

Single-qubit Gate

A qubit state can be modified by applying quantum gates on qubits. Every single-qubit gate is a rotation $R_{\hat{n}}(\theta)$ on the Bloch sphere along an axis $\hat{n} = (n_x, n_y, n_z)$ ($|\hat{n}|^2 = 1$) by an angle θ. In the logical basis, such a quantum gate can be represented by the unitary matrix

$$R_{\hat{n}}(\theta) = \cos\left(\frac{\theta}{2}\right) I - i\sin\left(\frac{\theta}{2}\right)(n_x X + n_y Y + n_z Z),$$

where

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \ Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \ Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.15}$$

are the standard Pauli gates. For example, $R_x(\pi)$, $R_y(\pi)$, and $R_z(\pi)$ rotate the qubit by π along the x, y, and z axis, respectively. The effect of $R_x(\pi)$ is to exchange $|0\rangle$ and $|1\rangle$ of the target qubit, which is called bit-flipping. The effect of $R_z(\pi)$ is to negate the coefficient of $|1\rangle$ of the target qubit, i.e., change the phase of the qubit, which is called phase-flipping.

A quantum gate U updates the qubit state from $|\psi_i\rangle$ to $|\psi_o\rangle$. This process can be mathematically described using the matrix-vector multiplication

$$|\psi_o\rangle = U\,|\psi_i\rangle\,.$$

Take the $R_x(\pi)$ gate applied on the qubit with the state $|\psi_i\rangle$ as an example. Since

$$R_x(\pi) = -i\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \text{and} \quad |\psi_i\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

The output state $|\psi_o\rangle$ will be:

$$|\psi_o\rangle = R_x(\pi)\,|\psi_i\rangle = -i\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = -i\begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}. \tag{2.16}$$

As shown in Eq. 2.16, the global phase of a quantum gate turns into that of the qubit state, which has no observable effects either. Hence, we can also ignore the global phase of a quantum gate and write

$$R_x(\pi) = -i\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Multi-qubit Gate

Apart from quantum gates operating only a single qubit, there are also gates operating more than one qubit. The two most common two-qubit gates are the controlled-not (CNOT) gate and the controlled-phase (CPhase) gate with the corresponding matrices in the two-qubit logical basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$:

$$\text{CNOT} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \text{CZ} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

The effect of a two-qubit gate applied on two qubits is also described by matrix-vector multiplication. For example, two qubits with the original state $|\psi_i\rangle$ with

$$|\psi_i\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

will be transformed to the state $|\psi_o\rangle$ by a CNOT gate:

$$|\psi_o\rangle = \text{CNOT} \cdot |\psi_i\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_3 \\ \alpha_2 \end{bmatrix}.$$

This process can be understood intuitively using the following mapping:

$$\begin{aligned} |00\rangle &\rightarrow |00\rangle; \\ |01\rangle &\rightarrow |01\rangle; \\ |10\rangle &\rightarrow |11\rangle; \\ |11\rangle &\rightarrow |10\rangle. \end{aligned}$$

In other words, the value of qubit q_0 in the basis is flipped if the state of qubit q_1 in the same basis is 1 (the so called controlled-not). Note, if the initial state is a superposition of the four states, the CNOT gate performs this mapping for all four states simultaneously, which is some intrinsic parallelism provided by quantum mechanics. CPhase gate is also called CZ gate because it can also be understood as applying a Z gate on qubit q_0 (or phase-flipping q_0) if the state of qubit q_1 in the same basis is 1:

$$\begin{aligned}
|00\rangle &\rightarrow |00\rangle ; \\
|01\rangle &\rightarrow |01\rangle ; \\
|10\rangle &\rightarrow |10\rangle ; \\
|11\rangle &\rightarrow -|11\rangle .
\end{aligned}$$

Both the CNOT gate and the CZ gate can map certain product states onto entangled states. In general, Two- or more-qubit gates are essential to create entanglement among qubits for quantum computing.

Extending the previous understanding, we can define an n-qubit gate, which is described by a $2^n \times 2^n$ matrix. For example, the Toffoli gate, which is also called the Controlled-Controlled NOT (CCNOT) gate, applies on three qubits with the behavior described by a 8×8 matrix:

$$\text{Toffoli} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.17}$$

2.1.3 Quantum Circuit

Analog to classical circuit which describes how operations are done over classical bits, a *quantum circuit* can be used to illustrate the process of applying quantum gates on qubits. In the quantum circuit, every qubit is represented by a horizontal line, and one operation by a block on the qubits. For example, applying a single-qubit gate U_s on qubit q_0 can be represented using the quantum circuit shown in Figure 2.2. If the qubit state before the gate U_s is $|\psi_i\rangle$, then the qubit state after the gate is $|\psi_o\rangle = U_s |\psi_i\rangle$.

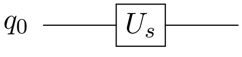


Figure 2.2: Quantum circuit of applying a single-qubit gate U_s on qubit q_0.

We can also use a quantum circuit to describe the m-qubit gate U_m as shown in Figure 2.3.

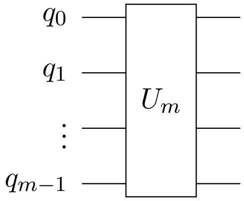
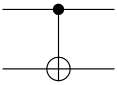
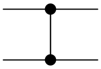
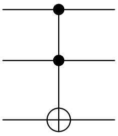


Figure 2.3: Quantum circuit of a multi-qubit gate U_m.

Some quantum circuit symbols are defined to intuitively represent the commonly-used quantum gates, as shown in the Table 2.1.

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

Gate Name	Gate notation	Matrix representation
CNOT		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
CZ		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
Toffoli		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

A quantum circuit can describe multiple quantum gates applied on multiple qubits. An example is shown in Figure 2.4, with the H, T, $T^\dagger$, and S gates

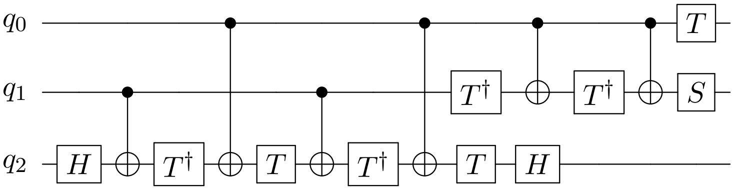


Figure 2.4: Example of a quantum circuit, which is the decomposition of the Toffoli gate as explained in Section 2.1.4.

defined as following:

$$\begin{aligned} H &\equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, & T &\equiv \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}, \\ T^\dagger &\equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0\\ 0 & e^{-i\pi/4}\end{bmatrix}, & S &\equiv \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/2}\end{bmatrix}. \end{aligned} \tag{2.18}$$

Quantum operations applied on the qubits earlier are placed on the left side to the operations applied later. Hence, the quantum circuit in Figure 2.4 describes the following operation sequence:

$$\begin{aligned} &T_{q_0} \cdot S_{q_1} \cdot \mathrm{CNOT}_{q_0,q_1} \cdot T^\dagger_{q_1} \cdot \mathrm{CNOT}_{q_0,q_1} \cdot H_{q_2} \cdot T^\dagger_{q_1} \cdot T_{q_2} \cdot \\ &\mathrm{CNOT}_{q_0,q_2} \cdot T^\dagger_{q_2} \cdot \mathrm{CNOT}_{q_1,q_2} \cdot T_{q_2} \cdot \mathrm{CNOT}_{q_0,q_2} \cdot T^\dagger_{q_2} \cdot \\ &\mathrm{CNOT}_{q_1,q_2} \cdot H_{q_2} \end{aligned} \tag{2.19}$$

Note, the order of gates in the mathematical expression is inverse to that in the quantum circuit as shown in Figure 2.4, which is consistent with the order of multiply multiple consecutive matrices with an input vector.

2.1.4 Universal Quantum Gate Set

A classical computing process can be generally treated as a mapping from n-bit value x to m-bit value y:

$$f : x \to y \tag{2.20}$$

According to Boolean algebra, we can represent any given function f with a set of three gates $S_{\mathrm{UC}} = \{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ using some methods such as Karnaugh Mapping [69]. Hence, S_{UC} is *universal* in classical computation.

In analogy, a set of quantum gates is called universal [40] if it can be used to approximate any m-qubit operation U_m to arbitrary precision, where $m \geq 1$. A commonly-used universal quantum gate set is $\{H, T, \text{CNOT}\}$.

For example, a Toffoli gate can be decomposed using $\{H, T, \text{CNOT}\}$ with the quantum circuit as shown in Figure 2.4, the mathematical expression as shown in Eq. 2.19, or a quantum assembly language (QASM) description [50] as shown in Listing 2.1. Note, the $T^\dagger$ and S gates can be implemented using seven and two concatenated T gates, respectively.

```
H          q2
CNOT       q1, q2
TDAGGER    q2
CNOT       q0, q2
T          q2
CNOT       q1, q2
T          q2
CNOT       q0, q2
TDAGGER    q1
T          q2
CNOT       q0, q1
H          q2
TDAGGER    q1
CNOT       q0, q1
T          q0
S          q1
```

Listing 2.1: QASM code for the decomposition of the Toffoli gate.

As shown in Listing 2.1, the quantum operation is written down in the order that they appear in the quantum circuit from left to right. Every quantum operation occupies a line in QASM. One quantum operation contains one operation name and a list of parameters consisting of the target qubits.

2.2 Quantum Algorithms

In this section, we illustrate how to design an algorithm to solve a problem using the mechanisms as explained in the previous section. We take Deutsch's algorithm [12] as the example to illustrate this process because it is simple to understand and complex enough to reveal the features of a quantum algorithm. Later, we summarize some features of quantum algorithms.

2.2.1 Bernstein-Vazirani Problem

To illustrate the power of quantum computing, Bernstein and Vazirani proposed the following problem in 1993 [13]. Given is the function f which takes an n-bit value x as input:

$$f(x) = x \cdot s = \left(\sum_{i=0}^{n-1} x_i \cdot s_i \right) \mod 2 \tag{2.21}$$

where x_i (s_i) is the i-th bit of x (s). The problem is to find out the n-bit value of s. It is assumed that a black box (or *Oracle*) $\mathcal{O}$ is given which has implemented the function f. During a *query*, the oracle $\mathcal{O}$ returns the result of $f(x)$ when fed with an input x.

A classical implementation of $\mathcal{O}$ can only return one-bit information in each query. Hence a classical algorithm needs to query $\mathcal{O}$ for n times to determine all n bits of s. As shown in [13], it is possible to design a quantum algorithm which can determine s by a single query to a quantum implementation of $\mathcal{O}$ using $n + 1$ qubits. Figure 2.5 illustrates this quantum algorithm for the case $n = 8$.

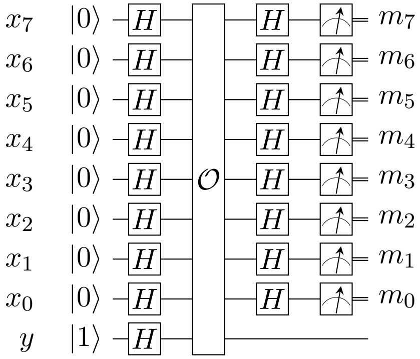


Figure 2.5: Quantum algorithm for Bernstein-Vazirani problem.

As shown in Figure 2.5, a quantum algorithm typically consists of four stages:

1. initializing qubits to some state easy to prepare, which is usually the computational basis;
2. preparing the superposition of the input data by superposing the qubits;
3. process all input data in parallel *using entanglement*; and
4. computational result extraction.

In the Bernstein-Vazirani algorithm, the top n qubits are prepared into the $|0\rangle$ state and the bottom qubit the $|1\rangle$ state during initialization. The overall state of $n+1$ qubits can be written as:

$$|\psi_0\rangle = |0\rangle\,|0\rangle \cdots |0\rangle\,|1\rangle \tag{2.22}$$

The second step, preparing superposition is achieved by applying transversal H gates on every qubit, resulting in the following state:

$$\begin{aligned}
|\psi_1\rangle =& H\,|0\rangle \cdot H\,|0\rangle \cdots H\,|0\rangle \cdot H\,|1\rangle \\
=& \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \cdot \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \cdot \cdots \cdot \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \cdot \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \\
=& \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{n-1} |x\rangle \left(|0\rangle - |1\rangle\right).
\end{aligned}$$

The oracle $\mathcal{O}$ implements a function g based on f:

$$g : (x,\ y) \rightarrow (x,\ [y + f(x)] \mod M), \tag{2.23}$$

where $M = 2^m$ and m is the number of qubits used by y. As an example, using reversible circuit synthesis, we can implement the oracle for function g with $s = \overline{10110010}$ using the circuit as shown in Figure 2.6. Oracle for other value of s can be implemented in a similar way.

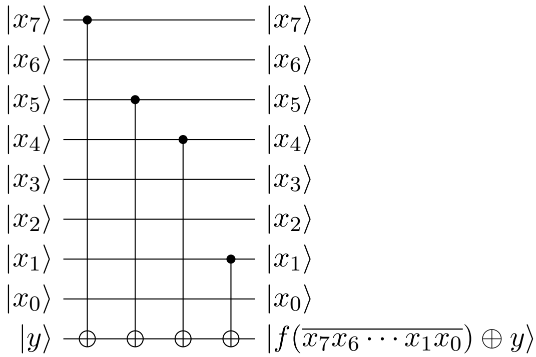


Figure 2.6: Oracle used in the Bernstein-Vazirani algorithm.

Ruled by the linearity of quantum mechanics, if the input data to a quantum oracle is superposed, the output of the oracle is the superposition of the result for each input data. In other words, the oracle $\mathcal{O}$ applies the function g on each input data $|x\rangle |j\rangle$ ($j \in \{0, 1\}$) simultaneously:

$$\begin{aligned} |0\rangle |j\rangle &\rightarrow |0\rangle |f(0) \oplus j\rangle , \\ |1\rangle |j\rangle &\rightarrow |1\rangle |f(1) \oplus j\rangle , \\ &\vdots \\ |n-1\rangle |j\rangle &\rightarrow |n-1\rangle |f(n-1) \oplus j\rangle . \end{aligned}$$

The resultant state after the oracle $\mathcal{O}$ can be written as:

$$\begin{aligned} |\psi_2\rangle =& \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{n-1} |x\rangle (|f(x)\rangle - |f(x) \oplus 1\rangle) \\ =& \frac{|0\rangle + (-1)^{s_{n-1}} |1\rangle}{\sqrt{2}} \otimes \cdots \otimes \frac{|0\rangle + (-1)^{s_1} |1\rangle}{\sqrt{2}} \otimes \\ & \frac{|0\rangle + (-1)^{s_0} |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \end{aligned} \tag{2.24}$$

For brevity, the deduction procedure which converts the addition format of the state $|\psi_2\rangle$ into the product format is shown in Section 2.7.

After the oracle, the result s has been encoded into the phase of one qubit of x. The last stage, computation result extraction, is done by a smart post-processing and measurements. The goal of the post-process is to convert the result encoded in the phase of each qubits into encoded in the basis state. It is done by applying transversal H gates on qubits of x. It can be easily verified that

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

so we can get:

$$|\psi_3\rangle = |s_{n-1}\rangle \otimes \cdots \otimes |s_1\rangle \otimes |s_0\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{2.25}$$

So the final measurement on qubits $\{x_i\}$ can return the unknown value s.

2.2.2 Characteristics Analysis

From the Bernstein-Vazirani algorithm, we could analyze some characteristics of quantum algorithms.

- Different to classical computers where data is stored and processed in different units, qubits are the place where data is both stored and processed in quantum computing. In other words, quantum computing is a kind of computation in memory. It hints that the conventional von Neumann architecture might not be the most suitable architecture for quantum computing.
- A quantum algorithm usually contains four stages: initialization, preparing superposition, data processing using entanglement, and computation result extraction.
- In classical computation, initialization data is copied to a memory which can be used later. In quantum computing, there is no clear definition of initialization data because qubits are usually prepared to the basis state during initialization. If there is any data used during the computation, it is input by applying corresponding quantum operations on qubits.
- The measurement can only reveal a small amount of data. This fact means that quantum computers are usually more suitable for problems of which both the input and output are short, but the computation process is complex [70].
- Quantum speedup comes from: 1) the exponential big state space allowed provided by superposition as shown by $|\psi_1\rangle$, and 2) every single quantum operation applied on entangled qubits can affects all superposed states simultaneously as shown by the oracle applied on $|\psi_1\rangle$.
- Unlike classical computation, where any data can be processed individually if not made in parallel intentionally, the parallelism of quantum operations is mandatory. This is the source of quantum speedup as well as a difficulty for designing new quantum algorithms.
- Since quantum computing is still much more expensive than classical computing, a valid quantum algorithm is required to have a lower complexity than its best classical counterparts [40]. This forms another challenge in designing quantum algorithms and one of the reasons why there are only a limited number of quantum algorithms till now [71].

2.3 Hardware Implementation

The work of this thesis focuses on transmon qubits [61] in planar circuit quantum electrodynamics [62].

2.3.1 Superconducting Qubits

Figure 2.7 shows images at various length scales of a transmon (Q) [63]. The transmon is a lumped-element nonlinear LC resonator consisting of an interdigitated capacitor in parallel with a pair of Josephson junctions providing nonlinear inductance. We use the ground state (first-excited state) of this circuit as the qubit $|0\rangle$ ($|1\rangle$) state. The transition frequency f_{Q} between these states can be tuned over several gigahertz on nanosecond timescales by controlling the flux through the loop between the two Josephson junctions using the proximal flux-bias line (port P_{F}).

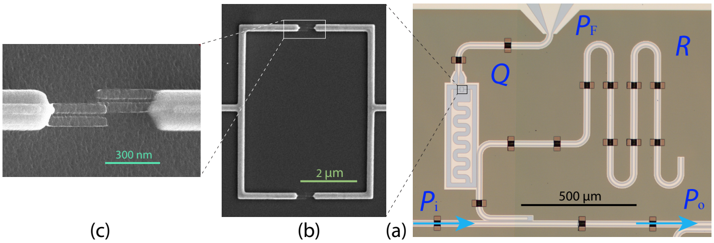


Figure 2.7: Images at various scales of a transmon qubit coupled to a readout resonator in a planar circuit quantum electrodynamics chip. (a) Qubit (Q), resonator (R), flux-bias line (P_{F}), feedline input (P_{i}), and feedline output (P_{o}). (b) Zoom-in on the two Josephson junctions of the qubit. The magnetic flux threaded through the loop sets the qubit transition frequency f_{Q}. (c) Zoom-in on one of the two Josephson junctions.

Qubit measurement exploits the qubit-state dependent fundamental frequency f_{R} of a coplanar waveguide resonator (R) which is capacitively coupled both to the transmon and to a feedline. A pulsed measurement (typically $300\ \mathrm{ns}$ - $2\ \mu\mathrm{s}$) of transmission through the feedline (from input port P_{i} to output port P_{o}) near the fundamental of R interrogates the qubit state, projecting it to $|0\rangle$ or $|1\rangle$. Demodulation, integration, and discrimination of the transmitted signal is used to infer the measurement result.

Single-qubit gates are performed by applying calibrated microwave pulses (with typical pulse length of $20\ \mathrm{ns}$) at f_{Q} to the feedline. These pulses are commonly generated by single-sideband modulation of a carrier using an I-Q mixer and envelope functions generated by an arbitrary waveform generator. The in-phase (I) and quadrature (Q) components of the envelope functions of x/y-rotations are shown in Figure 2.8. In this figure, the I and Q components both contain a -50 MHz single-sideband (SSB) modulation. The envelopes and the phase of the carrier determine the rotation axis along the equator of the Bloch sphere, and the amplitude of the pulse determines the rotation angle. Standard calibration routines are employed to determine the amplitude of specific pulses (e.g., π and $\pi/2$), and to correct for mixer imperfections. As explained in Section 2.1.4, arbitrary single-qubit gates can be decomposed into x- and y-axis rotations albeit at the cost of longer operation sequences.

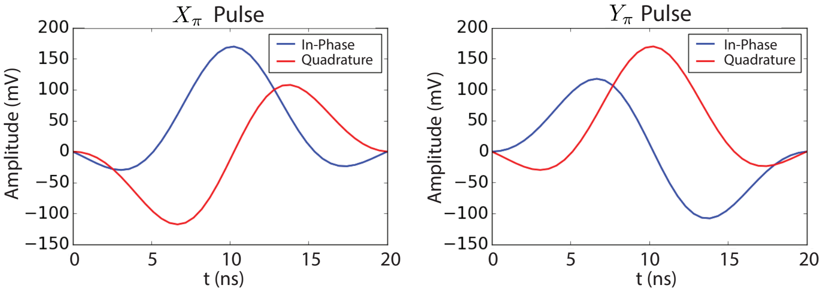


Figure 2.8: In-phase and quadrature envelopes of X_π and Y_π pulses, including -50 MHz single-sideband modulation.

In circuit quantum electrodynamics, the most common two-qubit gate is the CZ gate. Such a gate can be performed between qubits coupled to a common resonator or capacitor. It is realized by applying suitably calibrated pulses of typical duration ~ 40 ns to the flux-bias line that move the frequencies of qubits close to interact for a certain amount of time. We avoid going into further detail on CZ gates here as these are not part of this thesis. Please see [72–74] for details.

2.3.2 Implication on the Control

In classical processors, logical gates or functional units are implemented by a circuit composed of electronic elements such as transistors, which locates at a fixed position on the chip. To perform an operation, the operated data travels from the registers to the input ports of the corresponding functional unit implementing this operation, and the result is automatically generated by the functional unit. We say data is travelling and operations are stationary for classical computers (see the top half of Figure 2.9).

This is on the contrary for quantum computers where data is stored in qubits. Since most kinds of qubits are stationary, data carried by the qubits is also stationary[1]. To perform an operation, pulses are generated by an external con-

---

[1] Photon-based quantum computers are an exception, where data is encoded in photons which travel through different lens for operation. Throughout this thesis, we exclude photon-based quantum computers from our discussion.

troller and then applied on qubits. As a consequence, data is transformed in place. We say operations are travelling and data is stationary for quantum computers (see the bottom half of Figure 2.9).

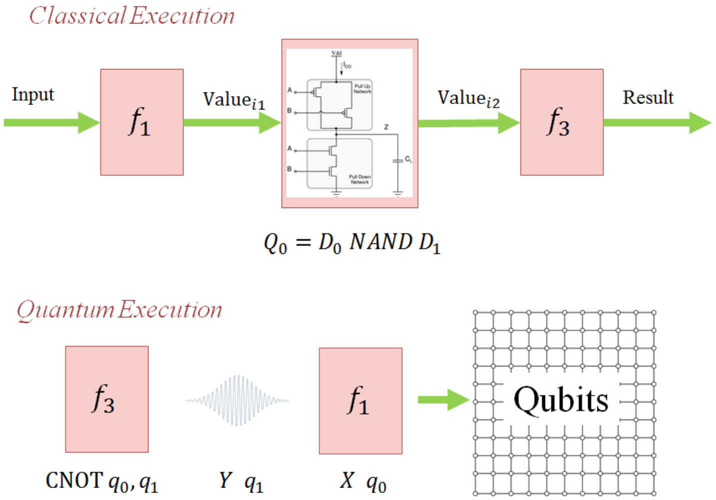


Figure 2.9: In classical execution, data travels undergoing operations one by one. In quantum execution, operations travel applying on qubits one by one.

In classical digital processors, data is stored and processed in binary format, which is represented by digital signals. The storage unit, registers or cache, and the processing unit, arithmetic/logic unit, can be implemented using the same basic elements, transistors. Apart from storage and processing, the control signals of the storage and the processing unit are also digital signals. The controller can be also implemented with transistors. The homogeneity among the storage, processing and control enables an easy integration of the storage unit, the processing unit and the control unit on a single chip.

In contrast, data is stored and processed in qubits which can be entangled. The data evolves in a format encoded in superposed states. The control medium over qubits are analog pulses, which are usually generated by arbitrary waveform generators. The heterogeneity between quantum data and quantum control leads to the requirement of separate design of the quantum controller to the quantum processor allocating qubits as shown in Figure 2.10.

To satisfy this requirement and as shown in Figure 2.11, a prevailing quantum controller is to use arbitrary waveform generators (AWG), such as Tektronix AWG5000 series [75], to generate analog control signals, and data collections

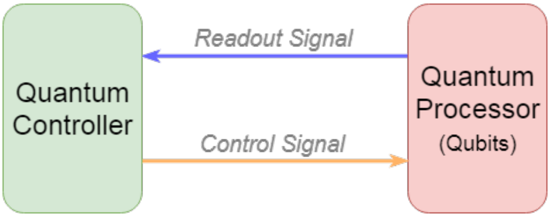


Figure 2.10: Different to classical processors, a dedicated control unit is required to constructed which is separated from the quantum processor locating qubits.

cards, such as AlazarTech PCI Express digitizers [76], to digitize the readout signals.

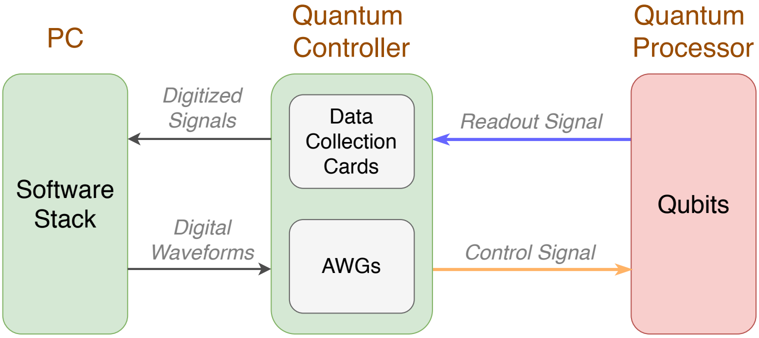


Figure 2.11: Quantum controller based on AWGs and data collection cards. Colored lines denote analog signals and black lines denote digital signals.

On the one hand, with such kind of AWGs, pulses for quantum operations are generated and combined by software on the PC into long waveforms according to the timing of each pulse. These waveforms are uploaded to the memory of the AWGs and later output by the AWGs autonomously. On the other hand, software-based integration over the digitized data is required to determine the measurement result. Both processes highly rely on the software running on the PC.

We take the T_1 experiment as an example to illustrate how to perform quantum experiments or algorithms using the quantum controller based on AWGs and data collection cards. The T_1 experiment is used to calibrate the relaxation time (T_1) of a qubit. This experiment involves the following steps:

1. Initialize the qubit in $|0\rangle$ by waiting several times T_1;
2. Excite the qubit to the $|1\rangle$ state by applying an X_π pulse;

3. Wait for a time τ;
4. Measure the qubit state, get measurement reuslt $R = 0$ or 1;
5. Repeat steps 1 to 4 with τ varying from $t_{\min}$ to $t_{\max}$ in K steps;
6. Repeat the above steps N times;
7. Calculate the raw fidelity $F_{|1\rangle}|_\tau = \sum_{j=0}^{N-1} R_j/N$ for each interval τ.
8. Extract T_1 by fitting the function $ae^{-\tau//T_1} + b$ to the $F_{|1\rangle}|_\tau$ data, with a, b, and T_1 as fitting parameters.

Steps 1 through 4 (5) are seen as one iteration (round) of the experiment. The required waveforms are shown in Figure 2.12.

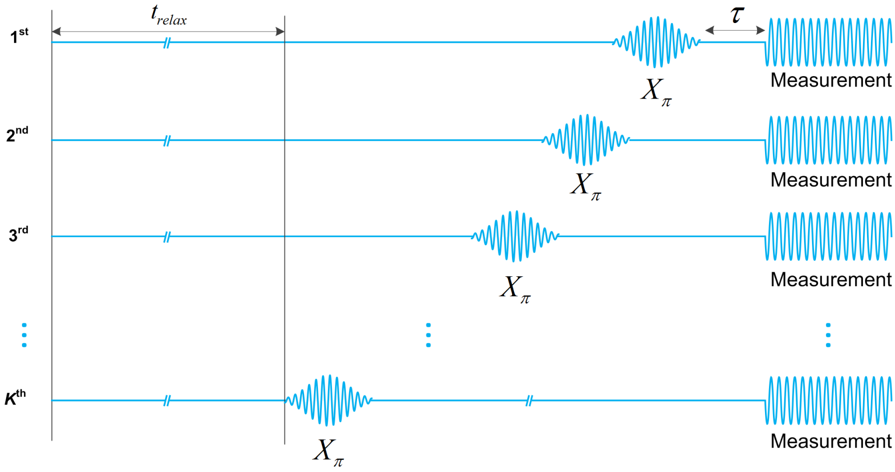


Figure 2.12: All waveforms required by a T_1 experiment, which consists of multiple iteration with each iteration of a different idling time (τ) between the X_π pulse and the measurement.

To use AWG5014 (one Tektronix AWG5000 series product) to perform this experiment, a commonly-used method is to prepare all the K waveforms at once, with each waveform corresponding to one τ. After all K waveforms are uploaded to the memory of AWG5014, AWG5014 generates waveform 1 to K consecutively and repeat this process until N rounds of waveforms have been generated.

As discussed in Section 3.3, this method suffers from:

- Awkward pulse generation because slight change in the timing of operations would require a re-generation of the entire waveform;
- High memory consumption to store long waveforms required by long experiments such as this T_1 experiment or Randomized Benchmark-

ing [77, 78]; and

- Difficult to implement feedback control because waveform uploading would take a time longer than the qubit coherence time.

2.4 Quantum Error Correction

Quantum computing is error prone because of two reasons. First, physical qubits have a very short coherence time, which means that qubits can loose its state during computation. Second, imperfect quantum operations can introduce errors when applied on qubits. Both facts calls for QEC to enable fault-tolerant quantum computing.

However, QEC is more challenging than classical error correction because of three reasons:

- According to the no-cloning theorem, an unknown quantum state cannot be copied, which makes it impossible to implement error correction by duplicating quantum states.
- Quantum errors are continuous, which cannot be corrected by directly applying a fixed set of operations.
- Quantum measurement may destroy the information stored in qubits, which means that error correction cannot utilize direct measurement on qubits storing the information.

The basic idea of QEC [37] is to use several physical imperfect qubits to compose more reliable units called logical qubits based on a specific QECC [79] such as surface code [39]. Such encoding does not need to clone the qubit state, as it is done by entangling several qubits called **data qubits**. Furthermore, possible errors in the logical qubit are detected by measuring some ‘helper’ qubits called **ancilla qubits**. In this way, the information in data qubits can be preserved. Ancilla are used to measure the stabilizers of nearby data qubits, which projects continuous errors into discrete errors. The stabilizer measurement results allow identifying whether there are errors, and if yes what kind (bit-flip, phase-flip or both) and in which qubit(s) the error(s) are. This kind of measurement is called error syndrome measurement (ESM). Since quantum errors accumulate as time elapses, ESM has to be done repeatedly.

It is worth noting that errors do not need to be corrected immediately. Instead, errors are tracked in classical logic using a technique called **Pauli Frame** [80]. Quantum operations and measurements are translated by the Pauli Frame, preserving the correctness of quantum computation.

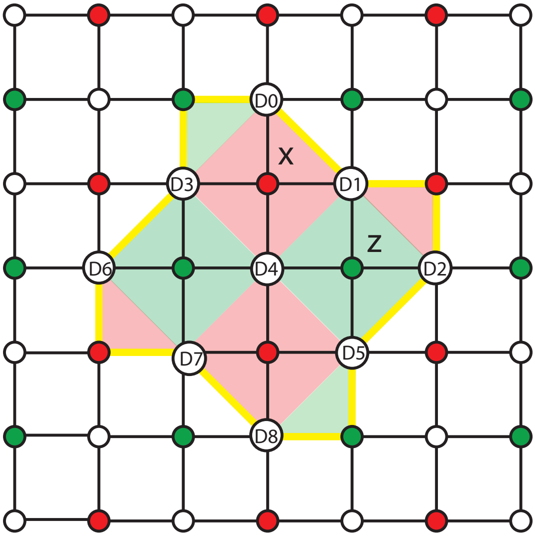


Figure 2.13: Implementation of a distance-3 surface code, which comprises 17 qubits.

One of the most promising and currently very popular QECC is **surface code** [39]. In surface code, qubits are arranged in a regular 2D lattice which only enables nearest-neighbour (NN) interaction (Figure 2.13). The NN lattice architecture is one of the most-promising structure both theoretically and experimentally [36, 42, 43, 81]. Surface code has an error threshold of $\sim 1\%$ meaning that it can tolerate a physical error rate up to 0.01.

In Figure 2.13, open circles represent data qubits and green and red filled circles correspond to Z and X ancilla qubits, respectively. In surface code, two kinds of stabilizers ($XXXX$ and $ZZZZ$) are measured, with the corresponding ESM circuits shown in Figure 2.14. The $XXXX$ ($ZZZZ$) stabilizer measurement can be used to detect phase-flip (bit-flip) errors using X (Z) ancilla. As we mentioned, these circuits will be repeatedly applied during computation. The interval between the starting point of two consecutive ESM is called a Surface Code Cycle (SC cycle). In surface code, the measurement results $+1/-1$ coming from several rounds of ESM, are then forwarded into classical logic where errors are identified (quantum error detection, QED) by using decoding algorithms, such as Blossom algorithm [82].

An important metric of a QECC is the code distance which is the minimum number of physical operations required to perform a logical operation, or the length of the shortest error chain that is undetectable [39]. For example, among all logical operations, a logical X gate on a distance-3 surface code costs three

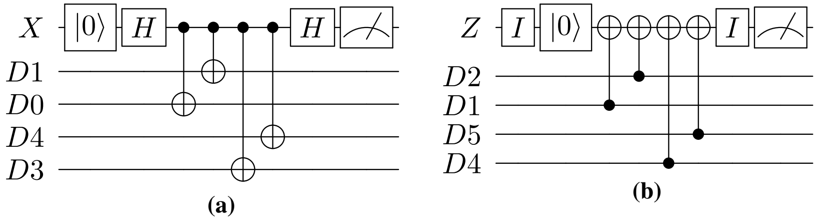


Figure 2.14: ESM circuit for the (a) $XXXX$ and (b) $ZZZZ$ stabilizer measurement, taking as example an X and Z ancilla as shown in Figure 2.13.

physical gates, which is the minimum number of physical gates required by any logical operation. In addition, three errors on a distance-3 surface code logical qubit can change the logical state without a detectable syndrome.

A set of logical operations are defined for surface code to enable fault-tolerant quantum computing. The operations include initialization, measurement, X, Z, H, T, and CNOT.

The initialization into the logical $|0\rangle$ ($|1\rangle$) state is implemented by:

- Resetting all data qubits in the $|0\rangle$ ($|1\rangle$) state.
- Waiting for three Surface Code Cycle, in each cycle a round of ESM is performed.
- The ESM result of the three Surface Code Cycle can help figure out error(s) that may occur. After correcting these errors, the final result is then a logical $|0\rangle$ ($|1\rangle$) state.

Measuring a logical qubit is implemented by measuring all data qubits (transversal measurement) in the Z basis. The product of the data qubit measurement outcomes ($+1$/-1) will yield the logical qubit measurement result. Note, errors during physical measurement can be diagnosed by classically checking the parity of the data qubits [83].

The logical X (Z) operations are implemented by applying a chain of X (Z) operations on the data qubits [39]. Taking the surface code in Figure 2.13 as an example, a logical X (Z) operation is performed by applying three physical X (Z) operations $X_2X_4X_6$ ($Z_0Z_4Z_8$).

The logical H gate is implemented by applying physical a H gate on every data qubit transversely ($H_0H_1H_2 \cdots H_8$). It is worth noting, after a logical H gate applied on the logical qubit, an X (Z) stabilizer will be turned into a Z (X) stabilizer, and the implementation of the logical X (Z) gate will be changed to $X_0X_4X_8$ ($Z_2Z_4Z_6$), taking the logical qubit in Figure 2.13 as an example.

The logical CNOT gate is implemented by performing pair-wise CNOT gates between data qubits of the two logical qubits. However, due to 2D NN constraint, it cannot be directly implemented on the 2D square lattice due to the NN constraint. A lattice-surgery-based approach can be used which is compliant with the NN constraint and explained in Chapter 5.

The implementation of the logical T gate requires magic state injection, where required are the states $(|0\rangle + e^{i\pi/4}|1\rangle)/\sqrt{2}$ and $(|0\rangle + i|1\rangle)/\sqrt{2}$, and logical CNOT and H gates. This process introduces significant overhead and is explained in detail in [39].

QEC allows fault-tolerant computation and is thus a fundamental part of large-scale quantum computing system. The drawback is that it dramatically increases the number of physical qubits. It also creates a large control overhead and requires a continuous and close interaction between the quantum chip and the classical platform that makes the quantum computing process more complex.

2.5 Related Research

2.5.1 System Perspective

To construct a workable, fully programmable, universal, circuit-model-based quantum computer, clarifying the required components of a quantum computer is essential.

Jones *et al.* [84] defined a layered control stack that step-by-step maps quantum applications into operations performed on physical two-level systems with imperfections. Virtual qubits with a particular error rate, quantum error correction, and logical qubits are the three abstraction layers in the between to support standard fault-tolerant quantum computing. Like the Open Systems Interconnection model (OSI model) [85], it is a conceptual model which characterizes and standardizes the communication functions of a fault-tolerant quantum computer without regard to its underlying internal structure and technology.

Van Meter *et al.* [86] proposed a design that consists of research areas including quantum complexity theory, quantum algorithms, QEC theory and implementation, interconnections and qubit storage, etc. This design is more a description of the research fields that are needed to address the quantum challenges rather than an implementable architecture.

Recently, addressing the correct abstractions that expose key low-level details to enable quantum software and hardware co-design, Chong, Franklin and Martonosi [87] defined the design tool flows and abstraction stack for quantum computing, which highlights the requirement of collaboration between classical and quantum parts.

In these works, the term ‘architecture’ is used to refer a system-level picture that can help understand how to construct a workable quantum computer. However, little attention has been paid in these works on translating the computation process as described by the quantum software into signals generated by the hardware to drive qubits, which drove us to propose our system stack for quantum computing as shown in Section 2.6.

2.5.2 Quantum Programming Languages

The first step of quantum computing is the program written in high-level or low-level quantum programming languages.

High-Level Languages

Various high-level quantum programming languages and quantum compilers have been proposed or developed to enable efficient description of quantum applications. Examples of imperative quantum programming languages based on C/C++ include Quantum Computation Language (QCL) [88], Q Language [89], and Scaffold [90], etc. Especially, Scaffold is supported by the quantum compiler ScaffCC [53], which is based on the widely-used LLVM infrastructure [91]. Another imperative language is the quantum while language [92], which is based on the while language [93] and highlights a formal specification of this language. Examples of functional quantum programming languages are Quipper [52] embedded in Haskell, LIQ$Ui\left|\right\rangle$ [51] embedded in F#, and ProjectQ [94] embedded in Python, etc. Treating the quantum computer as an accelerator, the domain-specific programming language Q# [70] defines a heterogeneous quantum computing model, in which a full quantum program comprises a host program described by a conventional programming language, like C#, and a quantum kernel described by Q#.

All these programming languages mostly aim at describing large-scale quantum computation at a high level in an efficient way, which is later translated by the compiler into a representation in a low-level quantum programming language.

Low-Level Languages

The development of low-level description of quantum applications started with QASM [40, 50] to provide an intuitive representation of the quantum circuit. As an interface to the quantum hardware, quantum physical operations language (QPOL) [50] is suggested to describe the execution of quantum applications with technology-dependent properties. However, no clear format or instructions are defined in QPOL. To enable hardware simulation that can quantitatively evaluate quantum architectures, Balensiefer [49] proposed a virtual instruction set architecture (virtual-ISA) based on the von-Neumann architecture, which is high level and cannot be directly executed on any physical machine. Also, branches are not allowed in the quantum program to enable easy simulation.

Being preliminary, QASM, QPOL, and the virtual-ISA are limited in several aspects, which attracts increasing attention of researchers in recent years. Addressing the exploded code size for large-scale quantum applications, Hierarchical QASM with Loops (QASM-HL) was proposed [53], which incorporates classical constructs like loops and function definition to suppress the code size. To enable analysis and optimization over programs with quantum-classical mixed instructions, a quantum instruction language, Quil, was proposed to describe programs that can run on the quantum abstract machine (QAM) [95]. QAM is similar to the hypothetical computer as used by Knuth Donald in his book *The Art of Computer Programming* [96]. Quil is at a level higher than normal QASM formats since it contains structured descriptions for both quantum gates and program routines, which should be compiled into a lower-level format for execution. OpenQASM [97] was proposed to support efficient description of quantum applications that can be executed by IBM cloud quantum computing platform, Quantum Experience [64]. OpenQASM is an assembly language that allows structured definitions of quantum gates based on a universal definition for single-qubit unitaries and the two-qubit CNOT gate. OpenQASM has limited support for classical operations although the **if** statement is allowed.

All these low-level languages target to be mathematically equivalent to the circuit model in an efficient way, with or without interacting with classical computing resources. They cannot be translated into an executable binary format that can be directly executed by a quantum control microarchitecture, which leaves a gap between quantum software and hardware.

2.5.3 Quantum System Structure

For every promising quantum technology, a scalable architecture has been proposed to organize qubits in a quantum system. The first attempt was made by Cirac and Zoller [98]. They suggested confining ion-based qubits in a linear trap, which can be operated by laser beams. Addressing the scalability issues raised by technical obstacles, Kielpinski, Monroe, and Wineland [99] proposed to organize qubits into an array based on techniques already demonstrated, which can support massive parallel gate operations for large-scale quantum computers.

Due to the relatively high fault-tolerance threshold, the surface code became a prevailing QECC to implement a fault-tolerant quantum computer. Surface code requires operating a two-dimensional array of qubits synchronously and in parallel. Comprehensive control over the qubits with sufficient operation fidelity and parallelism is required to support surface code, including initialization, quantum gates, and measurements.

DiVincenzo [42] proposed a two-dimensional (2D) array architecture with nearest-neighbor coupling for superconducting qubits which can effectively support surface code. As one step further, Versluis *et al.* [81] proposed to scale up this architecture by repeating the quantum hardware and coherent control of an eight-qubit unit cell. This architecture can execute the error correction cycle of a monolithic surface code, which can be either planar or defect-based surface code. The core trick for scaling up this architecture is to adopt the selective-broadcasting scheme as proposed by Assad and Dickel *et al.* [100]. This scheme enables independent control over qubits of the same frequency, which can reduce the electronic control resource, e.g., arbitrary waveform generator channels. Beside 2D architecture, Brecht *et al.* [101] proposed a three-dimensional (3D) multilayer microwave integrated quantum circuit platform for superconducting qubits to couple a large number of circuit components through controllable channels while suppressing any other interactions.

Loss and DiVincenzo [102] proposed a scalable semiconductor-based architecture which supports a universal set of single- and two-qubit gates for quantum computing. Kane [103] proposed an architecture based on nuclear spins of donor atoms with logical operations on individual spins performed by externally applied electric fields, and measurements by using currents of spin-polarized electrons. Addressing the fidelity loss due to swaps operations required by non-local quantum operations, Hollenberg *et al.* [41] proposed a quasi-2D architecture for nuclear spin qubits which enables subinterfacial transport of electron spins to support non-local qubit interactions.

Hill *et al.* [43] proposed an architecture with a novel shared-control paradigm to address the fabrication and control challenge in supporting surface code using nuclear spin qubits. Targeting spin qubits based on single electrons in gate-defined quantum dots, Li *et al.* [44] proposed a crossbar network architecture based on shared control and a scalable number of lines which enables qubit coupling beyond nearest neighbors, providing prospects for non-planar quantum error correction protocol.

In these words, the term ‘architecture’ is used to refer the structure in which qubits are encoded, connected, controlled and measured and so on. These works also define the the control over qubits, which requirement should be satisfied by the quantum control microarchitecture. The quantum control microarchitecture we propose in this thesis is mostly based on the quantum chip as proposed by [81].

2.5.4 Efficiently Using Qubits

On the one hand, researchers in the physics community mostly focus on how to fabricate quantum chips with more qubits with well-defined control by adopting a bottom-up method. On the other hand, researchers in the computer architecture community pay more attention to efficiently using qubits to flexibly perform universal, fault-tolerance quantum computing by adopting a top-down method [38, 45–48, 104–106]. Research on this topic is highlighted by the following observations:

- Thousands or millions of qubits are assumed to be available;
- The mapping of quantum computation or error correction tasks to different qubits or qubit regions is highly investigated to improve performance as well as reduce hardware resource consumption;
- The mapping process is largely inspired by the von-Neumann architecture, where the computation region is usually separated from the memory region;
- Since ancilla factory used for quantum error correction can consume more than half the qubits, with the percentage even reaching 93% [48] in a particular situation, it fundamentally affects the way to organize qubits and the computation process;
- Customizing the architecture for target quantum algorithms, e.g., Shor’s factoring algorithm, can reduce the hardware resource requirement substantially.

Since millions of qubits are far beyond the reach of the NISQ technology, there has started research on efficiently mapping the qubits as used in small-scale or intermediate-scale quantum programs to physical qubits with limited interconnection on a quantum chip [107–109].

In these works, the terms ‘architecture’ and ‘microarchitecture’ are used to refer the way how qubits are organize into regions for various purposes to improve the efficiency or fidelity of the quantum algorithm. This is different to this thesis where ‘architecture’ mainly refers the quantum instruction set architecture and ‘microarchitecture’ refers the quantum control microarchitecture which applies physical control – e.g., microwaves – over qubits.

2.5.5 Quantum Control Microarchitecture

Limited previous work talked about the microarchitecture used to perform control over qubits. Oskin *et al.* [38] proposed an overall architecture for fault-tolerant quantum computer implementation with concatenation code, which mainly consists of three parts: quantum arithmetic logic unit (QALU) to perform operations, quantum memory to store information, and a dynamic scheduler to orchestrate the behavior of the previous two parts. This paper firstly called for the interleaved execution of classical instructions and quantum instructions in the dynamical scheduler, which is simply assumed to be a classical microprocessor. Though maybe sufficient for qubits based on trapped ions, a classical, general-purpose microprocessor is insufficient when precise timing control at nanosecond scale is required, e.g., by superconducting qubits (see Section 3.3.2).

Addressing the huge instruction bandwidth required to apply physical operations for quantum error correction, Tannu *et al.* [110] proposed an architecture with a dedicated programmable microcode engine to continuously generate instructions for quantum error correction, which can reduce the instruction bandwidth between the instruction memory and the microarchitecture controlling qubits.

These works either focus on defining the general relationship between modules of the control microarchitecture or the design of one particular module to improve the performance of the microarchitecture. However, the details of some modules are usually incomplete and these works do not define a fully functioning control microarchitecture for NISQ devices.

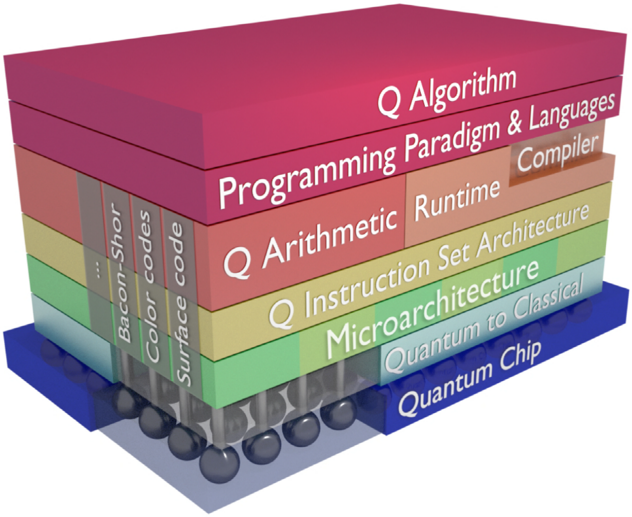


Figure 2.15: Overview of quantum computer system stack

2.6 Quantum System Stack

To construct a fully programmable quantum computer, a key question is: what layers are required to seamless connect quantum software and hardware which enable quantum algorithms to be executed on the quantum processor? Defining a layered system stack allows developing the functionality of the quantum computer as independently as possible from other layers and independent of the underlying quantum technology.

We propose a high-level view of the quantum system stack consisting of multiple layers as shown in Figure 2.15. A *quantum algorithm* is designed to solve particular problems with a performance that should surpass the best classical algorithm. To take advantage of both classical and quantum computing, a quantum algorithm can be described as a quantum-classical hybrid program which contains a host program and multiple quantum kernels. During execution, the host program invokes the quantum kernel(s) to accelerate a particular part of the computation, and performs classical computation when no quantum speedup exists.

A *quantum programming paradigm* is required to support flexible and efficient description of quantum applications, which may contain one or multiple high-level programming languages. The host program can written in a classical programming language such as C++, and the quantum part is written in a high-

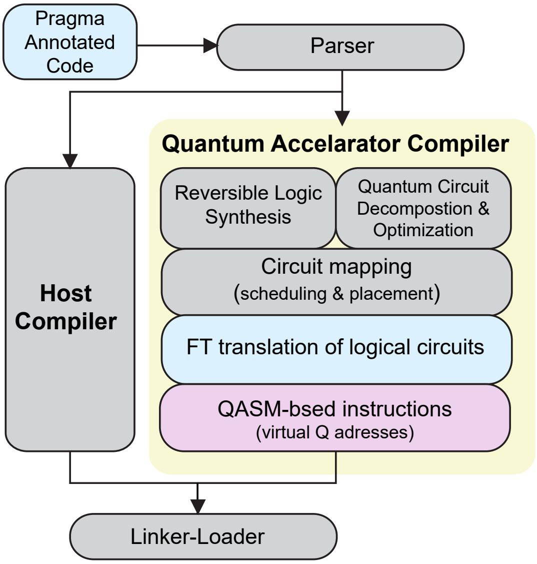

Pragma Annotated Code
Parser
Quantum Accelarator Compiler
Host Compiler
Reversible Logic Synthesis
Quantum Circuit Decompostion & Optimization
Circuit mapping (scheduling & placement)
FT translation of logical circuits
QASM-bsed instructions (virtual Q adresses)
Linker-Loader

Figure 2.16: Compiler infrastructure

level quantum programming language such as Scaffold [90] or Q# [70]. If QEC is required, qubits (quantum operations) at this level are defined as logical qubits (logical quantum operations).

As illustrated in Figure 2.16, the *compiler* infrastructure consists of a conventional host compiler such as the GNU Compilation Collection (GCC) and a quantum compiler such as ScaffCC [53]. The former compiles the classical part into classical code that can be executed by a classical processor, like the Intel Xeon processor. The latter works on the quantum part and generates quantum code that can be executed by the quantum processor.

During compilation, the quantum compiler performs reversible logic synthesis [111, 112], quantum gate decomposition and optimization [40, 113–115], and circuit mapping and scheduling [107, 108, 116]. When QEC is used, logical qubits and operations as used at the programming language level are also translated in a fault-tolerant way into physical qubits and operations which perform the same quantum computing tasks. As represented by the third dimension of the stack in Figure 2.15, the fault-tolerant implementation is guided by the choice of QECC.

All instructions in the quantum code as generated by the quantum compiler should belong to a *Quantum Instruction Set Architecture (QISA)*, which is the interface between quantum software and hardware. QISA should abstract away as much hardware details as possible but expose the necessary hardware features to the algorithm designer and programmer to enable platform-specific optimization. As stated above, when QEC is used, QISA should be able to efficiently describe physical operations on physical qubits required by the fault-tolerant implementation. Some architectural support might be required to achieve high efficiency for QEC. Since classical instructions can be used to guide the program flow and enable a hierarchical description of the program which can reduce the program code size, the quantum code should allow interleaving quantum instructions and classical instructions as suggested by [53, 70]. The classical instructions executed by the quantum processor are called *auxiliary classical instructions*.

The quantum control *microarchitecture* executes instructions belonging to the QISA. Apart from decoding the quantum instructions into required control signals with precise timing, the control microarchitecture is also responsible for real-time quantum error detection and correction when QEC is applied. It processes error syndromes generated by the stabilizer measurement over the data qubits to identify possible errors, based on which the required corrections are made by updating the Pauli frame or by sending the appropriate corrective operations when required.

Finally, based on the specific *quantum technology* – e.g., superconducting qubits – control signals are translated into required pulses, and sent to the quantum chip via the quantum-classical interface. The *quantum-classical interface* is responsible for all the conversions between the analog qubit plane and the digital layers in the system stack. The quantum-classical interface together with the quantum chip are technology dependent.

Adopting a bottom-up method, we analyzed the requirement of controlling qubits using flexible high-level programming. Based on the requirement, a quantum control microarchitecture is proposed. This work is introduced in the next chapter.

2.7 Appendix

This appendix shows the deduction procedure in Eq. 2.24, which converts the summation format of the state $|\psi_2\rangle$ into the product format.

After applying the oracle $\mathcal{O}$, We have

$$|\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{n-1} |x\rangle \left(|f(x)\rangle - |f(x) \oplus 1\rangle\right)$$

Since

$$|f(x)\rangle - |f(x)\oplus 1\rangle = \left\{ \begin{array}{ll} |0\rangle - |1\rangle\,, & \text{if } f(x)=0 \\ |1\rangle - |0\rangle\,, & \text{if } f(x)=1 \end{array} \right\} = (-1)^{f(x)}\left(|0\rangle - |1\rangle\right),$$

let $M(j,b) = x \,\&\, (b\cdot 2^j)$, where x has the binary format $\overline{x_{n-1}\cdots x_1x_0}$ and the operator '&' means bit-wise AND. Then we can rewrite $|\psi_2\rangle$ as

$$\begin{aligned}
|\psi_2\rangle =& \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{n-1}(-1)^{f(x)}|x\rangle\left(|0\rangle - |1\rangle\right) \\
=& \frac{1}{\sqrt{2^{n+1}}}\sum_{x_i\in\{0,1\}}|x_{n-1}\cdots x_1x_0\rangle(-1)^{f(\overline{x_{n-1}\cdots x_1x_0})}\left(|0\rangle - |1\rangle\right) \\
=& \frac{1}{\sqrt{2^{n+1}}}\sum_{x_i\in\{0,1\},i\neq j}\Big((-1)^{f(M(j,0))}|x_{n-1}\cdots x_{j+1}\rangle|0\rangle|x_{j-1}\cdots x_0\rangle + \\
& (-1)^{f(M(j,1))}|x_{n-1}\cdots x_{j+1}\rangle|1\rangle|x_{j-1}\cdots x_0\rangle\Big)\left(|0\rangle - |1\rangle\right) \\
=& \frac{1}{\sqrt{2^{n+1}}}\sum_{x_i\in\{0,1\},i\neq j}|x_{n-1}\cdots x_{j+1}\rangle\Big((-1)^{f(M(j,0))}|0\rangle + \\
& (-1)^{f(M(j,1))}|1\rangle\Big)|x_{j-1}\cdots x_0\rangle\left(|0\rangle - |1\rangle\right)
\end{aligned}$$

According to Eq. 2.21, we have the following relationship:

$$f(M(j,1)) = \left\{ \begin{array}{lll} f(M(j,0)) & \text{when} & s_j = 0, \\ 1 - f(M(j,1)) & \text{when} & s_j = 1. \end{array} \right.$$

Hence, ignore the global phase, we can further rewrite $|\psi_2\rangle$ as:

$$\begin{aligned}
|\psi_2\rangle =& \frac{1}{\sqrt{2^{n+1}}}\sum_{x_i\in\{0,1\},i\neq j}|x_{n-1}\cdots x_{j+1}\rangle\left(|0\rangle + (-1)^{s_j}|1\rangle\right)|x_{j-1}\cdots x_0\rangle\left(|0\rangle - |1\rangle\right) \\
=& \frac{|0\rangle + (-1)^{s_{n-1}}|1\rangle}{\sqrt{2}} \otimes \cdots \otimes \frac{|0\rangle + (-1)^{s_1}|1\rangle}{\sqrt{2}} \otimes \\
& \frac{|0\rangle + (-1)^{s_0}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}
\end{aligned}$$

Note. Section 2.6 is based on the following paper:

X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, **A Heterogeneous Quantum Computer Architecture**, *Proceedings of the ACM International Conference on Computing Frontiers (CF'16)*, ACM, 2016, pp. 323-330.

3

Experimental Quantum Microarchitecture

3.1 Introduction

To construct a fully programmable quantum computer based on the circuit model [40], a system stack [117] composed of several layers is required (Figure 2.15). Quantum algorithms are formulated and then described using a high-level quantum programming language [51, 52, 88, 90, 94]. Depending on the choice of quantum error correction code [79], such as surface code [39], the compiler [50, 51, 53] takes that description as input, performs optimization [51, 54, 55, 118, 119] and generates a fault-tolerant implementation of the original quantum algorithm. Next, it realizes the algorithm using instructions [49, 50, 53, 95, 97] belonging to a quantum instruction set architecture (QISA). Just like in classical architectures [120], the QISA is the interface between software and hardware. A control microarchitecture is needed to decode the quantum instructions into required control signals with precise timing as well as real-time quantum error detection and correction [121, 122]. Finally, based on the specific quantum technology – e.g., superconducting qubits [26–28], trapped ions [24, 25], spin qubits [29], nitrogen-vacancy centers [32, 33], etc. – control signals are translated into required pulses, and sent to the quantum chip via the quantum-classical interface.

In current experiments, quantum processors are controlled with well-defined electrical signals, e.g., microwave-frequency and baseband pulses, which require accurate parameters and timing. To satisfy the strict requirements on control signals, dedicated electronic devices are typically used to interface with the quantum processor. However, existing control methods introduce high resource consumption, long configuration times, and control complexity, all of which scale poorly with the number of qubits [60]. Although high-level languages offer flexibility, quantum compilers typically generate instructions that are not directly executable on a quantum processor. It is a challenge to design a control microarchitecture that accepts a set of instructions output by a compiler and translates them into the interface required by a quantum processor.

Motivated by heterogeneous computing, we propose a control microarchitecture, named QuMA, for a superconducting quantum processor based on the circuit model. QuMA accepts quantum-classical mixed code and enables flexible and precise-timing control over a quantum processor. The four concepts at the core of QuMA are:

- Codeword-based event control scheme: every event including pulse generation and measurement is assigned with an index, which is called a codeword. These events are triggered by corresponding codewords at runtime. This scheme abstracts the control of quantum processors using complex analog pulses into a simple interface consisting of only handy binary signals, providing the foundation for flexible control via instructions.
- Queue-based event timing control: in this scheme, events with precise timing decoded from instruction execution are first buffered in a group of queues and then triggered at expected timing. It allows that events are triggered at deterministic and precise timing while the instructions are executed with non-deterministic timing.
- Multilevel instruction decoding: quantum instructions are successively translated into microinstructions, micro-operations, and finally codewords with accurate timing. It enables using technology-independent instructions to control operations on qubits.
- Quantum microinstruction set: we design and implement a low-level quantum microinstruction set (QuMIS) which enables flexible control of quantum operations.

In addition, we implement QuMA on a field-programmable gate array (FPGA). We experimentally validate QuMA by conducting a standard gate-characterization experiment on a superconducting qubit, which is called *AllXY* [123, 124]. The control, initially specified in a high-level programming language, is converted to our proposed instructions by a quantum compiler.

The chapter is structured as follows. Section 3.2 presents previous work related to quantum control microarchitecture. After stating the challenges of controlling quantum processors using instructions in Section 3.3, Section 3.4 details how QuMA addresses these challenges in a systematic way with three proposed mechanisms. Section 3.5 discusses the advantages and scalability of QuMA. The implementation and experimental validation of QuMA and QuMIS are shown in Sections 3.6 and 3.7, respectively. Before the conclusion in Section 3.9, Section 3.8 discusses the potential impact of QuMA.

3.2 Related Work

Several quantum programming languages [51, 52, 88, 89, 94] and compilers [50, 51, 53] exist in which quantum algorithms can be written and compiled into a series of instructions. These quantum compilers [50, 90, 125] all generate a variant of quantum assembly language (QASM)-based instructions that belong to the quantum instruction set. Although several quantum instruction sets have been proposed, such as a von Neumann architecture-based virtual-instruction set architecture [49], quantum physical operations language (QPOL) [50], Hierarchical QASM with Loops (QASM-HL) [53], Quil [95], and OpenQASM [97], they are intermediate representations of quantum applications without considering the low-level constraints to interface with the quantum processor. They all lack an explicit control microarchitecture that implements the instructions set and allows the execution of such instructions on a real quantum processor.

Previous papers discussing quantum (micro-) architecture can be roughly divided into three groups. The first group discusses how to physically design and fabricate a quantum processor based on a specific technology, such as trapped ions [25, 47, 49, 99], superconducting qubits [42, 101], spin qubits [43], etc. The second group [38, 45–47, 54, 105] studies how to organize qubits into multiple regions for different computational purposes to reduce the required hardware resources and communication overhead, and to maximize parallelism. The third group takes a high-level view to discuss research domains [86] and quantum abstraction [84]. All of these works use the term microarchitecture differently from this thesis.

An example of control microarchitecture as viewed in this thesis is [117], where emphasis is placed on the definition of technology-independent and technology-dependent functions in which the microcode unit plays an essential role. The microcode approach was first introduced by Wilkes [126] to emulate a relatively complex machine instruction as a sequence of micro-operations, called a microprogram. The microprogram can be permanently stored or cached in a control store. It enables flexible complex instruction definition using the same hardware implementation. Vassiliadis *et al.* [127] extended the microcode method to a three-level translation from machine instructions to microinstructions and finally to micro-operations. A microinstruction decoded into one (multiple) micro-operation(s) is called vertical (horizontal).

The microcode method is a computational model that also maps quite well onto quantum computing because: (1) there are frequently-used routines in quantum computing, such as error correction, which impact system performance significantly but can be well optimized via carefully tuning the microcode for these routines, as proposed by [46]; (2) most quantum algorithms frequently use more complex operations which cannot, at least in the foreseeable future, be directly implemented by a quantum processor. In this thesis, we adopt the microcode approach in the proposed microarchitecture to enable flexible technology-independent instruction definition.

3.3 Microarchitectural Challenges

3.3.1 Motivational Example

We use the *AllXY* experiment [124] as an example to illustrate the microarchitectural challenges when controlling superconducting qubits. This experiment, although simple, requires flexible control over the qubit and is sensitive to control errors such as timing inaccuracy. Hence, it can reveal some of the essential features of a microarchitecture to control a superconducting quantum processor.

The *AllXY* experiment is a simple test of the calibration of single-qubit gates, which are realized by microwave pulses. Different pulse errors (amplitude, frequency, etc.) produce distinct signatures that are easily recognized. The qubit (initialized in the $|0\rangle$ state) is subjected to two back-to-back single-qubit gates and measured (Figure 3.1). In each round, we run 21 different gate pairs: ideally, the first 5 return the qubit to $|0\rangle$, the next 12 drive it to $\frac{1}{\sqrt{2}}\left(|0\rangle + e^{in\pi/2}|1\rangle\right)$ with $n \in \{0, 1, 2, 3\}$, and the final 4 drive it to $|1\rangle$. By averaging the measurements results for each pair over N rounds (we take $N = 25600$ in experiment), we can extract the fidelity of the qubit to the $|1\rangle$ state, and compare to the ideal staircase signature. Algorithm 1 shows the required procedure to perform the *AllXY* experiment.

Algorithm 1: Pseudo code of the *AllXY* experiment.

```
Data: gate[21][2] = $\{\{I, I\}, \{R_x(\pi), R_x(\pi)\},$
    $\{R_y(\pi), R_y(\pi)\}, \{R_x(\pi), R_y(\pi)\}, \{R_y(\pi), R_x(\pi)\},$
    $\{R_x(\pi/2), I\}, \{R_y(\pi/2), I\}, \{R_x(\pi/2), R_y(\pi/2)\},$
    $\{R_x(\pi/2), R_y(\pi/2)\}, \{R_x(\pi/2), R_y(\pi)\},$
    $\{R_y(\pi/2), R_x(\pi)\}, \{ R_x(\pi), R_y(\pi/2)\},$
    $\{R_y(\pi), R_x(\pi/2)\}, \{R_x(\pi/2), R_x(\pi)\},$
    $\{R_x(\pi), R_x(\pi/2)\}, \{R_y(\pi/2), R_y(\pi)\},$
    $\{R_y(\pi), R_y(\pi/2)\},\{R_x(\pi), I\}, \{R_y(\pi), I\},$
    $\{R_x(\pi/2), R_x(\pi/2)\}, \{R_y(\pi/2), R_y(\pi/2)\}\};$
for ($j = 0$; $j < N$; $j++$) do
    for ($i = 0$; $i < 21$; $i++$) do
        Init the qubit; // by waiting multiple $T_1$ ($t_{Init}$).
        Apply gate[i][0] on the qubit;
        Apply gate[i][1] on the qubit;
        $S_{j,i}$ = measure(qubit);
    end
end
$F_{|1\rangle}|_{\mathrm{meas},i} \leftarrow \sum_{j=0}^{N-1} S_{j,i}/N$;
```

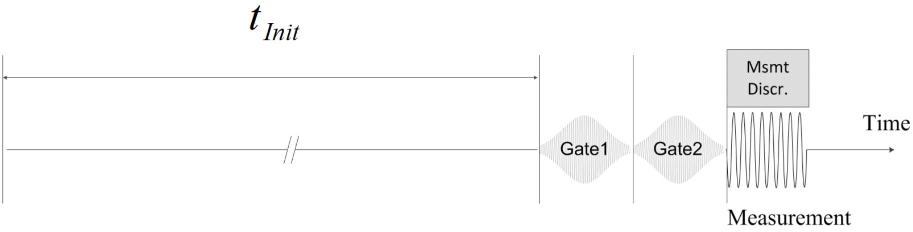


Figure 3.1: Waveforms and timings for one round of the $AllXY$ experiment.

3.3.2 Complex Analog Waveform Control

In classical computers, data and control signals are both binaries. In contrast, the input and output signals of quantum processors are both complex analog signals. The measurement outcome of qubits resides in the output analog signals from the quantum processor, while quantum operations on qubits (input signals) are performed by sending analog pulses that have well-defined but variable envelope, frequency, duration, timing, etc. For example, the X gate on a transmon qubit can be implemented using a 20 ns Gaussian pulse modulated to the frequency of the qubit with a particular phase.

A popular method to produce the required pulses uses arbitrary waveform generators. Before executing quantum algorithms, the pulses are calibrated and placed in the memory of these generators as arrays of amplitude values for each sample. A pulse lasting for a time T_d requires the memory to store $N_s = 2 \cdot T_d \cdot R_s$ samples for both in-phase (I) and quadrature (Q) components, where R_s is the sampling rate, typically ~ 1 GSample/s. Each sample can consist of ~ 12 bits, representing the vertical resolution of the amplitude.

Measurement Result Discrimination

As described in Section 2.3.1, measurement results are contained in an analog signal $V_a(t)$. To discriminate the result for a qubit q, dedicated data-acquisition boards are commonly used to digitize $V_a(t)$ and perform integration and discrimination in software as follows:

$$S_q = \int V_a(t) W_q(t) dt, \text{ and } M_q = \begin{cases} 1 & \text{if } S_q > T_q; \\ 0 & \text{otherwise.} \end{cases}$$

Here, $W_q(t)$ and T_q are a calibrated weightfunction and threshold for q, respectively. S_q is the integration result and M_q the final binary measurement result. The software-based method is disadvantageous because of two reasons. First, the long latency of the software-based method (hundreds of microseconds) makes real-time feedback control for superconducting qubits impossible, since latency well below the typical qubit coherence time ($< 100\ \mu$s) is required. The feedback control determines the next operations based on the result of measurements and is critical in many quantum algorithms, e.g., a specific implementation [128] of Shor's factoring algorithm [129]. Second, the implied hardware resource consumption cannot scale up to a large number of qubits. A scalable measurement discrimination method with short latency constitutes a challenge.

Flexible Combination of Operations

Quantum algorithms and even basic quantum experiments, such as *AllXY*, require combining multiple quantum operations. To generate the required operation combinations, current arbitrary waveform generators first upload long waveforms combining different pulses with appropriate timing and later play them. A drawback of this method is that even a small change to the operations requires a new upload of the entire waveform which costs significant memory and upload time. To generate the 21 combinations in the *AllXY* experiment, 21 different waveforms must be uploaded. With more qubits and more complex algorithms, the combination of operations can be more, which asks for more waveforms, leading to more memory consumption and larger uploading latency. Therefore, this method does not easily scale to a large number of qubits.

Furthermore, the execution of quantum programs requires more flexible feedback control, which cannot be supported by the autonomous arbitrary waveform generators as these devices cannot change a waveform to incorporate dynamically determined operations. Therefore, it is a requirement to define a flexible and scalable way to combine multiple smaller pulses, such that any sequence can be easily programmed, changed and executed when necessary.

Accurate Timing Control

Instructions in classical processors are usually executed with non-deterministic timing on a nanosecond timescale due to (1) process switching and system calls in the software layer, (2) indefinite communication latency including memory access, (3) static and dynamical instruction reorder, (4) pipeline stall and flushing, etc. However, the non-deterministic timing typically does not matter and the program can run correctly as long as the relative order of inter-dependent instructions is preserved.

In contrast, precise timing on nanosecond timescales is critical to quantum operations. As discussed in Section 2.3.1, when a fixed single-sideband modulation is used, the timing of pulses must be accurate to maintain the carrier phase, which sets the rotation axis of single-qubit gates. For example, given a fixed $50\ \mathrm{MHz}$ single-sideband modulation in the *AllXY* experiment, applying the modulation envelope of an x rotation $5\ \mathrm{ns}$ later will produce a y rotation instead. Besides, some quantum experiments require operations to be applied at a particular point in time. For example, the pulses implementing the two single-qubit gates and the measurement must be applied on the qubit back-to-back. To provide the appropriate timing precision, dedicated hardware is needed where again scalability in terms of the number of qubits is an additional requirement.

Using instructions to specify the timing of operations is more promising. However, it is challenging to use non-deterministic instruction execution to generate pulses with deterministic and precise timing.

3.3.3 Instruction Definition

The instruction set architecture is the interface between hardware and software and is essential in a fully programmable classical computer. So is QISA in a programmable quantum computer.

As explained in Section 3.2, existing instruction set architecture definitions for quantum computing mostly focus on the usage of the description and optimization of quantum applications without considering the low-level constraints of the interface to the quantum processor. It is challenging to design an instruction set that suffices to represent the semantics of quantum applications and to incorporate the quantum execution requirements, e.g., timing constraints.

It is a prevailing idea that quantum compilers generate technology-dependent instructions [50, 90, 125]. However, not all technology-dependent information can be determined at compile time because some information can only be generated at runtime due to hardware limitations. An example is the presence of defects on a quantum processor affecting the layout of qubits used in the algorithm. In addition, the following observations hold: (1) quantum technology is rapidly evolving, and more optimized ways of implementing the quantum gates are continuously explored and proposed; a way to easily introduce those changes, without impacting the rest of the architecture, is important. (2) depending on the qubit technology, the kind, number and sequence of the pulses can vary. Hence, it forms another challenge to microarchitecturally support a set of quantum instructions which is as independent as possible of a particular technology and its current state of the art.

3.4 Quantum Microarchitecture

In this section, we describe the Quantum MicroArchitecture (QuMA) as shown in Figure 3.2. QuMA is a heterogeneous architecture which includes a classical CPU as a host and a quantum coprocessor as an accelerator.

As proposed in [117], the input of QuMA is a binary file generated by a compiler infrastructure where classical code and quantum code are combined. The classical code is produced by a conventional compiler such as GCC and executed by the classical host CPU. Quantum code is generated by a quantum compiler and executed by the quantum coprocessor.

As shown in Figure 3.2, the host CPU fetches quantum code from the memory and forwards it to the quantum coprocessor. In the quantum coprocessor, executed instructions in general flow through modules from left to right. The execution controller performs register update, program flow control and streams quantum instructions to the physical execution layer. The physical microcode unit translates quantum instructions into microinstructions using the Q control store. These are further decomposed into micro-operations by the quantum microinstruction buffer (QMB). The timing of each micro-operation is also determined by the physical microcode unit. Based on the output of quantum microinstruction buffer, the timing control unit triggers micro-operations at a deterministic timing. The analog-digital interface converts digitally represented micro-operations into corresponding analog pulses with precise timing that perform quantum operations on qubits, as well as analog signals containing measurement information of qubits into binary signals. Required modulation and demodulation with radio-frequency carrier waves are also carried out in the quantum-classical interface.

In order to address the challenges described in the previous section, three schemes are introduced in QuMA. (i) The codeword-based event control scheme is implemented by the codeword-triggered pulse generation unit (CTPG), which produces analog input to the quantum processor based on the received codeword triggers, and the measurement discrimination unit (MDU) converting the analog output from the quantum processor into binary results. (ii) The queue-based event timing control scheme is implemented by the timing control unit, which issues event triggers with precise timing to the measurement discrimination unit and the micro-operation unit (u-op unit). (iii) A multilevel instruction decoding scheme, which successively decodes a quantum instruction into microinstructions at the Q Control Store, micro-operations at the quantum microinstruction buffer, and finally codeword triggers at the micro-operation unit. The complex analog waveform control challenge is addressed by (i) and (ii) whereas the instruction definition is addressed by (iii).

3.4.1 Codeword-Based Event Control

The analog-digital interface (Figure 3.2) is at the boundary of analog signals and digital signals in QuMA, which is technology-dependent. As shown in Figure 3.2, from left to right , the micro-operation unit and the codeword-triggered pulse generation unit translate codeword triggers into pulses representing quantum operations on the qubits with a fixed latency. From right to left, analog measurement waveforms from the quantum processor are discriminated into binary results by the measurement discrimination unit. In this way, the analog-digital interface abstracts the complex analog waveform generation

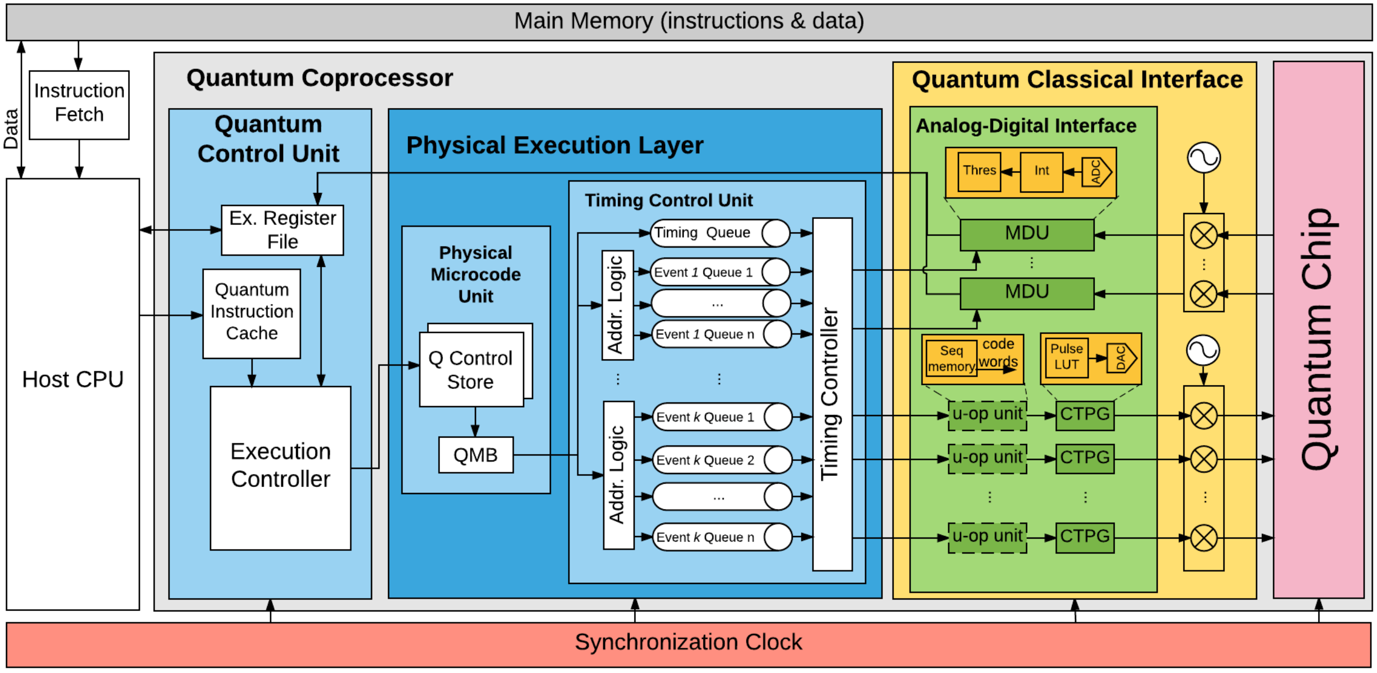


Figure 3.2: Overview of the Quantum MicroArchitecture (QuMA).

and puts forward the responsibility of codeword control with precise timing to the upper digital layers. Therefore, it enables controlling analog pulse generation using instructions. Fast and flexible feedback control is also possible in principle because the codeword-triggered pulse generation scheme does not require the waveform to be uploaded at runtime and codeword triggers with precise timing can be efficiently generated dynamically.

Codeword-Triggered Pulse Generation

From experiments, we observe that the pulses for a fixed and small set of quantum operations can be well defined and used after calibration. They are also called primitive operations because they are sufficient for many quantum computing experiments. Based on this, we introduce the codeword-triggered pulse generation scheme in QuMA to generate pulses corresponding to primitive operations. In codeword-triggered pulse generation, well-defined primitive pulses instead of entire waveforms are uploaded to the memory. The memory is organized as a lookup table and each entry in the lookup table, indexed by means of a codeword, contains the sample amplitudes corresponding to a single pulse. The codeword-triggered pulse generation unit converts a digitally stored pulse into an analog one only when it receives a codeword trigger. An example of the lookup table content for single-qubit operations is shown in Table 3.1.

Table 3.1: An example of the lookup table content of a codeword-triggered pulse generation unit for single-qubit gates.

Codeword	0	1	2	3
Pulse	I	$R_x(\pi)$	$R_x(\frac{\pi}{2})$	$R_x(-\frac{\pi}{2})$
Codeword	4	5	6	$\cdots$
Pulse	$R_y(\pi)$	$R_y(\frac{\pi}{2})$	$R_y(-\frac{\pi}{2})$	$\cdots$

The codeword-triggered pulse generation scheme has a modest memory requirement since it only needs to store a small number of pulses for the well-defined primitive operations. In the *AllXY* experiment, only the pulses for 7 operations need to be stored, which only consumes the memory for $7 \times 2 \times 20\ \mathrm{ns} \times R_s$ samples (in total 420 Bytes), instead of 21 waveforms each containing two operations, that are $21 \times 2 \times 2 \times 20\ \mathrm{ns} \times R_s$ samples (in total 2520 Bytes). When more complex combination of operations is required, the memory consumption will remain the same and the memory saving will be more significant. The small memory footprint provides a scalable path for controlling a larger number of qubits.

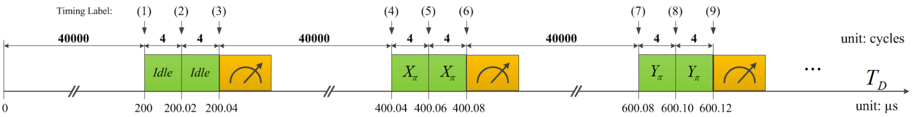


Figure 3.3: Operations of the *AllXY* experiment in the timeline. Measurement pulse generation and measurement result discrimination overlap in time and are shown using the same meter box.

The delay between the codeword trigger and the pulse generation is required to be fixed and short in the codeword-triggered pulse generation unit. The fixed delay ensures that the flexible combination of the pulses with precise timing can be achieved by flexibly generating the corresponding codeword triggers at precise timing. In the *AllXY* experiment, by issuing the codeword triggers for the two gates with an interval of 20 ns, the pulses for the two gates can be played out exactly back to back.

Measurement Discrimination

Recent experiments have demonstrated measurement discrimination using a customized FPGA [63], achieving a short latency $< 1\ \mu$s which enables real-time feedback control. This method also costs modest hardware exhibiting better scalability. Adopting this idea, we introduce hardware-based measurement discrimination units in the analog-digital interface. The measurement discrimination unit translates the analog signal containing measurement information of a single qubit into a binary measurement result. Once the measurement discrimination unit for qubit q receives a codeword trigger, it starts the measurement discrimination process and generates a binary result R_q. R_q can be subsequently forwarded to the quantum control unit for feedback control or reading back.

Recent experiments have also demonstrated combining the measurement result of multiple qubits into one analog signal [27, 130]. This can reduce the number of required measurement discrimination units and exhibits better scalability.

3.4.2 Queue-Based Event Timing Control

The timing control unit divides the microarchitecture into two timing domains: the non-deterministic timing domain and the deterministic timing domain, which are on the left and right side of the timing control unit in Figure 3.2, respectively. In the non-deterministic timing domain, the quantum control unit and physical execution layer execute instructions and feed quantum operations to the queues in an as-fast-as-possible fashion. In the deterministic timing domain, quantum operations in the queue are emitted to the analog-digital interface with deterministic and precise timing. To this end, queue-based event timing control is introduced.

To illustrate the working principle of queue-based event timing control, the operations of the *AllXY* experiment with corresponding timing are shown in Figure 3.3. The horizontal axis labels mark the time points in microseconds when a corresponding operation takes place. Each time point is assigned a timing label, which is the number in brackets on the top. The bold numbers above the double-arrow lines indicate intervals between two time points in cycles. Here and throughout the rest of this chapter, a cycle time of $5\ \mathrm{ns}$ is used.

The timing control unit implements queue-based event timing control in QuMA. It consists of a timing queue, multiple event queues, and a timing controller. The timing queue buffers the time points with corresponding timing labels. The location of the time points can be designated in the timeline, e.g., by specifying the intervals between consecutive time points as shown in Figure 3.3 and the first column of Table 3.2. Each event queue buffers a sequence of events with a time point at which the event is expected to take place. The time point is indicated by the aforementioned timing label. An event can be a quantum gate, measurement, or any other operation. The timing controller maintains the clock of the deterministic timing domain (T_D), which can be started by an instruction or another source, e.g., an external trigger. When T_D reaches the assigned time point, the timing controller signals the queues to fire the events matching that time point and emits them to the analog-digital interface.

In order to better illustrate how queue-based event timing control works, we use the *AllXY* experiment. Three event queues are used in this experiment (see Table [2-4]): the Pulse Queue for single-qubit operations, the MPG Queue for measurement pulse generation, and the MD Queue for measurement discrimination. Besides the timing label for each event, the pulse queue contains the single-qubit operations, e.g., the I or X_π operation, to be triggered, and the MD queue contains the destination register, e.g., $r7$, to write back the measurement result. After executing a couple of instructions in the program and before T_D is started, the state of the queues is as shown in Table 3.2. The bottom of the table corresponds to the front of the queues. After T_D is started, a counter in the timing controller starts counting. When the counter reaches the

Table 3.2: Queue state of the *AllXY* experiment when $T_D = 0$.

Timing Queue	Pulse Queue	MPG Queue	MD Queue
⋮	⋮	⋮	⋮
(4, 6)			
(4, 5)			
(40000, 4)	$(X_\pi, 5)$		
(4 , 3)	$(X_\pi, 4)$		
(4, 2)	$(I, 2)$	(6)	$(r7, 6)$
(40000, 1)	$(I, 1)$	(3)	$(r7, 3)$

Table 3.3: Queue state of the *AllXY* experiment when $T_D = 40000$.

Timing Queue	Pulse Queue	MPG Queue	MD Queue
⋮	⋮	⋮	⋮
(4, 6)			
(4, 5)			
(40000, 4)	$(X_\pi, 5)$		
(4, 3)	$(X_\pi, 4)$	(6)	$(r7, 6)$
(4, 2)	$(I, 2)$	(3)	$(r7, 3)$

Table 3.4: Queue state of the *AllXY* experiment when $T_D = 40008$.

Timing Queue	Pulse Queue	MPG Queue	MD Queue
⋮	⋮	⋮	⋮
(4, 6)			
(4, 5)	$(X_\pi, 5)$		
(40000, 4)	$(X_\pi, 4)$	(6)	$(r7, 6)$

first interval value in the timing queue, i.e., 40000, the corresponding timing label, i.e., 1, is broadcast to all event queues. At the same time, the counter resets and restarts. Since the pulse queue contains that same label, 1, at the front of the queue, the operation I is fired to the analog-digital interface. The queue state then turns into Table 3.3. The second I operation is issued in the same way when the counter reaches the next interval value, 4. After the counter reaches the third interval value, 4, the timing label 3 is broadcast and the MG Queue triggers the measurement pulse generation and the MD queue triggers a measurement discrimination process of which both associated timing labels are 3. The queue state then turns into Table 3.4. The rest can be done in the same manner.

3.4.3 Multilevel Instruction Decoding

Combining the codeword-based event control scheme and queue-based event timing control enables other stages in QuMA to focus on flexibly decoding the quantum instructions and filling the queues as fast as possible without worrying about complex analog waveform control with rigid timing constraints. In this subsection, we first give an overview of the instruction definition and then discuss the multilevel decoding scheme for the quantum instructions.

Instruction Definition

The quantum code is written with instructions in the Quantum Instruction Set (QIS). An example of QIS instructions is shown in Table 3.6. QIS contains auxiliary classical instructions and quantum instructions. Auxiliary classical instructions are used for basic arithmetic and logic operations and program flow control. Quantum instructions describe which and when quantum operations will be applied on qubits. By including auxiliary classical instructions, QIS can support feedback control based on measurement results and a hierarchical description of quantum algorithms which can significantly reduce the program code size [55].

Instruction Decoding

To support a technology-independent quantum instruction set definition, we adopt a multilevel instruction decoding approach in which quantum instructions, especially that for quantum gates, are successively decoded into quantum microinstructions, micro-operations and finally codeword triggers to control codeword-triggered pulse generation to generate pulses. For example, Table 3.6 shows four decoding steps for the instructions of the *AllXY* experiment. From the QIS on, time is calculated in cycles. Due to the simplicity of the *AllXY* experiment and for the sake of code efficiency, the inner loop as shown in Algorithm 1 is unrolled. The execution of quantum instructions starts from the execution controller.

Execution Controller This unit executes the auxiliary classical instructions in the QIS and streams quantum instructions to the physical microcode unit. By executing the auxiliary classical instructions in the execution controller, the same quantum instruction can be issued to the physical microcode unit multiple times and each time with expected parameters computed at runtime.

Table 3.5: The format of QIS instructions, quantum microinstructions. Taking the *AllXY* experiment as an example (Part I).

QIS	QuMIS
# Input to the execution controller	# Input to the QMB
mov r1, 0	# round 0:
mov r2, 25600	Wait 40000
mov r3, ResultMemAddr	Pulse {q0}, I
mov r15, 40000	Wait 4
Outer_Loop:	Pulse {q0}, I
QNopReg r15	Wait 4
Apply I, q0	MPG {q0}, 300
Apply I, q0	MD {q0}, r7
Measure q0, r7	# round 1:
Load r9, r3[0]	Wait 40000
Add r9, r9, r7	Pulse {q0}, X180
Store r9, r3[0]	Wait 4
	Pulse {q0}, X180
QNopReg r15	Wait 4
Apply X180, q0	MPG {q0}, 300
Apply X180, q0	MD {q0}, r7
Measure q0, r7	. . .
Load r9, r3[1]	
Add r9, r9, r7	
Store r9, r3[1]	
...	
add r1, r1, 1	
bne r1, r2, Outer_Loop	

For example, the *QNopReg r15* instruction in the QIS is used to specify the initialization time. Each of the 21 *QNopReg r15* instructions will be issued once per round. Every time it is issued, it reads a waiting time from the register r15, which results in a *Wait 40000* instruction. If the register value is updated using auxiliary classical instructions, the waiting time specified in the *Wait* instruction can be calculated at runtime. In this way, it enables a compact and flexible description of quantum algorithms.

Physical Microcode Unit Quantum instructions are translated into a sequence of microinstructions in the physical microcode unit based on the microprograms uploaded into the Q control store. The timing for each quantum operation is also determined at this stage. For now and as shown in Table 3.7, the microinstruction set, QuMIS, consists of the following instructions: i) the

Table 3.6: The format of micro-operations and codeword triggers. Taking the *AllXY* experiment as an example (Part II).

Micro-operations	Codeword Triggers
# Input to the u-op units	# Input to the MDU or CPTG
	# Δ is the delay of the u-op unit
T_D = 40000:	$T_D = 40000 + \Delta$:
I sent to u-op unit0	CW 0 sent to CTPG0
T_D = 40004:	$T_D = 40004 + \Delta$:
I sent to u-op unit0	CW 0 sent to CTPG0
T_D = 40008:	T_D = 40008:
# MPG and MD bypass this stage	CW 7 sent to CTPG5 # Msmt
	MD(r7) sent to MDU0
T_D = 80008:	$T_D = 80008 + \Delta$:
X_π sent to u-op unit0	CW 1 sent to CTPG0
T_D = 80012:	$T_D = 80012 + \Delta$:
X_π sent to u-op unit0	CW 1 sent to CTPG0
T_D = 80016:	T_D = 80016:
# MPG and MD bypass this stage	CW 7 sent to CTPG5 # Msmt
...	*MD(r7)* sent to MDU0
	...

Wait instruction used to specify the interval between consecutive time points, ii) the **Pulse** instruction used to apply quantum gates on qubits; iii) the **MPG** instruction used to generate the measurement pulse; iv) the **MD** instruction used to trigger the measurement discrimination process.

In the quantum microinstruction buffer (QMB), quantum microinstructions for quantum gates are decomposed into separate micro-operations with timing labels and push them into the queues in the timing control unit as shown in Table 3.2. Due to the simplicity of measurements in terms of instruction control, quantum microinstructions for measurement pulse generation or measurement discrimination can be directly translated into codeword triggers to control the codeword-triggered pulse generation unit or the measurement discrimination unit bypassing the micro-operation unit. The timing control unit then emits the micro-operations at the expected timing. The **Pulse** and **MPG** instructions are both horizontal instructions, which can trigger the operation on multiple qubits at the same time.

Let us illustrate these concepts using the CNOT gate. A CNOT gate with a control qubit c and a target qubit t can be decomposed in the following way [40]:

$$\text{CNOT}_{c,t} = R_y(\pi/2)_t \cdot CZ \cdot R_y(-\pi/2)_t.$$

Table 3.7: QuMIS instructions.

Assembly Format	**Description**
Wait *Interval*	Wait for the number of cycles indicated by the immediate value *Interval*.
Pulse $(QAddr_0, uOp_0)[, (QAddr_1, uOp_1), \ldots]$	Apply the micro-operation uOp_i on each of the qubit(s) specified by the address $QAddr_i$.
MPG *QAddr*, *D*	Generate the measurement pulse for the qubits specified by the address *QAddr*. *D* indicates the duration of the measurement pulse in number of cycles.
MD *QAddr*, *$rd*	Discriminate the measurement results of the qubits specified by *QAddr* and store the result into register *$rd*.

Adopting the microcoded approach for the instruction *CNOT qc, qt* applying on superconducting qubits results in Listing 3.1.

```
Pulse   {qt},      Ym90
Wait     4
Pulse   {qt, qc}, CZ
Wait     8
Pulse   {qt},      Y90
Wait     4
```

Listing 3.1: Microprogram for the physical instruction *CNOT qc, qt*.

By utilizing horizontal microcode, one quantum instruction can be translated into multiple microinstructions and one microinstruction into multiple micro-operations. This allows flexible emulation of complex, technology-independent instructions using technology-dependent primitives.

Micro-Operation Unit At the micro-operation unit, each micro-operation is translated into a sequence of codeword triggers with predefined latency, which further makes associated codeword-triggered pulse generation units generate primitive operation pulses. For each predefined micro-operation uOp_i, the micro-operation unit stores a sequence Seq_i comprising of codewords and timing. Seq_i has the following format:

$$\mathrm{Seq}_i : \ ([0,\ cw_0];\ [\Delta t_1,\ cw_1];\ [\Delta t_2,\ cw_2];\ \ldots),$$

where Δt_j represents the interval between codeword triggers cw_{j-1} and cw_j. Once the micro-operation uOp_i is triggered, the micro-operation unit starts to output codeword cw_j after waiting for Δt_j cycles sequentially as defined in the sequence Seq_i. Since the timing controller fires the micro-operation at precise timing, the codeword triggers are also generated at precise timing.

For example, a Z gate can be decomposed into a Y gate followed by an X gate since $Z = X \cdot Y$ (up to an irrelevant global phase). The micro-operation unit can perform the translation for superconducting qubits using the following sequence given the lookup table content as listed in Table 3.1:

$$\mathrm{Seq}_Z: \;\; ([0,\ 1];\;\; [4,\ 4]).$$

The micro-operation unit allows the emulation of commonly-used quantum operations which are not directly implementable using primitive operations. Moreover, it reduces the communication between the timing control unit and the analog-digital interface. This is especially helpful when the timing control unit and the analog-digital interface are implemented in different electronic devices for performance and scalability.

3.5 Evaluation

To evaluate QuMA, we make a comparison between QuMA and the architecture of the Raytheon BBN APS2 system, which is a commercial device that has been recently demonstrated [130, 131] for superconducting qubits. Then we discuss the scalability limitation of QuMA.

The APS2 system has a distributed architecture consisting of nine individual APS2 modules and a trigger distribution module (TDM) that can fully control up to eight qubits. A quantum application is translated into multiple binary executables running in parallel on each of the APS2 modules. A binary is composed of separated program flow control instructions and output instructions. Instead of instructions with explicit quantum semantics, low-level output instructions are used, such as waveform with a physical memory address. Idle waveforms are used to implement precise timing between operations, and the TDM distributes trigger signals to perform parallelism/synchronization of multiple outputs via an interconnect network. The main disadvantage are that no output instructions can be processed when synchronization is required, and the interconnect network is cumbersome and fragile when scaling up to tens of qubits where multiple APS2 systems are required [130].

In contrast, QuMA employs a centralized architecture, in which: (i) only one binary executable is required for controlling multiple qubits, (ii) quantum semantics and timing of operations are explicitly defined at the instruction level, (iii) parallelism/synchronization of outputs is achieved by triggering events at specific timing points, which is neither dependent on another module nor limited by the interconnect network. These three points contribute to a relatively simple compilation model for QuMA. As explained in Section 3.4.2, QuMA decouples the timing of executing instructions and performing output. So it can maintain fully deterministic timing of the output and maximally process instructions during waiting. Since data is gathered in a single place (the register file), it is natural to extend QuMA to a heterogeneous computing platform by adding extra data exchange instructions to interact with the host CPU and the main memory.

Regarding scalability, QuMA is not limited by the analog-digital interface and the timing control unit, as their size scales linearly to the number of qubits and can be implemented in a distributed way. However, the limited time for executing instructions in quantum computers may form a challenge in QuMA when more qubits ask for a higher operation output rate while only a single instruction stream is used. A Very-Long-Instruction-Word (VLIW) architecture [120] can be adopted to provide much larger instruction issue rate. In addition, by optimizing the microcode unit and the micro-operation unit, it is possible to use less quantum instructions to describe more quantum operations, which can relax the instruction issue rate requirement.

3.6 Implementation

In this section, we discuss the quantum control box, where the aforementioned mechanisms have been implemented.

3.6.1 Quantum Control Box

The quantum control box, as shown schematically in Figure 3.4, consists of four FPGA boards. One board implements the Master Controller and the other three boards implement a two-channel arbitrary waveform generator (AWG) each.

The master controller is implemented using an Arrow BeMicro CV A9 board holding an Altera Cyclone V 5CEFA9 FPGA chip. It connects to two 8-bit resolution analog-to-digital converters (ADC) that digitize analog measurement signals from the quantum chip. The master controller has eight digital outputs

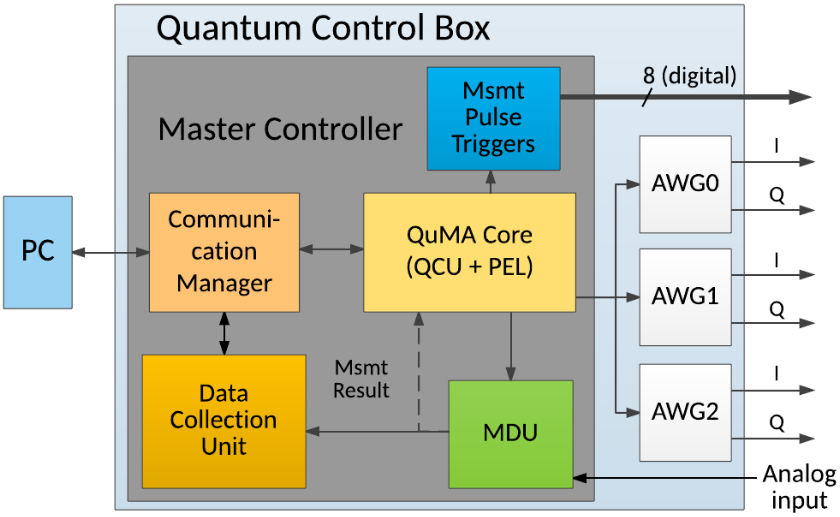


Figure 3.4: Schematic of the CBox firmware architecture. The QuMA core is implemented in the Master Controller. Dashed lines indicate functionality to be added in the future.

used for triggering measurement pulse generation and triggers the pulse generation of each AWG via a pair of Low-Voltage-Differential-Signaling wires.

Inside the MC, the QuMA core implements the quantum control unit and the physical execution layer of QuMA. The digital output unit converts the measurement operation tuple $(QAddr, D)$ received from the QuMA core into ‘1’ state with a duration of D cycles for the eight digital outputs masked by $QAddr$. The measurement discrimination unit (MDU) can discriminate the measurement result of a single qubit. The data collection unit can collect K consecutive integration results of a single qubit for N rounds, calculate and store the average of K integration results across the N rounds:

$$\bar{S}_i = \left(\sum_{j=0}^{N-1} S_{i,j} \right) /N \ , \quad i \in \{0, 1, \cdots, K-1\}.$$

After the data collection process is done, the PC can retrieve the averaging integration results $\{\bar{S}_i\}$.

Each AWG is implemented using a Terasic DE0-Nano board holding an Altera Cyclone IV EP4CE22F FPGA chip and uses two 14-bit resolution digital-to-analog converters (DAC) to generate the in-phase and quadrature components of qubit control pulses. Each AWG includes a micro-operation unit and a codeword-triggered pulse generation unit. The implemented codeword-triggered pulse generation unit has a fixed delay of 80 ns from the codeword trigger to the output pulse.

All FPGAs, ADCs, and DACs are clocked at 200 MHz, except for communication and data collection, which run at 50 MHz. The MC communicates with the PC via USB. The MC communicates to the AWGs, e.g., uploading the lookup table content of the codeword-triggered pulse generation unit.

3.6.2 QuMA Implementation

The QuMA implementation in the control box in shown in Figure 3.6. In view of the running physics experiments, it slightly differs from the microarchitecture presented in Section 3.4. We have partially implemented the system including the quantum instruction cache, the execution controller, part of the physical microcode unit, the timing control unit and the quantum classical interface. The rest is planned for future release. Due to the absence of a fully functioning physical microcode unit, the high-level quantum instructions of the QIS are not implemented yet. A combination of the auxiliary classical instructions in the QIS and QuMIS (see Table 3.7) is loaded into the quantum instruction cache.

We have designed a quantum programming language OpenQL based on C++ with a compiler that can translate the OpenQL description into the auxiliary classical instructions and QuMIS instructions.

The execution controller incorporates a classical pipeline to execute auxiliary classical instructions. The register file in this pipeline contains runtime information related to quantum program execution. QuMIS instructions are dispatched to the physical microcode unit after reading register values. The physical microcode unit can determine the timing of QuMIS instructions and decompose QuMIS instructions into micro-operations. A full implementation of the physical microcode unit is still under development. The timing control unit implements the queue-based event timing control scheme (as described in Section 3.4.2). The measurement pulse triggers pulse modulated microwave carrier generators in the other devices block to produce the measurement pulse for qubits.

3.7 Experimental Results

We have performed various quantum experiments on a qubit to validate and verify the design of QuMA and QuMIS, including T_1, T_2 Ramsey, T_2 Echo, *AllXY*, and randomized benchmarking [78] experiments. Considering the readability and page limitation, we only show the *AllXY* experiment in the thesis.

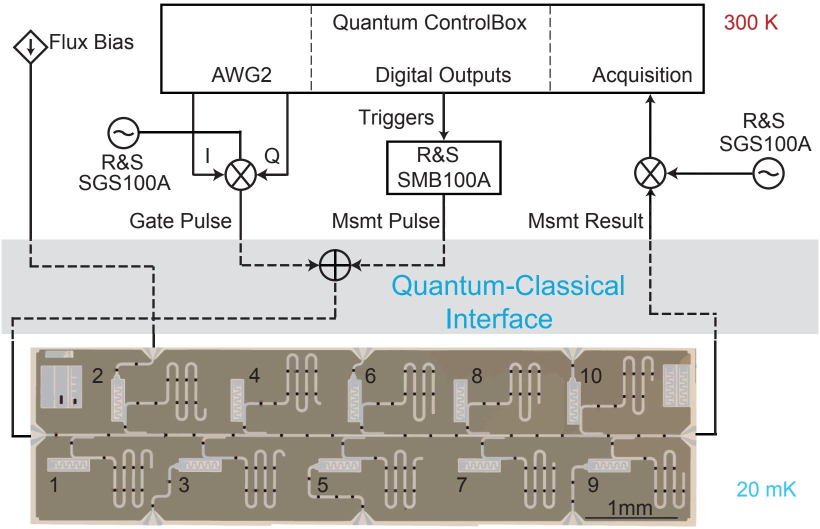


Figure 3.5: Experimental setup used for validation of the microarchitecture.

Figure 3.5 shows the experimental setup. All classical electronics are at room temperature. The quantum chip, operating at 20 mK, contains 10 transmon qubits with dedicated readout resonators all coupled to a common feedline. The measured qubit (labeled 2) has transition frequency $f_\mathrm{Q} = 6.466$ GHz, and the coupled resonator has fundamental $f_\mathrm{R} = 6.850$ GHz (for qubit in $|0\rangle$) (further detailed in [63]). To perform single-qubit gates, we use one microwave source [Rohde & Schwarz (R&S) SGS100A] to generate a 6.516 GHz carrier and control box AWG 2 to produce the in-phase and quadrature components (including -50 MHz single-sideband modulation) that define the pulse envelope. To generate the measurement pulse, we trigger a 6.849 GHz carrier (generated by a R&S SMB100A) using the control box digital output 1. The transmitted feedline signal is demodulated to an intermediate frequency of 40 MHz using a 6.809 GHz local oscillator (another R&S SGS100A). Prior to the experiment, the qubit pulses are calibrated and uploaded into con-

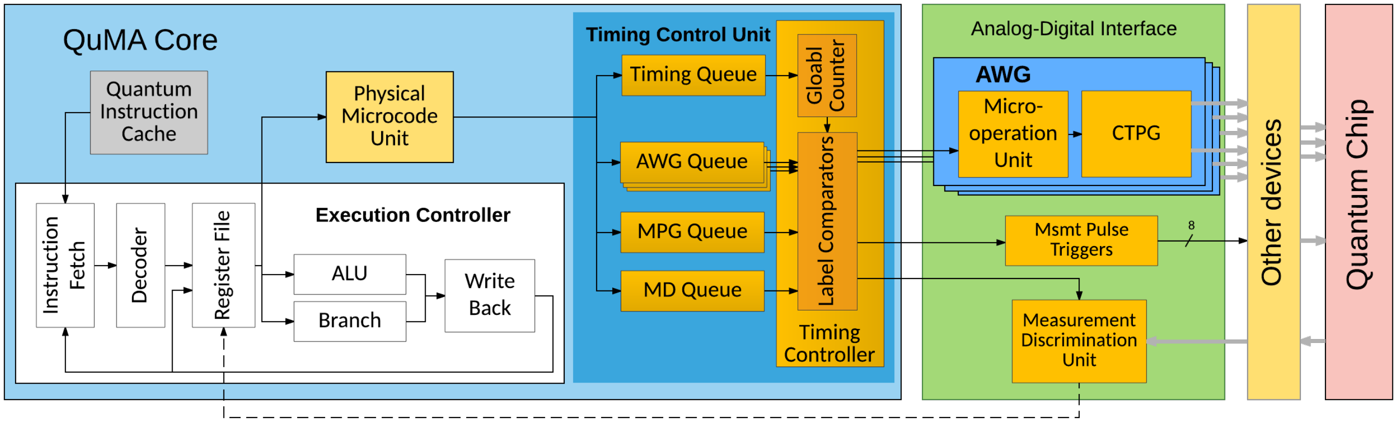


Figure 3.6: Schematic of the implemented QuMA. The thick gray lines are analog signals while the dark thin lines are digital signals. Dashed lines indicate functionality to be added in the future.

trol box AWG 2. Since the operations in the *AllXY* experiment are primitive, the micro-operation unit simply forwards the codewords to the wave memory without translation.

```
MOV  r15,  40000     # 200 us
MOV  r1,   0         # loop counter
MOV  r2,   25600     # number of averages

Outer_Loop:
  QNOPREG    r15        # Identity, Identity
  Pulse     {q2}, I
  Wait       4
  Pulse     {q2}, I
  Wait       4
  MPG       {q2}, 300
  MD        {q2}
(repeat the previous 7 instructions once again)

  QNOPREG    r15        # X180, X180
  Pulse     {q2}, X180
  Wait       4
  Pulse     {q2}, X180
  Wait       4
  MPG       {q2}, 300
  MD        {q2}
(repeat the previous 7 instructions once again)

  QNOPREG    r15        # Y180, Y180
  Pulse     {q2}, Y180
  Wait       4
  Pulse     {q2}, Y180
  Wait       4
  MPG       {q2}, 300
  MD        {q2}
(repeat the previous 7 instructions once again)

  ...

  ADDI    r1, r1, 1
  BNE     r1, r2, Outer_Loop
```

Listing 3.2: QuMIS Program to perform *AllXY* experiment.

The QuMIS program used to perform the *AllXY* experiment is generated from a OpenQL description and is shown in Listing 3.2. In this experiment, each of the 21 combinations is measured twice to make a direct visual distinction between systematic errors and low signal-to-noise ratio. Figure 3.7 shows the measurement results. The red staircase shows the ideal signature of perfect pulsing. The results of the 0-th (18-th and 19-th) combination are taken as the calibration point $\bar{S}_{|0\rangle,r}$ ($\bar{S}_{|1\rangle,r}$). Using the calibration points to rescale the signal, we obtain the fidelity $F_{|1\rangle}|_i$ corrected for readout error:

$$F_{|1\rangle}|_{\text{meas},i} = \left(\bar{S}_i - \bar{S}_{|0\rangle,r}\right) / \left(\bar{S}_{|1\rangle,r} - \bar{S}_{|0\rangle,r}\right).$$

We loop over these $K = 42$ pulse combinations over $N = 25600$ rounds. The data acquisition unit performs the required averaging of measurement results for each K.

This experiment uses the instructions generated from the high-level language OpenQL description to control the operations on the qubit. Only 7 pulses including the *Identity* operation are stored in the lookup table of the codeword-triggered pulse generation unit, regardless of the number of combinations of operations. It has a moderate memory consumption to store $140\ \mathrm{ns} \times R_s$ samples exhibiting a better scalability compared to the conventional method. From the experiment result, we can see that the measured fidelity for each combination matches well with the ideal readout fidelity. Since the *AllXY* experiment is sensitive to imperfection of the pulses and the timing, it demonstrates that the right pulses are generated and the precise timing of operations is well preserved.

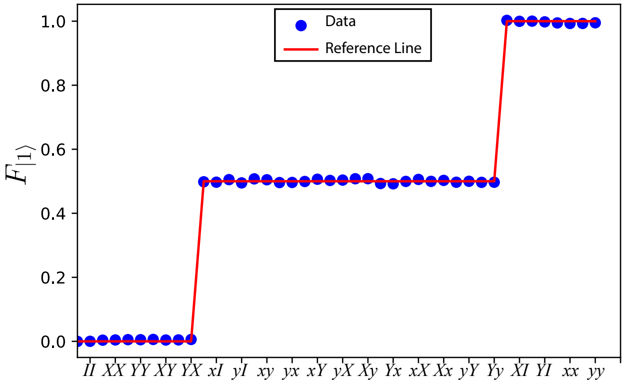


Figure 3.7: The *AllXY* result of qubit 2. In the label, each X/Y (x/y) denotes a rotation by π ($\pi/2$) around the x/y axis of the Bloch sphere.

3.8 Potential Impact

QuMA fills the gap between quantum compilers and quantum hardware by providing a control system that translates quantum code into low-level analog signals that operate on the qubits. In addition, QuMA makes a move towards the first definition of an executable QISA. As shown in the next chapter, we improved the microcode unit by enabling the translation from a single instruction to multiple operations on different qubits. An executable QISA, named eQASM, is also defined on top of QuMIS. With certain low-level information exposed in eQASM, such as timing, the quantum compiler can generate executable instructions for real devices.

Some quantum algorithms for near-term devices ask for quantum-classical mixed computation, such as a variational eigenvalue solver [15]. Because data can be gathered into the register file in QuMA, it is natural to construct a heterogeneous computing platform with a classical host and a quantum coprocessor by adding extra data exchange instructions to interact with the host CPU and the main memory.

The verification of quantum software design creates a challenge. QuMA can assist the verification of quantum software and the estimation of their performance by simulating the generated instructions targeting QuMA. To this end, an architecture simulator for QuMA is required, which can simulate the execution of the instructions respecting hardware constraints and generate operations for each qubit with timing information. These timed operations can then be fed to a qubit state evolution simulator, such as QX [132] or QuantumSim [68]. In this way, the correctness of quantum software can be checked at both the architecture level and the qubit state level. Our previous work on the Quantum Platform Development framework (QPDO) [133] is a step towards building the required architecture simulator.

Programmable AWGs became available recently in industry [131, 134]. In these devices, the analog channels are coupled to a processor with a large memory. Instead of instructions with explicit quantum semantics, low-level instructions are used to generate the output, such as the waveform instruction, which takes a physical memory address as parameter. A distributed architecture with a synchronization mechanism is assumed to provide more analog channels. The required hardware resources go up almost linearly to the number of qubits. In contrast, QuMA is a centralized architecture with quantum semantics and timing of operations explicitly defined at the instruction level. It does not depend on an external synchronization mechanism and can scale up to control tens of qubits. By adopting the codeword-triggered pulse generation scheme, the AWG complexity can be reduced, which costs modest hardware. Also, the requirement for multiple control processors can be eliminated, making a simple compilation model and again asking for less hardware resources.

In recent years, quantum processors with more qubits are being produced. More qubits, in general, ask for more operations per unit time on average, which requires more operations to be fed into the queues. Only one instruction stream in QuMA results in a limited instruction issue rate, just as in classical processors. The limited instruction issue rate might be insufficient to issue all instructions in time that describe the required operations, which forms a bottleneck of QuMA. It is possible to make use of conventional processor design methods to optimize the non-deterministic timing do-main without affecting the deterministic timing of the output. Inspired by conventional processor design techniques, such as the Intel Streaming SIMD Extensions (SSE), we proposed a Single-Operation-Multiple-Qubit (SOMQ) execution fashion for QuMA in our recent research. Together with a very-long-instruction-word architecture (VLIW) update, we implemented the digital part of the improved QuMA in a device capable of controlling seven qubits. With a slight change to the configuration, such as VLIW width, the device can be, in principle, extended to control at least 49 qubits, which can form a distance-5 surface code logical qubit [39].

3.9 Conclusion

We have proposed and developed QuMA, a microarchitecture that takes the compiler generated instructions as input to flexibly control a superconducting quantum processor. Three mechanisms are introduced in QuMA to enable flexible control over quantum processors: i) codeword-based event control, ii) precise queue-based event timing control, and iii) multilevel instruction decoding pulse control mechanism. We have also designed and implemented the quantum microinstructions set QuMIS which can well describe quantum operations on qubits with precise timing.

We implemented a QuMA processor prototype on a FPGA. We have validated this microarchitecture by performing a successful *AllXY* experiment on a superconducting qubit, using a combination of the auxiliary classical instructions and QuMIS instructions which are generated by OpenQL. QuMA enables flexible definition of quantum experiments by a straightforward change in the input program.

We expect QuMA to spark a new line of research on a flexible and scalable approach to control near-term and future quantum chips. Building a quantum control microarchitecture and defining the required QISA can help the design of the control hardware, as well as the quantum software.

Note. The content of this chapter is based on the following papers:

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**, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE/ACM, 2017, pp. 813-825.

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, **A Microarchitecture for a Superconducting Quantum Processor**, *IEEE Micro*, vol. 38, pp. 40-47, 2018.

4

eQASM: An Executable QISA

4.1 Introduction

Quantum computing can accelerate solving some problems which are inefficiently solved by classical computers, such as quantum chemistry simulation [10, 14]. The goal is to develop a quantum computer with Noisy Intermediate-Scale Quantum (NISQ) technology [57] (without quantum error correction [79]), whose capability goes beyond that of state-of-the-art classical computers [56]. This capability is also termed quantum supremacy [58]. To this end, a fully programmable quantum computer based on the circuit model should be constructed of several layers [87, 117]. These layers form the full stack, which includes the quantum algorithm, quantum language, quantum compiler, quantum instruction set architecture (QISA), quantum control microarchitecture, quantum-classical interface, and quantum chip. Compared to the flourishing of research at the opposite ends of the stack, relatively less research has been dedicated to the low-level description of quantum applications with the required control microarchitecture for NISQ technology.

4.1.1 Related Work and the Challenges

To address the poor scalability of previous quantum control paradigms based on directly operating on waveforms and the problem that no control microarchitecture supports the execution of existing quantum assembly languages on real hardware (including QASM [50], a virtual instruction set [49], QASM-HL [53], Quil [95], OpenQASM [97], f-QASM [92], and cQASM [135]), we proposed the quantum control microarchitecture QuMA implementing a quantum microinstruction set QuMIS to bridge the gap between quantum software and hardware.

However, QuMIS is unsatisfactory for three reasons. First, instructions in QuMIS do not support feedback based on qubit measurement results, which is vital for circuit-model-based quantum computing applications such as active qubit reset [136], teleportation [137], quantum gate decomposition [119], and Shor's factoring [128]. For example, active qubit reset requires measuring the qubit followed by an X gate if the qubit measurement result is $|1\rangle$ (this process is also called binary control). Teleportation requires performing a subprogram (containing an X and Z gate) conditioned on the result of measurements on two qubits. In addition, feedback is necessary for fault-tolerant quantum computing where a key application is the implementation of non-Clifford gates (e.g., the T gate [79]). Feedback has been demonstrated in multiple experiments [63, 138–140] using customized hardware, but not yet using a (micro)architectural solution.

A second drawback of QuMIS is limited scalability. A QuMIS program has a relatively low instruction information density because (1) an explicit waiting instruction is required to separate any two consecutive timing points; (2) each target qubit of a quantum operation occupies a field in the instruction, making the instruction width a limitation for the number of target qubits in a single instruction; (3) two parallel and different operations cannot be combined into a single instruction. The required number of quantum operations per cycle in general increases as the number of qubits grows; fetching all instructions for an increasing number of quantum operations from memory and applying them on qubits on time forms a challenge given the limited instruction issue rate (the *quantum operation issue rate problem*) [110, 141, 142].

Third, QuMIS is limited in flexibility because QuMIS instructions are low level and tightly bound to the electronic hardware implementation. Compared to existing quantum assembly languages, QuMIS instructions are microinstructions without explicit quantum semantics. Thus, QuMIS does not qualify as a QISA, and it remains an open challenge to design an executable QISA with quantum semantics which is scalable and supports runtime feedback.

4.1.2 Contributions

In this chapter, we propose an executable QISA based on QASM, named *executable QASM* (eQASM). eQASM can be generated by the compiler backend from a higher-level representation, like OpenQASM or cQASM. eQASM contains both quantum instructions and auxiliary classical instructions to support quantum program flow control. eQASM supports a set of discrete quantum operations. The contributions of the chapter are the following:

- **Runtime Feedback:** eQASM proposes two kinds of feedback with required microarchitectural mechanisms to implement them: *fast conditional execution* for simple but fast feedback, and *comprehensive feedback control* (CFC) for arbitrary user-definable feedback;
- **Operational implementation:** eQASM is a QISA framework with the definition focusing on the assembly level and the basic rules of mapping assembly to binary. It requires customized instantiation for the binary format targeting a particular platform, which allows the pursuit of flexibility and practicability;
- **Increased quantum operation issue rate:** eQASM adopts *Single-Operation-Multiple-Qubit* (SOMQ) execution, Very-Long-Instruction-Word (VLIW) architecture and a more efficient method for explicit timing specification, which can considerably alleviate the quantum operation issue rate problem when compared to QuMIS;
- **Configurable QISA at compile time**: As opposed to the classical instruction set architecture (ISA) whose operations are defined at ISA design time, eQASM enables the programmer to configure allowed quantum operations at compile time, leaving ample space for compiler-based optimization.

We instantiate eQASM into a 32-bit instruction set targeting a seven-qubit superconducting quantum processor and implement it using a control microarchitecture derived from QuMA as proposed in [141]. We validated eQASM by performing several experiments over a two-qubit superconducting quantum processor using the implemented microarchitecture.

This chapter is organized as follows. Section 4.2 introduces the heterogeneous quantum programming model adopted by eQASM and an overview of eQASM. The quantum instructions of eQASM with related mechanisms are explained in Section 4.3. Section 4.4 describes the instantiation of eQASM targeting a seven-qubit quantum processor as well as its microarchitecture and implementation. Section 4.5 shows the experiments, and Section 4.6 concludes.

4.2 eQASM Overview

To our understanding, it is viable to integrate quantum computing in a similar way as a GPU or an FPGA in a heterogeneous architecture. The quantum part can be seen as a coprocessor used to accelerate particular classically-hard tasks. This section introduces the eQASM programming and compila-

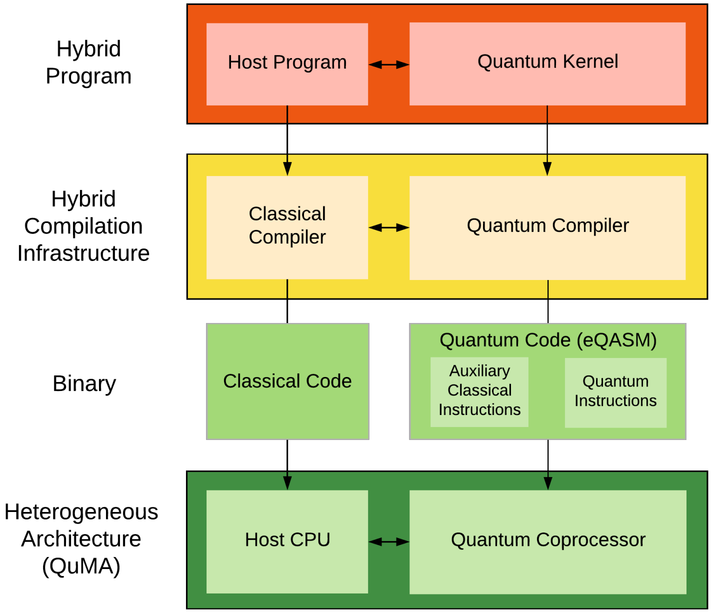


Figure 4.1: Heterogeneous quantum programming and compilation model.

tion model, the design guidelines for eQASM, the architectural state, and an overview of instructions.

4.2.1 Programming and Compilation Model

OpenCL [1] is an open industry standard for classical heterogeneous parallel computing which served as the basis for defining eQASM, of which the programming and compilation model is shown in Fig. 4.1.

A quantum-classical hybrid program contains a host program and one or more quantum kernels with the quantum kernel(s) accelerating particular parts of the computation. The host program is described using a classical programming language, such as Python or C++, and the quantum kernels are described using a quantum programming language, such as Scaffold [90] or Q# [70]. A hybrid compilation infrastructure compiles the host program into classical code using a conventional compiler such as GCC, which is later executed by the classical host CPU.

The quantum compiler, such as OpenQL [141], compiles the quantum kernels in two steps. First, quantum kernels are compiled into QASM, or a similar format mathematically equivalent to the circuit model. This format is hardware independent and can be ported across different platforms for quantum algorithms. Most of the hardware constraints are taken into account in the second step, where the compiler performs scheduling and low-level optimization. The output is the quantum code consisting of eQASM instructions. The quantum code contains quantum instructions as well as auxiliary classical instructions to support comprehensive quantum program flow control including runtime feedback [136, 143]. After the host CPU has loaded the quantum code into the quantum processor, the quantum code can be directly executed. In the rest of this paper, we focus on the quantum processor, i.e., the microarchitecture in charge of controlling qubits. The interaction between the classical processor and the quantum processor a research topic outside the scope of this paper.

4.2.2 Design Guidelines

The design of eQASM focuses on being executable on real hardware providing user-definable feedback. It should be capable of describing quantum applications for various quantum technologies and not bound to particular electronic control setup. Calibration experiments usually occupy a considerable ratio of the time using qubits in the NISQ era. Examples include measuring the relaxation time of qubits (T_1 experiment) and calibrating the parameters (amplitude, phase, frequency, etc.) of pulses for quantum operations, and so on. They need to use uncalibrated or uncommon quantum operations and explicitly change the timing of operations. eQASM is also expected to help quantum experiments required to calibrate qubits and quantum operations. The design of eQASM is guided by five main principles:

1. eQASM should include classical instructions to support quantum program flow control including runtime feedback;
2. eQASM should contain well-defined methods to specify the timing of quantum operations;
3. Low-level hardware information should be abstracted away from the eQASM assembly as much as possible to avoid eQASM being stuck to a particular hardware implementation;
4. 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;
5. Different experiments and radical compiler-based optimization techniques such as quantum optimal control [144, 145] may use a differ-

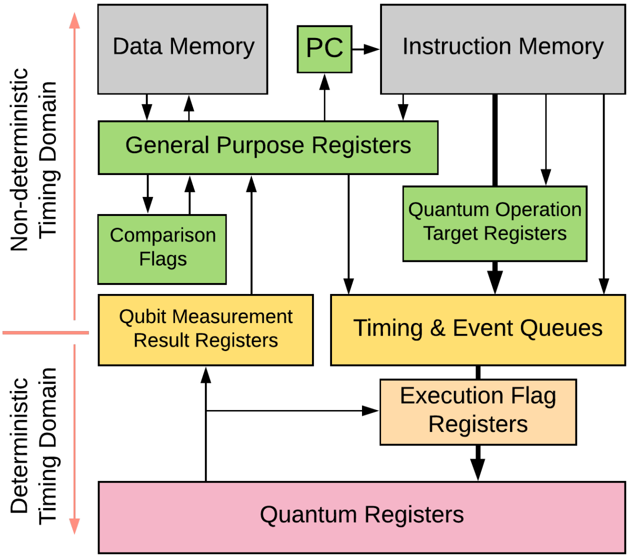


Figure 4.2: Architectural state of eQASM. Arrows indicates the possible information flow. The thick arrows represent quantum operations, which reads information from the modules passed through.

ent set of quantum operations, which can be uncalibrated or uncommon. eQASM should be flexible to allow different quantum operations via configuration.

4.2.3 Architectural State

As shown in Fig. 4.2, the architectural state of the quantum processor includes: a data memory, an instruction memory, a program counter (PC), a general purpose register (GPR) file, comparison flags, a quantum operation target register file, timing and event queues, a qubit measurement result register file, an execution flag register file, and a quantum register file.

Data Memory

The data memory can buffer intermediate computation results and serve as the communication channel between the host CPU and the quantum processor.

Instruction Memory & Program Counter

The eQASM instructions are stored in the instruction memory, and the Program Counter (PC) contains the address of the next eQASM instruction to fetch.

General Purpose Registers

The general purpose register (GPR) file is a set of 32-bit registers, labeled as `Ri`, where `i` is the register address.

Comparison Flags

The comparison flags store the comparison result of two general purpose registers which are used by comparison and branch related instructions (see Table 4.1).

Quantum Operation Target Registers

Each quantum operation target register can be used as an operand of a quantum operation. Since most quantum technologies support physical operations applied on up to two qubits, there are two types of quantum operation target registers: single-qubit target registers for single-qubit operations [including measurement (MEASZ)], and two-qubit target registers for two-qubit operations.

Each single- (two-)qubit target register can store the physical addresses of a set of qubits (allowed qubit pairs). An *allowed qubit pair* is a pair of qubits on which we can directly apply a physical two-qubit gate. A single- (two-)qubit target register is labelled as `Si` (`Ti`), with `i` being the register address. eQASM does not define the format of target registers (see Section 4.3.3 for a discussion).

Timing and Event Queues

To support explicit timing specification of quantum operations, eQASM adopts a queue-based timing control scheme [141]. The timing and event queues are used to buffer timing points and operations generated from the execution of quantum instructions (see Section 4.3.1). Together with the qubit measurement result registers, it separates the processor into two timing domains, the deterministic one and the non-deterministic one.

Qubit Measurement Result Registers

Each qubit measurement result register is 1-bit wide, and stores the result of the last finished measurement instruction on the corresponding qubit when it is valid (see Section 4.3.6). It is labeled as `Qi`, where `i` is the physical address of the qubit.

Execution Flag Registers

Sometimes, the execution of a quantum operation depends on a simple combination of previous measurement results of this qubit [63, 138]. To this end, each qubit is associated with an execution flag register, which contains multiple flags derived automatically by the microarchitecture from the last measurement results of this qubit. The execution flag register file is used for fast conditional execution (see Section 4.3.5).

Quantum Register

The quantum register is the collection of all physical qubits inside the quantum processor. Each qubit is assigned a unique index, known as the *physical address*. Since data in qubits can be superposed, eQASM does **not** allow direct access to the quantum data at the instruction level. Instead, users can measure qubits using measurement instructions and later access the results in the qubit measurement result registers.

4.2.4 Instruction Overview

Quantum technology is evolving rapidly and is still far away from a stable state. To avoid the format of eQASM being stuck to a specific quantum technology implementation with particular properties, the definition of eQASM focuses on the assembly level and introduces basic rules of mapping the assembly code to binary instructions. The binary format is defined during the instantiation of eQASM targeting a concrete control electronic setup and quantum chip. This fact enables the eQASM assembly to be expressive while leaving considerable freedom to the (micro)architecture designer to pursue microarchitectural practicability and performance.

An eQASM program can consist of interleaved quantum instructions and auxiliary classical instructions. An overview of the eQASM instructions is shown in Table 4.1. Since the host CPU can provide classical computation power, auxiliary classical instructions are simple instructions to support the execution of quantum instructions. Complex instructions (e.g., floating-point instructions) are not included.

The top part of Table 4.1 contains the auxiliary classical instructions. There are four types: *control*, *data transfer*, *logical*, and *arithmetic* instructions. These are all scalar instructions. The function `sign_ext(Imm, 32)` sign extends the immediate value `Imm` to 32 bits. The operator `::` concatenates the two bit strings. The `CMP` instruction sets all comparison flags based on the comparison result of GPR `Rs` and `Rt`. The `BR` instruction changes the PC to `PC + Offset` if the specified comparison flag is '`1`'. To enable arithmetic or logical operations on the comparison result, the `FBR` instruction fetches the specified comparison flag into GPR `Rd`. The `FMR` instruction supports comprehensive feedback control and is explained in Section 4.3.6.

The bottom part of Table 4.1 contains the quantum instructions. There are three types of instructions:

- Waiting instructions used to specify timing points (`QWAIT`, `QWAITR`),
- Quantum operation target register setting instructions (`SMIS`, `SMIT`), and
- Quantum bundle instructions, which consist of the specification of a small waiting time and multiple quantum operations.

These quantum instructions have several features based on the following four observations:

- Many quantum experiments, such as the T_1 experiment, require changing the timing of operations explicitly. Also, the timing of operations can significantly impact the fidelity of the final result as quantum errors accumulate during computation (see Section 4.5). eQASM can explicitly specify the timing of quantum operations to support quantum experiments and compiler-based timing optimization. The timing model is explained in Section 4.3.1.
- Different quantum experiments or algorithms may require a different set of physical quantum operations. To allow using different sets of quantum operations, quantum operations are specified by programmers at compile time via configuration (see Section 4.3.2) instead of being defined at QISA design time. This flexibility reserves ample space for

Table 4.1: Overview of eQASM Instructions.

Type	**Syntax**	**Description**
Control	`CMP Rs, Rt`	Compare GPR `Rs` and `Rt` and store the result into the comparison flags.
	`BR <Comp. Flag>, Offset`	Jump to `PC + Offset` if the specified comparison flag is ‘`1`’.
Data Transfer	`FBR <Comp. Flag>, Rd`	Fetch the specified comparison flag into GPR Rd.
	`LDI Rd, Imm`	Rd = sign_ext(Imm[19..0], 32).
	`LDUI Rd, Imm, Rs`	Rd = Imm[14..0]::Rs[16..0].
	`LD Rd, Rt(Imm)`	Load data from memory address `Rt + Imm` into GPR `Rd`.
	`ST Rs, Rt(Imm)`	Store the value of GPR `Rs` in memory address `Rt + Imm`.
	`FMR Rd, Qi`	Fetch the result of the last measurement instruction on qubit `i` into GPR `Rd`.

Logical	`AND/OR/XOR Rd, Rs, Rt`
	`NOT Rd, Rt`
		Logical and, or, exclusive or, not.

Arithmetic	`ADD/SUB Rd, Rs, Rt`	Addition and subtraction.

Waiting	`QWAIT Imm`
	`QWAITR Rs`
		Specify a timing point by waiting for the number of cycles indicated by the immediate value `Imm` or the value of GPR `Rs`.

Target Specify	`SMIS Sd, <Qubit List>`
	`SMIT Td, <Qubit Pair List>`
		Update the single- (two-)qubit operation target register `Sd` (`Td`).

Q. Bundle	`[PI,] Q_Op [| Q_Op]*`	Applying operations on qubits after waiting for a small number of cycles indicated by `PI`.

compiler-based optimization. Only single- and two-qubit operations are allowed, and more-qubit operations should be decomposed into single- and two-qubit operations by the compiler [51, 55, 113–115].

- To alleviate the quantum operation issue rate problem, eQASM adopts SOMQ execution, which supports applying a single quantum operation on multiple qubits (see Section 4.3.3), and a VLIW architecture which can combine multiple different quantum operations into a quantum bundle (see Section 4.3.4).
- Two kinds of feedback are supported. Fast conditional execution performs a Go/No-go decision for every single-qubit operation based on a execution flag of the target qubit (see Section 4.3.5). To be more flexible, CFC allows programmers to define arbitrary feedback by redirecting the program flow based on the measurement results (see Section 4.3.6).

4.3 Architecture

In this section, we construct the assembly syntax of quantum operations by introducing the aforementioned mechanisms.

4.3.1 Timing Model

Queue-based Timing Control

eQASM adopts the queue-based timing control scheme proposed in [141] since it can support explicit timing specification. We briefly introduce this scheme and refer readers to the original paper for a detailed discussion.

In the queue-based timing control scheme, the execution of quantum instructions can be divided into a *reserve* phase in the non-deterministic timing domain and a *trigger* phase in the deterministic timing domain. A timeline is constructed by the reserve phase and consumed by the trigger phase: the result of executing quantum instructions in the reserve phase is consecutively creating new timing points on the timeline and associating events to them; the deterministic timing domain maintains a timer, and triggers all quantum operations associated with the timing point on the timeline that it reaches. Auxiliary classical instructions and mask setting instructions are not directly associated with timing points. The trigger phase is handled by the microarchitecture; we introduce the reserve phase in the following.

Timeline Construction

Quantum instructions fetched from the instruction memory form a quantum instruction stream. Instructions in the stream are executed in order; this constructs a timeline by generating consecutive timing points and assigning operations to them.

If the fetched instruction is a waiting instruction, `QWAIT Imm` or `QWAITR Rs`, a new timing point in the timeline is generated. The position of the new timing point is determined by the specification of the interval since the last generated timing point. The interval length comes from the immediate value `Imm` or GPR `Rs`. The first timing point of the timeline can be set by a dedicated instruction, or by an external trigger to the microarchitecture. Both waiting instructions use the unit cycle for the interval length.

If the fetched instruction is a quantum bundle instruction, the quantum operation(s) specified in the bundle instruction is associated with the last generated timing point. If multiple quantum operations are associated to the same timing point, these quantum operations will all start execution at that same timing point.

Based on our observation over some testbenches (see Section 3.6), short intervals between timing points are a common case. To improve the quantum operation issue rate, eQASM allows merging a `QWAIT PI` instruction followed by a quantum operation `<Quantum Operation>`

```
QWAIT PI
<Quantum Operation>
```

into a single instruction

```
[PI,] <Quantum Operation>.
```

Square brackets `[...]` indicate that the content inside is optional. `PI` is short for `pre_interval`, which specifies a short interval between last generated timing point and the one when the operations in this instruction are to be triggered. It defaults to 1 if not specified. Value 0 is acceptable to both the `PI` and the waiting instructions, which means that the following timing point is identical to the last timing point.

Example

Assuming the durations of quantum operations `q_op0`, `q_op1`, `q_op2`, and `q_op3` all equal one-cycle time, the following code triggers these four operations back-to-back.

```
LDI  r0, 1     # r0 <- 1
q_op0
q_op1          # Default PI = 1
QWAITR r0      # Register-valued waiting
0, q_op2
QWAIT 0        # Equivalent to NOP
1, q_op3       # Explicity PI = 1
```

4.3.2 Quantum Operation Definition & Decoding

Depending on the qubit technology and the algorithm to run, different quantum operations can be used. eQASM does not define a fixed set of quantum operations at QISA design time, such as $\{H, T, \text{CNOT}, \cdots\}$. Instead, the available quantum operations can be configured by the programmer at compile time.

Flexible quantum operation configuration is achieved through the configuration of the assembler, the microcode unit and the pulse generator of the microarchitecture: on the one hand, the assembler is configured to translate a quantum operation, e.g., the X gate, to the expected opcode, e.g., 0x01; on the other hand, the microcode unit translates the quantum opcodes into the expected microinstruction(s) using a microcode-based instruction decoding scheme [127]. Each microinstruction represents one or more micro-operations, which are finally converted into pulses by the pulse generator with precise timing applying operations on qubits. The assembler, the microcode unit, and the pulse generator should be configured consistently at compile time.

4.3.3 Address Mechanism

A quantum operation applied on multiple qubits is a common case. For example, quantum computation usually starts by preparing the superposition state from initialized qubits, which requires applying Hadamard gates on multiple qubits. eQASM uses SOMQ execution, which can apply a single quantum operation on multiple qubits at the same time. SOMQ is similar to classical single-instruction-multiple-data (SIMD) execution [146], with the operation target replaced by qubits. An instantiated eQASM can also be treated as an implementation of the previously proposed Multi-SIMD(k, d) architecture [54] but removing the assumption of SIMD regions that in each region only a single quantum operation can be applied.

SOMQ is based on an indirect qubit addressing mechanism. The `SMIS` or `SMIT` instruction first defines a set of quantum operation target(s) in a quantum operation target register. Then a quantum operation can use the target register as the operand:

```
<Operation Name>  <Target Register>.
```

Address of Allowed Qubit Pairs

Since a two-qubit operation, such as a **`CNOT`** gate, can operate on its qubits differently, two qubits with different orders, i.e., `(Qubit A,Qubit B)` and `(Qubit B,Qubit A)`, are treated as different allowed qubit pairs. The term *quantum chip topology* indicates the available qubits and allowed qubit pairs of a quantum chip (see Fig. 4.6 for an example). The quantum chip topology can be represented as a graph where each available qubit can be denoted as a vertex, and an allowed qubit pair as a directed edge. In the directed edge `( Qubit A, Qubit B)`, `Qubit A` is called the source qubit and `Qubit B` the target qubit of the pair.

Translation from Assembly to Binary

Since the efficiency of encoding the qubit list (qubit pair list) may depend on the target quantum chip topology, the designer can choose different binary encoding schemes for different target quantum processors during eQASM instantiation. In general, it is more efficient to put the address pairs in the instruction for a highly-connected quantum processor, while a mask format could be more efficient when the qubit connectivity is limited. For example, since at most two two-qubit gates can be applied and each qubit can be addressed with 3 bits in a fully connected 5-qubit trapped ion processor [25], only $2\times2\times3$ bits $= 12$ bits are required to specify the target of a two-qubit gate. This is more efficient than a mask of 20 bits with each bit in the mask indicating one of all 20 different allowed qubit pairs selected or not. In contrast, a mask of 6 bits is more efficient for the IBM QX2 [147], which also contains five qubits but has only six allowed qubit pairs.

Example

The following code sets the single-qubit target register `S7` to contain two qubits (0 and 1), and then applies an X gate on both qubits simultaneously.

```
SMIS S7, {0, 1}
Y    S7
```

The following code sets the two-qubit target register `T3` to contain two pairs of qubits (1, 3) and (2, 4), and then applies a **`CNOT`** gate on them.

```
SMIT T3, {(1, 3), (2, 4)}
CNOT T3
```

4.3.4 Very Long Instruction Word

Quantum Bundle Format

Apart from SOMQ, different operations are also allowed to be applied on different qubits in parallel. eQASM can combine parallel quantum operations into a *quantum bundle* in a VLIW format. We define parallel quantum operations as operations starting at the same timing point, regardless of the duration of each operation. The format of a quantum bundle is:

```
[PI,] <Quantum Operation> [| <Quantum Operation>]*
```

The vertical bar `|` is used to separate different quantum operations in the same bundle. The asterisk $\star$ means the item in square brackets can repeat for $n \geq 0$ times.

Translation from Assembly to Binary

In the assembly code, an arbitrary number of quantum operations can be combined into a single quantum bundle. However, a single instruction can accommodate only a few quantum operations because of the limited instruction width. The *VLIW width* of eQASM characterizes the number of quantum operations that can be put in a single instruction word, which is defined during eQASM instantiation. Matching this, a single quantum bundle can be broken into multiple quantum bundle instructions with `PI` being 0. If the number of operations is not a multiple of the VLIW width, quantum no-operations

(QNOP) fill up the last instruction. For example, given a VLIW width of 2, the bundle

```
PI, X S5 | H S7 | CNOT T3
```

can be decomposed by the assembler to two consecutive quantum bundle instructions

```
PI, X S5 | H S7
0, CNOT T3 | QNOP.
```

Example

In the code as shown in Fig. 4.3, the instruction `QWAIT 10000` initializes both qubits by idling them for 200 μs (assuming a cycle time of 20 ns). Line 6 applies a Y gate on both qubits using SOMQ. Line 7 is a VLIW instruction, which applies an X_{90} and X gate on each qubit. In this paper, X_{90} (Y_{90}) denotes the gate rotating the quantum state along the x- (y-)axis by a $\pi/2$ angle. $X_{\mathrm{m}90}$ ($Y_{\mathrm{m}90}$) denotes similar gates but with the rotation angle of $-\pi/2$. Line 8 measures both qubits using SOMQ. According to the `PI` value, the Y gate happens immediately after the initialization, followed by the X_{90} and X gates 20 ns later and the measurement 40 ns later. The 1 μs waiting time (line 9) ensures no operations happening during the measurement.

```
1  SMIS S0, {0}
2  SMIS S2, {2}
3  SMIS S7, {0, 2}
4  ...
5  QWAIT    10000
6  0, Y     S7
7  1, X90   S0   | X   S2
8  1, MEASZ S7
9  QWAIT    50
10 ...
```

Figure 4.3: Part of the code for a two-qubit *AllXY* experiment, which is used in validating eQASM in Section 4.5.

4.3.5 Fast Conditional Execution

Fast conditional execution allows executing or canceling a single-qubit operation when the micro-operation is triggered. The decision is made based on the value of a selected flag in the execution flag register corresponding to the target qubit. The value of the execution flag is derived by the microarchitecture using predefined combinatorial logic from the last measurement results of the same qubit. Once there returns a measurement result for a qubit, the corresponding execution flags are updated automatically. If the execution flag is ‘`1`’, then the operation executes; otherwise, it is canceled. A selection signal is required for each micro-operation to select which execution flag to use, which can be generated by the microcode unit, or specified by an instruction field [49]. Except for the default execution flag that should always be ‘`1`’, which and how many execution flags there are, should be defined during eQASM instantiation (see Section 3.4 for an example).

Example

In one instantiation of eQASM, the quantum operation `C_X` uses the execution flag which is ‘`1`’ if and only if (**iff**) the last measurement result of the qubit is $|1\rangle$. Figure 4.4 shows the code for the active qubit reset experiment, where qubit 2 is put in an equal superposition using an X_{90} gate after initializing it in the $|0\rangle$ state by idling it for 200 μs. After a measurement, a conditional `C_X` gate is applied to reset the qubit. Qubit 2 is measured again to read out the final state for verification.

```
SMIS    S2, {2}
QWAIT   10000
X90     S2
MEASZ   S2
QWAIT   50
C_X     S2
MEASZ   S2
```

Figure 4.4: eQASM program for active qubit reset. This experimental result is shown in Section 4.5.

4.3.6 Comprehensive Feedback Control

CFC allows adjusting the program flow based on measurement results of any qubits to enable arbitrary user-defined feedback. This flexibility comes at the cost of longer feedback latency. We propose a three-step mechanism to implement CFC:

1. A measurement instruction is applied on the condition qubit `i`. At the moment that this measurement instruction is issued, `Qi` is invalidated. At the moment the measurement result is available, it is written in `Qi`. `Qi` turns back to valid if there are no more pending measurement instructions on qubit `i`.
2. The `FMR Rd, Qi` instruction fetches the value of the quantum measurement result register `Qi` into GPR `Rd`. If `Qi` is invalid, `FMR` should wait until `Qi` gets valid again. Thereafter, the value of `Qi` can be fetched into `Rd`. `Qi` remains valid until qubit `i` is measured again.
3. GPR `Rd` is then used in a `BR` instruction to select the program flow to follow. Note, multiple `FMR` and `BR` instructions can be combined to support more complex feedback logic.

Example

The eQASM program shown in Fig. 4.5 first measures qubit 1. If the measurement result is 1, a Y gate is applied on qubit 0, otherwise, an X gate is applied.

4.4 Instantiation & Implementation

This section introduces an instantiation, microarchitecture, and implementation of eQASM.

4.4.1 Target Superconducting Quantum Chip

The quantum chip topology of the target seven-qubit superconducting quantum chip is shown in Fig. 4.6. It is part of a two-dimensional square lattice as proposed in [81]. It can implement a distance-2 surface code [39], which can detect one physical error. In this figure, a vertex represents a qubit, and a directed edge represents an allowed qubit pair. Numbers besides the vertex (edge) are the addresses of qubits (allowed qubit pairs). For example, allowed

```
  SMIS  S0, {0}
  SMIS  S1, {1}
  LDI   R0, 1
  MEASZ S1
  QWAIT 30
  FMR   R1, Q1      # fetch msmt result
  CMP   R1, R0      # compare
  BR    EQ, eq_path # jump if R0 == R1
ne_path:
  X     S0    # happen if msmt result is 0
  BR  ALWAYS, next # this flag is always ‘1’
eq_path:
  Y     S0    # happen if msmt result is 1
next:
  ...
```

Figure 4.5: eQASM program using CFC.

qubit pair 0 has qubit 2 as the source qubit and qubit 0 as the target qubit. The feedlines are used to measure the nearby coupled qubits. Qubit 0, 2, 3, 5, and 6 (1 and 4) are coupled to feedline 0 (1). Each feedline has an input port and an output port. Besides, each qubit is connected to a microwave port and a flux port, which are not shown in Fig. 4.6.

Operations supported by this quantum processor include measurements, single-qubit x- or y-axis rotations, and a two-qubit controlled-phase (CZ) gate. A typical gate time is 20 ns for single-qubit gates and $\sim$ 40 ns for two-qubit gates. The duration of a measurement is typically 300 ns - 1 μs. A cycle time of 20 ns is used in this instantiation.

4.4.2 eQASM Instantiation Design Space Exploration

To determine a suitable eQASM instantiation configuration for the target quantum processor [a single- (two-)qubit gate time of 1 (2) cycle(s), and a measurement time of 15 cycles], we perform analysis over three benchmarks using a quantum control architecture simulator derived from the previously proposed QPDO [133]. Because substantial time is spent on calibrating qubits before running applications with NISQ technology, the first benchmark we select is the widely-used calibration experiment randomized benchmarking

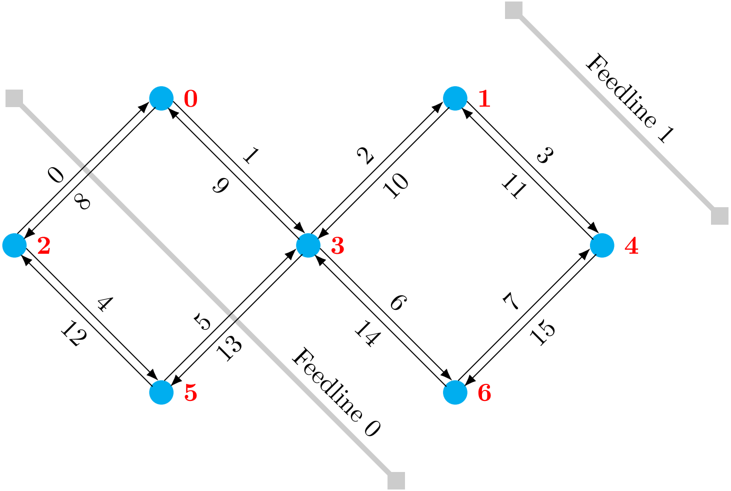


Figure 4.6: Quantum chip topology of the target seven-qubit superconducting quantum chip. Numbers in red are the physical addresses of qubits. The numbers along the direct edges are addresses of the allowed qubit pairs.

(RB) [77, 78], which might be limited by the high memory consumption when the required waveform for control is plainly stored in memory. Each qubit is subject to 4096 single-qubit Clifford gates which have been decomposed into x and y rotations. Because every gate happens immediately following the previous one, randomized benchmarking cannot reveal timing patterns of quantum operations in real quantum algorithms, where the parallelism is limited by two-qubit gates. Addressing this, we also select two benchmarks from ScaffCC [53] as the representatives of small-scale quantum algorithms that might be executed with NISQ technology: a parallel algorithm (Ising model using 7 qubits, IM) which has $< 1\%$ two-qubit gates, and a relatively sequential algorithm (Grover's algorithm to calculate the square root using 8 qubits, which is the minimum number of qubits required, SR), which has $\sim 39\%$ two-qubit gates. The evaluation metric is the total number of instructions.

We investigate the impact of the VLIW width (w), three timing-specification methods, and SOMQ on the number of instructions. The three timing-specification methods include: the QuMIS fashion (specifying every timing point using separate `QWAIT` instructions, ts_1); including `QWAIT` in the quantum bundle instruction at the place of a quantum operation (ts_2); and using `PI` with various bit widths (w_{PI}) to specify a small waiting time and using separate `QWAIT` instructions to specify longer waiting times (ts_3). The simulation results are shown in Fig. 4.7.

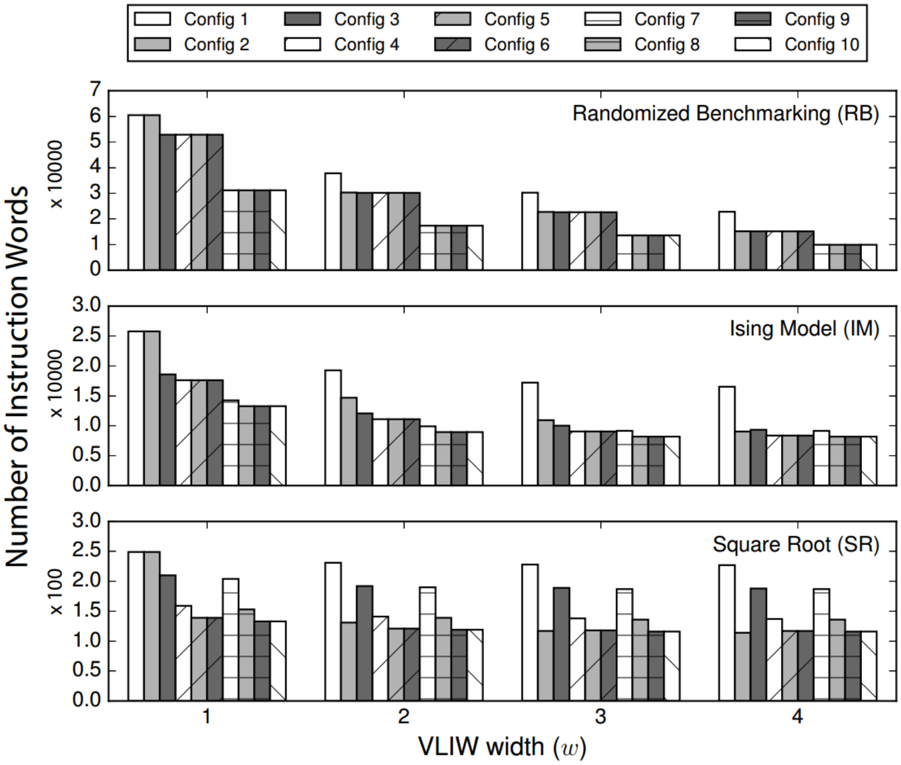


Figure 4.7: Number of instructions for various architecture configurations for randomized benchmarking, Ising model, and square root.

Config 1 is (ts_1, no `PI`, no SOMQ), and Config 1 with $w = 1$ is chosen as the baseline. By increasing w from 1 to 4, the number of instructions can be reduced up to 62% (RB). Benchmarks with substantial parallelism (RB and IM) benefit more from a big w. The instruction reduction in SR ($\sim 8\%$) indicates that large w slightly improves quantum applications with limited parallelism.

Config 2 is (ts_2, no `PI`, no SOMQ). A minimum w of 2 is required by ts_2 to distinguish it from ts_1. Compared with Config 1, by including the `QWAIT` operation as part of a quantum bundle instruction, Config 2 can reduce the number of instructions by 20 - 33% (RB), 24 - 45% (IM), 43 - 50% (SR) by varying w from 2 to 4. SR benefits most because of two reasons. First, due to its sequential nature, it has relatively more `QWAIT` instructions. Second, limited parallelism in this algorithm leaves potential VLIW slots unused, which can be filled by `QWAIT` instructions.

Config 3/4/5/6 is (ts_3, $w_{\mathrm{PI}} = 1/2/3/4$, no SOMQ). Config 3 can reduce the number of instructions by 13 - 33% for RB and 28 - 44% for IM with w varying from 1 to 4 compared with Config 1. Since the intervals between operations in RB and IM are mostly close to 1, further increasing w_{PI} up to 4 bits introduces marginal benefit. Config 3 reduces the number of instructions of SR by $\sim 17\%$ regardless of w. Further increasing w_{PI} to 3 or 4 bits can reduce the number of instructions of SR by up to 48%. Like SR, quantum algorithms are scheduled to be executed in a time as short as possible. This result of Config 3-6 suggests that most of the waiting time is short and can be encoded in a 3-bit `PI` field. Note that Config 3/4/5 is also more beneficial than Config 2 when $w = 1$ or $w = 2$.

Config 7/8/9/10 is (ts_3, $w_{\mathrm{PI}} = 1/2/3/4$, SOMQ). Our analysis assumes that the target registers can always provide the required qubit (pair) list, and therefore shows the theoretical maximum benefit that can be obtained by SOMQ. Compared to Config 3/4/5/6, SOMQ can introduce a maximum reduction of 42% (Config 8, $w = 2$) in the number of instructions for RB, while it can only reduce at most 4% instructions for SR (Config 8, $w = 1$). Regardless of w_{PI}, SOMQ can help reduce the number of instructions of IM by $\sim$ 24, 19, 9, and 2% for different w. This fact suggests that SOMQ is more effective for highly parallel applications, especially when w is small. An application that would benefit significantly from SOMQ is quantum error correction, which requires performing well-patterned error syndrome measurements repeatedly presenting high parallelism. As not shown in the figure, we also analyzed the number of effective quantum operations in each quantum bundle for Config 9, which is 1.795, 2.296, and 3.144 for RB, 1.485, 1.622, and 1.623 for IM, and 1.118, 1.147, and 1.147 for SR with w varying from 2 to 4, respectively. It indicates that with the existence of SOMQ, $w > 2$ is not highly required for many quantum applications (RB is a special case with extreme parallelism).

As a result of the analysis, our eQASM instantiation adopts Config 9 (ts_3, $w_{\mathrm{PI}} = 3$, SOMQ) with $w = 2$. A width of 32 bits is used by all instructions for the memory alignment. Two instruction formats are used: the single format with the highest bit being ‘`0`’ and the bundle format with the highest bit being ‘`1`’. Single format instructions use the other 31 bits to encode a single instruction, including all auxiliary classical instructions, and `SMIS`, `SMIT`, `QWAIT(R)` instructions. For brevity, we only present the format of quantum instructions as shown in Fig. 4.8.

There are 32 single- (two-)qubit target registers, and the target register address width is 5 bits. The target registers use a mask format. The mask is 7- (16-)bit wide in the single- (two-)qubit target register. Each bit in the mask of the value

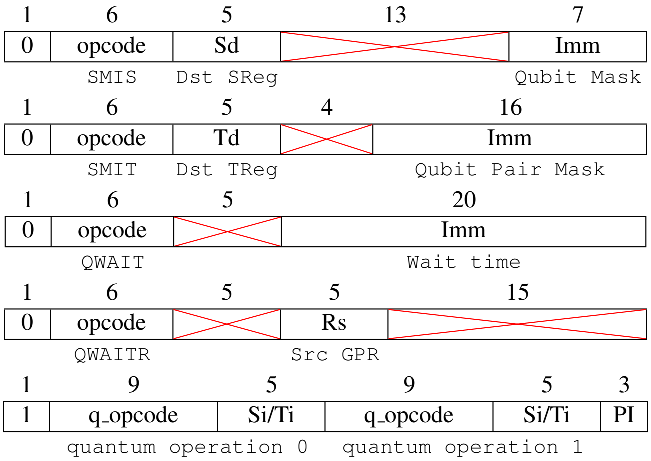


Figure 4.8: Format of the `SMIS` and `SMIT` (top two), `QWAIT` and `QWAITR` (middle two), and quantum bundle (bottom) instruction.

'`1`' indicates that the corresponding qubit (allowed qubit pair) is selected. In the `QWAIT(R)` instruction, only the least significant 20 bits of the `Imm` field or GPR `Rs` are used to specify the waiting time. In the quantum bundle instruction, each quantum operation occupies 14 bits and the q_opcode is 9 bits.

4.4.3 Microarchitecture

QuMIS is implemented by the control microarchitecture QuMA with codeword-based event control, queue-based event timing control and multi-level instruction decoding [141]. Adopting these three mechanisms, we redesign a quantum control microarchitecture, QuMA_v2, implementing the instantiated eQASM as shown in Fig. 4.9. It supports all features of eQASM. The classical pipeline maintains the PC and implements the GPR file and the comparison flags. The execution flag register is maintained by the fast conditional execution module. The classical pipeline fetches and processes instructions one by one from the instruction memory. All auxiliary classical instructions are processed by the classical pipeline while quantum instructions are forwarded to the quantum pipeline for further processing.

The timestamp manager processes the `QWAIT(R)` instructions and the `PI` field to generate timing points. The quantum pipeline contains a VLIW front end with two VLIW lanes, each lane processing one quantum operation. The `SMIS` (`SMIT`) instructions update the corresponding target registers in each VLIW

Table 4.2: Definition of the micro-operation selection signal.

Value	Operation to Select	Value	Operation to Select
'`00`'	None	'`10`'	μ_op_{tgt}
'`01`'	μ_op_{src}	'`11`'	μ_op_{s}

lane. Inside each VLIW lane, the q_opcode is translated by the microcode unit into one micro-operation (labeled as μ_op_{s}) for a single-qubit operation or two micro-operations (labeled as μ_op_{src} and μ_op_{tgt}) for a two-qubit operation. μ_op_{src} (μ_op_{tgt}) will be applied on the source (target) qubit of the target qubit pair. The configuration of the microcode unit is stored in the Q control store, which is implemented using a lookup table. The target register `Si` (`Ti`) is read for a single- (two-)qubit operation.

The quantum microinstruction buffer resolves the mask-based qubit address and associates the quantum operations to the last generated timing point. It resolves the qubit address in two steps.

First, the mask stored in `Si` (`Ti`) is translated into seven two-bit micro-operation selection signals OpSel_i, where $i = 0, 1, \cdots, 6$, with each signal for one qubit. Table 4.2 lists the meaning of every case of the micro-operation selection signal. For single-qubit operations, OpSel_i is set to '`11`' ('`00`') if the i-th bit in the mask is '`1`' ('`0`'). For a two-qubit operation, OpSel_i is set to '`00`' if qubit i is not contained in any selected allowed qubit pair. Otherwise, OpSel_i is '`01`' ('`10`') if the target qubit pair contains qubit i as the source (target) qubit. Take qubit 0 as an example. It is connected to edges 0, 1, 8, and 9. When edge 0 or 9 (1 or 8) is selected in the mask, qubit 0 is the target (source) qubit and should be applied with μ_op_{tgt} (μ_op_{src}). In other words, OpSel_0 should be '`10`' ('`01`'), and can be generated using a simple `OR` ($\vee$) logic:

$$\mathrm{OpSel}_0 = (\mathtt{Ti}[0] \vee \mathtt{Ti}[9]) :: (\mathtt{Ti}[1] \vee \mathtt{Ti}[8]).$$

The assembler should check the validity of two-qubit target register values. For example, it is invalid if two edges connecting to the same qubit are selected in the same `T` register.

Second, based on OpSel_i, either none or one micro-operation is output for qubit i. This step is fully parallel.

The operation combination module also works in a two-step fashion. First, since each VLIW lane outputs none or one micro-operation for each qubit, the operation combination module merges both micro-operations from both VLIW lanes. If both VLIW lanes output one micro-operation on the same

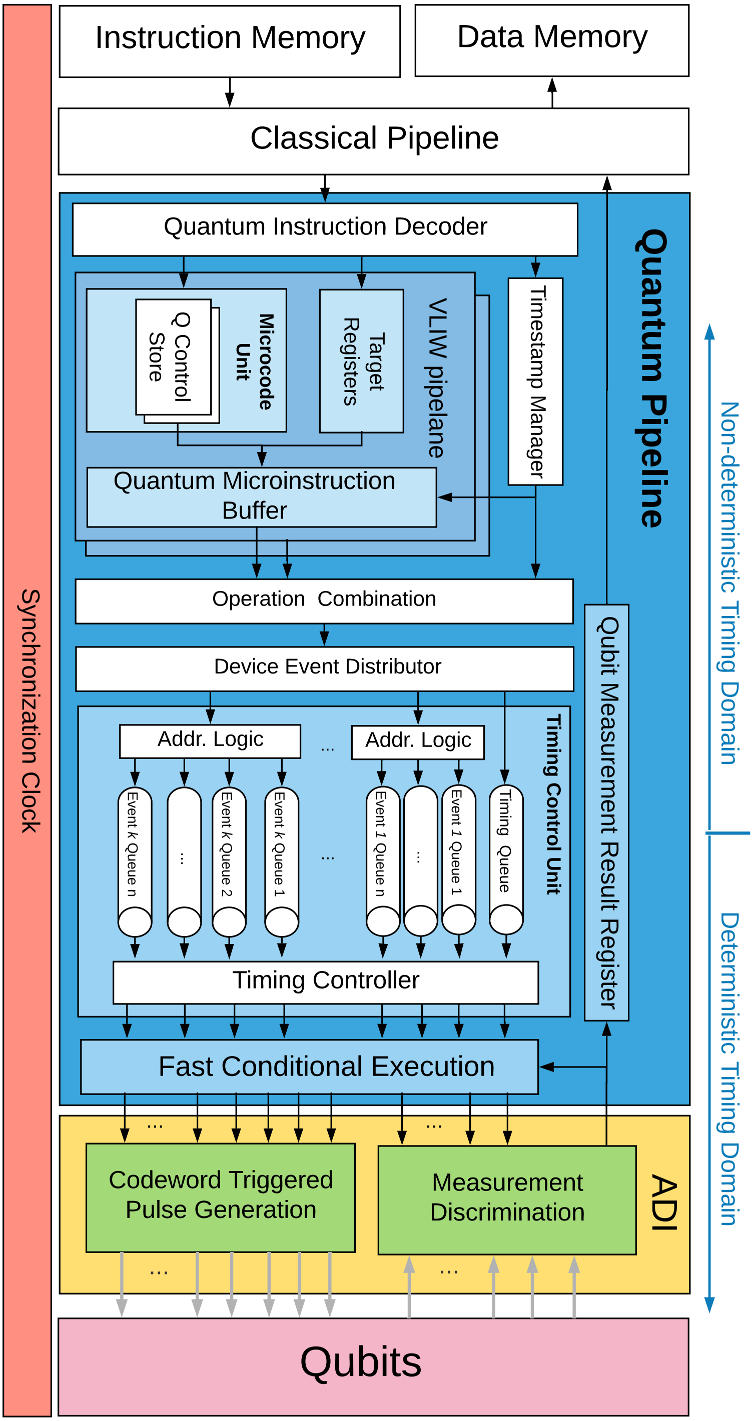


Figure 4.9: Quantum microarchitecture implementing the instantiated eQASM for the seven-qubit superconducting quantum processor.

qubit, an error is raised, and the quantum processor stops. Second, as explained in Section 4.3.4, a long quantum bundle requires multiple quantum bundle instructions to describe it. The operation combination module buffers all micro-operations associated with the same timing point. Only when it detects that all quantum operations in the same quantum bundle have been collected, the operation combination module sends the buffered micro-operations to the device event distributor. This detection can be done, e.g., by recognizing a new timing point generated by the timestamp manager which is different to the one associated to the buffered micro-operations. Also, if two different quantum bundle instructions specify a quantum operation on the same qubit, an error is raised, and the quantum processor stops.

As shown in Section 3.6, operating a qubit may require the collaboration of multiple electronic devices in the analog-digital-interface, and a single device may also control multiple qubits. Hence, the micro-operations should be reorganized into *device operations* to trigger the corresponding devices. The device event distributor reorganizes multiple micro-operations associated with the same timing label into different device operations. After that, each device operation with the associated timing label is buffered at an event queue of the timing control unit awaiting execution. The timing controller then triggers every device operation at its expected timing point.

After the device operations have been triggered by the timing controller, fast conditional execution is performed based on the selected execution flags of the target qubits. The execution flag selection signal comes from the microcode unit configured by the programmer. Only device operations for qubits of which the selected execution flag is '`1`' are released to the analog-digital interface (ADI). In this eQASM instantiation, four types of combinatorial logic are used to define the execution flags:

1. '`1`' (the default for unconditional execution);
2. '`1`' **iff** the last finished measurement result is $|1\rangle$;
3. '`1`' **iff** the last finished measurement result is $|0\rangle$;
4. '`1`' **iff** the last two finished measurements get the same result.

Note, the last finished measurement result refers to the result of the last finished measurement instruction on this qubit when these flags are used. It is irrelevant to the validity of the quantum measurement result register. Once there returns a measurement result for a qubit from the analog-digital interface, the fast conditional execution unit immediately update the execution flags corresponding to that qubit.

To support CFC, a counter `Ci` is attached to each qubit measurement result register `Qi`, with an initial value of 0. Once a measurement instruction acting on qubit i is issued from the classical pipeline to the quantum pipeline, `Ci` increments by 1. If the measurement discrimination unit writes back a measurement result for qubit i, `Ci` decrements by 1. `Qi` is valid only when `Ci` is 0. If `Ci` is not 0 when the instruction `FMR Rd, Qi` is issued, the pipeline is stalled until `Ci` 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.

4.4.4 Implementation

The hardware structure implementing the microarchitecture (Fig. 4.10) consists of a Central Controller responsible for orchestrating three modules containing slave devices for microwave control, flux control, and measurement.

The Central Controller is a digital device built with an Intel Altera Cyclone V SOC 5CSTFD6D5F31I7N Field Programmable Gate Array (FPGA) chip. The Central Controller implements the digital part of the microarchitecture (left to the ADI in Fig. 4.9). The timing controller and fast conditional execution module work at $50\ \mathrm{MHz}$ to get a cycle time of $20\ \mathrm{ns}$. The other parts work at $100\ \mathrm{MHz}$.

Single-qubit x and y rotations are performed by applying microwave pulses to the qubits. The pulses are generated by Zurich Instruments High Density Arbitrary Waveform Generators (HDAWG) and modulated using a Rohde & Schwarz (R&S) SGS100A microwave source. A custom-built vector switch matrix (VSM) is responsible for duplicating and routing the pulses to the respective qubits as well as tailoring the waveforms to the individual qubits [100] using a qubit-frequency reuse scheme that allows for efficient scaling of the microwave control module [81].

Flux pulses that implement two-qubit CZ gates and single-qubit z rotations are performed by applying pulses generated by an HDAWG on the dedicated flux lines for each qubit.

The measurement discrimination unit is implemented using two Zurich Instruments Ultra-High-Frequency Quantum Controllers (UHFQC) connected to the two feedlines shown in Fig. 4.6. The UHFQC has two analog outputs that can be used to generate the measurement pulses and two analog inputs to sample the transmitted signals from which the UHFQC can infer the measurement result. The measurement pulses going to (coming from) the qubits are modulated (demodulated) using a single R&S SGS100A. All analog ports operate at $1.8\ \mathrm{GSa/s}$ allowing for simultaneous measurement of up to 9 qubits per

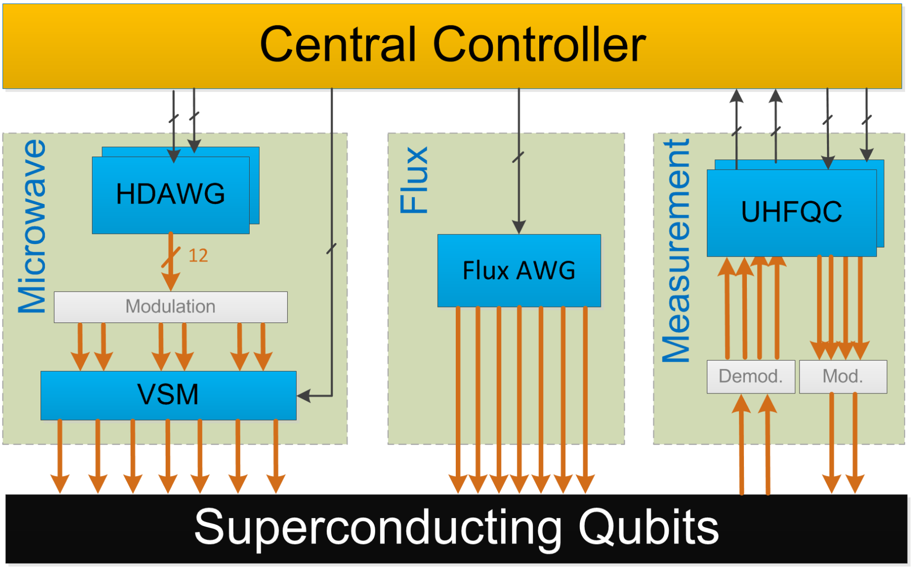


Figure 4.10: Hardware structure implementing the instantiated eQASM for the seven-qubit superconducting quantum processor. Thin (thick) lines represent digital (analog) signals.

feedline using frequency multiplexing techniques [148].

The Central Controller connects to the UHFQCs and HDAWGs via a 32-bit digital interface working at 50 MHz. Since measurement results are sent from the UHFQC to the Central Controller, 16 bits of the connection are sent from the Central Controller to the UHFQC and the other 16 bits the other way around. All operations on UHFQCs and HDAWGs are codeword triggered. The routing of microwave pulses by the VSM is controlled through seven digital signals with a sampling rate of 400 MSa/s.

4.5 Experiment

Since the target seven-qubit quantum chip is still under test at the time of writing, we replaced the quantum chip of this microarchitecture with a two-qubit superconducting quantum processor to validate the eQASM design. The two qubits are interconnected and coupled to a single feedline. A configuration file is used to specify the quantum chip topology with the two qubits renamed as qubit 0 and 2. It is used by the quantum compiler and the assembler. eQASM programs used to perform the experiments as described below are all compiled from OpenQL descriptions with corresponding quantum operation configuration.

We first used eQASM to perform some single-qubit calibration experiments which utilize uncalibrated operations. For example, the Rabi oscillation [124] applies an x-rotation pulse on the qubit after initialization and then measures it. A sequence of fixed-length x-rotation pulses with variable amplitudes are used. Each pulse in the sequence is uploaded to the codeword triggered pulse generation unit of the microarchitecture and configured to be an operation `X_Amp_i` in eQASM. As a result, this experiment calibrated the amplitude of the X gate pulse. Together with other experiments, the fidelity of single-qubit quantum operations used later reached 99.90$\%$ as measured in the following RB experiment. It is worth mentioning that we observed considerable speedup in performing these experiments with the eQASM control paradigm in practice.

eQASM is then configured to include single-qubit gates $\{I, X, Y, X_{90}, Y_{90}, X_{\mathrm{m}90}, Y_{\mathrm{m}90}\}$ and a two-qubit CZ gate for the following experiments. The *AllXY* experiment is typically used to calibrate single-qubit gates. In *AllXY*, pairs of single-qubit gates are chosen from the set $\{I, X, Y, X_{90}, Y_{90}\}$ and applied in such a way that the expected measurement outcomes produce a characteristic staircase pattern that is highly sensitive to gate errors (red line in Fig. 4.3). In the two-qubit *AllXY* experiment, the control pulses are applied on each qubit simultaneously. The sequence is modified to distinguish the qubits on which it is applied: each gate pair in the sequence is repeated on the first qubit while the entire sequence is repeated on the second qubit. The fidelity of qubit to the $|1\rangle$ state can be extracted by averaging the measurement results for each gate pair over N rounds and correcting for readout errors. The eQASM program for one routine of this experiment is shown in Fig. 4.3. Figure 4.11 shows the final measurement result of the entire experiment (blue dots), which matches well with the expectation (red line). This demonstrates that the timing control, SOMQ, and VLIW of eQASM work properly in the experiment.

To evaluate the impact of the timing of operations on the error rate, we use single-qubit randomized benchmarking, a technique that can estimate the average error rate for a set of operations under a very general noise model [77, 78]. In this experiment, a sequence of k random Clifford gates are applied on a qubit initialized in the $|0\rangle$ state. Before measurement, a Clifford is chosen that inverts all preceding operations so that the qubit should end up in the $|0\rangle$ state with survival probability $p(k)$. By performing this experiment for different k and averaging over many randomizations, the Clifford fidelity F_{Cl} can be extracted from the exponential decay. Because each Clifford gate is decomposed

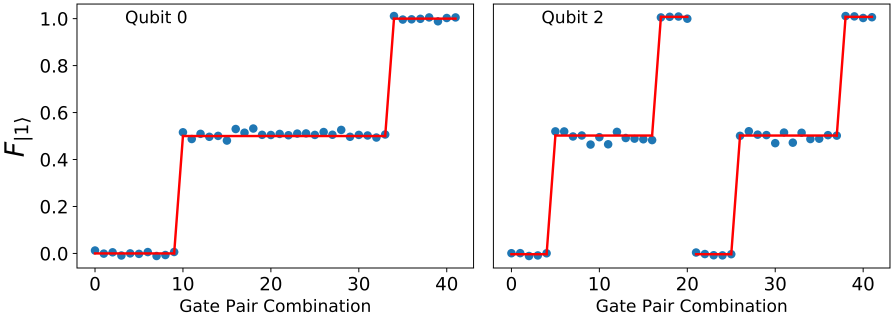


Figure 4.11: Two-qubit *AllXY* result, corrected for readout errors.

into primitive x- and y-rotations the gate count is increased by 1.875 on average. The average error rate per gate, ϵ, is then calculated as $\epsilon = 1 - F_{\mathrm{Cl}}^{1/1.875}$.

Single-qubit randomized benchmarking was performed for different intervals between the starting points of consecutive gates (320, 160, 80, 40, and 20 ns). As shown in Fig. 4.12, the average error per gate decreases by a factor of ~ 7, from 0.71% to 0.10% when decreasing the interval from 320 ns to 20 ns. This demonstrates the significant impact of timing on the fidelity of the final computation result, which substantiates the requirement of explicit specification of timing at QISA level to enable platform-specific optimization and especially scheduling by the compiler.

Fast conditional execution is verified by the active qubit reset experiment with qubit 2 using the code as shown in Fig. 4.4. We find the probability of measuring the qubit in the $|0\rangle$ state after conditionally applying the `C_X` gate to be 82.7%, limited by the readout fidelity. We verified CFC by connecting the Central Controller and the UHFQC. The eQASM program used is shown in Fig. 4.5. The UHFQC is programmed to generate alternative mock measurement results for qubit 0. The alternation between X and Y operations is verified by detecting the output digital signals using an oscilloscope. We also measured the feedback latency of fast conditional execution and CFC, which are ~ 92 ns and ~ 316 ns, respectively. The feedback latency is defined as the time between sending the measurement result into the Central Controller and receiving the digital output based on the feedback from the Central Controller.

As a proof of concept of performing quantum algorithms using eQASM, we executed a two-qubit Grover's search algorithm [16, 72]. The algorithmic fidelity, i.e., correcting for readout infidelity, is found to be 85.6% using quantum tomography with maximum likelihood estimation. This fidelity is limited

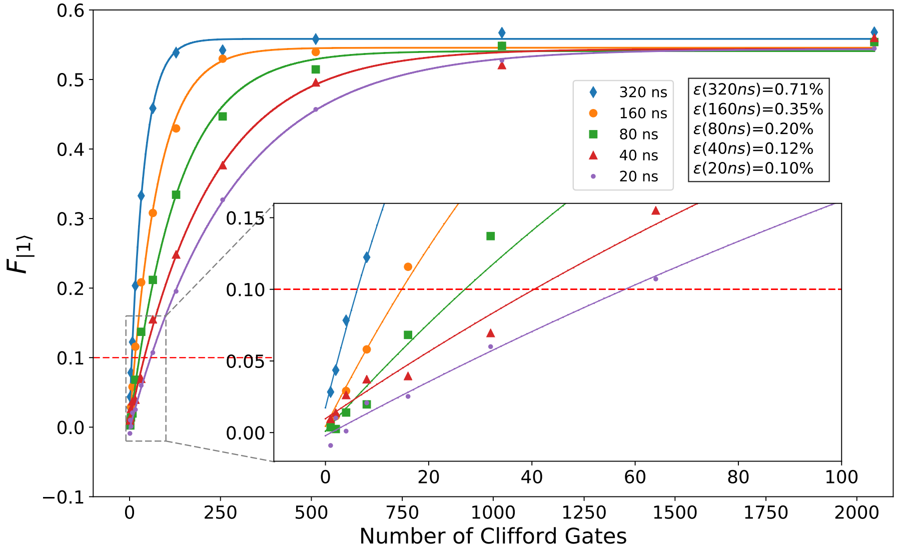


Figure 4.12: Single-qubit randomized benchmarking results for different intervals between gates. Dashed line indicates a 10% error rate for visual reference.

by the CZ gate.

4.6 Conclusion

In this chapter, we have proposed eQASM, a QISA that can be directly executed on a control microarchitecture after instantiation. With runtime feedback, eQASM supports full quantum program flow control at the (micro)architecture level [136, 143]. With efficient timing specification, SOMQ execution, and VLIW architecture, eQASM alleviates the quantum operation issue rate problem, presenting better scalability than QuMIS. Quantum operations in eQASM can be configured at compile time instead of QISA design time, which can support uncalibrated or uncommon operations, leaving ample space for compiler-based optimization. Low-level hardware information mainly appears in the binary of a particular eQASM instantiation, which makes eQASM assembly expressive. It is worth noting that by removing the timing information in the eQASM description, the quantum semantics of the program can be kept and further converted into another executable format targeting another hardware platform.

As validation, eQASM was instantiated into a 32-bit instruction set targeting a seven-qubit superconducting quantum chip, and implemented using a quantum microarchitecture. eQASM was verified by several experiments with this microarchitecture performed on a two-qubit chip. The efficiency improvement observed in using eQASM to control quantum experiments broadens the scope of application of quantum assemblies.

Future work will include performing verifying comprehensive feedback control with qubits and controlling the originally targeted seven-qubit superconducting quantum processor with the implemented microarchitecture. Also, it will be interesting to instantiate eQASM to control other quantum processors, including superconducting quantum processors with a different quantum chip topology, and altogether different quantum hardware, such as spins in quantum dots [31], nitrogen vacancy centers [33].

Note. The content of this chapter is based on the following paper:

X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, 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**, *arXiv:1808.02449*, 2018. [To appear on HPCA'19]

5

Fault-Tolerant Quantum Microarchitecture

5.1 Introduction

Due to the short decoherence time of qubits and the erroneous quantum operations, fault-tolerant quantum computing (FTQC) based on quantum error correction (QEC) is essential to implement large-scale quantum algorithms. The basic idea of QEC is to encode quantum information into a *logical qubit* using a group of *physical qubits* according to some encoding scheme called quantum error correction code (QECC). To achieve fault-tolerance, it requires periodically detecting and (if necessary) correcting possible quantum errors through a highly patterned process called error syndrome measurement (ESM). In addition, quantum operations on such logical qubits should be implemented by a series of physical operations in such a way that individual errors of physical operations will not ruin the information stored in the logical qubits. As a consequence, FTQC dramatically increases the number of required physical qubits and the number of physical operations.

However, quantum control microarchitectures and QISAs proposed by recent research [141, 149] mainly target to control the Noisy Intermediate-Scale Quantum (NISQ) devices with around fifty to hundreds of qubits [57], where quantum error correction is not applied. The QISA required by FTQC can be different to that required by NISQ technology because of the following reasons.

- Quantum algorithms targeting NISQ technology directly operate on individual physical qubits without QEC. In contrast, quantum algorithms operates on logical qubits in FTQC. The microarchitecture should support not only logical operations but also individual physical operations which are required to implement some logical operations such as initialization [39].

- QEC introduces more complex classical computational tasks at runtime, such as quantum error decoding and error tracking using Pauli frame [133], which require the support of new instructions in the QISA and the control microarchitecture.
- The quantum error correction process requires repeated operations on qubits, which significantly increase the number of quantum operations on qubits per unit time, which aggravates the quantum operation issue rate problem [141, 142] confronting the control microarchitecture.

It is an open challenge to develop a scalable and flexible control microarchitecture which can satisfy the requirement of quantum error correction, and fault-tolerant logical operations. This chapter envisions a fault-tolerant microarchitecture, FT_QuMA, for planar surface code with logical operations implemented by lattice surgery [150]. The main contributions of this chapter are the following:

- We introduce the concept *virtual memory* into quantum computing with microarchitectural support, which contributes to a clean compilation model independent of the actual physical addresses of qubits that can vary from device to device;
- We propose a scheme to support quantum error detection and correction at the microarchitecture level, which can enable flexible planar-surface-code-based fault-tolerant logical operations implemented by lattice surgery;
- We propose a hardware mechanism that substantially reduces the codesize of the executable to enable efficient execution of quantum instructions.

This chapter is organized as follows. Section 5.2 introduces the basics of FTQC and planar surface code with logical operations implemented by lattice surgery. The envisioned fault-tolerant quantum control microarchitecture, FT_QuMA, which supports the execution of quantum instructions with error correction, is described in Section 5.3. We discuss the architectural design choices needed for the quantum plane in Section 5.4 and conclude in Section 5.5.

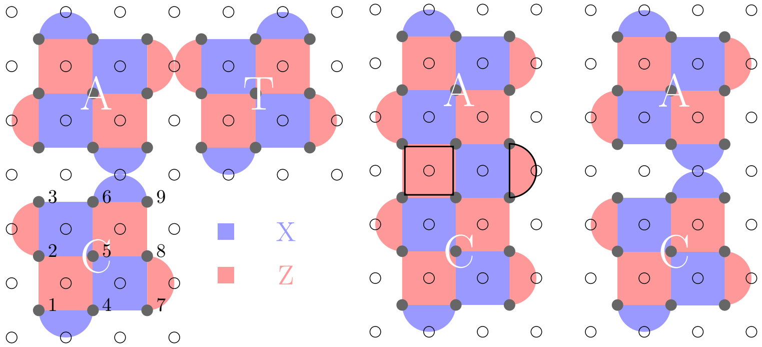
5.2 Fault-Tolerant quantum computing

5.2.1 Quantum Error Correction

Qubits are fragile and quantum operations are erroneous. For example, the state-of-the-art superconducting qubits have a coherence time of around one hundred microseconds [151] and gate fidelity ranging from 99.5 - 99.9% [26]. QEC can protect quantum states against errors by encoding one logical qubit into several physical qubits, called *data qubits*. Logical operations can be implemented by a series of physical operations on the physical qubits in such a way so that individual errors can be detected and corrected to achieve fault-tolerance. Non-destructive ESM is periodically performed with the assistance of ancillary qubits, called *ancilla qubits*, to discretize quantum error and extract the error syndromes. Afterwards, quantum error decoding is applied to find the likely errors leading to the observed syndromes. The capability of a QECC to detect and correct errors is characterized by the *distance* d of this QECC. The distance d is defined as the minimum number of physical operations required to implement a logical operation.

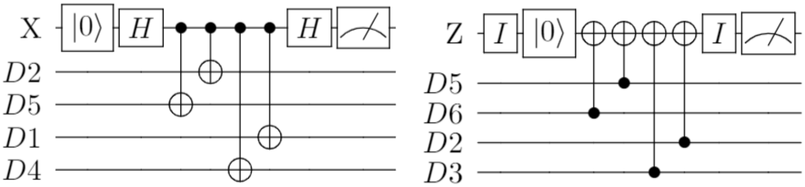
Surface code [152, 153] is a 2D topological stabilizer code of which ESM is realized by measuring low-weight stabilizers, which is applicable on near-term quantum devices with limited connectivity. Moreover, surface code has high tolerance to errors with error threshold $\sim 1\%$, which can be achieved by several quantum technologies such as superconducting qubits. This paper focuses on planar surface code and investigate its implications on a quantum microarchitecture.

The qubit layout of a distance-3 planar surface code is shown in Fig. 5.1a (Lattice ‘C’). It consists of two types of qubits, data qubits (solid circles) for storing computational information, and ancilla qubits (open circles) used to detect errors. Each ancilla qubit is coupled to 2 or 4 data qubits, depending on position of the ancilla qubit in the logical qubit lattice. Each data qubit is connected with two differently colored ancillas, with each corresponding to one type of *stabilizers*: X-stabilizers for detecting Z errors, and Z-stabilizers for detecting X errors. The circuits for performing X- and Z-stabilizer measurement are shown on Fig. 5.2, in which the operations for performing ESM are highly patterned. Decoding algorithms such as minimum weight perfect matching [154–156] will be used to identify the possible errors with high likelihood [121]. Rather than physically performing these corrections which will introduce more errors to the quantum system, errors can be tracked by classical control logic using the technique called *Pauli frame* [80] that can be implemented in the microarchitecture [133].



(a) **(b)** **(c)**

Figure 5.1: (a) The qubit layout for performing a CNOT gate between two logical qubits. Each logical qubit is encoded by a distance-3 rotated planar surface code where data qubits are on the vertices (solid circles) and ancilla qubits are on the plaquettes (open circles). The blue (red) squares and semi-circles represent stabilizers of the form $X(Z)^{\otimes 4}$ and $X(Z)^{\otimes 2}$, respectively. 'C' is the control qubit, 'T' is the target qubit, and 'A' is the ancilla. The joint M_{ZZ} on qubit 'A' and 'C' is realized by first performing ESM on the entire lattice in (b) for a merge and then performing ESM separately on the two lattices in (c) for splitting.



(a) **(b)**

Figure 5.2: The ESM circuits (a) and (b) for measuring X- and Z-stabilizers ($X_{D5,D2,D4,D1}$ and $Z_{D6,D3,D5,D2}$).

5.2.2 Fault-tolerant Logical Operations

For surface codes, an implementable universal set of logical operations include the preparation, measurement, Pauli, CNOT, H, S and T gates. In planar surface code, logical Pauli gates, preparation and measurement can be implemented transversally [39], i.e. applying bitwise operations to data qubits in the code and then performing $0-d$ rounds of ESM to decode errors. For example, the logical X and Z on a distance-3 surface code (e.g., lattice 'C' in Fig. 5.1a) can be realized by performing $X_{D1}X_{D2}X_{D3}$ and $Z_{D1}Z_{D4}Z_{D7}$ followed by 3 rounds of ESM, respectively.

In order to be able to perform logical CNOT gates in a 2D layout with only nearest-neighbour (NN) interactions, [157] proposes a measurement-based procedure which can be implemented by the circuit shown in Fig. 5.3 [158]. The qubit layout for performing such a logical CNOT gate on distance-3 planar surface code is shown in Fig. 5.1. The joint measurement M_{XX} or M_{ZZ} is realized by first performing a merge operation and then a split operation using a technique called lattice surgery [150, 159]. For example, the merge operation for M_{ZZ} in Fig. 5.3 is realized by performing d rounds of ESM on the merged lattice in Fig. 5.1b. This means the two lattices 'A' and 'C' are integrated into one single lattice. Similarly, the split operation is implemented by performing d rounds of ESM individually on each lattice 'A' and 'C' in Fig. 5.1c. The splitting procedure divides the merged lattice back into two lattices. In addition, one needs to read out the result of each joint measurement by multiplying the outcomes of all new stabilizers during its merge, which will be used for further logical Pauli corrections (Fig. 5.3). For instance, the measurement result of M_{ZZ} is the multiplication of the outcomes of the two blocked Z stabilizers in Figure 5.1b. In total, $\sim 3d$ rounds of ESM are performed to realize a lattice-surgery-based CNOT gate.

It is worth to mention that the decoding for lattice-surgery-based operations may differ from the decoding for the standard error correction cycles (we refer interested readers to [160] for more technical details). For the merge operation in Figure 5.1b, the decoding for detecting Z errors is based on the measurement outcomes of all the X-stabilizers on the entire merged lattice, but the decoding for detecting X errors is only based on the measurement outcomes of all the Z-stabilizers on the original two separate lattices.

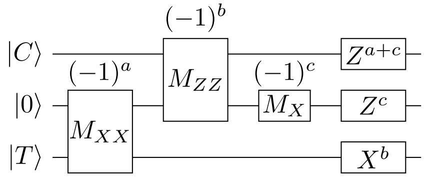


Figure 5.3: The circuit to realize a measurement-based CNOT gate.

The logical H gates on planar surface code in principle can be implemented transversally. Afterwards, the X- and Z-stabilizers will be exchanged, that is, the lattice is rotated by 90 degrees. One needs to keep track of the orientation of each lattice for generating the correct ESM circuits. In addition, a rotation procedure is required if a logical CNOT will be performed between two lattices which are not in the same orientation (The 'C' and 'T' qubits have the same lattice orientation in Fig. 5.1a). Several papers have proposed lattice-surgery-based procedures to rotate a planar surface code [121, 159]. However, different approaches have different latency and use different types of lattices (shapes and sizes), it is not clear which one has better performance. Investigating rotation procedures is beyond the scope of this paper, we will ignore this operation in this work. The last element for implementing a fault-tolerant universal gate set on planar surface code is magic state preparation using lattice surgery [159], which is used to realize the logical S and T gates [153].

Summary: The transversal logical preparation, measurement, Pauli, H gates, and magic state preparation on planar surface code are implemented by first performing some physical operations and then performing $0 - d$ rounds of ESM on the lattice of objective qubit. The lattice-surgery-based logical CNOT and rotation procedure are both realized by consecutively performing d rounds of ESM on several different lattices. It means the most frequent operation unit of surface-code-based quantum computing is a full round of ESM on a specific lattice. In the following sections, we will discuss how to efficiently support the execution of ESM and fault-tolerant operations in the control microarchitecture.

5.3 Fault-Tolerant Control Microarchitecture

Figure 5.4 gives an overview of the proposed fault-tolerant control microarchitecture FT_QuMA. It is updated from QuMA_v2 with the dashed blocks highlighting the modules and mechanisms introduced to support large-scale FTQC. The QECC chosen is planar surface code with logical operations implemented with lattice surgery. The same as QuMA_v2, instructions accepted by FT_QuMA describe quantum operations at the physical level. Logical operations should be translated by the compiler into a fault-tolerant implementation consisting of instructions describing physical operations.

To support large-scale quantum computing, FT_QuMA adopts *virtual memory* as used by classical computers to enable a clear compilation model. FT_QuMA supports fault-tolerant quantum computing by providing essential features for flexible logical operation description at the physical level, efficient ESM circuit generation, QED, and Pauli frame, etc. Note, the user can turn off all modules related to QEC, and FT_QuMA can work properly as well, but at the physical level without fault-tolerance support. Modules shared by QuMA_v2 and FT_QuMA work in the same way unless otherwise specified.

5.3.1 Qubit Addressing

Virtual Address and Physical Address

To enable a simple compilation model for large-scale fault-tolerant quantum computing, two kinds of addresses are used: the virtual address and the physical address. It is assumed that the quantum compiler works with qubits on a virtual 2D array with NN interaction among qubits. The virtual address has the format $q_v = (i, x, y)$, where i is the index address of the qubit and (x, y) is the Cartesian coordinates of the qubit on the virtual array. The index address can be calculated from the Cartesian coordinates and vice versa, or a look-up table can be used to store the mapping between each other. The Cartesian coordinates are included as part of the address because they are important in determining the qubit type (data qubit, X ancilla, or Z ancilla) and the operations required by QEC, which will be discussed in Section 5.3.2.

As shown in Fig. 5.5, the virtual qubit array (red frame) is mapped to a lattice of the same size on the physical qubit array during the initialization before execution. Every qubit also gets a physical qubit address $q_p = (\hat{i}, \hat{x}, \hat{y})$ at this step. Assuming the mapping process keeps the orientation of the axes of the qubit plane, then the mapping can be determined by, e.g., recording the physical coordinates $q_p = (\hat{x_0}, \hat{y_0})$ of the virtual qubit (0, 0, 0). The Cartesian coordinates between the virtual address and the physical address can be translated using the relationship:

$$(\hat{x}, \hat{y}) = (x, y) + (\hat{x_0}, \hat{y_0})$$

The physical address of qubits may vary when executing the program on different platforms. By using the virtual address, the compilation process can be independent of the actual physical qubit address, which contributes to a cleaner compilation model.

Microarchitectural Support

The quantum instructions input to FT_QuMA are physical instructions with virtual addresses. As shown in Fig. 5.4, modules in the virtual address do-

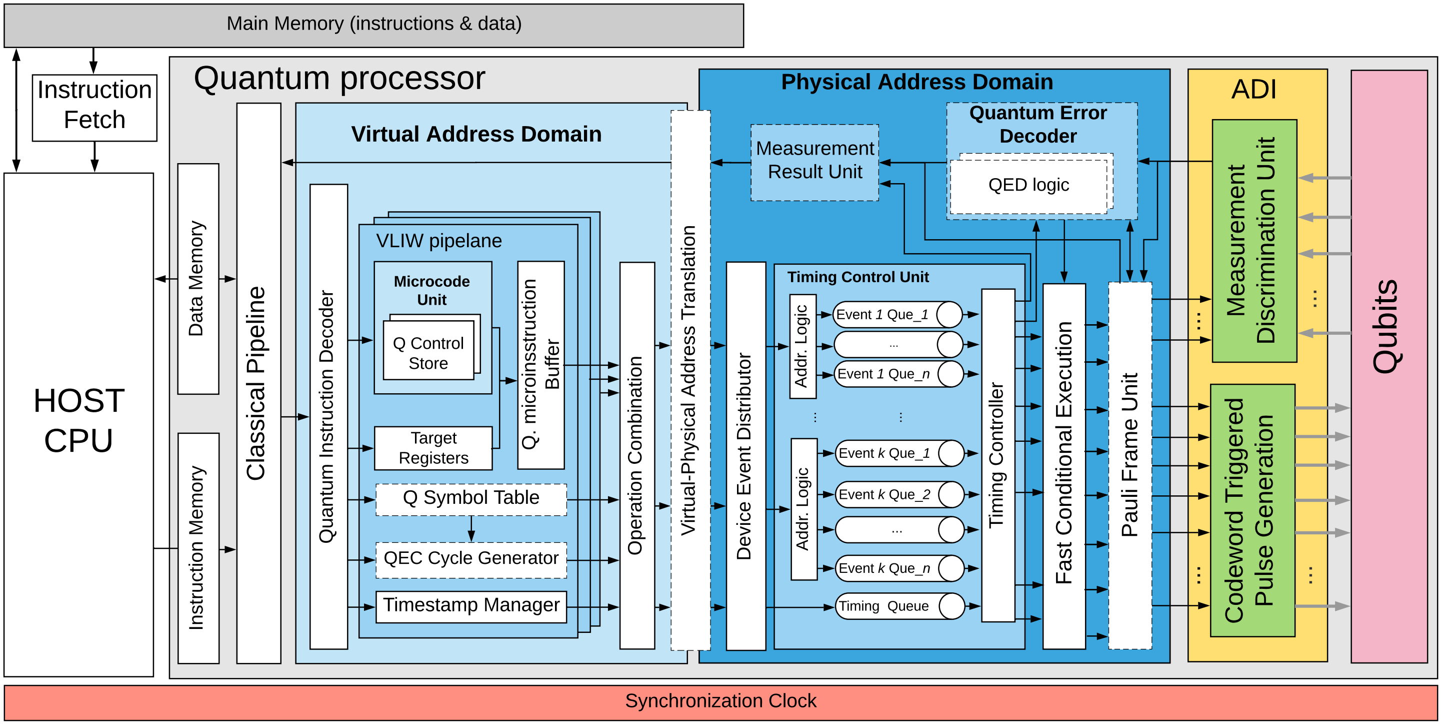


Figure 5.4: Overview of the Envisioned Fault-Tolerant Quantum Control Microarchitecture, FT_QuMA.

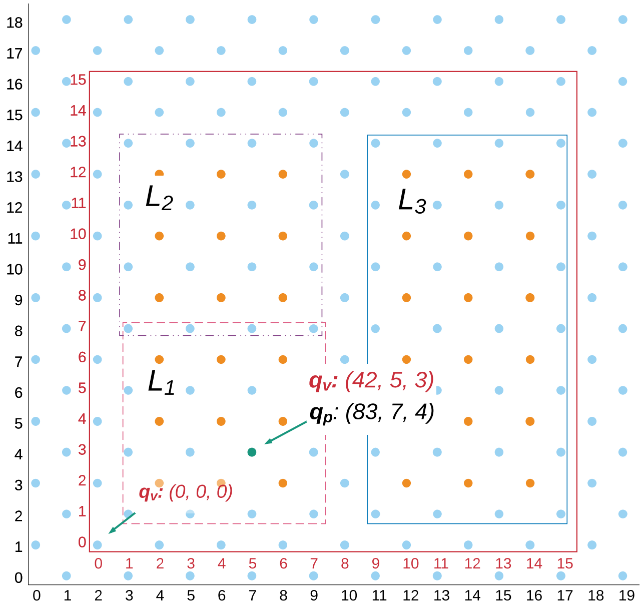


Figure 5.5: Virtual qubits mapped on physical qubits. Red is for the virtual address space and black for the physical address space.

main works in the same way as explained in Chapter 4.4. They are relatively technology-independent and work with the virtual address. The virtual-physical address translation module translates the virtual address into the physical address according to the mapping in the initialization. In the physical address domain, the modules are mostly technology-dependent and work with the physical address. The measurement discrimination unit returns measurement results associated with physical address, which will be translated by the virtual-physical address translation module into virtual address and later sent to the classical pipeline for further process. In this way, the virtual address domain is clearly separated from the physical address domain.

Note, the virtual address and physical address are only used by the physical qubits, and each logical qubit uses only one but unique logical address throughout the microarchitecture, which will be explained in the next subsection.

5.3.2 Fault-Tolerant Logical Operations

As explained in Section 5.2.2, logical operations can be either implemented by applying transversal physical quantum operations on the qubits or by lattice surgery. In both operations, the keys to achieve fault-tolerance is to 1) perform ESM on the active stabilizers, and 2) decode and correct the errors at runtime. Since lattice surgery needs to turn on and off some stabilizers at different QEC cycles, which should be supported by the control microarchitecture. Besides, the microarchitecture should be able to decode quantum errors at real-time as well, which is required by QEC.

Targeting these goals, FT_QuMA maintains a qubit symbol table (Q symbol table), which keeps track of the status of all logical qubits which is used to determine active stabilizers. The qubit symbol table is introduced in Section 5.3.2. Section 5.3.2 explains the working principle of the QEC cycle generator to efficiently generate quantum operations used for ESM. The quantum error decoder module of FT_QuMA is used to decode quantum errors at runtime, which is explained in Section 5.3.3. FT_QuMA also contains a logical measurement unit used to deduce the measurement result of logical qubits, which is explained in Section 5.3.4. The Pauli frame unit tracks quantum errors of data qubits at runtime, which is explained in 5.3.5.

Qubit Symbol Table

The microarchitecture should be able to know what logical qubits are alive, where they are, and the type of each physical qubit. The qubit symbol table tracks which logical qubits are *alive (info1)*. As shown in Fig. 5.5, there are three alive logical qubits, L_1, L_2, and L_3.

To generate the instructions for ESM, the following configuration of logical qubits should be updated at runtime:

- Since a logical qubit can vary its size during lattice surgery, the *size* $(d_1 \times d_2)$ *(info2)* of each logical qubit should be recorded, where d_1 and d_2 are the number of **data qubits** of this logical qubit along x- and y-axis, respectively. As shown in Fig. 5.5, logical qubit L_1 has a size of 3×3 while logical qubit L_3 has a size of 3×6.
- The *location* of each logical qubit *(info3)* is essential to determine which physical qubits are used to implement this logical qubit. The location of each logical qubit can be determined by recording the virtual coordinates $(x_{L,0}, y_{L,0})$ of the **data qubit** at the bottom left corner of this logical qubit. For example, the location of logical qubit L_1 is $(2, 2)$.
- A logical qubit with the same data qubits can use different ancilla qubits for different purposes in lattice surgery. Take the distance-3 surface code as an example and as shown in Fig. 5.6, there are in total four different flavours

for the same qubit, which can be distinguished using two flags: the *base ancilla type* and the *chirality (info4)*. *Info4* is required to determine which physical qubits are used as ancilla and what types they are.

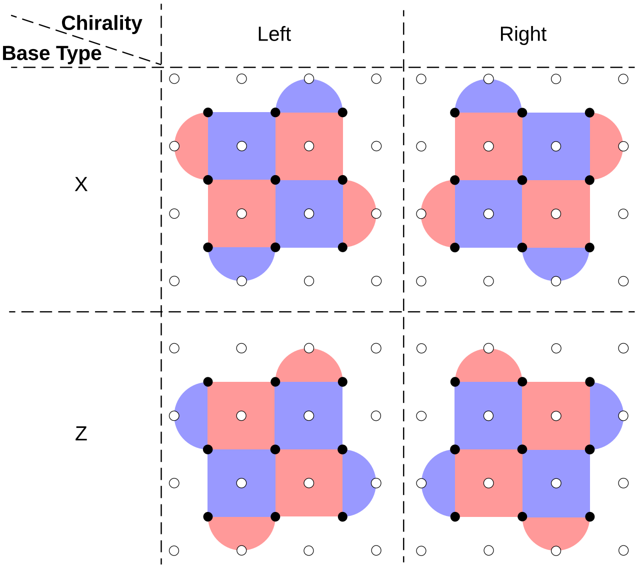


Figure 5.6: Four flavours of a distance-3 surface code logical qubit using the same data qubits. It is assumed that redundant ancilla qubits are not used. The chirality of the logical qubit is left (right) when the physical qubit at $(x_{L,0}+1, y_{L,0}-1)$ $((x_{L,0}+3, y_{L,0}-1))$ is used as an ancilla qubit, where $(x_{L,0}+1, y_{L,0}-1)$ is the location of the logical qubit. The base ancilla type is X (Z) when the ancilla qubit at $(x_{L,0}+1, y_{L,0}-1)$ or $(x_{L,0}+1, y_{L,0}-1)$ is an X (Z) ancilla.

Note, some logical operation implementation may rotate the lattice (Section 5.2.2), which procedure would require more information. Since it is not yet clear if the lattice rotation is essential or which rotation scheme should be used, the microarchitectural support for the lattice rotation will be included in the future work. The qubit symbol table should contain all *info1* to *info4*, with an example qubit symbol table shown in Table 5.1. When necessary, another table could be provided to inversely map the physical or virtual qubit address to logical qubit address.

Based on the qubit symbol table, the hardware can deduct which physical qubits are used by each logical qubit, and the type of each physical qubit. Furthermore, all operations required by ESM can be determined according to the ESM circuits as shown in Fig. 5.2.

Table 5.1: An example of the qubit symbol table, with the content recording the status of logical qubits in Fig. 5.5.

valid	Logical QID	Location	Size	Base Ancilla Type	Chirality
1	1	(2, 2)	(3, 3)	Z	Left
0	2	(2, 8)	(3, 3)	X	Right
1	3	(10, 2)	(3, 6)	Z	Right

QEC Cycle Generator

As explained in Section 5.2, most of the physical operations are used to perform ESM, which is required by QEC and fault-tolerant logical operations. We perform a simple investigation in the number of physical operations for ESM, followed by the introduction to the QEC cycle generator.

In the following estimation, it is assumed that there are N_{L} logical qubits encoded in planar surface code, each with a distance of d. A logical qubit with a distance of d consists of $d \times d$ data qubits and $d^2 - 1$ X- or Z-stabilizers. As d increase, the weight-2 stabilizers only occupies around $2/d$ of all stabilizers, and most stabilizers are weight-4 stabilizers, with the corresponding ESM circuit realized by applying eight or six physical operations (Fig. 5.2). During lattice surgery, some stabilizers on the edge of logical qubits can be turned on or off, which may slight increase or decrease the number of physical operations required by ESM. For simplicity, it is assumed the total number of physical operations for ESM is not affected by lattice surgery. ESM should be performed with the active stabilizers of logical qubits repeatedly at every cycle, no matter a logical operation is being performed or not. Hence, it suffices for us to roughly estimate the physical operations required to perform one round of ESM:

$$N_{\mathrm{ins}}^{\mathrm{c}} \approx 7 \times N_{\mathrm{L}} \times d^2. \tag{5.1}$$

$N_{\mathrm{ins}}^{\mathrm{c}}$ can explode as the d and/or N_{L} increase. Though the quantum compiler can generate the instructions for these physical operations, it cannot scale up because the quantum operation issue rate is a bottleneck for the QuMA microarchitecture [141, 142]. In other words, the required instructions are too many to be fetched from the instruction memory for processing due to a limited instruction bandwidth. It highly requires the microarchitecture to generate physical operations for ESM to alleviate the quantum operation issue rate problem.

FT_QuMA introduces a special instruction, `Gen_ESM L_i`, which triggers the hardware to generate all operations to perform ESM over the logical qubit L_i.

This method is feasible based on the following observation. Given the location and size of a logical qubit in the qubit symbol table, the data qubits of this logical qubit can be determined. Based on the base ancilla type and chirality, the ancilla qubits and their type (hence the stabilizers) can be determined. It is then possible for the microarchitecture to automatically generate the quantum operations with expected timing to implement the ESM circuit for each stabilizer according to Fig. 5.2.

During the fault-tolerant implementation of the original quantum algorithm, the compiler generates a `Gen_ESM L_i` instruction for every round of ESM on every alive logical qubit in the binary. Once `Gen_ESM L_i` instruction is fetched during execution, this instructions triggers the QEC cycle generator after instruction decoding. The QEC cycle generator reads the location, size, base ancilla type, and the chirality of the target logical qubit L_i, based on which the physical operations with precise timing for ESM can be generated and sent to the timing control unit awaiting execution. By using the proposed method, the required number of instructions ($N_{\mathrm{ins}}^{\mathrm{h}}$) used for one round of ESM is substantially reduced to

$$N_{\mathrm{ins}}^{\mathrm{h}} = N_{\mathrm{L}}. \tag{5.2}$$

The comparison between these two methods are shown in Fig. 5.7. When $d = 30$, $N_{\mathrm{ins}}^{\mathrm{h}}$ is four orders of magnitude less than $N_{\mathrm{ins}}^{\mathrm{c}}$.

It is worth mentioning, Tannu *et al.* [110] also proposed a similar idea independently, which adopts a microcode-based method to generate the instructions for ESM in the hardware. The microcode-based method can support not only planar surface code but also defect-based surface code.

5.3.3 Quantum Error Decoding

After a particular number of rounds of ESM have been applied on a logical qubit, the quantum error decoder should performs decoding algorithm to infer the possible errors based on the syndromes, i.e, the stabilizer measurement results. Although the Pauli frame can keep track of the identified errors, errors should be still identified as soon as possible. The qubit measurement results stored in the measurement result unit should contain the correct values after error correction to support runtime feedback. For example, in an implementation of the Shor's algorithm [128], some X gates are conditioned on the results of previous measurements to reduce the number of required qubits. Therefore, the error decoding process is required to be fast enough so that the computation

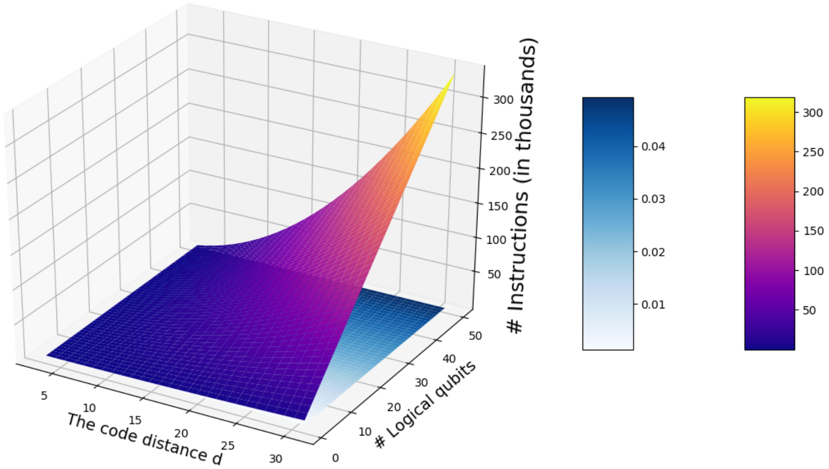


Figure 5.7: The number of instructions for performing one round of ESM on all alive logical qubits through compiler generation (top) and hardware generation (bottom).

will not be delayed, leading to more accumulated errors or even a failure of the computation. Ideally, the quantum error decoder decodes the error syndromes generated by d rounds of ESM using less time than the duration of these ESM circuits. To achieve such high speed, the decoding unit can decode errors based on, e.g., Blossom algorithm [82] or neural network [161, 162].

Microarchitectural Support

FT_QuMA introduces another special instruction, `Decode_ESM L_i` to trigger the QED unit to perform decoding over the error syndromes of the last rounds of ESM for the logical qubit L_i. Once the `Decode_ESM L_i` instruction is fetched during execution, the QEC cycle generator reads the entry of the logical qubit L_i in the qubit symbol table, which information is finally sent to trigger the QED unit at the expected timing point by the timing control unit. The quantum error decoding unit contains multiple instances of the decoding unit. Each decoding instance is triggered by one `Decode_ESM L_i` instruction with the logical qubit information. Based on this information, the decoding instance can determine which physical qubits are used as X or Z ancilla qubits, and detect the errors that may happen in that logical qubit. Note, a distributed implementation of the QED unit can be easily parallelized which presents better scalability.

5.3.4 Measurement Result Unit

There are two kinds of logical measurements used by planar surface code based on lattice surgery: the measurement of a single logical qubit and the joint measurement of two logical qubits. The measurement of a single logical qubit is realized by first measuring all the data qubits and later classically checking the parity of these data qubits to perform error decoding. After removing the detected errors, the logical measurement result is calculated by multiplying the measurement outcomes of all data qubits of this logical qubit. The joint measurement of two logical qubits is realized by a merge operation followed by a split operation using lattice surgery (Section 5.2.2). After error correction, the joint measurement outcome is calculated by multiplying the outcomes of the new measured stabilizers.

Microarchitectural Support

The `Desc_Result L_i` instruction is used to trigger the measurement result discrimination for a single logical qubit L_i. After decoding, address translation and awaiting execution, every `Desc_Result L_i` instruction fetched from the instruction memory finally triggers the measurement result unit at a precise timing to detect and remove errors, and calculate the logical measurement result for logical qubit L_i. Note, the entry of this logical qubit in the qubit symbol table is also sent to the measurement result unit.

The measurement result can be read by the classical pipeline for feedback, e.g., directing the following program flow. The result can also be sent to the host CPU via the data memory. Note that, the logical measurement unit also stores the latest measurement result of each physical qubit to enable feedback at the physical level.

5.3.5 Pauli Frame

The Pauli frame mechanism [80] allows tracking Pauli errors of qubits in the classical logic without physically correcting them.

As described in [133], the Pauli frame unit only need to work for the data qubit. To distinguish data qubits and ancilla qubits, the Pauli frame unit maintains a qubit symbol table without virtual address in the deterministic-timing domain, which records the real-time logical qubit information. Whenever the qubit symbol table in the virtual address domain gets updated, then same update information is send to the timing control unit, and later updates the Pauli frame unit at a precise timing.

The Pauli frame unit stores a record for every data qubit. The record is the Pauli operators (X, Z, or X and Z) that should have been applied on the corresponding qubit, which can come from the quantum program or quantum error correction. The Pauli frame unit contains an arbiter, which directs the quantum operations released by the fast conditional execution unit.

All quantum operations on the ancilla qubits are ignored by the arbiter, which directly triggers the codeword triggered pulse generation unit (CTPG) to apply corresponding pulses on the qubit. Quantum operations on data qubits are processed as follows based on the type of this operation:

- Pauli gate: The arbiter redirect this operation to update the Pauli record of the target qubit. No operation is sent to the quantum-classical interface (QCI).
- Clifford gate but not Pauli: The arbiter sends this operation to the QCI. In addition, the Pauli record of the target qubit should be updated correspondingly.
- Non-Clifford gate: Since Pauli gates does not commute with non-Clifford gates, two steps are required. First, The Pauli record(s) of the target qubit(s) are flushed, i.e., the Pauli operators in the record are released to the QCI to apply corresponding pulses on the qubit(s). Second, the non-Clifford gate is released to the QCI and applied on the target qubit(s). To ensure the timing of the following operations is not interrupt, FT_QuMA requires reserving an extra time slot before every non-Clifford gate to allocate up to two single-qubit gates (X and Z).
- Measurement: The arbiter sends this operation to the QCI to trigger physical qubit measurements. After the measurement finishes, the measurement result generated by the measurement discrimination unit (MDU) are inverted if there is an X operation in the Pauli record for the target qubit. The final result is then sent to the logical measurement for further processing. Finally, the Pauli record for the target qubit is removed.

5.4 QUBE: The qubit plane

Finally, we discuss the way the quantum plane can be organized and what infrastructure is necessary for fast execution and error correction. A first point that needs to be addressed is whether or not a von-Neumann-like architec-

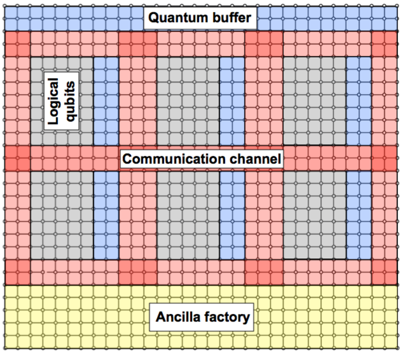


Figure 5.8: Example QUBE architecture.

ture should be implemented in the quantum plane as advanced by many papers [38, 45–47, 54, 99, 105, 106]. Just like with classical von Neumann architectures, the quantum plane is then split into specialized regions for processing and communication and others to store quantum states. There are two main arguments to not go in that direction: first, the computational paradigm of quantum logic can be seen as the dual of the classical one. The qubit states represent the information as it has been computed up to now and all future operations are applied on these qubits. In principle, there is no movement required of the qubit states to an ALU like component as the quantum gates are directly applied on the qubits. So the logic is streaming through the qubits rather than the opposite. A second reason is that by introducing a von Neumann architecture, we also introduce may the parallelization challenges, such as the memory wall issue, that have proven to be very difficult to solve for conventional architectures.

However, as shown in Figure 5.8, it may still be useful to create specialized regions, for instance to transport qubit states to different parts of the quantum plane. Ancilla factories may also be necessary to create, e.g. EPR pair used for teleporting states [104] and special ancilla states used for implementing fault-tolerant T and S gates [48].

What functional specialization of the qubit is necessary and how rigid or flexible that should be is still an open issue. The two extreme views are that the qubit plane can be seen as completely undefined for which an ad-hoc infrastructure will be generated, or that the architecture is completely pre-defined as proposed in [54, 106]. In this sense, it is important to investigate what the trade-offs are for both choices given different benchmarks or applications.

5.5 Conclusion

In this chapter, we envisioned a control microarchitecture that can support FTQC based on planar surface code with logical operations implemented by lattice surgery. We ported the concept virtual memory from classical computing to the quantum computing to provide a clean compilation model, which is independent of the actual physical addresses of qubits that can vary from device to device. With the support of ESM circuit generation and QED at runtime, the codesize of the fault-tolerant implementation of the quantum program can be significantly reduced.

Future work will involve the development of the hardware blocks in FT_QuMA that support the virtual memory and fault-tolerant quantum computing, and verify FT_QuMA on our developing full-stack simulator, called quantum virtual machine, which can simulate not only the qubit state evolution but also the execution of quantum instructions on the microarchitecture.

Note. The content of this chapter is based on the following papers:

X. Fu, L. Lao, C. G. Almudever, and K. Bertels, **A Control Microarchitecture for Fault-Tolerant Quantum Computing**. [In preparation.]

L. Riesebos, X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels. **Pauli Frames for Quantum Computer Architectures**, *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, ACM, 2017, p. 76.

6

Quantum (Micro)architecture Simulator

Both quantum software and electronic devices in our lab used to perform quantum algorithms or experiments are far from mature and are still evolving rapidly. While developing the quantum software and electronic devices, we are confronted with the challenges in the verification of the quantum software stack, and the design, development, and verification of the microarchitecture. The challenges ask for a quantum microarchitecture simulator to automate these processes.

6.1 Challenges in QuMA Development

QuTech Control Box (CBox) can control one or two qubits with the quantum control microarchitecture QuMA, which can execute QuMIS instructions. Since the number of QuMIS instructions is limited, which are low-level and tightly bound to the hardware, the design of QuMA is relatively simple. To address the challenges in controlling more qubits and the inherent quantum operation issue rate problem of QuMA (as discussed in Section 4.1), we designed an executable QISA, eQASM, which is instantiated to a 32-bit instruction set to control seven qubits and implemented by QuMA_v2.

Compared to QuMA, QuMA_v2 outputs a larger number of signals to orchestrate the behavior of more analog devices. The VLIW architecture, SOMQ execution, and microwave selective broadcasting, all ask for more complex addressing and dynamic checking logic 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 the higher complexity of control microarchitecture, which pose a big challenge in its design, development, and verification.

These challenges drove us to develop a new method that can boost the design, development, and verification of future quantum control microarchitectures. Inspired by simulators for designing classical processors such as gem5 [163], a cycle-accurate (micro)architecture simulator would also be required for the quantum control microarchitecture.

This chapter is organized as follows. In Section 6.2, We introduce the quantum microarchitecture development flow, analyze the drawbacks of this flow, and show potential improvements over it. Section 6.3 introduces the design of QuMAsim. The three potential applications of QuMAsim and how QuMAsim supports them are shown in Section 6.4.

6.2 Quantum Microarchitecture Development Flow

This section starts by briefly introducing the microarchitecture development procedure we adopt. After showing the drawbacks of this method, potential improvements are introduced.

6.2.1 QuMA Development Flow

The quantum control microarchitecture is used to execute the instructions belonging to a given QISA and apply operations on the qubits performing the described computation steps. While the underlying analog devices and quantum chip are evolving, the quantum control microarchitecture would require modification accordingly. To fit the requirement for fast modification and deployment, the quantum control microarchitecture is implemented based on Field Programmable Gate Array (FPGA), instead of the application-specific integrated circuit (ASIC).

The development of QuMA targeting a particular eQASM instantiation is similar to the development of classical processors targeting a given classical ISA. It contains the following steps:

1. **eQASM Instantiation**: Instantiate eQASM into a concrete QISA targeting the given quantum chip topology, and configure the assembler for this concrete QISA;
2. **Interface Requirement Specification**: Specify the electronic devices used to control the quantum chip, and the interface (digital signals) requirement between the physical execution layer and the analog-digital

interface;

3. **Design**: Design the microarchitecture which accepts this concrete QISA and outputs required signals to control the analog-digital interface;
4. **Development**: Describe the designed microarchitecture at *register-transfer level* (RTL) using a hardware description language (HDL), such as VHDL;
5. **Verification**: Verify the correctness of the HDL description by running a set of eQASM programs in simulation using tools like Mentor Graphics QuestaSim [164]. Usually, a testbench is required. The correctness is done by checking the output signals of the microarchitecture for the given eQASM program.

6.2.2 Drawbacks of the QuMA Development Flow

Without the assistance of a quantum (micro)architecture simulator, the design, development, and verification of QuMA can be inefficient and error-prone because:

- (**Design**) The design space exploration (DSE) can be limited by time when only a diagram-based method is available, and the relationship among modules might not be fully respected;
- (**Development**) it is usually time-consuming to write HDL to describe the entire system directly; and
- (**Verification**) it requires the hardware programmer to check the correctness of multiple signals in simulation for every eQASM program in the benchmark.

6.2.3 Potential Improvements

Based on the experience with the design of classical processors, all the three steps (DSE, development, and verification) can be boosted via automation.

The DSE can be performed via a configurable simulator for the quantum control microarchitecture, like simulators for classical processors such as gem5 [163]. QuMA differs from normal classical processors by separating the non-deterministic timing domain and the deterministic timing domain. This difference makes simulators for classical processors cannot directly simulate the behavior of QuMA. A dedicated simulator for QuMA is required to perform design space exploration.

The HDL implementation can be boosted using high-level synthesis [165]. But the HDL code generated by high-level synthesis usually suffers from limited performance. Since the implemented QuMA is supposed to run at a frequency greater than 100 MHz on nowadays FPGA to satisfy the quantum operation throughput requirement, the quality of the HDL code must be high.

The verification of the HDL implementation can be automated using the universal verification methodology (UVM) [166], which would require a simulator to reference values for the checking points for each input program.

6.3 Design of QuMAsim

A simulator that describes the target system at a relatively low level (preferably at the register-transfer level) can help the automation of the DSE, development, and verification of QuMA. We call this simulator QuMAsim .

As shown in Figure 4.10, the analog part of QuMA uses commercial products, and our research pays more attention to the digital part of QuMA. Hence, QuMAsim mostly simulates the behavior of the digital part of QuMA. Qubit state evolution can be simulated via another simulator, like QX [132] or QuantumSim [68].

In this section, we clarify the goal of QuMAsim and then briefly introduce the implementation of QuMAsim.

6.3.1 Requirement of QuMAsim

Before developing QuMAsim, the simulation of the microarchitecture (using QPDO or QUAPA) is independent of the microarchitecture implementation.

We have developed an architecture simulator based on Python, Quantum Platform Development framewOrk (QPDO) [133]. QPDO has a layered structure with each layer implementing a particular functionality, and different layers can be combined to provide different control stack, e.g., with and without Pauli frame to control a distance-3 logical qubit based on the planar surface code. In the following development, L. Riesebos upgraded QPDO to the QUantum μArchitecture Performance Analyzer (QUAPA) based on SystemC. 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 regard to the internal structure of each module. Both QPDO and QUAPA take as input a stream of quantum operations in a format similar to cQASM [135] instead of binary instructions considering simulation flexibility. Neither QPDO nor QUAPA is synthesizable. There leaves a wide gap between the microarchitecture simulation and the HDL implementation.

QUAPA can perform DSE for QuMA implementation at a relatively high level (transaction level). As a result, QUAPA generates a set of overall configurations of QuMA satisfying the working requirement of given conditions. For example and as shown in Section 4.4, what VLIW width and frequency of the non-deterministic timing domain are required to achieve required quantum operation issue rate for potential applications on given quantum chip topology. This high-level information can guide the eQASM instantiation process and help set the hardware implementation target.

QuMAsim contributes to the DSE in the second step. Given the configuration of the entire architecture, QuMAsim can describe the concrete microarchitecture at a relatively low level. To enable easy implementation and feasibility verification of microarchitectural ideas, QuMAsim should be capable of modeling the internal structure of modules of interest at the register-transfer level to pursue accuracy while describing the function of modules of less interest at a relatively high level to pursue fast implementation. (**requirement 1**).

QuMAsim should describe the microarchitecture at a low level (preferably RTL). The code of QuMAsim by itself should be synthesizable, and the generated HDL code is of high quality, which can be fine-tuned manually to achieve the required performance (**requirement 2**).

To enable verification over the HDL implementation, QuMAsim should accept the same input as the microarchitecture, i.e., binary instructions of the instantiated eQASM (**requirement 3**). The checkpoints used for the verification should cover related features of QuMA. In the non-deterministic timing domain, QuMAsim should correctly reflect the architectural state update. In the deterministic timing domain, QuMAsim should reflect not only the value change but also the moment of the value change for each output signal since timing is crucial (**requirement 4**).

6.3.2 QuMAsim Implementation

Overview

To satisfy the requirements as discussed in Section 6.3.1, we choose SystemC to implement QuMAsim because:

- Compared to C/C++, SystemC supports a simpler simulator description for the target system;

- SystemC can describe the target system at various abstract levels, which can support fast implementation as well as precise modeling;
- SystemC can describe the target system at the register-transfer level which can help better HDL code generation;
- Translation from SystemC description to HDL description is well supported by commercial products, such as Mentor Graphics Catapult [167].

Modeling

As an initial step, QuMAsim models the digital part of QuMA_v2 as shown in Figure 4.9 at the register-transfer level.

To enable flexible DSE, we try to decouple different parts of the system from each other as much as possible based on the functionality of each part. The classical pipeline takes charge of fetching instructions, executing classical instructions, and issuing quantum instructions to the quantum pipeline. The classical pipeline issue quantum instructions (quantum waiting instructions and quantum bundles) to the quantum pipeline as output, and reads the quantum measurement result register as input. The classical pipeline is modeled as one part in QuMAsim.

In QuMAsim, the quantum pipeline is divided into two parts: the technology-dependent part and less-technology-dependent part. The less-technology-dependent part contains modules between the classical pipeline and the device event distributor. In this part, quantum instructions are decoded, and quantum operations are routed to the target qubit. This part generates quantum micro-operations for each qubit with precise timing as output.

The technology-dependent part contains all modules right to the operation combination module. Because this part should be customized for the target system. For example, the device event distributor and the number of queues highly depends on what analog devices are used and how they interconnect in the analog-digital interface. This part receives the measurement result from the analog-digital-interface and outputs trigger signals (device operations as explained in Section 4.4), which are the final output of QuMAsim.

QuMAsim is designed to be a QuMA instantiation framework. Most modules are objects that are dynamically created in QuMAsim. During instantiation, QuMAsim reads the required configuration from a file and then creates each module with required size, structure, etc. The interface between modules can also be configured via a configuration file. Note, though this method can work for a series of hardware structure, it requires further improvement because different quantum chip topology would require a rather different implementation for the technology-dependent part and cannot be fully automatically generated from the configuration file.

Logging

In classical processor verification, tracking register-content change is the major method. However, tracking register-content change is insufficient to verify QuMA since the timing of output signals in the deterministic timing domain also matters. A new method that enables checking the timing of signals is required. To this end, we define a format that can be used to log digital events with timing information, which is called Timing Event Logging Format (TELF) as shown in Listing 6.1.

```
[metadata]
cycle_time = 20 ns  # supported unit: ps, ns, us, ms, s

[data]
"clock_cycle",           <key0>,      <key1>, ...
<number of cycle 0>,   <value>,    <value>, ...
<number of cycle 1>,   <value>,    <value>, ...
...
```

Listing 6.1: Format of a TELF file.

A TELF file contains metadata and data two parts, which are indicated by `[metadata]` and `[data]`, respectively. The metadata part is used to specify configuration information like the duration of one cycle, etc. Any characters after the ‘#’ symbol in a line are treated as the comment. The data part records the value of signals at a given cycle using the Comma-Separated Values (CSV) format. The first non-comment line in the data part should be the header line which defines the content of each column. The first key in the header line must be `clock_cycle`, which indicates that the first column contains the time information. An indefinite number of keys are allowed in the header line, with each key corresponding to one signal. From the second line on, every line records the value of each signal at a particular time point. The time point is specified by which cycle it is in. Multiple formats are supported for each value, including four-value logic, numeric value, and string. A four-value logic value can be ‘`0`’, ‘`1`’, ‘`X`’, or ‘`Z`’. Numeric value can be represented as binary (starting with `0b`), decimal (default), or hexadecimal (starting with `0h`). A string should be put in a pair of quotation marks (`"`).

6.4 Potential Applications

QuMAsim is still under development at the time of writing. Here, we show the potential applications of QuMAsim .

6.4.1 Design Space Exploration

As explained in Section 6.3.1, after QUAPA determines some overall configuration of QuMA, e.g., the VLIW width, QuMAsim can help designing the concrete quantum control microarchitecture. For example, to support controlling a quantum chip with 17 qubits that can form a distance-3 surface code as shown in Figure 6.1, we need to upgrade QuMA_v2.

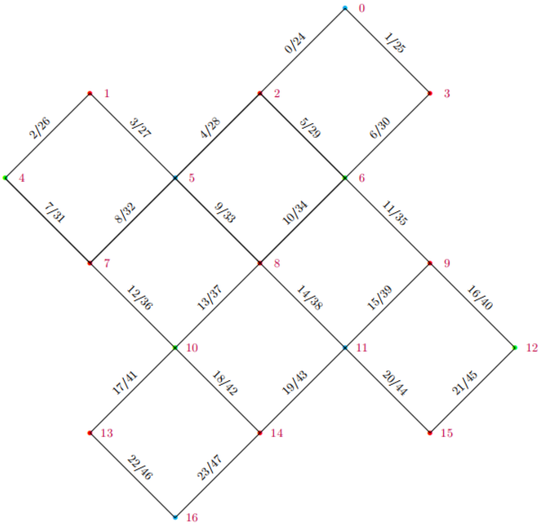


Figure 6.1: Quantum chip topology of the target 17-qubit superconducting quantum processor.

According to the analysis with QUAPA, a 32-bit eQASM instruction set with a VLIW width of 2 is still workable. Since more qubits (17) and allowed qubit pairs (48) are present in this quantum chip topology, the `SMIS` and `SMIT` in-

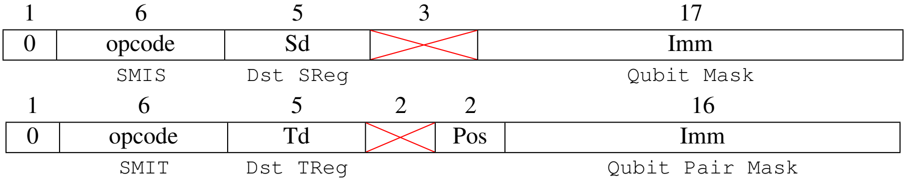


Figure 6.2: Format of the `SMIS` and `SMIT` for the 17-qubit quantum processor.

struction should be modified to support longer masks. In the `SMIS` instruction, the 17-bit mask can be put in the lower 17 bits. But for `SMIT` instruction, it is not possible to put the 48-bit mask into a single instruction. Hence, we use three `SMIT` instructions to specify a single two-qubit mask, with each specifying one-third of the mask. The new instruction format is shown in Figure 6.2.

Before implementing the entire new microarchitecture for the 17-qubit quantum processor, we validate the microarchitecture by implementing it in QuMAsim at first. This method is more advantageous than direct HDL implementation since QuMAsim allows describing components of less interest at a relatively high level. Even for modules that require to change, a functional implementation considering latency can enable a fast validation, which can be further converted into RTL description. According to the RTL description, hardware resource required by the microarchitecture can be estimated.

Furthermore, QuMAsim can help estimate the frequency and power of the QuMA processor at an early stage, which can save development time. This functionality is still under development.

6.4.2 RTL Verification

As shown in Figure 6.3, QuMAsim can support the verification of QuMA design automation by providing reference values for the checking points for given programs.

The checking is done by comparing the value and timing information for signals of interest generated from both the HDL implementation and QuMAsim in the TELF format. To enable easy recording of signals, we provide TELF recording libraries in both VHDL and SystemC.

To perform easy comparison, we also provide a library in Python to check the difference of two TELF input. For signals of which the timing is not important, the comparison unit only checks the values of them. For signals of which the timing is important, both value and timing are checked. Since the absolu-

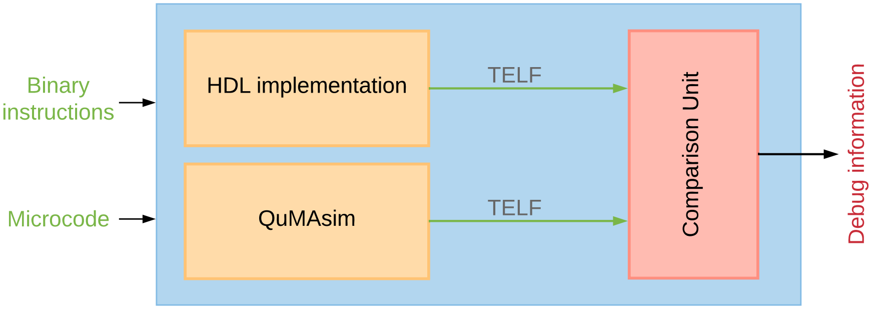


Figure 6.3: QuMAsim as used in a verification platform.

tion timing can vary in different simulation environments, the comparison unit checks the relative timing of different events.

6.4.3 Full Stack Quantum Simulator

It is still an open challenge to verify quantum software, including quantum algorithms, programming languages, and compilers etc. Though it is possible to verify some quantum software over the quantum hardware as we have in the lab, it is highly required to develop methods enabling verification over the software independently of the hardware. There are two reasons:

- Most of the time, the hardware we use is still under development and unreliable;
- It is inefficient or sometimes even impossible to test all newly-developed quantum software on the hardware.

To this end, QuMAsim can be used to construct a full stack quantum simulator for NISQ technology, called *Quantum Virtual Machine (QVM)*, as shown in Figure 6.4. The software layers in QVM are the same as those in a real quantum computer: quantum applications (algorithms or experiments) are described by a high-level language, which are compiled by the quantum compiler into eQASM instructions. The hardware layers in QVM are made of various simulators:

- QuMAsim simulates the execution of eQASM instructions and generate timed device operations;
- The electronic simulator translates the device operations into pulses with precise timing representing quantum operations;
- A qubit state simulator simulates qubit state evolution subject to the

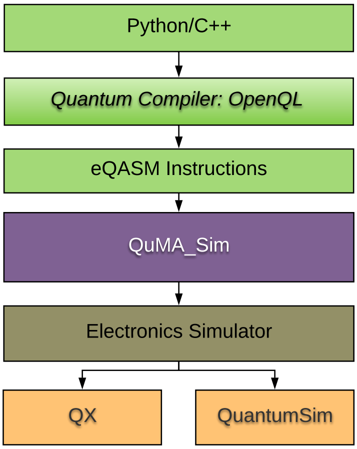


Figure 6.4: QVM: a full stack quantum simulator.

quantum operations generated by the electronic simulator. If the qubit state simulator has an error model extracted from experiments, such as QuantumSim [68], then it can simulate the execution of quantum applications under realistic noise.

Apart from checking the semantics of quantum applications are kept at various levels from quantum programming language to quantum operations applied on qubits, QVM can check: 1) if the microarchitectural constraints are respected or not; 2) if the timing requirement is satisfied or not. With the assist of QVM, the architectural constraints can be fully exposed to the algorithm designer and compiler designer for NISQ technology.

QVM can model a real measurement process and how measurement results affect the following program execution. This is especially useful for quantum subroutines which take advantage of feedback, such as qubit initialization based on feedback which is required by logical qubit initialization [39] and quantum gate decomposition using Repeat-Until-Success [119].

The fidelity of quantum algorithms can be significantly different when taking or not into account the timing of operations. By utilizing QVM, the timing of operations can be simulated, which can help investigate the impact of timing on the final fidelity of an algorithm and compiler-based optimization techniques.

6.5 Conclusion

In this chapter, we propose QuMAsim for quantum control microarchitecture. Based on SystemC, QuMAsim can simulate model the microarchitecture at a low level, which helps automating the design space exploration. By introducing the TELF format, the HDL simulation result can be compared against QuMAsim execution result, which enables using UVM to perform automatic verification over the QuMA series quantum control microarchitecture. By connecting to the qubit state simulator, such as QuantumSim, the quantum virtual machine is constructed, which enables verification over the quantum software stack while considering constraints at various levels.

An on-going work is to use QuMAsim to verify the correctness of the HDL implementation for further quantum control microarchitecture, e.g., QuMA_v3 to control the 17-qubit quantum processor which implements a distance-3 surface code. While developing FT_QuMA, QuMAsim can be a very useful tool to perform design space exploration, which can check the feasibility of the proposed mechanisms. Lastly, it would be interesting to use QVM to verify the correctness of the software stack including small-scale quantum algorithms, quantum programming languages, and quantum compilers.

7

Conclusion & Outlook

7.1 Conclusion

The gap between quantum software and hardware triggers the research of this thesis. We start by proposing a quantum control microarchitecture, QuMA, which is featured by codeword-based event control, queue-based precise timing control and multi-level instruction decoding. Apart from being scalable in the hardware implementation, QuMA enables flexible definition of quantum algorithms and experiments by a straightforward change in the input program written in the quantum microinstruction set QuMIS.

Being a low-level instruction set, QuMIS has low information density, which is disadvantageous regarding the quantum operation issue rate problem confronting QuMA. Required is a QISA that can encode denser information. Also, real-time feedback based on the measurement result of qubits is essential for qubit initialization, quantum error correction, and many quantum algorithms, which however cannot be supported in a programmable way. We propose an executable QISA, eQASM, targeting the NISQ technology, which can not only embed high-level quantum semantics but also be directly executed by a quantum control microarchitecture. eQASM defines two kinds of feedback, fast conditional execution, and comprehensive feedback control, which are supported by the microarchitectural mechanisms as implemented in QuMA_v2. As opposed to classical ISA where the allowed operations are defined at ISA design time, eQASM is a framework for concrete QISAs, which requires instantiation targeting a particular platform, and enables quantum operation definition at compile time. In this way, eQASM leaves great space for microarchitecture implementation optimization and compiler-based program optimization. Since it is not clear which quantum technology will be used to construct future quantum computers, eQASM is not bound to specific quantum hardware and has the potential to support different quantum platforms.

Large-scale quantum computing proposes more challenge in the control microarchitecture, including complex qubit management, exploded number of instructions required by QEC, and non-trivial fault-tolerant logical operation implementation. To address these challenges, we propose FT_QuMA, which is a fault-tolerant quantum control microarchitecture targeting surface code with logical operations using lattice surgery. The virtual-physical address translation at runtime enables a clean compilation model without worrying about the physical address of qubits. Automatic operation generation for error syndrome measurement in the hardware can significantly reduce the number of instructions. FT_QuMA can support all operations at both the logical level and the physical level. When necessary, the QEC-related features can be turned off to get a microarchitecture working at the physical level.

To enable effective design space exploration and efficient verification of the HDL implementation for the quantum control microarchitecture, we designed and implemented a control (micro)architecture simulator QuMAsim at the register-transfer level in SystemC. A dedicated format, timing event logging format (TELF), has been defined to enable the logging and comparison of digital signals with precise timing. With the same quantum software used (quantum algorithm, language, compiler), by connecting QuMAsim to an electronic simulator which controls a qubit state simulator, such as QuantumSim [68], we can get a full stack simulator, called the quantum virtual machine (QVM). QVM can be used to verify the correctness of quantum software at various levels. Constraints at the architecture and microarchitecture level can also be simulated.

7.2 Outlook

We have developed QuMA_v2 to control a seven-qubit superconducting quantum processor, a direct follow-up step is to develop next generations of QuMA to control the 17-qubit (distance-3 surface code) and 49-qubit (distance-5 surface code) superconducting quantum processors. The 17-qubit version control microarchitecture (QuMA_v3) is currently under development.

Various quantum technologies are being developed for quantum computing, including superconducting qubits and trapped ions. However, it is still unknown which quantum technology will be used to build future quantum computers. Though QuMA originally targets superconducting qubits, it can also be adapted to operate on different quantum technologies; some changes are required, including the microcode unit, the number and width of queues, and the quantum-classical interface. So current QuMA implementations are based on FPGAs to enable an easy adaption to different underlying hardware. Though our recent experiment demonstrates that QuMA is capable of controlling spin qubits, it would be interesting to make the digital part of QuMA as technology-independent as possible. This technology-independence is beneficial for two aspects. First, it helps the implementation of QuMA based on the application-specific integrated circuit (ASIC), which in principle can achieve a higher frequency for the non-deterministic timing domain and significantly alleviate the quantum operation issue rate problem. Second, since the same device can be used to control different hardware, it helps the commercialization of the digital part of QuMA because the same product can be easily duplicated to control different analog devices and qubits without further modification.

To further scale up the system, a tiled architecture consisting of multiple QuMA nodes with each node controlling tens of qubits would be a potential solution. In such a tiled architecture, the mechanisms in QuMA are still valid, but a communication protocol among nodes and a compilation model for a tiled system requires investigation.

For solid state quantum systems that require low temperature, current methods allocate most electronics at room temperature, and coaxial cables are used to send analog signals to qubits that are in the cryogenic environment. The number of cables grows roughly linearly to the number of qubits. The footprint and the thermal conductance of the cables form a challenge for a large number of qubits [168]. Addressing this issue, some research [60] investigates allocating part of the electronics, such as waveform generators, in the 4K environment. Whether a part of QuMA can be allocated in the 4K environment highly depends on the available power budget and the power consumption of each component of the QuMA implementation.

Further simulation based on QUAPA or QuMAsim could help us to deeper understand the QuMA series microarchitecture. First, the quantum operation issue rate is a potential bottleneck for the QuMA series microarchitecture. It would be interesting to investigate the relationship between the configurations of QuMA (VLIW width, instruction width, qubit address encoding, etc.) and the maximum number of qubits that can be controlled with this QuMA. Second, an interesting application of QuMAsim is to demonstrate the feasibility of FT_QuMA. Last, we can check how technology independent the quantum compiler, the QISA, and the control microarchitecture are by replacing the electronic simulator in QVM to target different qubit technologies.

Bibliography

[1] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, pp. 66–73, 2010.

[2] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, p. 69, 2015.

[3] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, pp. 269–284, 2014.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of 44th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2017, pp. 1–12.

[6] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, "Memristor for computing: Myth or reality?" in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 2017, pp. 722–731.

[7] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H.-S. P. Wong, and S. Mitra, "Carbon nanotube computer," *Nature*, vol. 501, p. 526, 2013.

[8] M. A. Manheimer, "Cryogenic computing complexity program: Phase 1 introduction," *IEEE Transactions on Applied Superconductivity*, vol. 25, pp. 1–4, 2015.

[9] H. Horii, J. H. Yi, J. H. Park, Y. H. Ha, I. G. Baek, S. O. Park, Y. N. Hwang, S. H. Lee, Y. T. Kim, K. H. Lee, U.-I. Chung, and J. T. Moon, "A novel cell technology using N-doped GeSbTe films for phase change RAM," in *Proceedings of Symposium on VLSI Technology*. IEEE, 2003, pp. 177–178.

[10] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982.

[11] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, pp. 230–265, 1937.

[12] D. Deutsch, "Quantum theory, the church–turing principle and the universal quantum computer," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 400, pp. 97–117, 1985.

[13] E. Bernstein and U. Vazirani, “Quantum complexity theory,” in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM, 1993, pp. 11–20.

[14] I. Kassal, J. D. Whitfield, A. Perdomo-Ortiz, M.-H. Yung, and A. Aspuru-Guzik, “Simulating chemistry using quantum computers,” *Annual Review of Physical Chemistry*, vol. 62, pp. 185–207, 2011.

[15] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O’brien, “A variational eigenvalue solver on a photonic quantum processor,” *Nature Communications*, vol. 5, p. 4213, 2014.

[16] 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.

[17] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, “Quantum machine learning,” *Nature*, vol. 549, p. 195, 2017.

[18] K. Lu, Y. Zhang, K. Xu, Y. Gao, and R. C. Wilson, “Approximate maximum common sub-graph isomorphism based on discrete-time quantum walk,” in *Proceedings of the 22nd International Conference on Pattern Recognition (ICPR)*. IEEE, 2014, pp. 1413–1418.

[19] K. K. H. Cheung and M. Mosca, “Decomposing finite abelian groups,” *Quantum Information & Computation*, vol. 1, pp. 26–32, 2001.

[20] D. P. DiVincenzo, “The physical implementation of quantum computation,” *arXiv:quant-ph/0002077*, 2000.

[21] The Class for Physics of the Royal Swedish Academy of Sciences, “Measuring and manipulating individual quantum systems,” the Royal Swedish Academy of Sciences, Citation of the Nobel Prize in Physics, 2012.

[22] N. A. Gershenfeld and I. L. Chuang, “Bulk spin-resonance quantum computation,” *Science*, vol. 275, pp. 350–356, 1997.

[23] D. G. Cory, A. F. Fahmy, and T. F. Havel, “Ensemble quantum computing by NMR spectroscopy,” *Proceedings of the National Academy of Sciences*, vol. 94, pp. 1634–1639, 1997.

[24] C. Monroe, D. Meekhof, B. King, W. M. Itano, and D. J. Wineland, “Demonstration of a fundamental quantum logic gate,” *Physical Review Letters*, vol. 75, p. 4714, 1995.

[25] S. Debnath, N. Linke, C. Figgatt, K. Landsman, K. Wright, and C. Monroe, “Demonstration of a small programmable quantum computer with atomic qubits,” *Nature*, vol. 536, pp. 63–66, 2016.

[26] 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*, vol. 519, pp. 66–69, 2015.

[27] 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*, vol. 6, p. 6983, 2015.

[28] 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.

[29] 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*, vol. 79, pp. 1217–1265, 2007.

[30] 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*, vol. 85, p. 961, 2013.

[31] 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*, vol. 555, p. 633, 2018.

[32] 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*, vol. 330, pp. 60–63, 2010.

[33] 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*, vol. 7, 2016.

[34] X.-C. Yao, T.-X. Wang, H.-Z. Chen, W.-B. Gao, A. G. Fowler, R. Raussendorf, Z.-B. Chen, N.-L. Liu, C.-Y. Lu, Y.-J. Deng, Y.-A. Chen, and J.-W. Pan, "Experimental demonstration of topological error correction," *Nature*, vol. 482, p. 489, 2012.

[35] D. Suter and T. Mahesh, "Spins as qubits: Quantum information processing by nuclear magnetic resonance," *The Journal of Chemical Physics*, vol. 128, p. 052206, 2008.

[36] A. D. Córcoles, E. Magesan, S. J. Srinivasan, A. W. Cross, M. Steffen, J. M. Gambetta, and J. M. Chow, "Demonstration of a quantum error detection code using a square lattice of four superconducting qubits," *Nature Communications*, vol. 6, p. 6979, 2015.

[37] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Physical Review A*, vol. 52, p. R2493, 1995.

[38] M. Oskin, F. T. Chong, and I. L. Chuang, "A practical architecture for reliable quantum computers," *Computer*, vol. 35, pp. 79–87, 2002.

[39] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, p. 032324, 2012.

[40] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2010.

[41] L. C. L. Hollenberg, A. D. Greentree, A. G. Fowler, and C. J. Wellard, "Two-dimensional architectures for donor-based quantum computing," *Physical Review B*, vol. 74, p. 045311, 2006.

[42] D. P. DiVincenzo, "Fault-tolerant architectures for superconducting qubits," *Physica Scripta*, vol. 2009, p. 014020, 2009.

[43] C. D. Hill, E. Peretz, S. J. Hile, M. G. House, M. Fuechsle, S. Rogge, M. Y. Simmons, and L. C. Hollenberg, "A surface code quantum computer in silicon," *Science Advances*, vol. 1, p. e1500707, 2015.

[44] R. Li, L. Petit, D. P. Franke, J. P. Dehollain, J. Helsen, M. Steudtner, N. K. Thomas, Z. R. Yoscovits, K. J. Singh, S. Wehner, L. M. K. Vandersypen, J. S. Clarke, and M. Veldhorst, "A crossbar network for silicon quantum dot qubits," *arXiv:1711.03807*, 2017.

[45] T. S. Metodi, D. D. Thaker, A. W. Cross, F. T. Chong, and I. L. Chuang, "A quantum logic array microarchitecture: Scalable quantum data movement and computation," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38).* IEEE/ACM, 2005, pp. 305–318.

[46] L. Kreger-Stickles and M. Oskin, "Microcoded architectures for ion-tap quantum computers," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2008, pp. 165–176.

[47] D. D. Thaker, T. S. Metodi, A. W. Cross, I. L. Chuang, and F. T. Chong, "Quantum memory hierarchies: Efficient designs to match available parallelism in quantum computing," in *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA)*, vol. 34. ACM/IEEE, 2006, pp. 378–390.

[48] N. Isailovic, M. Whitney, Y. Patel, and J. Kubiatowicz, "Running a quantum circuit at the speed of data," in *Proceedings of 30th Annual International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2008, pp. 177–188.

[49] S. Balensiefer, L. Kregor-Stickles, and M. Oskin, "An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures," in *Proceedings of 32nd International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2005, pp. 186–196.

[50] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *Computer*, pp. 74–83, 2006.

[51] D. Wecker and K. M. Svore, "LIQ$Ui|\rangle$: A software design architecture and domain-specific language for quantum computing," *arXiv:1402.4467*, 2014.

[52] 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.

[53] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[54] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *Proceedings of 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 2015, pp. 445–456.

[55] D. Kudrow, K. Bier, Z. Deng, D. Franklin, Y. Tomita, K. R. Brown, and F. T. Chong, "Quantum rotations: A case study in static and dynamic machine-code generation for quantum computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA).* ACM/IEEE, 2013, pp. 166–176.

[56] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, J. M. Martinis, and H. Neven, "Characterizing quantum supremacy in near-term devices," *arXiv:1608.00263*, 2016.

[57] J. Preskill, "Quantum computing in the NISQ era and beyond," *arXiv:1801.00862*, 2018.

[58] ——, "Quantum computing and the entanglement frontier," *arXiv:1203.5813*, 2012.

[59] J. Wu, Y. Liu, B. Zhang, X. Jin, Y. Wang, H. Wang, and X. Yang, "Computing permanents for boson sampling on TianHe-2 supercomputer," *arXiv:1606.05836*, 2016.

[60] J. M. Hornibrook, J. I. Colless, I. D. Conway Lamb, S. J. Pauka, H. Lu, A. C. Gossard, J. D. Watson, G. C. Gardner, S. Fallahi, M. J. Manfra, and D. J. Reilly, "Cryogenic control architecture for large-scale quantum computing," *Physical Review Applied*, vol. 3, p. 024010, 2015.

[61] 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*, vol. 76, p. 042319, 2007.

[62] A. Blais, R.-S. Huang, A. Wallraff, S. Girvin, and R. J. Schoelkopf, "Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation," *Physical Review A*, vol. 69, p. 062320, 2004.

[63] C. C. Bultink, M. A. Rol, T. E. O'Brien, X. Fu, B. Dikken, R. Vermeulen, J. C. de Sterke, A. Bruno, R. N. Schouten, and L. DiCarlo, "Active resonator reset in the nonlinear dispersive regime of circuit QED," *Physical Review Applied*, vol. 6, p. 034008, 2016.

[64] IBM Q team, "IBM Quantum Experience," https://www.research.ibm.com/ibm-q/, 2018.

[65] M. Takita, A. Córcoles, E. Magesan, B. Abdo, M. Brink, A. Cross, J. M. Chow, and J. M. Gambetta, "Demonstration of weight-four parity measurements in the surface code architecture," *Physical Review Letters*, vol. 117, p. 210505, 2016.

[66] Rigetti, "Rigetti Forest," https://www.rigetti.com/forest, 2018.

[67] Alibaba, "Superconducting quantum computer," http://quantumcomputer.ac.cn/index.html, 2018.

[68] 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*, vol. 3, p. 39, 2017.

[69] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, pp. 593–599, 1953.

[70] 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.

[71] J. Stephen, "Quantum algorithm zoo," list available at http://math.nist.gov/quantum/zoo, 2018.

[72] L. DiCarlo, J. M. Chow, J. M. Gambetta, L. S. Bishop, B. R. Johnson, D. I. Schuster, J. Majer, A. Blais, L. Frunzio, S. M. Girvin, and R. J. Schoelkopf, "Demonstration of two-qubit algorithms with a superconducting quantum processor," *Nature*, vol. 460, pp. 240–244, 2009.

[73] L. DiCarlo, M. D. Reed, L. Sun, B. R. Johnson, J. M. Chow, J. M. Gambetta, L. Frunzio, S. M. Girvin, M. H. Devoret, and R. J. Schoelkopf, "Preparation and measurement of three-qubit entanglement in a superconducting circuit," *Nature*, vol. 467, pp. 574–578, 2010.

[74] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, C. Neill, P. O'Malley, P. Roushan, A. Vainsencher, J. Wenner, A. N. Korotkov, A. N. Cleland, and J. M. Martinis, "Superconducting quantum circuits at the surface code threshold for fault tolerance," *Nature*, vol. 508, pp. 500–503, 2014.

[75] Tektronix, "Arbitrary waveform generator AWG5000 series," Available at https://www.tek.com/datasheet/awg5000-series, 2018.

[76] AlazarTech, "Alazartech PCI Express digitizers," Available at http://www.alazartech.com/, 2018.

[77] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical Review Letters*, vol. 106, p. 180504, 2011.

[78] J. M. Epstein, A. W. Cross, E. Magesan, and J. M. Gambetta, "Investigating the limits of randomized benchmarking protocols," *Physical Review A*, vol. 89, p. 062321, 2014.

[79] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, vol. 87, p. 307, 2015.

[80] E. Knill, "Quantum computing with realistically noisy devices," *Nature*, vol. 434, p. 39, 2005.

[81] 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*, vol. 8, p. 034021, 2017.

[82] A. G. Fowler, A. C. Whiteside, and L. C. L. Hollenberg, "Towards practical classical processing for the surface code," *Physical Review Letters*, vol. 108, p. 180501, 2012.

[83] L. Lao, "Simulation of fault-tolerant operations on rotated surface codes," Delft University of Technology, Technical Report, 2018.

[84] N. C. Jones, R. Van Meter, A. G. Fowler, P. L. McMahon, J. Kim, T. D. Ladd, and Y. Yamamoto, "Layered architecture for quantum computing," *Physical Review X*, vol. 2, p. 031007, 2012.

[85] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, pp. 425–432, 1980.

[86] R. Van Meter and C. Horsman, "A blueprint for building a quantum computer," *Communications of the ACM*, vol. 56, pp. 84–93, 2013.

[87] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, p. 180, 2017.

[88] B. Omer, "Structured quantum programming," *Information Systems*, p. 130, 2003.

[89] S. Bettelli, T. Calarco, and L. Serafini, "Toward an architecture for quantum programming," *The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics*, vol. 25, pp. 181–200, 2003.

[90] 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," Princeton University, Technical Report, 2012.

[91] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[92] S. Liu, X. Wang, L. Zhou, J. Guan, Y. Li, Y. He, R. Duan, and M. Ying, "Q$|$SI$\rangle$: A quantum programming environment," *arXiv:1710.09500*, 2017.

[93] C. Vasconcelos and A. Ravara, "The while language," *arXiv:1603.08949*, 2016.

[94] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: an open source software framework for quantum computing," *arXiv:1612.08091*, 2016.

[95] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," *arXiv:1608.03355*, 2016.

[96] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[97] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv:1707.03429*, 2017.

[98] J. I. Cirac and P. Zoller, "Quantum computations with cold trapped ions," *Physical Review Letters*, vol. 74, p. 4091, 1995.

[99] D. Kielpinski, C. Monroe, and D. J. Wineland, "Architecture for a large-scale ion-trap quantum computer," *Nature*, vol. 417, pp. 709–711, 2002.

[100] S. Asaad, C. Dickel, N. K. Langford, S. Poletto, A. Bruno, M. A. Rol, D. Deurloo, and L. DiCarlo, "Independent, extensible control of same-frequency superconducting qubits by selective broadcasting," *npj Quantum Information*, vol. 2, p. 16029, 2016.

[101] T. Brecht, W. Pfaff, C. Wang, Y. Chu, L. Frunzio, M. H. Devoret, and R. J. Schoelkopf, "Multilayer microwave integrated quantum circuits for scalable quantum computing," *npj Quantum Information*, vol. 2, p. 16002, 2016.

[102] D. Loss and D. P. DiVincenzo, "Quantum computation with quantum dots," *Physical Review A*, vol. 57, p. 120, 1998.

[103] B. E. Kane, "A silicon-based nuclear spin quantum computer," *Nature*, vol. 393, p. 133, 1998.

[104] M. Oskin, F. T. Chong, I. L. Chuang, and J. Kubiatowicz, "Building quantum wires: The long and the short of it," in *Proceedings of 30th Annual International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2003, pp. 374–385.

[105] E. Chi, S. A. Lyon, and M. Martonosi, "Tailoring quantum architectures to implementation style: a quantum computer for mobile and persistent qubits," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2007, pp. 198–209.

[106] M. G. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz, "A fault tolerant, area efficient architecture for shor's factoring algorithm," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, vol. 37. ACM/IEEE, 2009, pp. 383–394.

[107] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the ibm qx architectures," *arXiv:1712.04722*, 2018.

[108] M. Y. Siraichi, V. F. d. Santos, S. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 113–125.

[109] S. S. Tannu and M. K. Qureshi, "A case for variability-aware policies for nisq-era quantum computers," *arXiv:1805.10224*, 2018.

[110] S. S. Tannu, Z. A. Myers, P. J. Nair, D. M. Carmean, and M. K. Qureshi, "Taming the instruction bandwidth of quantum computers via hardware-managed error correction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE/ACM, 2017, pp. 679–691.

[111] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," *arXiv: quant-ph/0410184*, 2004.

[112] M. Amy, M. Roetteler, and K. M. Svore, "Verified compilation of space-efficient reversible circuits," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 3–21.

[113] D. Deutsch, A. Barenco, and A. Ekert, "Universality in quantum computation," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 449, pp. 669–677, 1995.

[114] S. Lloyd, "Almost any quantum logic gate is universal," *Physical Review Letters*, vol. 75, p. 346, 1995.

[115] D. P. DiVincenzo, "Quantum gates and circuits," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 454, pp. 261–276, 1998.

[116] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. Almudever, "Mapping of lattice surgery-based quantum circuits on surface code architectures," *arXiv:1805.11127*, 2018.

[117] X. Fu, L. Riesebos, L. Lao, C. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, "A heterogeneous quantum computer architecture," in *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. ACM, 2016, pp. 323–330.

[118] M. Amy, M. Roetteler, and K. Svore, "Verified compilation of space-efficient reversible circuits," *arXiv:1603.01635*, 2016.

[119] A. Paetznick and K. M. Svore, "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries," *Quantum Information & Computation*, vol. 14, pp. 1277–1301, 2014.

[120] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*. Elsevier, 2011.

[121] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, pp. 4452–4505, 2002.

[122] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $\mathcal{O}(1)$ parallel time," *Quantum Information and Computation*, vol. 15, pp. 145–158, 2015.

[123] J. M. Chow, L. DiCarlo, J. M. Gambetta, F. Motzoi, L. Frunzio, S. M. Girvin, and R. J. Schoelkopf, "Optimized driving of superconducting artificial atoms for improved single-qubit gates," *Physical Review A*, vol. 82, p. 040305, 2010.

[124] M. D. Reed, "Entanglement and quantum error correction with superconducting qubits," Ph.D. dissertation, Yale University, 2013.

[125] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *arXiv:1604.01401*, 2016.

[126] M. V. Wilkes, "The best way to design an automatic calculating machine," in *The early British computer conferences*. MIT Press, 1989, pp. 182–184.

[127] S. Vassiliadis, S. Wong, and S. Cotofana, "Microcode processing: Positioning and directions," *IEEE Micro*, vol. 23, pp. 21–30, 2003.

[128] S. Beauregard, "Circuit for Shor's algorithm using $2n + 3$ qubits," *arXiv:quant-ph/0205095*, 2002.

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

[130] C. A. Ryan, B. R. Johnson, D. Ristè, B. Donovan, and T. A. Ohki, "Hardware for dynamic quantum computing," *arXiv:1704.08314*, 2017.

[131] Raytheon BBN, "BBN Technologies Arbitrary Pulse Sequencer 2," Available at http://libaps2.readthedocs.org/en/latest/, 2017.

[132] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, "QX: A high-performance quantum computer simulation platform," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. IEEE, 2017, pp. 464–469.

[133] L. Riesebos, X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels, "Pauli frames for quantum computer architectures," in *Proceedings of the 54th Annual Design Automation Conference (DAC)*. ACM, 2017, p. 76.

[134] Keysight, "M3202A PXIe Arbitrary Waveform Generator, 1 GSa/s, 14 bit, 400 MHz," http://www.keysight.com/en/pd-2747446-pn-M3202A/pxie-arbitrary-waveform-generator-1-gs-s-14-bit-400-mhz?cc=US&lc=eng, 2017.

[135] 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," *arXiv:1805.09607*, 2018.

[136] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, pp. 527–586, 2004.

[137] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels," *Physical Review Letters*, vol. 70, p. 1895, 1993.

[138] D. Ristè, C. C. Bultink, K. W. Lehnert, and L. DiCarlo, "Feedback control of a solid-state qubit using high-fidelity projective measurement," *Physical Review Letters*, vol. 109, p. 240502, 2012.

[139] N. Ofek, A. Petrenko, R. Heeres, P. Reinhold, Z. Leghtas, B. Vlastakis, Y. Liu, L. Frunzio, S. M. Girvin, L. Jiang, M. Mirrahimi, M. H. Devoret, and R. J. Schoelkopf, "Extending the lifetime of a quantum bit with error correction in superconducting circuits," *Nature*, vol. 536, p. 441, 2016.

[140] L. Hu, Y. Ma, W. Cai, X. Mu, Y. Xu, W. Wang, Y. Wu, H. Wang, Y. Song, C. Zou, S. M. Girvin, L.-M. Duan, and L. Sun, "Demonstration of quantum error correction and universal gate set on a binomial bosonic logical qubit," *arXiv:1805.09072*, 2018.

[141] 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/ACM, 2017, pp. 813–825.

[142] ——, "A microarchitecture for a superconducting quantum processor," *IEEE Micro*, vol. 38, pp. 40–47, 2018.

[143] M. Ying, *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[144] J. Werschnik and E. Gross, "Quantum optimal control theory," *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 40, p. R175, 2007.

[145] N. Leung, M. Abdelhafez, J. Koch, and D. Schuster, "Speedup for quantum optimal control from automatic differentiation based on graphics processing units," *Physical Review A*, vol. 95, p. 042318, 2017.

[146] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, pp. 948–960, 1972.

[147] 5-qubit backend: IBM Q team, "IBM Q 5 Yorktown backend specification V1.1.0," Retrieved from https://github.com/QISKit/qiskit-backend-information/tree/master/backends/yorktown/V1, 2018.

[148] J. Heinsoo, C. Kraglund Andersen, A. Remm, S. Krinner, T. Walter, Y. Salathé, S. Gasperinetti, J.-C. Besse, A. Potočnik, C. Eichler, and A. Wallraff, "Rapid high-fidelity multiplexed readout of superconducting qubits," *arXiv:1801.07904*, 2018.

[149] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, 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," *arXiv:1808.02449*, 2018.

[150] A. J. Landahl and C. Ryan-Anderson, "Quantum computing by color-code lattice surgery," *arXiv:1407.5103*, 2014.

[151] IBM, "IBM Quantum Experience Device," https://quantumexperience.ng.bluemix.net/qx/devices.

[152] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, pp. 2–30, 2003.

[153] A. G. Fowler, A. C. Whiteside, and L. C. L. Hollenberg, "Towards practical classical processing for the surface code: timing analysis," *Physical Review A*, vol. 86, p. 042313, 2012.

[154] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of mathematics*, vol. 17, pp. 449–467, 1965.

[155] F. Barahona *et al.*, "Morphology of ground states of two-dimensional frustration model," *Journal of Physics A: Mathematical and General*, vol. 15, p. 673, 1982.

[156] V. Kolmogorov, "Blossom V: a new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, pp. 43–67, 2009.

[157] D. Gottesman, "Fault-tolerant quantum computation with higher-dimensional systems," in *Quantum Computing and Quantum Communications*. Springer Berlin Heidelberg, 1999, pp. 302–313.

[158] P. Aliferis and J. Preskill, "Fault-tolerant quantum computation against biased noise," *Physical Review A*, vol. 78, p. 052331, 2008.

[159] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, p. 123011, 2012.

[160] C. Vuillot, L. Lao, B. Criger, C. G. Almudever, K. Bertels, and B. M. Terhal, "Code deformation and lattice surgery are gauge fixing," *In preparation*, 2018.

[161] S. Varsamopoulos, B. Criger, and K. Bertels, “Decoding small surface codes with feed-forward neural networks,” *Quantum Science and Technology*, vol. 3, p. 015004, 2017.

[162] G. Torlai and R. G. Melko, “Neural decoder for topological codes,” *Physical Review Letters*, vol. 119, p. 030501, 2017.

[163] 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*, vol. 39, pp. 1–7, 2011.

[164] Mentor Graphics, “Questa Advanced Simulator,” https://www.mentor.com/products/fv/questa, 2018.

[165] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[166] S. Rosenberg and K. Meade, *A practical guide to adopting the Universal Verification Methodology (UVM)*. Cadence Design Systems, 2013.

[167] Mentor Graphics, “Catapult C/C++/SystemC high-level synthesis,” https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls, 2018.

[168] C. G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K. Bertels, “The engineering challenges in quantum computing,” in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. IEEE, 2017, pp. 836–845.

Summary

Quantum computers can accelerate solving some problems which are inefficiently solved by classical computers, such as quantum chemistry simulation. To date, quantum computer engineering has focused primarily at opposite ends of the required system stack: devising high-level programming languages and compilers to describe and optimize quantum algorithms, and building reliable low-level quantum hardware. Relatively little attention has been given to using the compiler output to fully control the operations on current experimental quantum processors.

Bridging this gap, we propose and build a prototype of flexible control microarchitecture, named QuMA, supporting quantum-classical mixed code for a superconducting quantum processor. The microarchitecture is based on three core elements: (i) a codeword-based event control scheme, (ii) queue-based precise event timing control, and (iii) a flexible multilevel instruction decoding mechanism for control.

However, QuMIS does not offer feedback control, and is tightly bound to the hardware implementation. Also, as the number of qubits grows, QuMA cannot fetch and execute instructions fast enough to apply all operations on qubits on time. Known as the quantum operation issue rate problem, this limitation is aggravated by the low information density of QuMIS instructions. To address these issues, we propose an executable quantum instruction set architecture (QISA), called eQASM, that can be translated from quantum assembly language (QASM), supports comprehensive quantum program flow control, and is executed on a quantum control microarchitecture. eQASM alleviates the quantum operation issue rate problem by efficient timing specification, single-operation-multiple-qubit execution, and a very-long-instruction-word architecture. The definition of eQASM focuses on the assembly level to be expressive. Quantum operations are configured at compile time instead of being defined at QISA design time.

Since qubits suffer from short decoherence time and quantum operations are erroneous, quantum error correction (QEC) is essential for large-scale fault-tolerant quantum computing, which requires highly patterned control over a large number of qubits. We envisioned a control microarchitecture, FT_QuMA, that can efficiently support fault-tolerant quantum computing based on planar surface code with logical operations implemented by lattice surgery. It highlights a two-level address mechanism which enables a clean compilation model

for a large number of qubits, and microarchitectural support for quantum error correction at runtime, which can significantly address the quantum program codesize and present better scalability.

As the complexity of quantum control microarchitecture increases, the design, development, and verification of the quantum control microarchitecture forms a challenge. Inspired by simulators as used in classical processor design, we propose and implemented a quantum (micro)architecture simulator QuMAsim. QuMAsim is a cycle-accurate (micro)architecture simulator based on SystemC, which can simulate the electronic part of the microarchitecture. By connecting QuMAsim with a qubit state simulator with a precise error model, such as QuantumSim [68], we can construct a full-stack simulator for NISQ technology, called *Quantum Virtual Machine* (QVM). Apart from the correctness of quantum program semantics at various levels, QVM can also verify low-level hardware constraints, including electronic control constraints and timing.

Samenvatting

Kwantumcomputers kunnen sommige problemen, die niet efficiënt opgelost kunnen worden met klassieke computers, versneld oplossen. Bijvoorbeeld kwantumchemie simulaties. Tot op heden heeft kwantumcomputertechnologie zich primair gericht op tegenovergestelde uiteinden van de vereiste systeemstack: het ontwerpen van programmeertalen met een hoog abstractieniveau en compilers om kwantumalgoritmen te beschrijven en te optimaliseren, en het construeren van betrouwbare laag-niveau kwantum hardware. Relatief weinig aandacht is besteed aan het gebruik van de compileruitvoer voor het volledig besturen van de benodigde operaties voor experimentele kwantumprocessors.

Om deze kloof te overbruggen stellen we een prototype en flexibele besturing microarchitectuur voor, genaamd QuMA, die een combinatie van kwantum en klassieke code ondersteund voor supergeleidende kwantumprocessors. De microarchitectuur is gebaseerd op drie kernelementen: (i) een door code aangestuurd en evenement gebaseerd besturingsschema, (ii) precieze en op wachtrij gebaseerde evenementtiming besturing, en (iii) een flexibel en meervoudigniveau decoderingmechanisme voor instructies.

QuMIS biedt echter geen teruggekoppelde besturing en is nauw verbonden met de hardware implementatie. Als het aantal qubits groter wordt kan QuMA niet snel genoeg instructies ophalen en verwerken om alle operaties op qubits op tijd uit te voeren. Dit probleem is bekend als het kwantumoperatie verwerkingssnelheid probleem en deze beperking wordt veroorzaakt door de lage informatiedichtheid van de QuMIS instructies. Om deze problemen aan te pakken stellen wij een uitvoerbare kwantuminstructieset architectuur (QISA) voor, genaamd eQASM, die van een kwantumassembleertaal vertaald kan worden, complexe kwantumprogramma stroombesturing ondersteund, en uitgevoerd kan worden op een kwantum besturing microarchitectuur. eQASM verlicht het kwantumoperatie verwerkingssnelheid probleem door gebruik te maken van een efficiënte tijdspecificatie, enkele-operatie-meerdere-qubits uitvoering, en een heel-lang-instructie-woord (VLIW) architectuur. De definitie van eQASM richt zich op het assembleertaalniveau om expressief te zijn. Kwantumoperaties worden tijdens het compileren geconfigureerd in plaats van tijdens de QISA-ontwerptijd.

Omdat qubits lijden aan korte decoherentietijd en kwantumoperaties fouten bevatten is kwantum foutcorrectie (QEC) essentieel voor grootschalige en fouttolerante kwantumberekeningen. Kwantum foutcorrectie vereist gestructureerde en repetitieve besturing over een groot aantal qubits. Wij voorzien

een besturing microarchitectuur, FT_QuMA, die efficiënte ondersteuning biedt voor fouttolerante kwantumberekeningen gebaseerd op plenaire surface code met logische operaties gebaseerd op lattice surgery. De nadruk ligt op het dubbel-niveau adresseringmechanisme wat ervoor zorgt dat het compilatiemodel voor een groot aantal qubits helder is en de ondersteuning vanuit de microarchitectuur voor kwantum foutcorrectie tijdens het uitvoeren van berekeningen. Deze technieken beperken de bestandgrootte van het kwantumprogramma en zorgen voor een betere schaalbaarheid.

Naarmate de complexiteit van de kwantum besturing microarchitectuur toeneemt wordt het ontwerp, het construeren en de verificatie van een kwantum besturing microarchitectuur een uitdaging. Geïnspireerd door simulatoren zoals gebruikt voor klassieke processorontwerpen stellen we een kwantum (micro)architectuur simulator voor. Deze simulator is geïmplementeerd en staat bekend als QuMAsim. QuMAsim is een cyclus-accurate (micro)architectuur simulator gebaseerd op SystemC en is in staat het elektronische deel van de microarchitectuur te simuleren. Door QuMAsim te verbinden met een kwantumsimulator met een nauwkeurig foutmodel, zoals QuantumSim, kunnen we een volledige systeemstack simulator construeren voor NISQ-technologie, genaamd Quantum Virtual Machine (QVM). Los van de correctheid van de kwantumprogramma semantiek op verschillende niveaus kan de QVM controleren of laag-niveau hardware beperkingen gehoorzaamd worden, inclusief elektronische besturingsbeperkingen en timing.

Curriculum Vitae

Xiang Fu was born on February 4th, 1990 in Yueyang, Hunan Province, China. In 2004, he started his high-school study at Yueyang County Number 1 high school, Yueyang, China. During this period, he developed his interest in physics.

He entered the Department of Electronic Engineering at Tsinghua University (THU), Beijing, China in 2007, and got his bachelor's degree in 2011. The mismatch between his major and his interest led him to a mentally painful state, which drove him to reconsider the meaning of his life. By studying philosophy with Prof. Zhengxiang Wei, especially Chinese traditional philosophy and dialectical materialism, he realized that the underlying Chinese culture, i.e., Confucianism, typically shaped his personality. He found his path to happiness, which requires him to achieve his value by connecting his interest with the requirement of the society. Also, he got relief by performing research under the supervision of Dr. Yue Wang. By the end of the bachelor study, it became clear that he wants to have a career in scientific research.

Later, he entered the College of Computer at National University of Defense Technology (NUDT) to study computer architecture under the supervision of Prof. Weixia Xu. In 2014, he received his master's degree in Computer Architecture. This period of study gave him a comprehensive understanding of computer architecture.

In 2014, he was awarded a 4-year scholarship from the China Scholarship Council (CSC) to pursue his Ph.D. at Delft University of Technology (TU Delft). Driven by interest, he switched his research direction from microelectronics to quantum computing and receives joint supervision from Prof. Koen Bertels and Prof. Leonardo DiCarlo. His Ph.D. studies focus on bridging the gap between quantum software and hardware by developing quantum control microarchitecture and quantum instruction set architecture. The results of this work are presented in the current dissertation.

He will continue research in quantum computing in China after his Ph.D. graduation.

List of Publications

Journal Papers

1. **X. Fu**, L. Lao, C. G. Almudever, and K. Bertels, A Control Microarchitecture for Fault-Tolerant Quantum Computing. [In preparation, invited by *Microprocessors and Microsystems*]
2. **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, A Microarchitecture for a Superconducting Quantum Processor, *IEEE Micro*, vol. 38, pp. 40-47, 2018. [**Top Picks from the 2017 Computer Architecture Conferences**].
3. M. A. Rol, C. C. Bultink, T. E. O'Brien, S. R. de Jong, L. S. Theis, **X. Fu**, F. Luthi, R. F. L. Vermeulen, J. C. de Sterke, A. Bruno, D. Deurloo, R. N. Schouten, F. K. Wilhelm, and L. DiCarlo. Restless Tuneup of High-fidelity Qubit Gates, *Physical Review Applied*, vol. 7, p. 041001, 2017.
4. C. C. Bultink, M. A. Rol, T. E. O'Brien, **X. Fu**, B. C. S. Dikken, R. F. L. Vermeulen, J. C. de Sterke, A. Bruno, R. N. Schouten, and L. DiCarlo, Active Resonator Reset in the Nonlinear Dispersive Regime of Circuit QED, *Physical Review Applied*, vol. 6, p. 034008, 2016.

Conference Proceedings & arXiv

1. **X. Fu**, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, 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, *arXiv:1808.02449*, 2018. [To appear on *Proceedings of 25th IEEE International Symposium on High-Performance Computer Architecture (HPCA'19)*]
2. **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, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE/ACM, 2017, pp. 813-825. [**Best Paper Award**].
3. L. Riesebos, **X. Fu**, S. Varsamopoulos, C. G. Almudever, and K. Bertels. Pauli Frames for Quantum Computer Architectures, *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, ACM, 2017, p. 76.
4. C. G. Almudever, L. Lao, **X. Fu**, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, and K. Bertels, The Engineering Challenges in Quantum Computing, *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*, IEEE, 2017, pp. 836-845.
5. N. Khammassi, I. Ashraf, **X. Fu**, C. G. Almudever, and K. Bertels, QX: A High-performance Quantum Computer Simulation Platform, *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*, IEEE, 2017, pp. 464-469.

6. **X. Fu**, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, A Heterogeneous Quantum Computer Architecture, *Proceedings of the ACM International Conference on Computing Frontiers (CF'16)*, ACM, 2016, pp. 323-330.