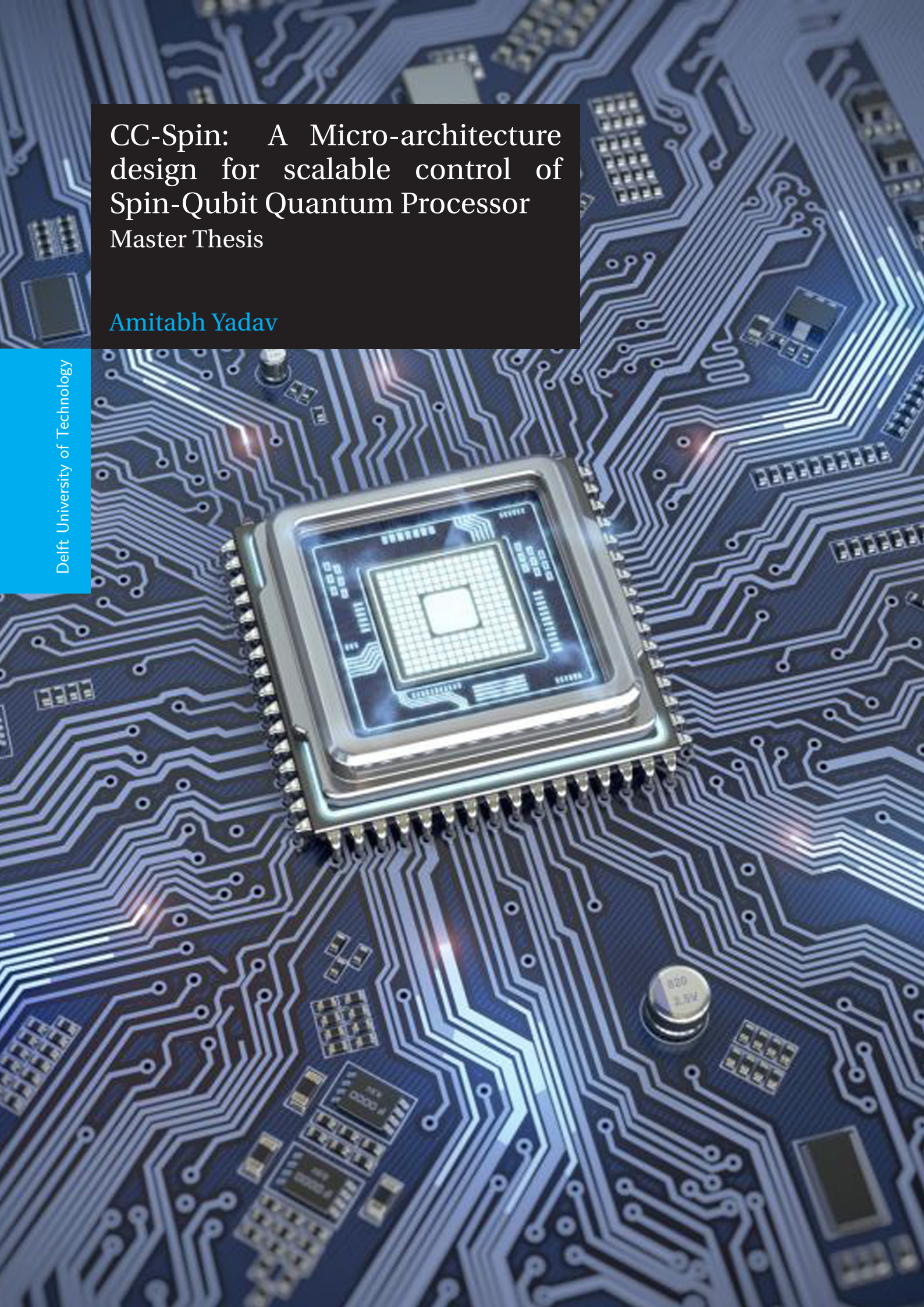
CC-Spin: A Micro-architecture design for scalable control of Spin-Qubit Quantum Processor

Master Thesis

Amitabh Yadav

Delft University of Technology



CC-Spin: A Micro-architecture design for scalable control of Spin-Qubit Quantum Processor

MASTER THESIS

by

Amitabh Yadav

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Wednesday August 28th, 2019 at 11:00 AM.

Supervisor:	Dr. ir. Nader Khammassi,	Intel Lab, Hillsboro, OR
	Prof. dr. Koen Bertels,	TU Delft
Thesis committee:	Prof. dr. Koen Bertels,	TU Delft
	Prof. dr. ir. Zaid Al-Ars,	TU Delft
	Prof. dr. ir. Fabio Sebastiano	TU Delft

This thesis is confidential and cannot be made public until August 28th, 2019.

Quantum & Computer Engineering

TU Delft

QuTech

intel

ॐ असतो मा सद्गमय।
तमसो मा ज्योतिर्गमय।
मृत्योर्मा अमृतं गमय।
ॐ शान्तिः शान्तिः शान्तिः॥

— बृहदारण्यक उपनिषद् (१.३.२८) —

Dedicated to Maa and Papa

Om, from falsehood lead me to truth. From (the ignorant state of) darkness lead me to the light (of spiritual knowledge). From (the world of) mortality lead me to immortality (of self-realization). Om peace, peace, peace. (Peace in three states of life - physical, mental and spiritual)

ACKNOWLEDGEMENTS

The last two years of my masters program at TU Delft has been an amazing journey. I would like to express my sincere gratitude to both my thesis supervisors **Dr. Nader Khammassi** and **Dr. Koen Bertels** for giving me the opportunity to conduct my master thesis research at the Quantum Computer Architecture Laboratory. The field of High-Speed Digital Design with a focus on Experimental Quantum Computing was the factor behind my interest in this particular masters thesis project.

During the course of my master thesis, I got the opportunity to learn about the experimental quantum processor control setup and the ideology behind development of microarchitecture for the same. I would like to thank Nader for his methodical guidance in approaching such a huge thesis topic. The thesis idea was initially proposed and initiated by Nader, when he dug up the old CBoxV3 and started working on the waveform generator units, and it was then under his supervision, I joined the project to work on the Master Controller Unit. I would like to thank **Prof. Koen Bertels** and **Prof. Carmina Almudever** for always encouraging and guiding me to successfully complete my thesis, and **Prof. Leiven Vandersypen** from the Applied Physics department for allowing me to collaborate with the Spin Qubit group. They gave me the opportunity to involve myself in a number of academic activities (both inside and outside of TU Delft) and involved me in the discussions of the entire Quantum and Computer Engineering (Q&CE) department. It is my honour and luck to be able to complete my master project under their supervision.

As a part of Q&CE and QuTech, I got to involve myself in the overall dynamics of the entire Quantum research community. The multiple discussions on all the related topics, be it theoretical or experimental, were always interesting and motivating to deliver a nice contribution to this research field; I truly felt part of the team.

I would like to thank everybody in the Q&CE department for all the discussions and fun moments in and around the 10th floor of the faculty of EEMCS. Special thanks to **Pinakin Padalia**, **Aritra Sarkar** and **Ahmed Abid Moueddene** for the great support and the in-depth discussions that helped me get the maximum out of my 'wide' project connecting the different layers of the Quantum Full-Stack. I would like to thank **Xiao Xue** and **Anne-Marije Zwerver** for their guidance in understanding the experimental details of the Spin-Qubit Control. I would like to thank my colleagues from the Quantum Computer Architecture Lab for all the discussions and fun moments. Special thanks to **Lingling Lao, Jeroen van Straten, Matthijs Brobbel, Dr. Ben Criger, Dr. Imran Ashraf, Dr. Xiang Fu, Leon Risebos, Dr. Savvas Varsamopoulos, Vieri Mattei, Miguel Moreira, Anneriet Krol, Neil Eelman, Daniel Moreno Manzano, Alejandro Morais** and **Diogo Valada** for all the spontaneous discussions that help me better understand quantum computing. I would also like to specially thank, **Joyce van Velzen** and **Anneke Verkerk** for helping me fulfill all the academic/administrative requirements with the group, and during my work as an RA and TA.

I would like to thank my fellow batchmates and friends at TU Delft, **Shivanand, Aurojyoti, Pradeep, Chirag, Adnan, Surya,** the complete ME house (**Abhairaj, Rishabh, Shardul, Shubham, Shrinidhi, Anirudh, Smit and Yash**), my 246 corridor-mates (**Shivam, Bhavya, Nimit, Knut and Sushmitha**) for those cherished moments during these two years.

Finally, I would like to thank **Maa and Papa**, **my beloved family, my professors, and all the friends** who have given me unconditional love and support over the last two years. During my time studying abroad, I can always rely on them to overcome the bad times in this period. Thank you all very much!

CC-Spin: A Micro-architecture design for scalable control of Spin-Qubit Quantum Processor

Abstract

Quantum Computing is an emerging field of technology with the promise that engineered quantum systems can address hard problems such as problems with exponential compute complexity in Chemistry, Genomics, Optimization and many more applications. Quantum Architecture is an area of research targeted for the NISQ-era quantum computing and little research has been done for development of a scalable classical control and read-out infrastructure for the quantum processors. The project is aimed at study of SoC/FPGA design methodology and architecture design for control of quantum processor. The targeted quantum hardware is the silicon spin-qubit quantum chip. The project is intended for understanding the design and working of a silicon-spin qubit for a computer (architecture) engineer. It further helps identify necessities for an architecture, Instruction Set requirements, bottlenecks and future challenges (specific to silicon spin quantum processor) that would help in better designs for new control architectures.

The objective of this thesis is directed towards addressing the architectural challenges for the quantum-classical hardware for controlling the NISQ-era quantum devices and beyond. We analyze the control infrastructure requirements and propose a micro-architecture and waveform generation methodology to integrate the physical device with the quantum compilation tool-chain.

CONTENTS

List of Figures xi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 A Full-stack Quantum Accelerator . . . 4
- 1.3 Objective . . . 6
- 1.4 Problem Solving with Quantum Computers . . . 6
- 1.5 Thesis Organization . . . 8

2 Quantum Technologies and Control 9
- 2.1 Quantum Circuit Model for Computation . . . 9
  - 2.1.1 Multiple Qubits . . . 10
  - 2.1.2 Measurements . . . 10
  - 2.1.3 Quantum Gates . . . 10
- 2.2 Spin-Qubits in Quantum Dots . . . 12
  - 2.2.1 2-Qubit processor in Si/SiGe Quantum Dot . . . 13
  - 2.2.2 Experimental Setup and Control . . . 14
  - 2.2.3 Scalable Physical Architectures in Spin Qubits . . . 16
- 2.3 Superconducting Qubit Control . . . 16
- 2.4 Quantum Control Hardware . . . 18
- 2.5 Si-Spin vs Superconducting Qubits . . . 20
- 2.6 Next level upgrades . . . 21
- 2.7 Quantum Control: A survey . . . 22
  - 2.7.1 Sinara Hardware . . . 22
  - 2.7.2 Raytheon APS2 and QDSP Hardware . . . 23
- 2.8 Conclusion . . . 24

3 CC-Spin Quantum Micro-architecture Implementation 25
- 3.1 Introduction . . . 25
- 3.2 Micro-architecture Overview . . . 26
  - 3.2.1 QISA . . . 28
  - 3.2.2 Quantum Pipeline . . . 32
  - 3.2.3 Quantum Microcode Unit . . . 35
  - 3.2.4 Timing Precise Execution Unit . . . 36
  - 3.2.5 Qubit Measurement . . . 36
  - 3.2.6 Comments on Micro-architecture for NISQ-era . . . 37
- 3.3 Enclustra Hardware . . . 37
- 3.4 Waveform Generation . . . 38
  - 3.4.1 IQ Signals . . . 38
  - 3.4.2 AWG Design Principles and Requirements . . . 39
  - 3.4.3 Using Programmable AWG . . . 40
  - 3.4.4 Using DDS based AWG Architecture . . . 41
  - 3.4.5 Multi-Device Synchronization . . . 44
- 3.5 Conclusion . . . 45

4 Hardware Testing & Results **47**
4.1 Implementation of FIFO based Timing Control Unit . . . 48
4.1.1 Asynchronous FIFO based Timing Control Unit . . . 48
4.1.2 Gray Code Counter Synchronization . . . 48
4.1.3 Implementation and Simulation . . . 49
4.2 Implementation of High-Speed LVDS links . . . 50
4.2.1 8b/10b encoder/decoder . . . 50
4.2.2 Implementation . . . 51
4.3 Implementation of FPGA controlled DAC-based AWG . . . 52
4.3.1 Using SignalTap . . . 52
4.3.2 Hardware Implementation . . . 53
4.4 Implementation of DDS-based AWG . . . 55
4.4.1 Functioning of DDS-Core . . . 56
4.4.2 DDS Serial Programming using Slave FPGAs . . . 56
4.4.3 Implementation . . . 57
4.5 Conclusion . . . 58

5 Conclusion & Future Work **61**
5.1 Implementation of CC Spin Micro-architecture . . . 61
5.2 Future Work . . . 62

A HDL Codes **65**
A.1 Binary-to-Gray Code Counter . . . 65
A.2 Asynchronous FIFO Design . . . 66
A.3 Quantum Pipeline . . . 73
A.4 True DAC based waveform generation demonstration . . . 75

B Appendix B: Set up and Use Instructions of SoC-FPGA **79**
B.1 Altera Quartus Prime . . . 79
B.2 SoC Design Guidelines . . . 80
B.2.1 In Qsys . . . 81
B.2.2 The Hardware . . . 82
B.2.3 Programming Abstraction on how we can get to the FPGA . . . 82
B.2.4 Cyclone V and Memory Mapping . . . 82
B.2.5 Using Quartus and QSYS . . . 83
B.2.6 Verilog — A Quick Recap . . . 84
B.3 SoC FPGA Design flow for Enclustra SoC-FPGA platform . . . 86

C Appendix C: Theory of Spin Qubits in Quantum Dots **89**

Bibliography **93**

LIST OF FIGURES

1.1 Algorithms to Quantum Machine Gap (Credits: Fred Chong, *"Closing the Gap between Quantum Algorithms and Machines with Hardware-Software Co-Design"*, Distinguished Lecture, UCLA, 2019) . . . 2
1.2 Ending Moore's Law . . . 3
1.3 System architecture with heterogeneous accelerators . . . 5
1.4 QCA Lab's Full-Stack Quantum Accelerator . . . 5
1.5 The Quantum Complexity classes, BQP and QMA. . . . 7

2.1 Spin-Qubit Chip . . . 13
2.2 Schematic of Si/SiGe double quantum dot device. . . . 13
2.3 Experimental Setup for Control of Two-Qubit Quantum Processor . . . 15
2.4 Scalable physical architectures in Spin Qubits: 8 qubits linear array (left) and 2×2 array of qubits (right) . . . 16
2.5 Transmon qubit in planar circuit-QED chip and the zoom in view of the Josephson Junction. Qubit (Q), resonator (R), flux-bias line (P_F), feedline input (P_i), and feedline output (P_o). Credits: *DiCarlo Lab, QuTech/TU Delft.* . . . 17
2.6 I/Q envelopes for X_π and Y_π rotations . . . 17
2.7 Experimental Setup for Control of Superconducting Quantum Processor . . . 18
2.8 Silicon Spin Qubit Control Rack (left) and Cryogenic Fridge (right) Setup. Credits: Vandersypen Lab, QuTech/TU Delft. . . . 19
2.9 Quantum Control Hardware. Credits: Vandersypen Lab, QuTech/TU Delft. . . . 19
2.10 Summary of Signals used in varous Qubit Technologies [1] . . . 21
2.11 Simplified Sinara Architecture . . . 22
2.12 Sinara Hardware ecosystem . . . 23
2.13 Raytheon APS2 System . . . 23

3.1 Full-Stack Quantum Instruction Execution Flow . . . 25
3.2 The system view of CC-Spin implementation using Arbitrary Waveform Generators (AWGs) - Modular AWG and/or DDS-DAC . . . 27
3.3 (a) Pulse Generation Timing Control, and (b) A general Instruction Format . . . 29
3.4 32-bit ISA (Design 1) uses Mask Index and Q-Channel Mark to construct a 32-bit channel mask (to address 32 qubits simultaneously) . . . 30
3.5 32-bit ISA (Design 2) - Using LUT for Channel Mask Expansion (10 bit address fetches the channel mask for 2^{10} = 1024 qubit control channels) . . . 31
3.6 eQASM format of Target Specify Instructions (top two), and Quantum Bundle (bottom three) [2] . . . 31
3.7 Implementation of Quantum control Pipeline using Arbitrary Waveform Generators. . . . 33
3.8 Simulation of Quantum Pipeline until Instruction Router . . . 35
3.9 Simulation of Quantum Pipeline with two FIFOs in the Timing Control Unit . . . 36
3.10 A general Quantum Approximate Optimization Algorithm architecture for Genome Sequencing Application (Credits: Aritra Sarkar, QCA/QuTech) . . . 37
3.11 I and Q signals phase shifted by 45° and modulated waveform . . . 39
3.12 Waveform Generation using a True AWG Architecture . . . 41

3.13 Waveform generation using DDS . . . 42
3.14 Detailed Block Diagram of AD9910 1GSPS, 14-bit, 3.3V CMOS DDS-DAC [3] . . . 43
3.15 Waveform Generation using a Direct Digital Synthesis DDS-DAC . . . 44
3.16 Implementation of Synchronized Waveform generation using AD9910 DDS-DAC . . . 45

4.1 Schematic of Experimental Setup . . . 47
4.2 Block diagram of Asynchronous FIFO . . . 49
4.3 Functional Simulation of Asynchronous FIFO . . . 49
4.4 8b/10b encoding . . . 51
4.5 Schematic Serial Data Transmission firmware using ALT_TX and ALT_RX SERDES Megafunctions . . . 51
4.6 Using Logic Analyzer to access data for single-ended and LVDS signalling . . . 52
4.7 Two-Cycle Multiplier-Based Architecture Timing Diagram . . . 52
4.8 Real-time ADC and DAC waveform obtained through Signal Tap . . . 53
4.9 sine waveform and Normalized Spectral Plot of DAC, generated using Signal Tap and plotted in MATLAB . . . 53
4.10 RTL Design of DAC-ADC implementation on Terasic DE10 Standard SoC-FPGA . . . 54
4.11 DAC based waveform generation Setup . . . 54
4.12 Output waveforms using Terasic Cyclone V and DAC board . . . 55
4.13 AD9910 DDS Evaluation Board Block Diagram . . . 55
4.14 AD9910 DDS Core Functional Block Diagram (Credits: AD9910 Data-sheet) . . . 56
4.15 AD9910 DDS Evaluation Board Setup . . . 58
4.16 Filtered Waveform Output AD9910 DDS-DAC Evaluation Board . . . 58

A.1 Quantum Program Execution Flow . . . 73
A.2 Filtered Waveform Output AD9910 DDS-DAC Evaluation Board when operated in RAM mode. . . . 74
A.3 Filtered Waveform Output AD9910 DDS-DAC Evaluation Board when operated in RAM mode-II . . . 74
A.4 IQ Sin wave 25MHz, V_{pp} = 2V from Terasic Cyclone V with DAC board . . . 77

C.1 2DEG in a Spin-Qubit quantum dot device. . . . 90
C.2 Initialization of Spin Qubit . . . 90
C.3 Spin-Qubit Read-out. . . . 91

Acronyms

ASIC Application Specific Integrated Circuit. 6

AWG Arbitrary Waveform Generator. 6, 26

CTPG Code-word Triggered Pulse Generation. 31

DAC Digital to Analog Converter. 6

DDS Direct Digital Synthesis. 21, 26

FPGA Field Programmable Gate Array. 6

FS Full-Stack. 25

HPS Hard Processing System. 38, 63

ISA Instruction Set Architecture. 28

LUT Lookup Table. 30

LVDS Low Voltage Differential Signalling. 34

MSB Most Significant Bit. 56

NISQ Noisy Intermediate Scale Quantum. vii, 1, 6, 8

QAOA Quantum Approximate Optimization Algorithm. 37

QCA Quantum Computer Architecture. 4

QEC Quantum Error Correction. 21

QISA Quantum Instruction Set Architecture. 25, 26

SERDES Serializer-Deserializer. 34

SoC System-on-Chip. 6

VQ Variational Quantum. 37

1

INTRODUCTION

> “Since the laws of quantum physics are reversible in time, we shall have to consider computing engines which obey such reversible laws."
> - Richard P. Feynman

Quantum Mechanics, to this day, remains one of the most successful yet most mysterious of the scientific theories ever developed. It's concrete foundations were laid during the period 1900 to 1920 by Erwin Schrodinger, Werner Heisenberg, Max Born among many others, and since then it has been used successfully in understanding particle physics and fundamental forces of nature. An interesting turn of events took place in 1970s when scientists started inquiring how information theory and fundamental computer science can be applied to quantum systems. The result was a new perspective that gave rise to questions, like: what are the fundamental physical limitations on space and time required to construct a quantum state? What makes quantum states difficult to understand and simulate by conventional classical means?

The foundations of quantum computing were laid in early 1980s by Richard Feynman [4, 5], Yuri Manin [6, 7], Paul Benioff [8, 9] and David Deutsch [10]. Feynman introduced the idea of "universal quantum simulator" with controllable quantum particles that can be used to simulate other quantum systems [4]. Deutsch presented a “universal quantum computer" using quantum parallelism [10]. In 1992, Deutsch and Josza presented the first quantum algorithm that surpassed it's classical counter-part and included entanglement [11]. Peter Shor (in 1994) demonstrated an algorithm for two major problems: the problem of finding prime factors of an integer and, the 'discrete logarithm' problem, that can be solved efficiently on a quantum computer [12]. This sparked a great interest in the scientific community as this was the first practically viable algorithm and could directly break the present-day public-key cryptography such as, the RSA-algorithm. In 1996, David DiVincenzo published a standard set of conditions for building a quantum computer, known as DiVincenzo's criteria [13].

Though we are far from constructing quantum computers that can irrefutably demonstrate quantum supremacy [14] by solving problems that are intractable on classical computers, the present day, Quantum Computers have matured enough to demonstrate noisy experimental realizations using different qubit technologies on which, low-depth quantum algorithms can be executed. The most popular implementations include Superconducting Qubits, Spin-qubits in semiconductor Quantum Dot, Trapped-ion Qubits, Nitrogen-Vacancy centers in diamond etc. The concept of utilizing noise-prone qubits (without quantum error correction) with limited-depth quantum circuit definition capable of performing useful calculations is termed as Noisy Intermediate Scale Quantum (NISQ) computing [15].

Alongside physical qubit representation, an active research is also being carried out in the development of quantum algorithms, quantum-classical high-level programming languages, compilers and quantum simulators. However, less focus has been placed on development of middle-ware necessary to connect the high-level abstraction to efficiently control the physical qubits. Figure 1.1 shows the trend in the number of qubit count over the years, and the minimum number of qubits required to execute some of the near-term quantum algorithms. As we progress towards higher qubit count, we would need a hardware/software co-design framework to address each physical qubit.

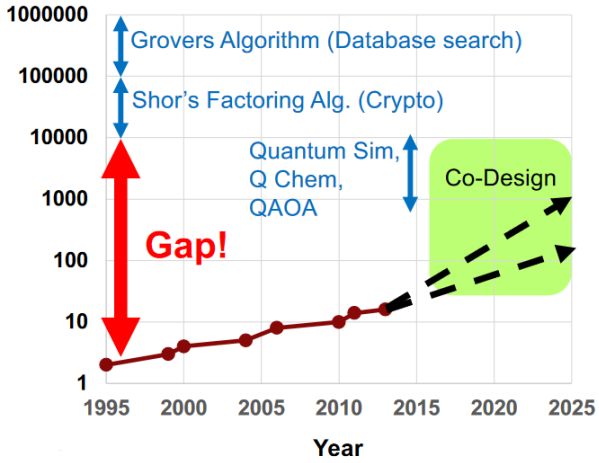


Figure 1.1: Algorithms to Quantum Machine Gap (Credits: Fred Chong, *"Closing the Gap between Quantum Algorithms and Machines with Hardware-Software Co-Design"*, Distinguished Lecture, UCLA, 2019)

1.1. MOTIVATION

Since the inception of electronic computing in late 1940s, the race to innovate and bring new technologies to the table has been unprecedented. The advances made in CMOS-based designs allowing integration of more number of transistors per unit area on a silicon die has provided the programmers, the independence and flexibility to create highly complex designs for computations. The rate of increasing integration was first observed by Intel co-founder, Gordon Moore in his 1965 paper [16], where based on the trend from 1959 - 1965, he predicted, *"there is no reason to believe it will not remain nearly constant for at least 10 years"*. It turned out that the prediction has been true for more than half-century and had been accepted as an 'economical law'. **Moore's Law** is the observation that the number of transistors on integrated circuits doubles approximately every two years. This is shown in figure 1.2 (Transistors trend).The miniaturization of transistor feature-size is what has enabled the larger computation power (smaller size transistors meant faster switching speeds), and larger memory capacity. The law has helped as a coordination device to plan ahead of time and develop future technologies and applications.

In 1974, Robert Dennard [17] formulated that, roughly as transistors get smaller, the necessary voltage and current is also scaled downward. Thereby, power stays proportional to the area of the transistor. This is known as **Dennard Scaling**. Therefore, with transistor's shrinking sizes, the voltage was reduced and circuits operated at higher frequencies at the same power. Moore's Law along with Dennard scaling, meant that Performance-per-Watt grows at the same rate, doubling about every two years. This trend is referred to as **Koomey's Law**.

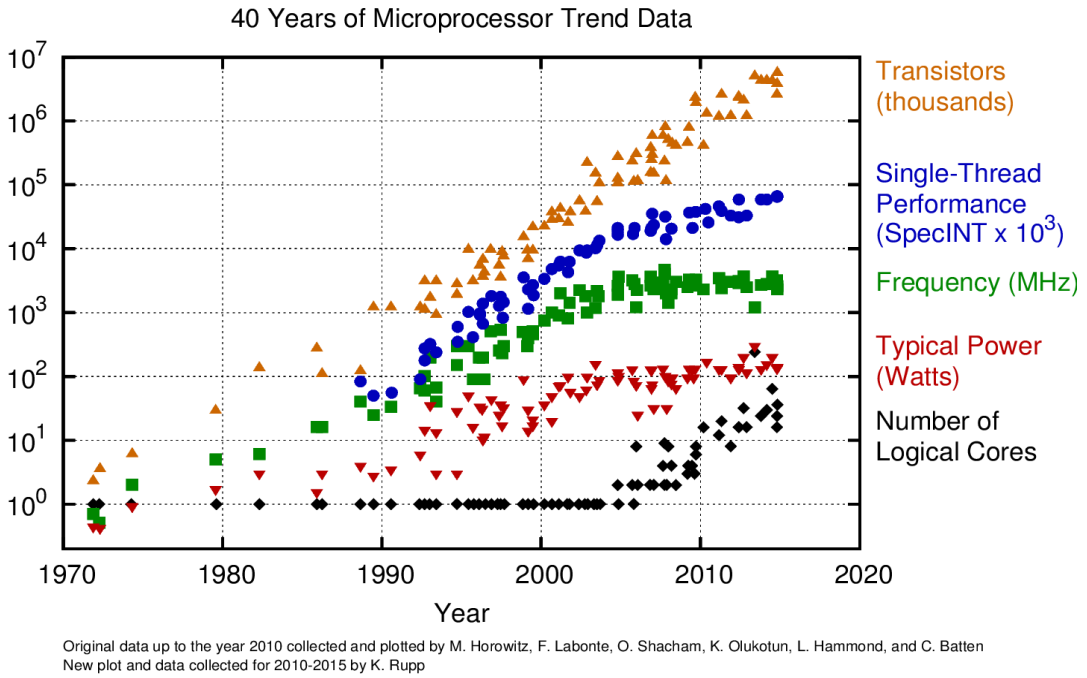


Figure 1.2: Ending Moore's Law

Since 2005, Dennard scaling (figure 1.2) started to break down. The primary reason for this is the Static Power Dissipation. Power dissipation is expressed as:

$$P = ACV^2 f + V I_{leak}$$

$$f_{max} \propto \frac{(V - V_t)^2}{V} \qquad I_{leak} \propto exp(-\frac{qV_t}{kT})$$

To increase the frequency, f_{max}, Threshold voltage V_t has to be reduced. Reducing V_t implies increase in I_{leak}, thus directly increasing static power dissipation. Therefore, we hit a *Power Wall*. At smaller sizes, current leakage causes the chip to heat up, which creates a risk of thermal runaway. As of 2018, Moore's law still appears to continue, but frequency has become constant and single-thread performance increase has also been gradual.

Emergence of Multi-core In the beginning of the 21st century, single-core had become too complex. The breakdown of Dennard scaling meant that increasing clock frequencies was no longer a viable solution to get better performance. Further, it was not possible to extract more Instruction Level Parallelism (ILP) (*ILP Wall*) and also, the gap in the processor-memory performance increased (*Memory Wall*). This lead chip manufacturers to turn to multi-core processor architectures as an alternative which led to the emergence of parallel computing. While multiple processor cores help in parallelism, the speedup is limited by the serial part of the program. **Amdahl's Law** helps to predict the speedup of applications when multiple processors are used. The law is given as follows:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

where, S is the maximum speedup possible using N processors. P is the proportion of program that can be made parallel, and *(1-P)* is the proportion that remains serial.

By increasing the core count, we benefit for parallel workloads, but it still results in increased overall power consumption. Therefore, to meet the power constraints it is required to keep only a

fraction of the chip active at any given point in time. The remaining (inactive) area is called *dark silicon*. Therefore, multi-core is not a ideal solution.

The homogeneous multi-core processors still dominates but companies like, Intel, Xilinx and Nvidia observed that heterogeneity was the right way forward to improve the computing power. GPU and FPGAs have helped execute huge workloads and are now seen as natural extensions of computer architecture. Such an architecture allows developing custom hardware architecture to achieve massive parallelism.

Alternate technologies and emergence of quantum computing Today's High Performance Supercomputers operate in PetaFLOPS (10^{15}) which is still not enough to meet the requirements for many scientific computing applications. These employ complex distributed architecture of compute units, but nevertheless, are based on (mostly) CMOS technology. The current state of the art transistor gate-length fabrication sizes has reached 14nm based on FinFET technology. As we proceed towards the limits of miniaturization of transistor gate lengths, quantum effects start to emerge rendering the correct transistor functioning no longer possible.

This raises scientific curiosity towards 'the next big revolution' in computing and thus, emerges an active interest to look into *Beyond-CMOS technologies*. Quantum Computing is one such highly investigated fields of research that rethinks computing methodology at most fundamental levels by employing quantum effects of 'superposition' and 'entanglement' to solve classically intractable problems.

1.2. A FULL-STACK QUANTUM ACCELERATOR

The concept of accelerator technology originates from the idea that any end-application contains multiple parts, and the properties of these parts are better executed by a particular accelerator (such as, FPGA, GPU, DSP, TPU etc). This is shown in figure 1.3. An accelerator has a co-processor (Device) communicating to a central processor (Host). The Device executes certain parts of the overall application much faster than the Host. Similarly, quantum processors are good at solving certain parts of the problem that are inefficient to be solved by classical processors. Furthermore, the quantum assembly language such as proposed in [18] and [19] allows user to specify the complete quantum algorithm - containing both classical computational elements and a 'quantum kernel'. The classical processor keeps the control over the total system and delegates the execution of certain parts to the present accelerators. The functioning of such an accelerator requires abstracting away the low-level (qubit technology) details from the algorithm developer and integrate the software (programming language and compiler) to the hardware (qubit control electronics, cryogenic electronics and qubits themselves).

Given the proposition of Noisy Intermediate-Scale Quantum technology [15], the Quantum Processing Unit (QPU) would prove to be elemental in accelerating calculations for numerous applications such as, simulation of molecules, Protein Folding, Genome Sequencing, Quantum Field Theory simulations, Quantum Machine Learning and many others. In order to execute complex algorithms, we require a system architecture connecting quantum circuit descriptions to the low-level pulses that operating on the physical qubits. Figure 1.4 is such an example of a 'full-stack' system architecture developed by the Quantum Computer Architecture Lab in Delft.

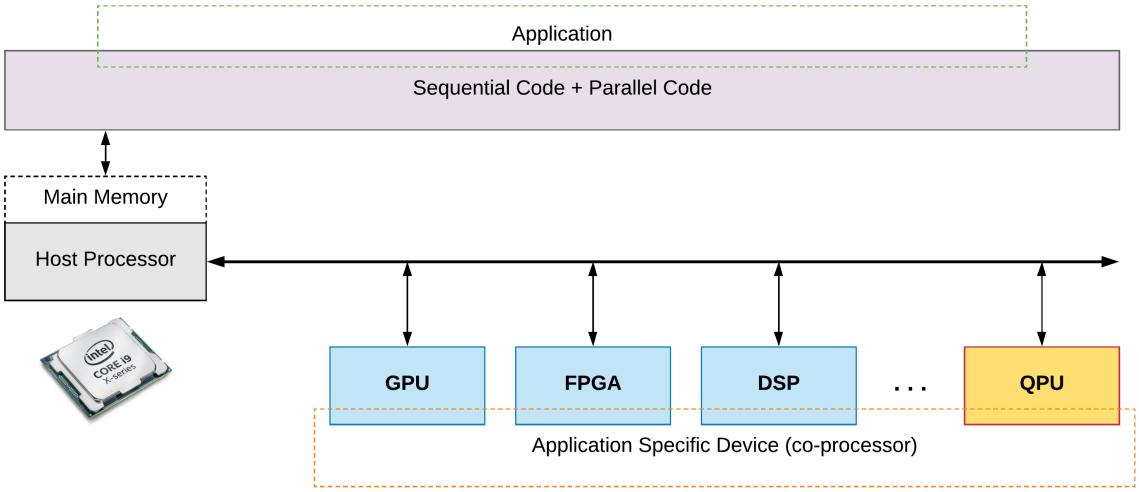


Figure 1.3: System architecture with heterogeneous accelerators

The quantum compilation toolchain consists of the OpenQL compiler, that translates circuit-level algorithmic descriptions and high-level quantum and classical libraries (for example, QFT, classical optimizers etc.) to a Quantum Assembly Language (cQASM) description, a low-level compiler translates the cQASM description to a platform-independent Quantum Instruction Set Architecture (QISA). The QISA layer contains both classical and quantum instructions. The micro-architecture layer constitutes the implementation of QISA, and is responsible for translating the eQASM instructions to analog waveform with precise timing specifications while being synchronous with multiple channels. The Quantum-to-Classical layer forms comprises of a analog electronic devices that is the interface for analog signals to act upon the physical qubits on the Quantum Chip.

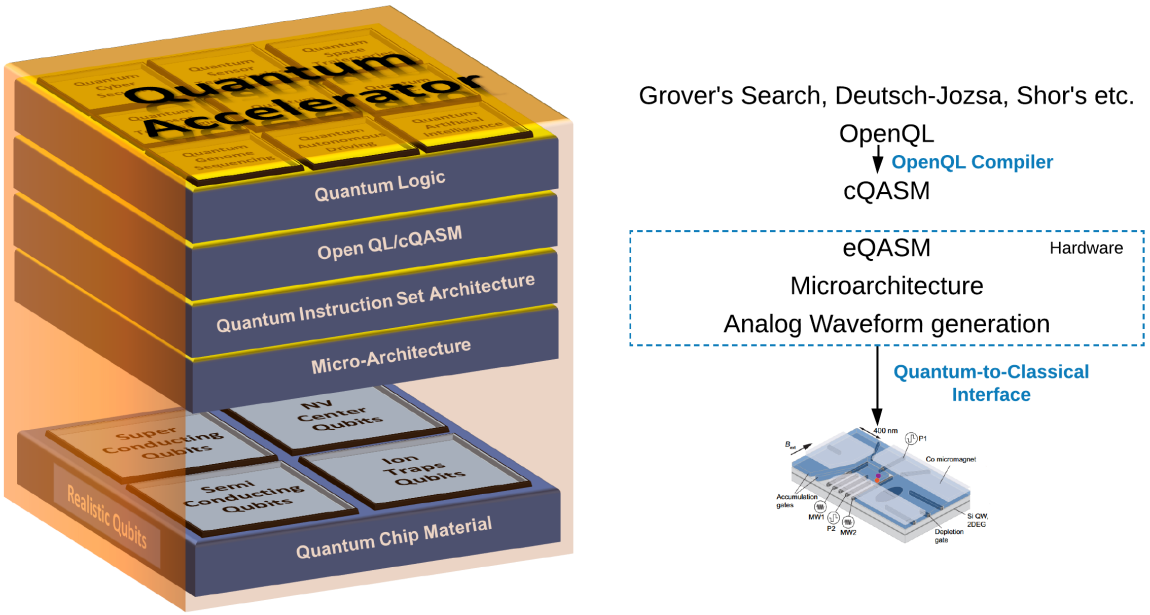


Figure 1.4: QCA Lab's Full-Stack Quantum Accelerator

1.3. OBJECTIVE

The objective of this thesis is directed towards addressing the architectural challenges for the quantum-classical hardware for controlling the NISQ-era quantum devices and beyond. We take into account two of the promising circuit-model based quantum technologies of the future namely, superconducting qubits and Spin-qubits in Quantum-dot, analyze the control infrastructure requirements and propose a micro-architecture to integrate the physical device with the quantum compilation toolchain.

The architecture prototype is developed on a SoC-FPGA platform in the form of a central-controller hardware, CC-Spin. The current version is targeted towards Spin-Qubit in Quantum Dot technology (hereafter referred to as, "Spin-qubit quantum processor"). The central controller interfaced to modular AWG units perform the waveform generation such that, the number of DAC channels scale linearly with the number of AWG modules on the central controller.

The results of the micro-architecture are presented as waveform generations applicable to Spin-Qubit achieved on an oscilloscope, thereby validating the applicability of the architecture to a Spin-qubit quantum processor. Towards the end, we analyze the scalability of such an architecture based on the parameters of number of channels, channel capacity and ability to be integrated as an ASIC.

The integration of such a micro-architecture is relevant not only to theorist and programmers but to the experimentalist as well. The micro-architecture reduces the high-resource consumption, reduces the control complexity, and allows faster prototyping and execution of experiments.

1.4. PROBLEM SOLVING WITH QUANTUM COMPUTERS

> “Quantum computers would not solve hard search problems instantaneously by simply trying all the possible solutions at once."
> - Scott Aaronson

Alan Turing in his 1936 paper suggested a model of computation, now known as 'Turing Machine'. Alonzo Church's and Alan Turing's work stated that, *any algorithmic process can be simulated efficiently using a Turing Machine*. This is known as the *Church-Turing Thesis*. While classical computers broadly accepted as 'Universal Turing Machines' are capable of simulating any other Turing Machines, it is impossible to do so *efficiently*. An efficient algorithm is one that executes in time polynomial in the size of the problem solved, while an inefficient requires super-polynomial (usually, exponential) time.

Church-Turing Thesis held strong until 1970s when when Robert Solovay and Volker Strassen presented their test for determining an integer being prime or composite using a *randomized algorithm*. Until his point the Turing Machine as a deterministic one that can efficiently solve algorithmic problems. However, randomized algorithms showed that there are efficiently solvable problems that cannot be solved efficiently on a Turing Machine. This challenge lead to a modification in the Church-Turing Thesis:

Any algorithmic process can be simulated efficiently using a probabilistic Turing Machine.

The idea to discover a single model of computation that could efficiently simulate problems that cannot be efficiently solved on the Turing machine motivated David Deutsch in 1985 to investigate the laws of physics to derive such a model of computation. The idea that the laws of physics are quantum mechanical, led to the proposition of building a computer based on the laws of quantum mechanics. The quest for searching problems that can be efficiently solved on a quantum computer (but have no efficient solution on (deterministic or probabilistic) Turing Machine) was a challenge at the time.

Deutsch-Jozsa Algorithm proposed in 1992 is one of the first examples of a quantum algorithm that is exponentially faster than any possible deterministic classical algorithm. However, the

remarkable result developed by Peter Shor in 1994 was the demonstration of quantum computational solution to two important problems — finding prime factors of integers and the discrete logarithm problem. Shor's Algorithm is a powerful result that shows quantum computers are more powerful than Turing Machines, and even probabilistic Turing Machines. Later, results by Lov Grover on performing unstructured database search further strengthened the proposition. Though the speed up is not spectacular, it has widespread applicability.

Though quantum computers can solve a number of problems efficiently that classical computers cannot, there still exists an inexhaustive list of problems that classical computers can solve, given enough time and memory (albeit never practically possible); and Turing machine can simulate quantum computers. Therefore, quantum computing model does not disprove the Church-Turing thesis.

To guess the exact type of problems that can be solved efficiently on quantum computers is hard. While *Complexity Theory* in computer science helps determine classes of algorithms that are efficiently solvable on classical computer, it's quantum counterpart, *Quantum Complexity Theory* lays the foundations of classes of problems that can be efficiently solved on a quantum computer. Figure 1.5 shows my reinterpretation based on [20] the classes of problems in Complexity theory and their relationship.

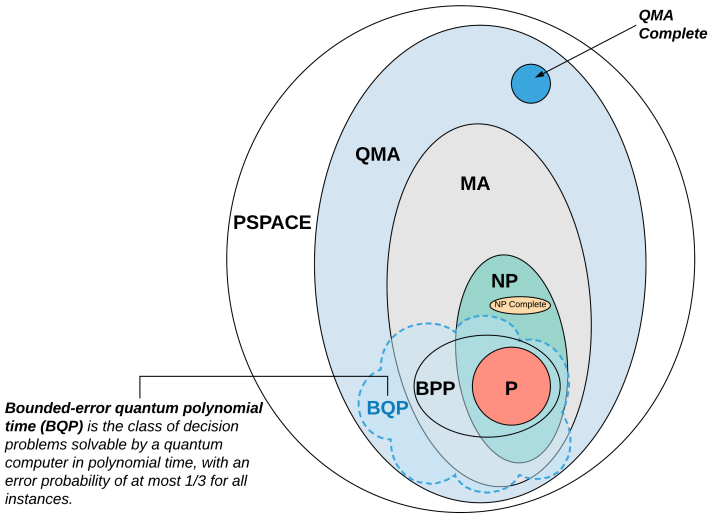


Figure 1.5: The Quantum Complexity classes, BQP and QMA.

Two of the important quantum complexity classes are BQP and QMA. These are the quantum analogue of classical complexity classes P and NP, respectively.
BQP (Bounded-Error Quantum Polynomial Time) class contains the set of decision problems that are efficiently solvable by quantum computers within a bounded error. Shor's factoring algorithm, discrete logarithm, simulation of quantum systems are problems in the BQP class.
QMA (Quantum Merlin Arthur) is the quantum complexity class containing the set of decision problems for which when the answer is Yes, there exists a quantum proof of polynomial-size that convinces a quantum verifier of this fact in polynomial-time with a high probability. And if the answer is No, all the polynomial-size quantum proof gets rejected by the quantum verifier with a high probability.

At the time of writing of this thesis, quantum computing applications are being investigated for non-trivial problems such as, quantum chemistry[21], pattern matching in genomics[22], quantum associative memory (QuAM) in high energy physics [23], quantum field theory simulations [24], cryptography [25], optimization problems (QAOA) [26] etc. However, one of the major applications of quantum computers in the coming decades would be to simulate quantum mechanical systems

that are too difficult to simulate on classical computers, as originally suggested by Richard Feynman in 1982.

1.5. THESIS ORGANIZATION

The thesis is divided into chapters intended to understand the requirement specifications for the development of a micro-architecture to perform control and read-out of a Spin-Qubit quantum processor.

In Chapter 1, We introduced the motivation behind taking the road to quantum computation, the full-stack approach and defining the class of problems quantum computers can solve better than the classical computers. We stated the objective of our thesis and it's role in the NISQ-era.

In Chapter 2, we will briefly introduce the quantum circuit model for universal quantum computation to provide the necessary background for the thesis. Then we will present the details of experimental realization of a Spin-qubit quantum processor. We also present a brief overview of Superconducting Quantum Processor along with low-level comparison of signal specifications for qubit control. We conclude the section with a survey of available control hardware platforms that are used for present-day quantum computers.

Chapter 3 presents a detailed design for CC-Spin quantum micro-architecture, an Instruction Set Architecture, and the quantum-classical pipeline and Waveform Generation methodologies using modular AWGs and DDS-DAC based AWGs. We also address the scalability aspects of the quantum micro-architecture and present the interpretation of requirements for near-term quantum computing and hardware bounds with respect to number of qubits.

In Chapter 4, the implementation of sub-components of the quantum micro-architecture, hardware implementation and testing methodology on SoC/FPGA is presented. We present the results as timing simulations and parameters of generated waveform on hardware implementation.

We conclude the thesis in Chapter 5, aggregating our findings and the proposing the required future work to integrate the micro-architecture with the quantum full-stack for NISQ-era and beyond.

2

Quantum Technologies and Control

In section 1.1, we discussed the diminishing performance returns drawback of technology scaling and multi-core processor architectures. As we scale down to the nanometres (nm) length-scales, the electromagnetic action scales approach Planck's constant ($h = 6.626 \times 10^{-34} J.s$). At that point, design must be based on quantum principles. A quantum computer inherently uses the laws of quantum mechanics notably, *quantum superposition*, *quantum entanglement* and *quantum tunnelling* to solve classically intractable problems. In this chapter we will discuss fundamentals of quantum information and control methodology of Spin Qubit in semiconductor Quantum-Dots Quantum Processor.

2.1. Quantum Circuit Model for Computation

The circuit model of quantum computing uses a sequence of quantum gates and measurement operators, to transform and measure the quantum states, respectively. Just like classical logic gates (e.g. AND, OR, NAND, NOT etc.) transform bits, the quantum logic gates are used to perform logical operations on qubits.

In quantum computing, a qubit is a quantum two-state physical system where (usually) the lower energy state is represented as Boolean 0, and the higher energy state is represented as Boolean 1. These states are written in the Dirac notation or *'bra-ket'* notation – $|0\rangle$ and $|1\rangle$ (pronounced as, ket 0 and ket 1, respectively). The qubit can also be in a *superposition* of the two states and this is represented as a linear combination of these states, as follows:

$$|\psi\rangle = \alpha_0|0\rangle + \alpha_1|0\rangle$$

The variables α_0 and α_1 are called *probability amplitudes* and are *complex numbers*. The probability of finding the qubit in state $|0\rangle$ is given by $|\alpha_0|^2$ and the probability of finding the qubit in state $|1\rangle$ is given by$|\alpha_1|^2$. The sum of these probabilities must be equal to 1.

$$|\alpha_0|^2 + |\alpha_1|^2 = 1$$

A qubit can be represented as a column vector in a 2-dimensional complex vector space. This means that state $|0\rangle$ and $|1\rangle$ can be represented in the vector notation, as follows:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Besides the *ket* notation, we also have *bra* notation, which is given by the conjugate transpose of the *ket column vector*. The *bra* notation is therefore a *row vector*. If $|\psi\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$ we will have $\langle\psi| = \begin{pmatrix} \alpha_0^* \alpha_1^* \end{pmatrix}$, where α_i^* is the conjugate of α_i.

The quantum states described are for a two-state system – $|0\rangle$ and $|1\rangle$. These states together form a *basis* and this pair is called the *computational basis states*. If the system is a d-level systems, we call these *qudits*.

2.1.1. MULTIPLE QUBITS

"Hilbert Space is a big place." - Carlton Caves

With an n-qubit system we can have an arbitrary superposition over 2^n basis-states with normalized complex amplitudes as coefficients and an irrelevant global phase. We can represent this as a vector in an n-dimensional Hilbert space. For example, with 2-qubits, we will have $2^n = 4$ computational basis states — $|00\rangle, |01\rangle, |10\rangle$and$|11\rangle$. This is represented as follows:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

In more general terms, a quantum state is represented as a sum of basis states, $|n_i\rangle$.

$$|\psi\rangle = \Sigma_i \alpha_i |n_i\rangle, \qquad \text{where } \alpha_i \in C^n$$

$$\Sigma_i |\alpha_i|^2 = 1$$

The sets of basis states are always *orthonormal*. This means that these basis states are unit vectors ($|\alpha_0|^2 + |\alpha_1|^2 = 1$) and are orthogonal to each other ($\langle\psi|\phi\rangle = 0$). With n, the dimension of the vector space C^{2n} increases exponentially. Therefore, the space C^d where $d = 2^n$ is also called the *state space* of n qubits. This is because the state space of two quantum states is combined by the *tensor product*.

$$|\psi_1\psi_2\rangle = |\psi_1\rangle|\psi_2\rangle = |\psi_1\rangle \otimes |\psi_2\rangle = \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_d \end{pmatrix} \otimes |\psi_2\rangle = \begin{pmatrix} \alpha_0|\psi_2\rangle \\ \vdots \\ \alpha_d|\psi_2\rangle \end{pmatrix}$$

With multiple qubits, we can also create *Entangled States*. Entangled states do not have a classical analogue. A joint state of two qubits can be $|\psi_1\rangle \otimes |\psi_2\rangle$. This state can be also called *separable state*, as we can break the components as tensor product of two individual states. Entangled state on the other hand, are states that are *not separable*.

Therefore, $|\psi\rangle$ is entangled iff, $|\psi\rangle \neq |\psi_1\rangle \otimes |\psi_2\rangle$, for any $|\psi_1\rangle$ and $|\psi_2\rangle$.

2.1.2. MEASUREMENTS

Quantum measurements are probabilistic in nature. This also presents the fact that qubits really are fundamentally different. A measurement, or readout operation, in some basis state 'collapses the wavefunction' to one of the basis states with some probability. This holds true for multiple qubits in computational basis as well. For example, measuring a general quantum state $|\psi\rangle = \Sigma_i \alpha_i |n_i\rangle$, we would readout the state $|x_i\rangle$ with probability $p_x = |\alpha_x\rangle|^2$.

2.1.3. QUANTUM GATES

Quantum computing can be considered as an In-Memory computing model. Quantum Gates (also called, Quantum Operators) in the circuit model quantum computing are unitary matrices that act on the data encoded in the qubits. A quantum gate acting on n qubits is a unitary matrix of size $2n \times 2n$. One of the important properties of circuit model is that the quantum gates are reversible. This means that the number of inputs to the quantum gate is equal to the number of outputs. A list of the commonly is single- and multi-qubit gates are given in table 2.1 and 2.2.

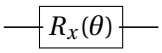
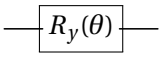
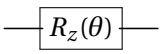
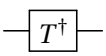
Name	Unitary	Circuit	Relations
Identity	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$		
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$		
Pauli-X	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$		X = iZY = HZH
Pauli-Y	$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$		Y = iXZ
Pauli-Z	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$		Z = iYX = HXH
Rotation-X	$R_x(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -isin\frac{\theta}{2} \\ -isin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$		
Rotation-Y	$R_y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$		
Rotation-Z	$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$		
T	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$		
T†	$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}$		$I = TT^\dagger$
Phase	$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$		$S = T^2$

Table 2.1: Commonly used Single-Qubit Gates

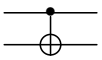
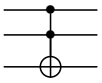
Name	Unitary	Circuit
Controlled NOT	$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	
SWAP	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	
Controlled Phase	$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$	
Toffoli	$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$	
Fredkin	$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	

Table 2.2: Commonly used Multi-Qubit Gates

In practical systems, a qubit is a physical object and is realized through different physical quantum mechanical systems. The Micro-architecture layer is the interface between the high level programming infrastructure and the low-level signal generation systems. This layer is interfaced to the qubits (at the Physical layer) via quantum-classical interface. Now we will introduce topics on control of two of the prominent technologies for realizing qubits, namely, Spin-Qubit in Semiconductor Quantum Dot and Superconducting Transmon Qubits.

2.2. SPIN-QUBITS IN QUANTUM DOTS

Quantum computing emerged from the idea that quantum degrees of freedom can be utilized to encode and process information. This lead the researchers to investigate ideas of fabrication, addressing and manipulation of single-quantum system, that could help in realization of the quantum computer. Today, we have many ways to realize single-quantum systems that can be used to encode qubits, however the challenge to solid-state quantum computation lies in eliminating unwanted noise, arising due to interaction of the the qubit with the environment and/or the host material.

Spin Qubits, first proposed in 1997 [27], is one of the Solid-state quantum computing methods that uses spins of electrons confined in quantum dots. For quantum dot based qubits, the charge

and nuclear spin noises lead to decoherence and gate errors. It has been demonstrated that these noises can be tackled by dynamical decoupling [28] and decoherence free subspaces[29, 30]. Noise can further be reduced and coherence time extended to seconds by growing better oxides and heterostructures[31], and by fabricating the spin qubit in Silicon (^{28}Si), reason being, it's naturally low abundance of nuclear spin isotopes which can be removed by isotopic purification[32]. Using such methodologies have resulted in high gate fidelities (99%) and the demonstration of two-qubit CZ (controlled phase) gate. Thereby having these ingredients, a two-qubit quantum processor can be fabricated in silicon. The details of Spin-Qubit in quantum dot theory along with it's advantages is given in appendix C.

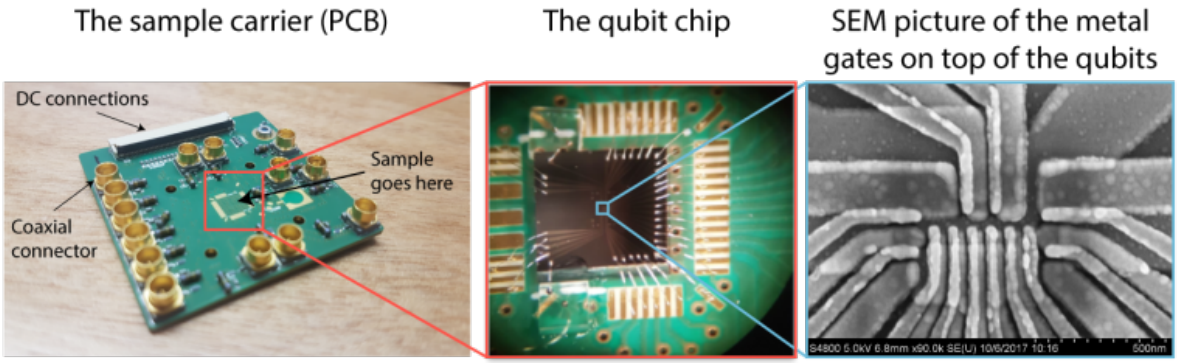


Figure 2.1: Spin-Qubit Chip

At Vandersypen Lab, two single-electron spin qubits in natural Si/SiGe double quantum dot (DQD) combining initialization, single- and two-qubit gate rotations and measurement has been demonstrated in 2018 [33]. Spin qubit in quantum dot based processors have demonstrated 99.9% fidelity in single qubit gates [34] and implementation of two-qubit gates [35]. While they have a huge potential for scalability, the current best quantum dot based processors are limited to two-, three- and four-qubit processors. In this chapter we will discuss the original Spin Qubit proposal (Loss-DiVincenzo Quantum Processor), the electrical control of Si/SiGe Spin-qubits in quantum dot, the requirements for a large-scale usable spin-qubit quantum processor and recent advances on scaling quantum dots to larger counts.

2.2.1. 2-QUBIT PROCESSOR IN SI/SIGE QUANTUM DOT

This section discusses the design for a two-qubit quantum processor in silicon as described in [33]. The schematic for the device is shown in Figure 2.2.

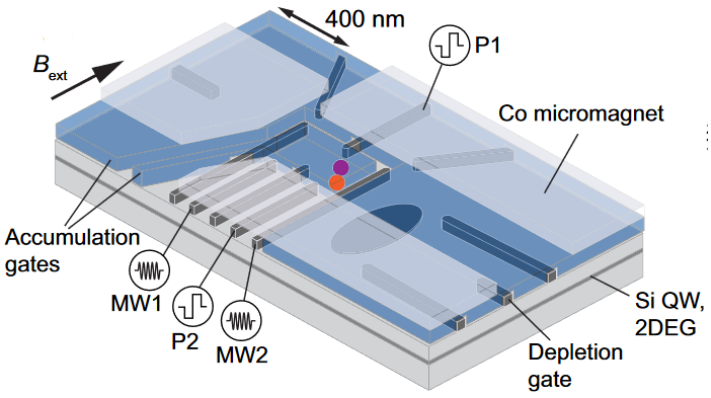


Figure 2.2: Schematic of Si/SiGe double quantum dot device.

Structural description of the Quantum Processor :

1. Accumulation Gates are used for the formation of two-dimensional electron gas (2DEG) in the quantum well of the Si/SiGe heterostructure.

2. Applying negative voltages to the Depletion Gates allows formation of double quantum dot (DQD) in the 2DEG. D1 and D2 are the approximate positions of the quantum dot (orange and purple dots, in Figure 2.2).

3. The external magnetic field $B_{ext} = 617mT$ causes the Zeeman Split (spin-down $|0\rangle$ and spin-up $|1\rangle$) in single electrons at D1 and D2.

4. The initialization and read out of Q2 is done by spin-selective tunneling to a reservoir. Whereas, Q1 is initialized by spin-relaxation hotspot and measured via Q2 using controlled rotation (CROT). The respective fidelities are given 2.3.

5. Three cobalt micromagnets are present on the top, that provide a magnetic field gradient having a component perpendicular to B_{ext}. This is used for Electric Dipole Spin Resonance (EDSR).

The magnetic field gradient from the cobalt micromagnets also helps separate the qubit frequencies ($f_{Q1} = 18.4GHz$ and $f_{Q2} = 19.7Ghz$) which allows individual control of the qubits. Rabi Frequencies of $f_R = \omega_R/2\pi = 2MHz$ is reported.

Single-Qubit gates are performed by IQ vector modulation [1] of microwave drive signals. X (and Y) gate rotations are $\pi/2$ rotations and X^2 (and Y^2) are π rotations, around $\hat{x}$ (and $\hat{y}$) on this processor platform.

The Two-qubit gate CZ (controlled phase) is also performed similarly. A table summarizing various qubit parameters is shown below, in table 2.3.

	Q1	**Q2**
Initialization Fidelity	>99%	>99%
Measurement Fidelity	73 %	81 %
T_1	$50ms$	$3.7 \pm 0.5ms$
T_2^*	$1.0 \pm 0.1\mu s$	$0.6 \pm 0.1\mu s$
T_{2Hahn}	$19 \pm 3\mu s$	$7 \pm 1\mu s$
Average Clifford Gate Fidelity	98.8 %	98.0 %

Table 2.3: Properties of qubits (Q1 and Q2) in (1,1) configuration (1 electron each in D1 and D2). Average Clifford Gate Fidelity is obtained by randomized benchmarking.

2.2.2. EXPERIMENTAL SETUP AND CONTROL

The experimental setup to control and measure two-qubit silicon-spin is shown in Figure 2.3.

We apply DC voltages to all the gate electrodes using room-temperature DACs via filtered lines. Tektronix 5014C AWG with 1 GHz clock rate is used to apply voltage pulses to plunger gates P1 and P2. These signals from AWGs pass through an RT low-pass filter and attenuators through different stages of the dilution fridge and get added to the DC signals via bias tees mounted on the PCB.

[1] Discussed in Section 3.4.1

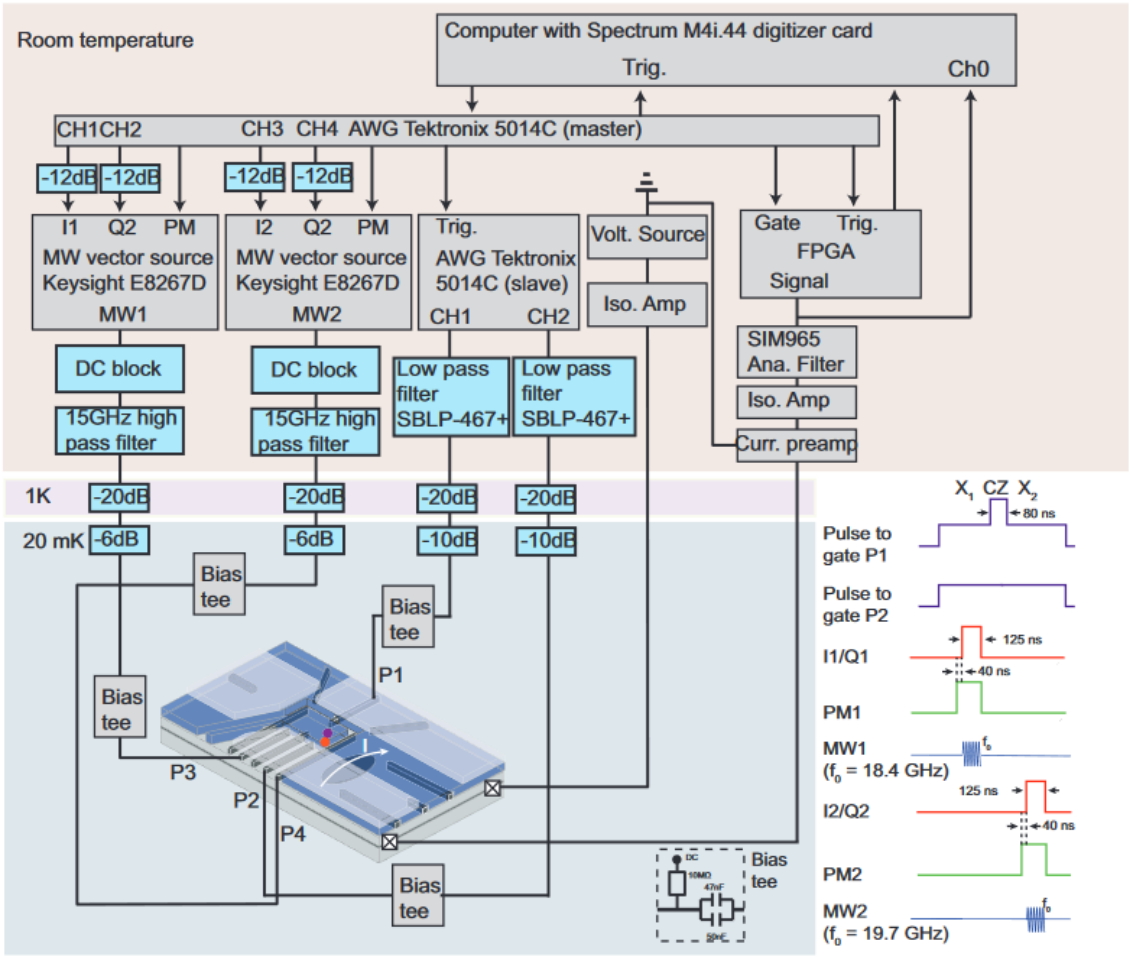


Figure 2.3: Experimental Setup for Control of Two-Qubit Quantum Processor

To perform EDSR on Q1 and Q2, two Keysight E8267D vector microwave sources, MW1 and MW2, are employed to generate microwave pulses (18 - 20 GHz) and are applied to respective qubits. The microwave signals pass through RT DC blocks, homemade 15 GHz high-pass filters, and attenuators at different stages of the fridge and are added to the DC signals via bias tees mounted on the PCB. I/Q vector modulation is used to control the output of the MW source (phase, frequency, amplitude, duration). Another Tektronix 5041C is used to generate I/Q signals which acts as the master device for the entire setup and provides trigger signals for the other devices. In addition to the vector modulation we employ pulse modulation to give an on/off microwave power output ratio of 120 dB.

While I/Q modulation can be used to output multiple frequencies, the bandwidth of the AWG was not enough to control both qubits with one microwave source due to their large separation in frequency (1.3 GHz). The sensor current, I, is converted to a voltage signal with a home-built preamplifier and an isolation amplifier is used to separate the signal ground with the measurement equipment ground to reduce interference. Following this, a 20 kHz Bessel low-pass filter is applied to the signal using a SIM965 analog filter.

An FPGA is used to analyze the voltage signal during the readout which assigns the trace to be spin-up if the voltage falls below a certain threshold. We can also use a Digitizer card via the PC to measure the voltage signal. The shape of the pulses generated by the AWGs and MW sources during qubit control with the typical timescales is also shown in Figure 2.3. Square pulses are used to perform the CZ gate and as the input for the I/Q modulation to generate MW pulses. The pulse

modulation is turned on 40ns before turning on the I/Q signal due to the time needed for the modulation to switch on.

2.2.3. SCALABLE PHYSICAL ARCHITECTURES IN SPIN QUBITS

Just like the performance of every individual transistor affects the performance of a processor, the performance of quantum processor depends on the quality of every qubit. Controlling electrostatically defined quantum-dot arrays gets harder with increasing number of qubits due to inconsistency of fabrication technology. New physical architectures and control methodology are therefore explored to arrange quantum dots, and precisely control the number of electrons present in each quantum dot and the inter-dot tunnel rate.

Scalable qubit topologies have been demonstrated as a proof-of-concept such as, In [36], an arrangement of 12 quantum dots in a linear array is demonstrated — 9 qubits in linear array and 3 as quantum-dot, where the charge sensors are used for performing read-out of adjacent quantum dots. Another recent method is described in [37] (Figure 2.4) that presents arrangement and control of a linear array of 8 quantum-dots in GaAs with 2 charge sensing dots. The control of this device is performed by using a series of plunger gates and barrier gates. The plunger gate are connected to high frequency lines and controls the electro-chemical potential of every individual quantum-dot, whereas the barrier gate controls the tunnel coupling between adjacent quantum-dots by using DC signals. Such unit cell structures of areas up to a few μm^2 can be repeated to form a linear array structure.

In experimental spin-qubit ICs, the qubits are placed in a row, in a chain formation. This is not an ideal placement scheme for large scale operation. For a scalable qubit operation, we would require a 2D array of qubits, such as demonstrated in [38, 39]. For example, in [38] (Figure 2.4), we have a 2×2 array of tunnel-coupled quantum dots. By changing plunger gate voltages using AWGs, the electrons are tunneled to the quantum dots from their individual reservoirs. A center gate creates a negative bias and suppresses tunnel couplings along the diagonals, separating the qubits. All the Plunger gate pairs are connected to high-frequency (approx. 1 GHz) lines for pulsing and fast sweeping. Two sets of 3 control lines are used control 2 quantum dots, that act as charge sensors to perform readout. A resonant RF circuit performs high-bandwidth (up to 3 MHz) charge sensing from each sensing dot.

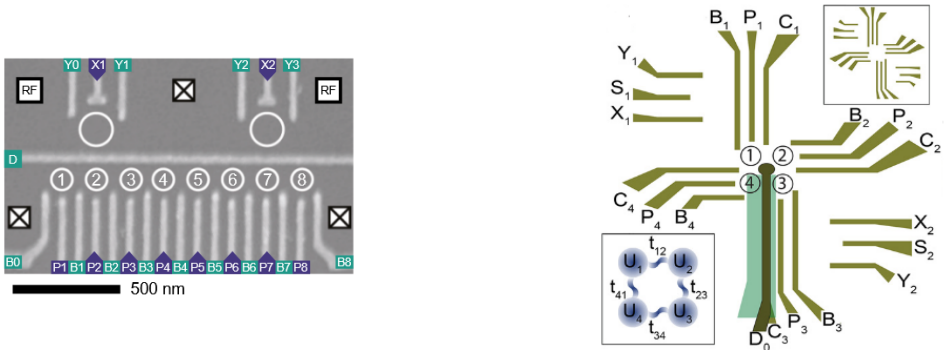


Figure 2.4: Scalable physical architectures in Spin Qubits: 8 qubits linear array (left) and 2×2 array of qubits (right)

2.3. SUPERCONDUCTING QUBIT CONTROL

Different Superconducting qubits can be made using different encoding methods such as using Flux, Phase or Charge of a superconducting resonator. The *Transmon* is one such superconducting qubit that encodes information in the *Charge* state of the non-linear LC resonator made from Josephson Junctions in a loop and a capacitor in parallel. The information is encoded in the two-lowest energy levels ($|0\rangle$ and $|1\rangle$). A unit Transmon qubit is large (typically, 0.1-1mm^2), is

typically operated at below 100mK and has coherence times typically, 18.7μs. The frequency of qubit is externally controllable over several gigahertz using the proximal flux-bias lines (P_F). The Transmon qubit is shown in Figure 2.5.

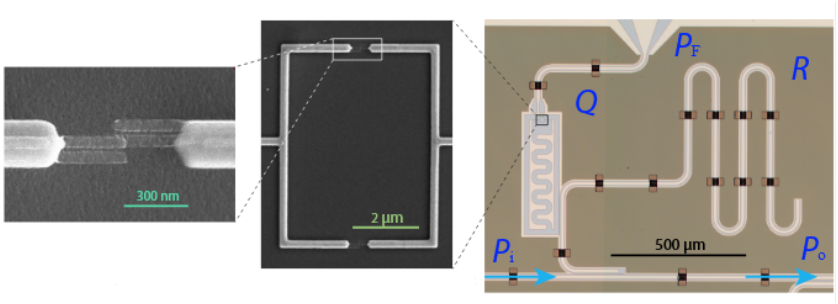


Figure 2.5: Transmon qubit in planar circuit-QED chip and the zoom in view of the Josephson Junction. Qubit (Q), resonator (R), flux-bias line (P_F), feedline input (P_i), and feedline output (P_o). Credits: *DiCarlo Lab, QuTech/TU Delft.*

Single Qubit Gates Single qubit gates (such as, X) are performed by applying calibrated microwave pulses at resonance frequency (typically, 4-6GHz and 20ns duration) to the feedline. The I/Q envelopes are created using Arbitrary Waveform Generators. The I/Q mixer performs Single Sideband (SSB) modulation of the carrier microwave signal. Example envelopes for performing X_π and Y_π rotation is shown in Figure 2.6.

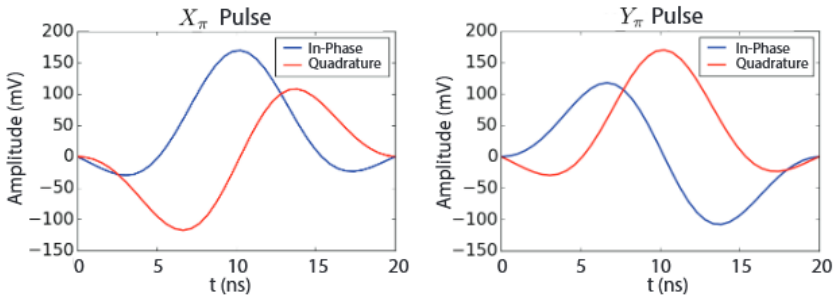


Figure 2.6: I/Q envelopes for X_π and Y_π rotations

Two-Qubit Gates Controlled Phase (CZ) and SWAP are the most common gates that can be performed on Transmon qubits that are coupled to a common resonator. Pulses of length ~40ns are applied to the flux-bias lines which causes the frequencies of qubits to come close and interact.

Measurement The co-planar waveguide resonator is capacitively coupled to the qubit and to the feedline. Frequency shift occurs to the resonator depending on the qubit state which can be measured by sending a 7-8GHz pulse for (300ns - 2μs). The result is then inferred after demodulation, integration and discrimination of the readout signal.

A typical control setup of superconducting qubits (as implemented by DiCarlo Lab, QuTech/TU Delft) is shown in Figure 2.7. For a complete description of Superconducting Hardware, please refer [40].

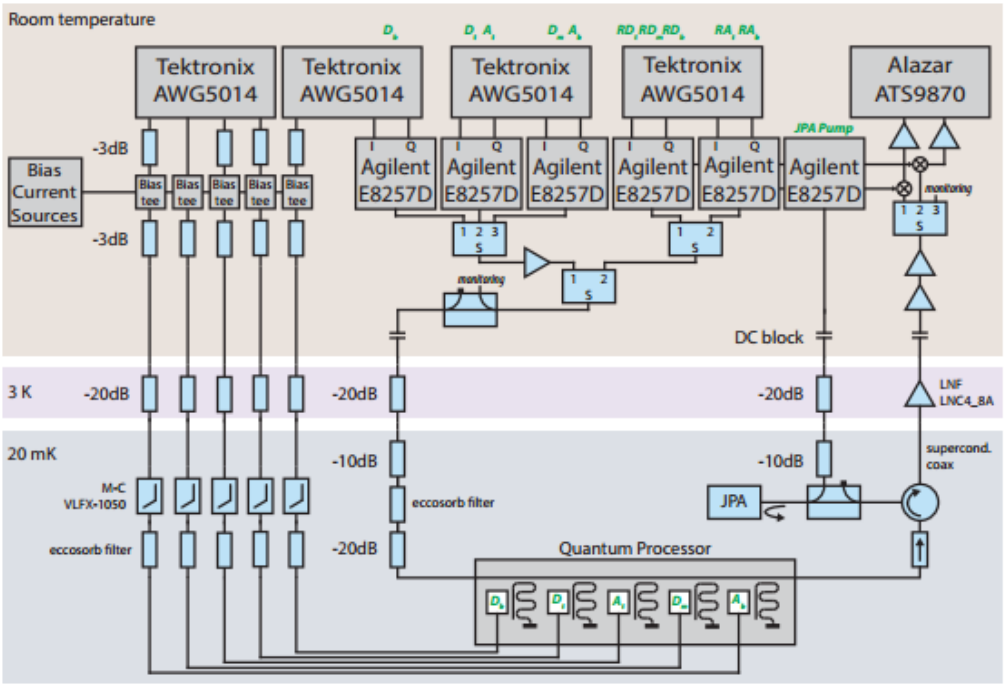


Figure 2.7: Experimental Setup for Control of Superconducting Quantum Processor

2.4. QUANTUM CONTROL HARDWARE

In this section, we describe the experimental setup used in the lab to control a 2-qubit chip. The main control rack is shown in Figure 2.8. The main components of the control rack are: Arbitrary Waveform Generator (Tektronix AWG5014C), Microwave Vector Source (Keysight E8267D), Multimeter (Keithley DMM6500 - not shown in Figure) and the Digitizer card installed with the Data Acquisition (DAQ) Unit.

For a single-electron qubit in Si/SiGe Quantum Dot, the spin of the electron is controlled by resonant microwave electric fields in a transverse magnetic field from a local micro-magnet; and the spin state is read-out in single shot mode. A description of the hardware used is given as follows:

TEKTRONIX AWG5014IC

The Tektronix AWG5014 model is used to generate IQ signals (In-phase/Quadrature Phase) using a 14-bit DAC converter with sample rates up to 1.2 GS/s. The hardware runs an Embedded Windows OS running RFXpress — a software, used for synthesis of digitally modulated baseband IQ and IF signals. The waveforms are preprogrammed in the memory (1-16,000 waveforms can be stored, with waveform length up to 32 Million points) and IQ signals are generated time-deterministically via the four output channels. In order to control one-qubit gates, 1 pair of IQ signals are required. These signals are DC signals that are sent to the IQ modulator through the SMA co-axial RF connectors.

KEYSIGHT E8267

The Keysight E8267D is microwave signal generator with high output power (@1Ghz: -130 dBm to +21 dBm), low phase noise(@1 GHz (20 kHz offset): -143 dBc/Hz), and capability to perform I/Q modulation (I/Q BW: 80 MHz to 4 GHz). The microwave souce generates high frequency sinusoidal waves (typically, 15 - 20GHz), modulates the incoming IQ signals from AWGs and acts on the high frequency lines on the qubit chip. The hardware is directly controlled using Signal Studio software that helps reduce the development time for creating complex signals using pulse building, noise power ratio (NPR), multi-tone signals etc.

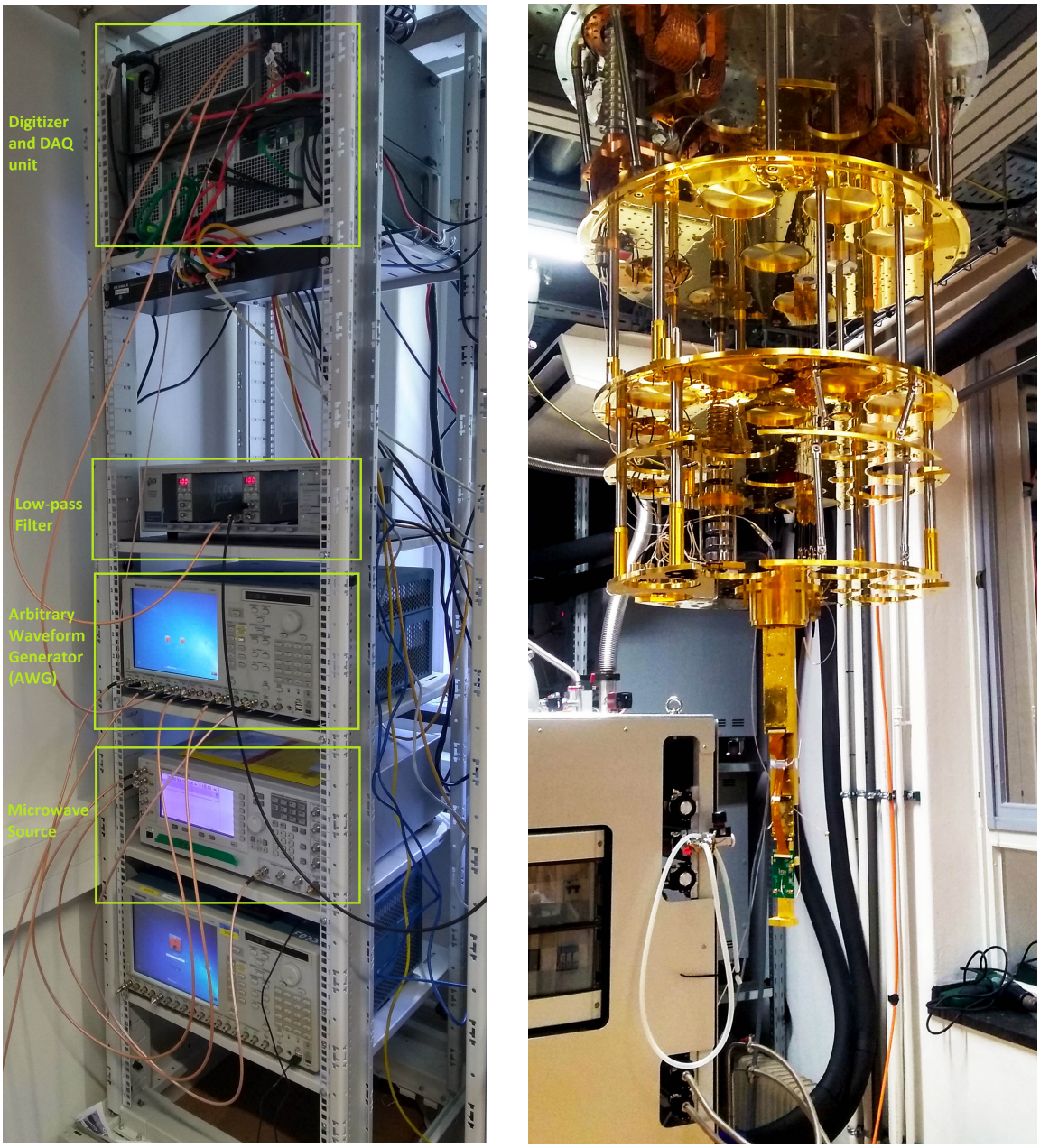
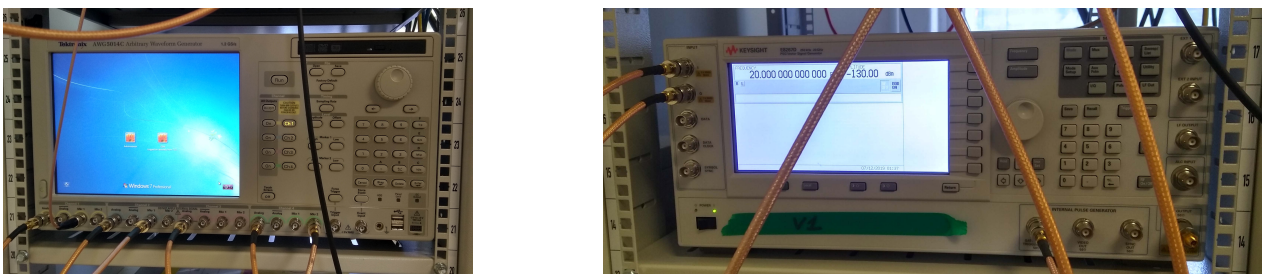


Figure 2.8: Silicon Spin Qubit Control Rack (left) and Cryogenic Fridge (right) Setup. Credits: Vandersypen Lab, QuTech/TU Delft.



(a) Tektronix AWG5014 (b) Keysight E8267

Figure 2.9: Quantum Control Hardware. Credits: Vandersypen Lab, QuTech/TU Delft.

MODULAR DAQ FRAMEWORK: QCODES

Microsoft developed QCoDeS is a python-based data acquisition (DAQ) framework that provides a robust architecture for fast and flexible data storage. It serves as a common framework for controlling physical systems with many degrees of freedom using a general purpose PC. A programmable interface allows definition of complex sweep functions over an n-dimensional parameter space, perform storage and post-process the acquired data by means of visualization tools.

The python programmable interface allows communication with the hardware through the transport layers. In QCoDeS, an experiment is composed of a *Loop* that performs sweep over one or multiple *Parameters* on one or multiple *Instruments*, performs measurement of the *Parameters* at every point, and stores the results in a *DataSet* (groups of arrays). We can also create an experiment by defining *Parameters* and by performing extended operations in *Loop* at every point in a sweep. *Parameters* are created for *Instrument* using `instrument.add_parameter()` and communicated to the Instrument. Further, Parameters can also be defined for *Meta Instruments* i.e. the higher level instruments that controls instruments at a lower hierarchy. QCoDeS helps promote faster development cycles, quick experimental setup and ease of development of hardware extensions. Details of QCoDeS are available at [41].

MEASUREMENT ENVIRONMENT: PYCQED

PycQED is a measurement environment developed by DiCarlo Lab at TU Delft for circuit-QED experiments and is built as a higher abstraction layer of QCoDeS i.e. it uses QCoDeS as the backend. The automatable software control is useful to control complex experiments. To use PycQED, we start QCoDeS, load the corresponding `[config file](init\config)`, set storage folders, create the *Instruments*, and load the values set in the *init*. The *init* folder is used to store configuration info and contains the script to create instruments and set up the experiments. Finally, the *Measurement Control* object is used to run the experiment (e.g.`MC.run()`). The *qubit* object is used to address the meta instrument and contains functions such as *measure, calibrate, find* and *tune*. For example, `qubit.measure_Rabi()` and `qubit.find_frequency_spec()`. PycQED is available at [42].

2.5. SI-SPIN VS SUPERCONDUCTING QUBITS

Usually multiple signals are required to control a single electron qubit in quantum dot. The following are commonly recurring requirements for the control signals:

1. an independently calibrated and tuned DC gate voltage on every site (typically up to $2V_{pp}$)
2. independently calibrated and tuned gate voltage pulses on every site (typically, few hundred mV and with sub-nanoseconds rise times)
3. independently calibrated and tuned microwave magnetic or electric fields at every site (typically -40 to -20 dBm, 1-50 GHz bursts of 10ns to 1μs duration)
4. a high precision of each of the control signals to achieve error rates comfortably below the 1% accuracy threshold.
5. Initialization, operations and read-out on timescales must be short compared to the relevant decoherence time.

Typically, precisely timed voltage pulse envelopes in baseband as I/Q signals and microwave sources are essential for single- and two- qubit gates. The measurement requires demodulation at microwave and baseband frequencies. Further, faster (low-latency) classical control feedback is required to perform feed-forward control (branching quantum instructions) on the quantum

processor. For example, QEC requires typically 100ns - 1μs measurement time before the qubit decoheres.

A table of signal requirements for various qubit technologies is summarized in Figure 2.10 [1].

Technology	T_2^*	1-Qubit gate	2-Qubit gate	Qubit read-out	DC-Biasing
Superconducting qubits (Transmons)	2.5 us	~ 6 GHz 20 ns	12ns 40 ns	7-8 GHz, ~ 1 us	flux-bias current
Single-electron spin qubits in a quantum dot	120 us	13-40 GHz ~ 1 us	~ 100 ns		gate voltage
Single-electron spin qubits in a donor system	160 us	30-50 GHz ~ 1 us	~ 100 ns		gate voltage
Singlet-triplet qubit	700 ns	~ 1 ns	~ 1 us		gate voltage
Exchange-only qubit	2.3 us	10 ns ...1 us	Sequence of pulses between different quantum dots		gate voltage
Hybrid qubit	< 10 ns	~ 100 ps	Sequence of pulses between different quantum dots		gate voltage

Figure 2.10: Summary of Signals used in varous Qubit Technologies [1]

2.6. NEXT LEVEL UPGRADES

The current methods of quantum control are apt for few qubit experiments but scaling the current approach to large qubit counts leads to expensive (time and resources) control infrastructure due to the varied qubit properties (such as, resonance frequency) due to process variations in fabrication. And although, qubit fidelities up to 99.9% is achievable, it is the absolute minimum for performing Quantum Error Correction (QEC). Further, the non-idealities in signal parameters and noise leads to reduced fidelity of qubit operations. The following solutions can be considered to address this issue:

1. Pulse shaping is required to obtain better gate and read out fidelities. For example, DRAG (Derivative Removal by Adiabatic Gate) Pulses requires pulse shaping capability. They are typically <10ns in duration and result in >0.999 fidelity.

2. Parameterized waveform generation. Modern Direct Digital Synthesis (DDS) based waveform generation allows specification of parameters such as, frequency, phase and amplitude for real-time signal generation. This approach is further detailed in Chapter 3.

3. Time Division Multiplexing (TDM) can be used to reduce the number of wires running from the controller to the qubit chip. Multiple qubits can addressed by multiplexing the control signals over a single line and using switching logic. A Vector Switch Matrix (VSM) used at DiCarlo lab serves this need. The drawback of TDM is that it limits the extent of parallel operation execution.

4. Frequency Division Multiplexing (FDM) of waveforms on a single line used to address different sub-blocks of different qubit frequencies.

5. SoC Controlled Architecture for feed-forward quantum operations and Variational Quantum Algorithms. While simple boolean-controlled branch statements can be handled by on-FPGA micro-architecture, complex branching logic (such as, those that require a

complete python subroutine execution) statements, used in variational quantum algorithms would require an on-chip processor(such as an SoC-FPGA architecture) to execute these sub-routines with low-latency. Further, Variational Quantum Algorithms also comprise of classical Optimization steps and periodic re-initialization of qubits that can be better achieved using an SoC-FPGA architecture.

2.7. QUANTUM CONTROL: A SURVEY

A number of initiatives have been taken by both, academia and industry, to develop scalable control architectures for promising quantum processor technologies, such as Superconducting Qubits, Spin Qubits and Ion-Trap Qubits Processor. In this section, we will explore some of the existing ideas in quantum "(micro-)architecture design". A study of these platforms enables us to systematically analyze and establish design points based on parameters of interest identified as per Spin-qubit control requirements. Two of the prominent hardware control infrastructures are summarized below - Sinara is from academia and APS2 system is from Industry.

2.7.1. SINARA HARDWARE

The Sinara hardware platform is a distributed modular control and readout platform developed by a consortium of universities, research labs and industry in Europe and North America. Sinara includes low phase - noise software-defined radio with 600MHz analog bandwidth and support for ARTIQ software controlled conditional execution (branching) support with microsecond latency. The hardware platform supports many other (more than 20) Sinara developed peripherals such as, multi-channel ADCs, DACs and ns-resolution digital I/O - all of which are based on insdustry-standard MTCA.4 and Eurocard Extension Modules (EEM). All the peripherals are controllable with the Advanced Real-Time Infrastructure for Quantum physics (ARTIQ). ARTIQ is the software control stack for Ion-Trap based quantum information processing devices that provides a high-level programming interface for writing quantum logic. It compiles the quantum code and executes it on FPGA-based hardware with nano-second timing resolution and microsecond branching latency.

The system is composed of the software-stack ARTIQ, the hardware-controller (Metlino/ Kasli) and waveform generation units controlled using slave FPGA units (RTIO core/containing I/O manager and FIFOs for timing control). A simplified view of the system architecture is shown in Figure 2.11. The inter-hardware connections are SFP based (fibre-optic) that supports data-rates as high as 12.5Gbps. The Sayma RTM blocks support drive the waveform generators containing 8 2.5GSPS DACs and 8 125MSPS ADCs based on MTCA.4 architecture. EEM sub-system is a low cost extension of the MTCA subsystem that does not require a low-jitter RF distribution. The sinara hardware ecosystem is shown in Figure 2.12.

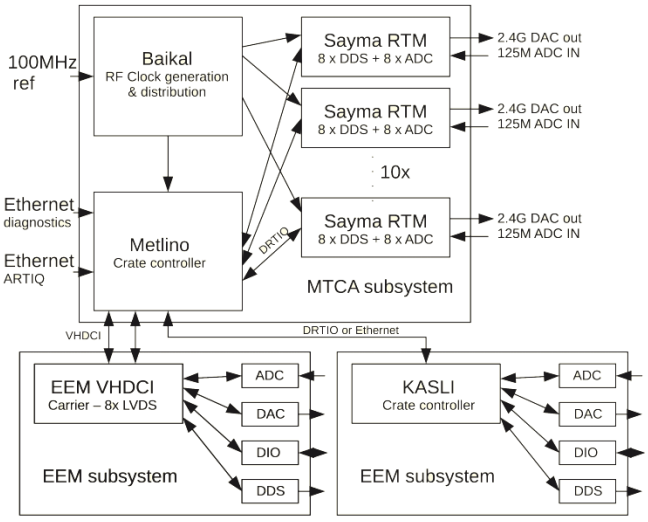


Figure 2.11: Simplified Sinara Architecture

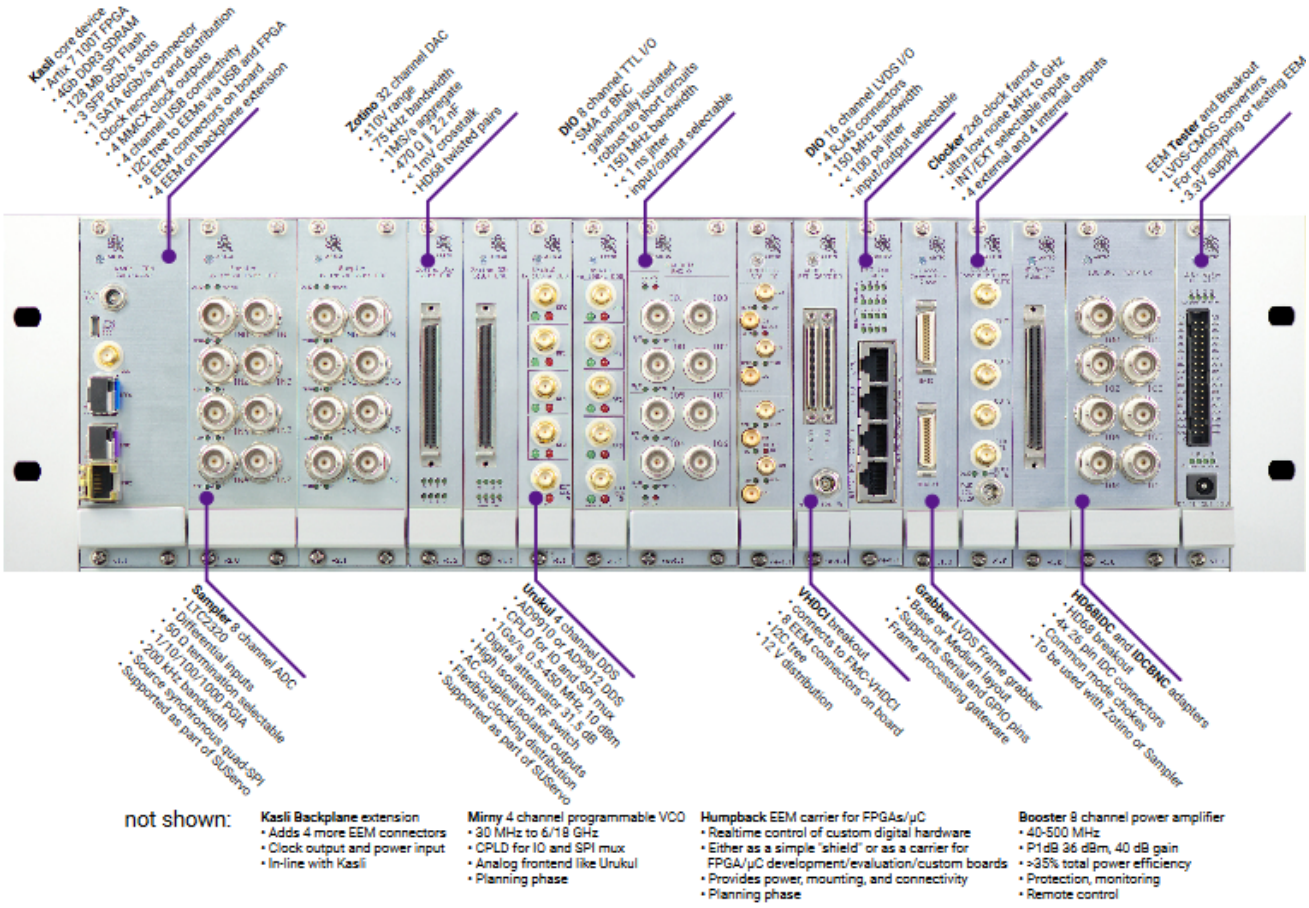


Figure 2.12: Sinara Hardware ecosystem

Critical Comments A distributed, modular and synchronized control of quantum hardware is the highlight of Sinara hardware ecosystem. The platform has been designed for modularity and flexibility in control. It supports very high-end hardware infrastructure such as the parameterized arbitrary waveform generation and industry-proven hardware standards. Critically examining the system, we find a lack of control-support for classical-quantum mixed code and integrability with QASM variants. Furthermore, the system still requires support for quantum libraries in the quantum compilation tool-chain. This makes the system difficult to integrate with compiled QASM code directly on quantum processors. However, the infrastructure does provide the flexibility to integrate it all.

2.7.2. RAYTHEON APS2 AND QDSP HARDWARE

Raytheon BBN Technologies has developed a control hardware and software stack Arbitrary Pulse Sequencer 2 (APS2) and QDSP framework for the control and readout, respectively of superconducting qubit quantum processor. An APS2 system comprises of 9 APS2 modules (with 18 14-bit 1.2 GSPS analog output channels) and a trigger distribution module (TDM)(as shown in Figure 2.13). The TSM captures the qubit state information on its FPGA and distributes the

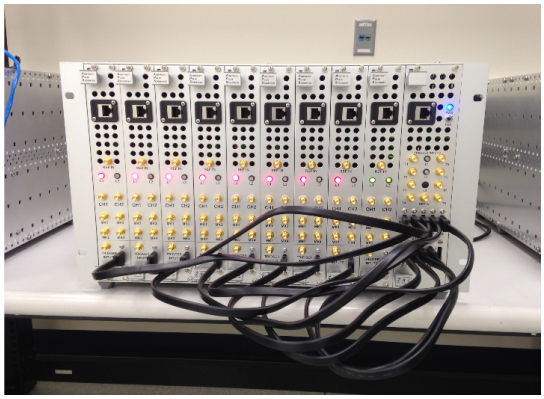


Figure 2.13: Raytheon APS2 System

compiled logic to APS2 units. The host controller is programmed via 1Gb Ethernet connection. A star network of SATA cables handle the intermodule communication between TDM and each of the APS2. QDSP is responsible for quantum state determination and data acquisition using FPGAs. They consolidate many DSP stages to 1 stage in order to perform rapid-readout.

Critical Comments APS2 system follows a distributed control infrastructure as well. The system is currently capable of controlling up to 8 qubits (in 2017) and no recent updates have been announced from the company. The hardware stack has its own inter-module synchronization methodology and provides low latency read-out for superconducting chips. However, some of the shortcomings of the system are: it does not state it's integrablity with available quantum compilation tool-chain, does not comment on integrability with different quantum technologies, it does not support comprehensive feedback control and has a complex compilation tool-chain comprising of generation of multiple binaries.

2.8. CONCLUSION

The details of the comparisons made in this chapter are crucial to the understanding of control requirements for Spin- and Superconducting quantum processors. We also took a look at two of the prominent quantum control architecture hardware (under development) to get a perspective of the state of the art. The idea of designing a QISA capable of supporting quantum operations and a flexible control over qubits is the inspiration for the development of a micro-architecture. In the next chapter, we shall discuss the CC-Spin micro-architecture design and it's implementation methodology.

3

CC-Spin Quantum Micro-architecture Implementation

The architecture of a machine connects the physical hardware to the applications that can run (on that hardware) and dictates how instructions are executed. The micro-architecture describes the way we leverage digital design techniques (such as, Datapath design, pipeline stages, number of registers etc.) to implement the defined quantum/classical Quantum Instruction Set Architecture (QISA). In this chapter we describe a control micro-architecture and it's specifications that is developed and implemented. In the next chapter we will present implementation of some of the building blocks of this micro-architecture.

3.1. Introduction

For the quantum algorithms to be comprehended by the quantum accelerator, requires a low-level representation of the quantum and classical mixed instructions that the classical control hardware controlling the quantum chip can understand and delegate its execution. This is known as the Quantum Instruction Set Architecture (QISA).

In the Full-Stack implementation, the OpenQL compiler performs Quantum Circuit Optimization, Scheduling, Qubit Mapping and Fault-Tolerant Circuit Implementation of the quantum program. The outputs of the compiler is the cQASM instructions (abbr. common QASM). cQASM is a quantum and classical mixed code that is a general instruction representation which allows this code to be executable on both, a quantum simulator and real quantum hardware (technology independent). For now, the current OpenQL compiler does not support features such as, C++ with quantum instructions, control flow with conditional branching, sub-circuit reuse etc. and is under development. The OpenQL compiler also consists of a low-level compiler that generates eQASM instructions (executable QASM). The compiled eQASM instructions are expressed in the Quantum Instruction

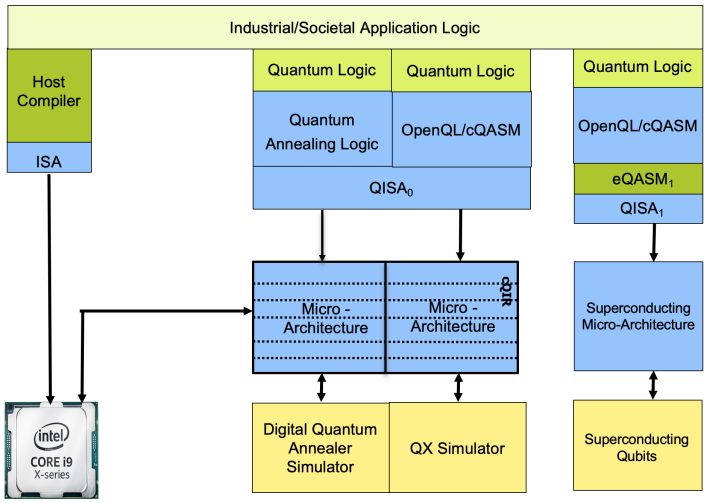


Figure 3.1: Full-Stack Quantum Instruction Execution Flow

Set Architecture (QISA) format — sufficient instructions to provide to the control hardware, responsible for initializing, controlling and performing readout of the qubits. The ultimate goal of the OpenQL-compiler is that it can produce two kinds of outputs. Based on cQASM, we generate an assembly version that can be executed on a simulator such as the QX-simulator. If an experimental chip is available, we can translate the cQASM into an executable version, called eQASM, that takes the low-level requirements of e.g. the semiconducting quantum chip into account. It can also take the requirements of a superconducting quantum chip into account and thus producing a different eQASM version. However, that is still in the process of being developed. For now and very related to the way it is expressed in the previous sentences, we require two instruction representation formats (cQASM and eQASM) because some of the hardware dependent timing constraints can be generated only at runtime. The low-level compiler (hardware compiler) takes these constraints into consideration and generates eQASM instructions. The control hardware that accepts eQASM instructions and performs precise and time-deterministic control-signal generation is the Micro-architecture (as shown in figure 3.1). A detailed execution flow diagram is also given in appendix A.3.

The Micro-architecture has a major role to play in the full-stack implementation of the quantum accelerator. The functions executed by the quantum accelerator are:

a) It creates the abstraction layer between the high-level programming and low-level control implementation on the quantum chip.
b) It can (ideally) be independent to the technology implementation (such as, superconducting qubit, silicon spin-qubit etc.) of the quantum chip; and abstracts away the low-level hardware details from the user.
c) Depending on specific requirements of the technology implementation, it can be reconfigured.
d) Generates high-speed signals (DC, Sinusoidal or Arbitrary) for controlling qubits and implements the timing control for precise and deterministic signal-based control of quantum chip. It is the layer on the full stack that separates the time-deterministic and time-non-deterministic parts of task execution.
e) Implements fast readouts, and readout-based control (feed-forward control) of the quantum chip.
f) Implements the classical control logic for quantum error decoding and correction.

We have divided this chapter into two main parts: the first part, describes the components of the micro-architecture independent of the back-end; and the second part, describes the waveform generation hardware architecture which can be AWG or DDS driven.

3.2. MICRO-ARCHITECTURE OVERVIEW

The design of micro-architecture for a specified QISA is divided into two steps: *Datapath Design* and *Control Unit Design*. The Datapath design refers to the definition of the Functional Blocks (such as General Purpose Registers, Bus Sizes, Microcode Unit etc.); whereas, the control unit contains instruction flow logic (such as, the Instruction Register, Program Counter, Decode Logic, Timing Control Unit). The control unit translates each QISA instruction to a sequence of actions, i.e. retrieving data from a registers, performing the Quantum/Classical operations, instruction sequencing, Timing-precise execution and result read-out and write-back. The control unit drives the functional units through 'signals' (electrical impulses) sent to the functional unit in 1 clock-cycle. In response, status signals are sent back by the functional units that indicates their current state. For example, a conditional jump is taken if the zero flag is 1 and the control unit alters program execution.

The problem of quantum micro-architecture design can also be approached with the same design considerations. A systems view of CC-Spin implementation for qubit control is shown in

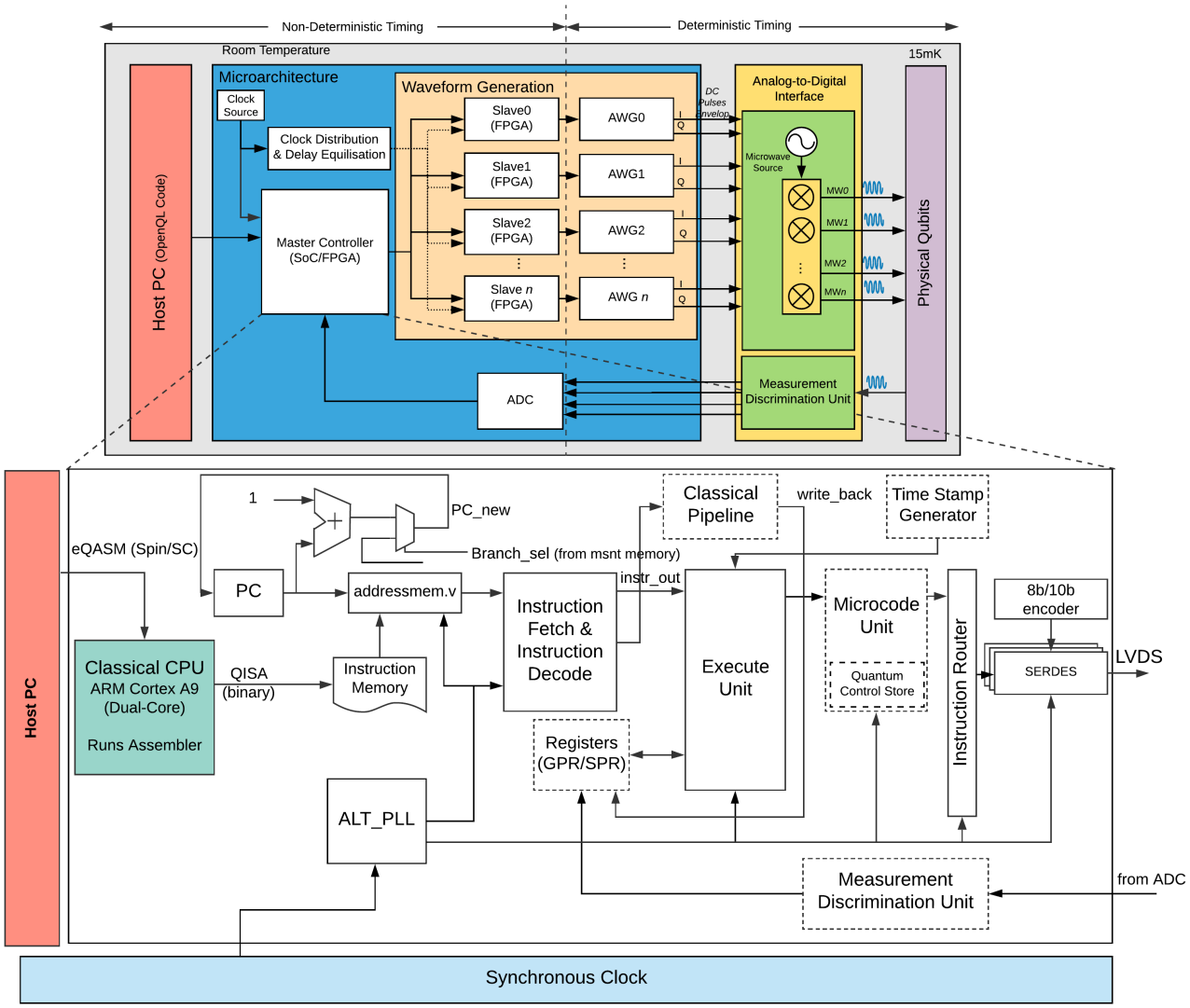


Figure 3.2: The system view of CC-Spin implementation using Arbitrary Waveform Generators (AWGs) - Modular AWG and/or DDS-DAC

figure 3.7. The flow of instruction is from left to right where the Host PC is responsible for quantum compilation and generation of QISA (such as, eQASM) instructions. The QISA, containing both classical and quantum instructions, is received by the *Master Controller*, implemented on Enclustra Mercury+ SA2 SoC-FPGA. The micro-architecture design, CC-Spin, containing classical and quantum pipelines, execute unit, microcode unit, instruction router and sequencer is implemented on the Master Controller. The execute unit is the control unit that directs the program flow for instruction sequencing and execution. The microcode unit is responsible for translating the QISA (binary) instructions (or, macro-instructions) from instruction memory to micro-instructions based on a *quantum control store* and further, to micro-operations. These micro-operations are routed to different SERDES (Serializer-Deserializer) units that transmit the 8b/10b encoded microcode to the slave FPGAs in the Waveform Generation Block.

The output of the micro-architecture is LVDS serial_out data (micro-operations) that is communicated to Waveform Generation Unit. The Waveform generation unit consists of slave FPGAs that implement the microcode buffer and the timing-precise execution unit for time-deterministic instruction execution (waveform generation), as shown in the figure. The Slave

FPGA drives the AWG units using serial (SPI for DDS)/parallel (14-bit parallel data for DAC-based) drivers. The AWG units are responsible for IQ (envelopes) and DC waveform generation.

The implementation of CC-Spin micro-architecture is based on using commercial off-the-shelf (COTS) components for waveform generation. These waveform generation/AWG units can be either: (a) Modular, in-house and custom-developed AWGs with limited memory and DC and IQ waveform generation capability; (b) a DDS-DAC based solution for waveform generation, such as AD9910 AWG; or, (c) Commercial Programmable Arbitrary Waveform Generators, with multi-channels and more flexibility. While commercial AWGs more flexible and programmable, they are expensive and bulky and not a long term solution for large-scale quantum control. In the following sections, we will discuss the implementation and working of each of the functional blocks of CC-Spin Micro-architecture.

3.2.1. QISA

The Instruction Set Architecture (ISA) is the highest level abstraction that is visible to a programmer and specifies the functions the system is capable of performing. ISA also helps to establish a common platform for multiple compilers. In the case of the design of a quantum micro-architecture, this level specifies the classical and quantum instructions without qubit technology-dependent constraints; and the programmer-visible architectural storage elements, such as the general-purpose registers, the program counter, and the status registers.

In a quantum micro-architecture, the ISA provides a common platform to execute all (quantum and classical) instructions. To understand this, we can compare the programming model of a quantum computer to that of a heterogeneous platform (such as, GPU or FPGA). The quantum instructions are a part of the 'quantum kernel' that are sequenced through the quantum pipeline.

QISA Design Requirements

The QISA implementation must meet the following requirements to qualify as a viable candidate for quantum control:

1. A QISA must include, both classical and quantum instructions, including run-time feedback execution.
2. The QISA must specify information of timing of quantum operations.
3. The QISA must not include hardware specific or qubit technology-dependent information i.e. it must not be restricted to a specific control hardware or a qubit-technology.
4. The instructions must be densely encoded to avoid instruction issue rate from becoming a bottleneck.
5. QISA must be flexible to accommodate different sets of quantum operations and support complex waveform generation.

QISA Design

In this thesis, we have proposed a 32-bit QISA and its implementation is capable of controlling from a few qubits up to 100s of qubits in Spin-based Quantum Processors in NISQ-era quantum computing. As stated in the previous chapter, quantum control comprises of a set of IQ and AC/DC electrical control signal generation. These control signals can be triggered by the micro-architecture (with deterministic timing, Delay of Trigger to Waveform Output being < 100 ns). To specify this triggering operation at the QISA layer, we propose a general instruction representation format that essentially specifies four things: *Instruction Type* (quantum/classical), *Opcode* (the quantum operation), *Quantum Channel Mask* (qubit mapping to channel number) and *Payload* (amplitude/phase/frequency information of (to be) generated waveform). An example of this general instruction format is as shown in figure 3.3 (b).

To understand this, let's take an example of the pseudo-code shown in figure 3.3 (a). *Pulse* and *wait* operations, are specified as Opcode. In any practical application, Pulse can be any of the

specified quantum operations (such as, X, Y, H, $R_x(\theta)$, $R_x(\theta)$, CNOT [1] etc.) and Wait instruction can be specified in either, Register addressing or Immediate addressing modes. The channel number is specified through the 8-bit quantum channel mask for Channel 1 to 8. For example, the `8'b10011011` asserts the Opcode specified pulse on channels 1, 2, 4, 5 and 8. Finally, the Payload contains different parameters of the waveform such as the DC voltage level to be generated, the phase for IQ signals etc.

As shown in the figure, all channels are initialized with 0V and upon assertion of the pulse operation, the operation stays active on both specified channels for the duration specified by the wait operation (2 units[2]). Next, only the pulse on channel 2 is modified, and 3 wait units. This does not affect channel 1's voltage output and it continues to hold it's value until next pulse operation is asserted viz. at time unit 5. Therefore, using an 8-bit channel mask, we can address up to 8 qubits. The mapping of channel number to a qubit can be fetched through the quantum control store (in the microcode unit) which is generated from the compilation step.

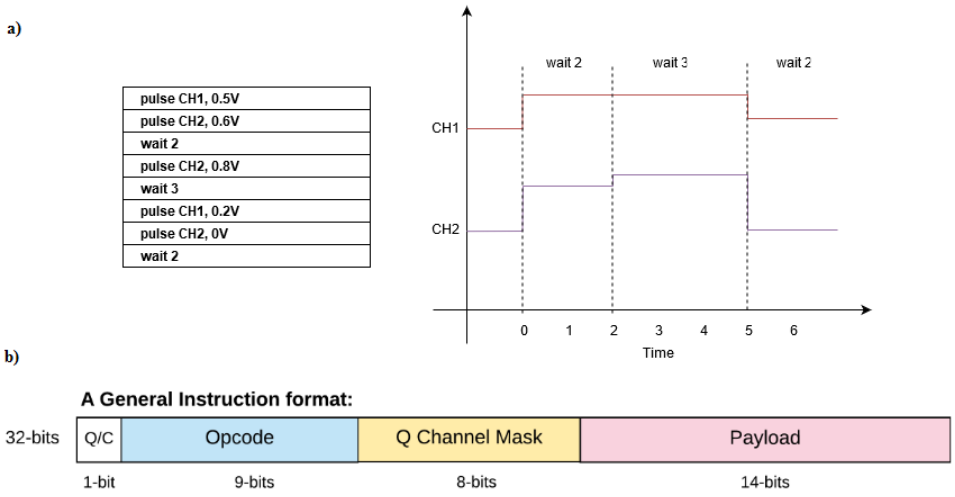


Figure 3.3: (a) Pulse Generation Timing Control, and (b) A general Instruction Format

Motivated by the previous example of the QISA representation, we have have proposed the following two QISA representation formats for addressing a large number of qubits. For simplicity reasons, we will only focus on the quantum instructions and the *wait* operation. The classical operations can be represented in any standard format by setting bit-31 to `0`. An example of classical instructions for quantum control is given in table 3.1. Note that, while addressing a large number of qubits simultaneously is essential for NISQ-era quantum control, other factors also need to be considered such as, efficient encoding of waveform parameters in the payload bits, Single-Instruction-Multiple-Qubit (SIMQ) control, high instruction-issue rate etc.

QISA - I In the first design, we reduce the number of Payload bits and introduce *Mask Index* bits. Using a simple decoder, we can expand the 8-bit channel mask to 32 bits and obtain a final channel mask bits by performing Bitwise OR operation, as shown in figure 3.4. This representation is moderately scalable (addresses up to 32 qubit channels). In the worst case, i.e. addressing all 32 channels simultaneously (such as creating superposition on all qubits), 4 instruction fetch cycles are required.

[1] Note that, in spin qubit implementation, CNOT gate requires triggering two channels with precise timing (phase) constraints.
[2] number of clock cycles

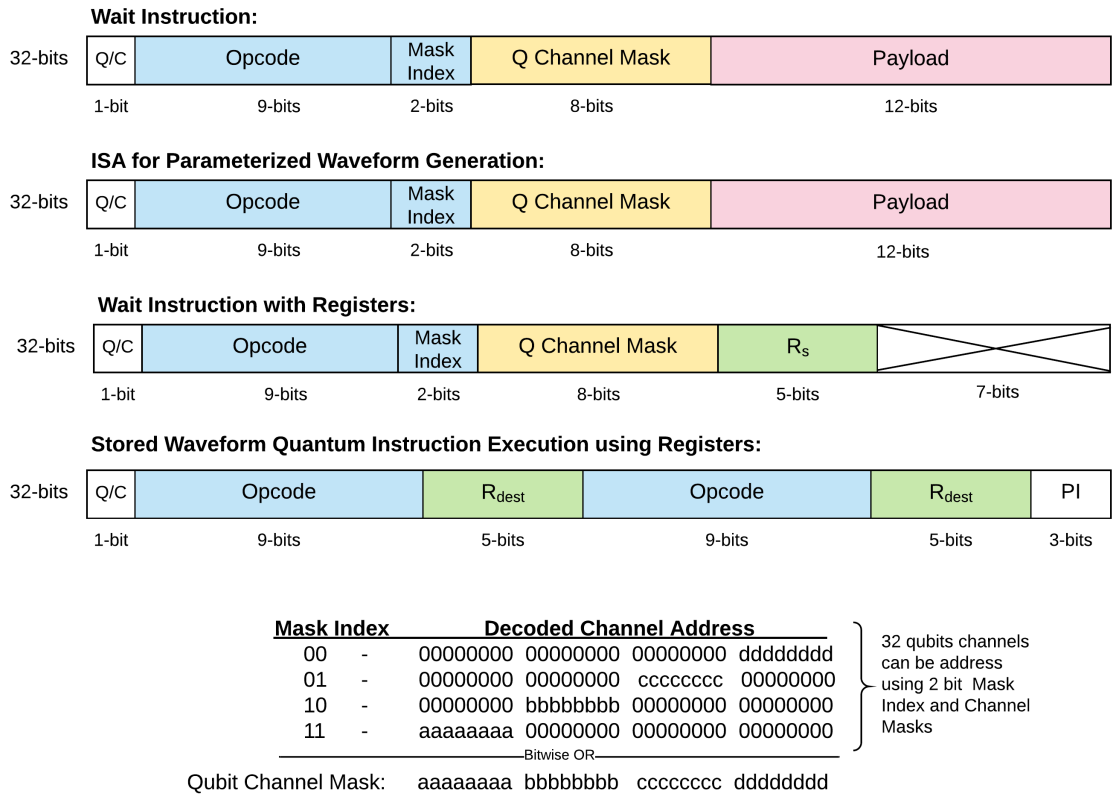


Figure 3.4: 32-bit ISA (Design 1) uses Mask Index and Q-Channel Mark to construct a 32-bit channel mask (to address 32 qubits simultaneously)

QISA -II In this representation, the channel mask bits are represented in standard binary number system that point to the channel address in a Lookup Table (LUT). The channel mask can then be fetched from the look-up table. The channel address is the input, and the channel mask value at that address is the output. For example,

`8'b00000001` ⟶ `Mask: 256'b0000...0001`
`8'b00000010` ⟶ `Mask: 256'b0000...0010`
`8'b00000011` ⟶ `Mask: 256'b0000...0100`
until... `8'b11111111` ⟶ `Mask: 256'b1000...0000`

LUT occupies memory but helps to realize complex digital logic functions fast, as it requires looking up a value from the table which occurs in a single clock cycle. Further, we can also use the LUTs for payload bits to specify different waveform parameters with a higher resolution.

In figure 3.5, we have shown the QISA - II instruction format for the wait-instruction (immediate addressing (first) and register addressing (third)) and quantum operation execution (waveform generation based on specified parameters (second), and triggering stored waveform on AWG (fourth). The Q Channel Address is 8-bits which maps to a 2^8 = 256 qubit channels. The disadvantage of this instruction format is that in one clock-cycle only one channel can be addressed. For n-qubits, n instructions need to be fetched in n clock-cycles. However, this occurs in the time-indeterministic side of the quantum pipeline and these operations can be buffered before executing them on the waveform generator.

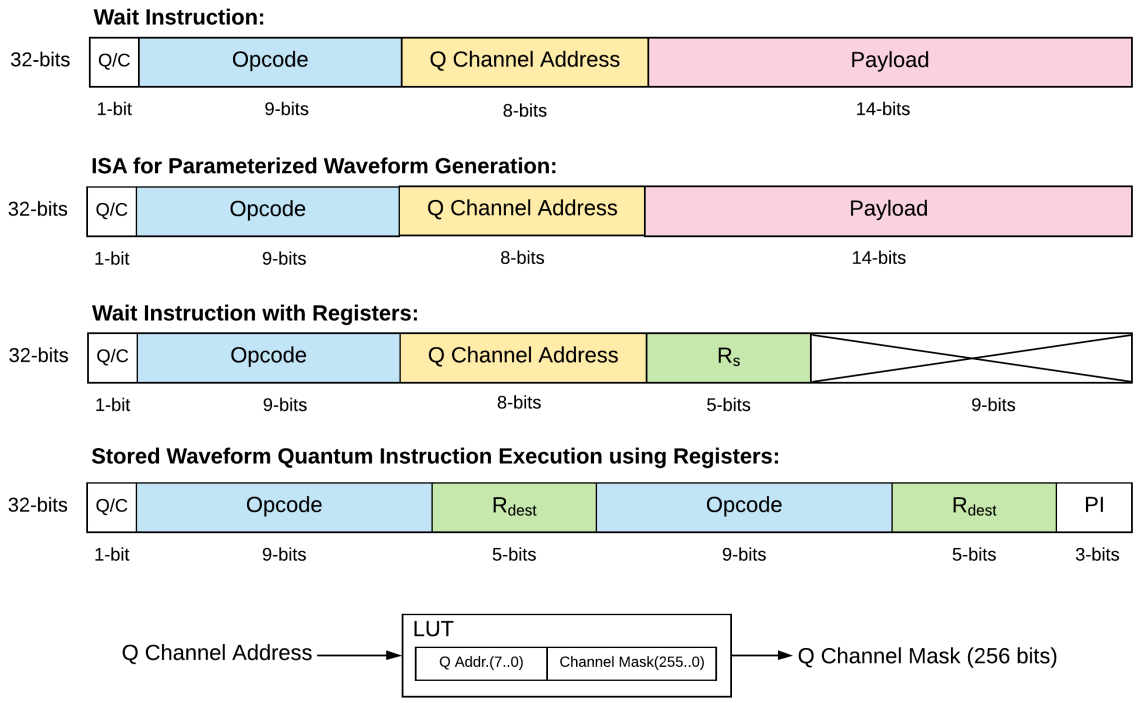


Figure 3.5: 32-bit ISA (Design 2) - Using LUT for Channel Mask Expansion (10 bit address fetches the channel mask for 2^{10} = 1024 qubit control channels)

This instruction format is helpful for: (1) addressing large number of qubits (up to 256 qubits) and (2) specifying waveform parameters in QISA. Further, if an arbitrary waveform generator provides complex waveform shaping capability, the payload bits can be modified accordingly to supply the parameters for the same. The QISA also supports indirect qubit addressing mechanism using SMIS and SMIT instructions for Code-word Triggered Pulse Generation (CTPG), such as proposed in [2] and shown in figure 3.6. The SMIS and SMIT instructions use a set of registers to specify multiple target qubits for the same single-qubit or two-qubit operation, respectively. The operation on the qubit can be then performed by specifying the operation Opcode and the target register.

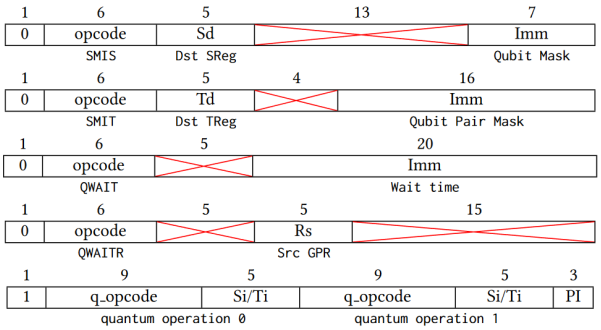


Figure 3.6: eQASM format of Target Specify Instructions (top two), and Quantum Bundle (bottom three) [2]

The detailed definition of ISA along with Opcode-representation is shown in 3.1. The QISA contains both classical and quantum instructions necessary for performing quantum computation. The requirements for the QISA to fulfill were identified from existing literature, such as, [43].

Table 3.1: eQASM - an example of Quantum-Classical Instruction Encoding Scheme [2]

Type	Syntax	Description
Control	`CMP Rs, Rt`	**CoMP**are GPR Rs and Rt and store the result into the comparison flags.
	`BR <Comp. Flag>, Offset`	(**BR**anch) Jump to `PC + Offset` if the specified comparison flag is '1'.
Data Transfer	`FBR <Comp. Flag>, Rd`	(**F**etch **B**ranch **R**egister) Fetch the specified comparison flag into GPR Rd.
	`LDI Rd, Imm`	(**LoaD** **I**mmediate) Rd = sign_ext(Imm[19..0], 32).
	`LDUI Rd, Imm, Rs`	(**LoaD** **U**nsigned **I**mmediate) Rd = Imm[14..0]::Rs[16..0].
	`LD Rd, Rt(Imm)`	(**LoaD** from memory) Load data from memory address `Rt + Imm` into GPR Rd.
	`ST Rs, Rt(Imm)`	(**ST**ore to memory) Store the value of GPR Rs in memory address `Rt + Imm`.
	`FMR Rd, Qi`	(**F**etch **M**easurement **R**esult) 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`	(**Q**uantum **WAIT** **I**mmediate/**R**egister) 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>`	(**S**et **M**ask **I**mmediate for **S**ingle-/**T**wo-qubit operations) 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.

For this master thesis, QISA-II is too complex to implement as a micro-architecture, and we will use figure 3.3's general format for its implementation and simple DC and Sine waveform generation, without taking into account the payload parameters.

3.2.2. QUANTUM PIPELINE

In this section, we will discuss the main components of the quantum pipeline. The implementation of the quantum pipeline is based on the general QISA format as described in 3.3. It is more than often an instance that some component of the hardware system do not work as assumed. Using the general format allows us to stick to a simpler pipeline for testing the hardware components and their performance. Moreover, in the HDL-code the pipeline is implemented by setting standard *parameters* for specifying variable elements in the code such as: bus_width, addr_width, memory_depth etc. This allows easy modification of the code for scaling of the same logic to accommodate hardware extensions. Further, all components are described in a *structural logic* which allows easy integration and continuous development on and around the pipeline.

IMPLEMENTATION

A detailed picture of the quantum pipeline is shown in Figure 3.7. The quantum instruction is specified in the QISA-format (such as eQASM) and is communicated to the ARM processor on the SoC-FPGA platform through TCP/IP Ethernet port. The executable `.qisa` file contains quantum instructions along with execution hardware timing constraints file. The ARM processor runs an assembler to translate the `.qisa` instructions to binary. The translation here is a direct one-to-one mapping. The assembler then communicates the 32-bit binary instructions `[31..0]` to the instruction memory on the FPGA side. The instruction-fetch stage of the firmware is implemented in the `addressmem.v`. The binary instruction contain both, the classical and quantum instructions, which is identified by the bit-31 of each `[31..0]`instruction where the number 0 represents a classical instruction and the number 1 refers to a quantum instruction. The instruction representation format of classical and quantum instructions are different and therefore requires two different pipelines for execution. An arbiter in the Instruction Decode stage separates the classical code from the quantum code. The classical instruction representation format is a MIPS-like instruction format and is executed in the classical pipeline. A summary of the supported classical instruction is given in table 3.1.

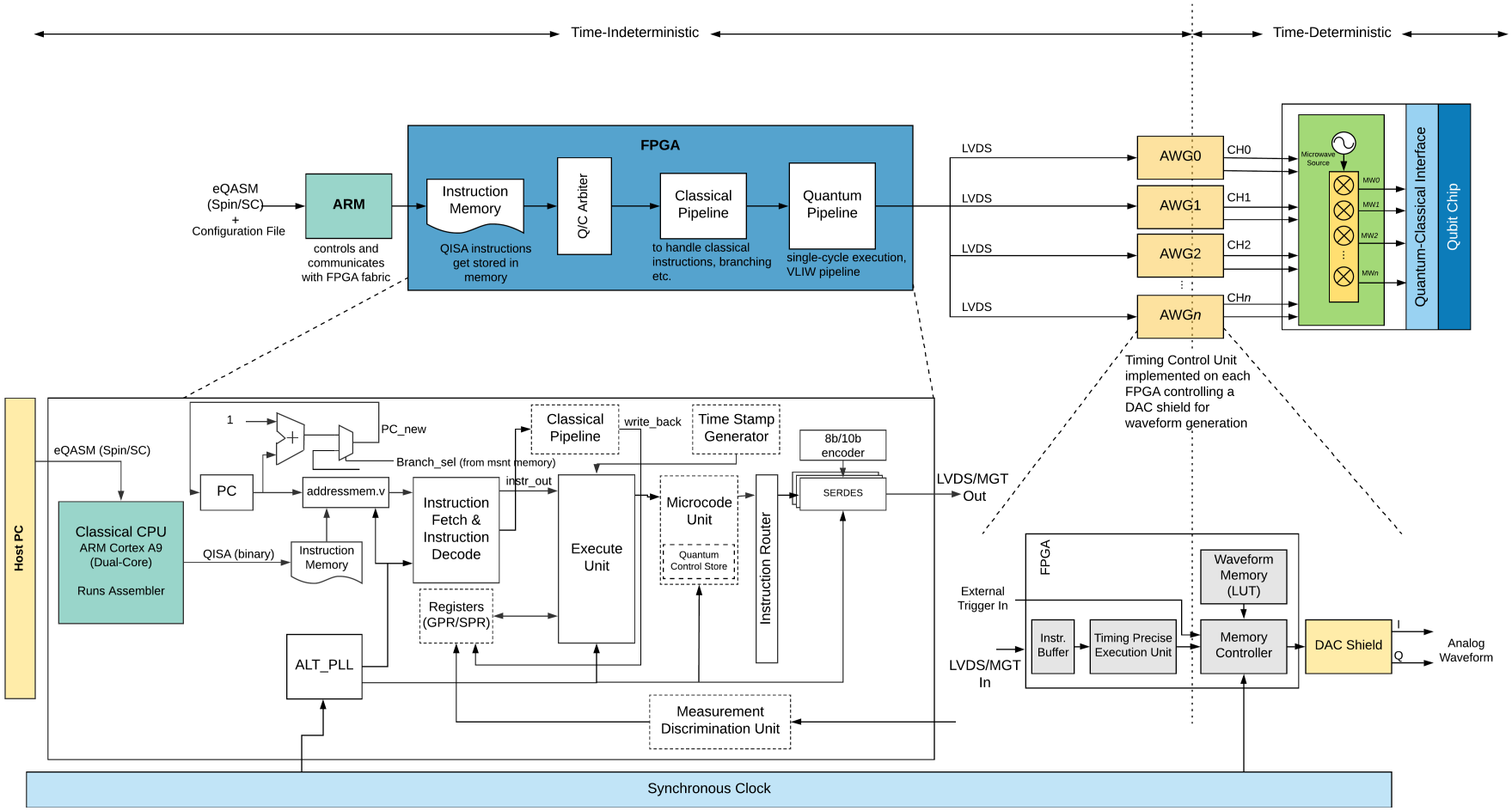


Figure 3.7: Implementation of Quantum control Pipeline using Arbitrary Waveform Generators.
a

[a] Dashed boxes in the pipeline are not yet implemented.

The Instruction Decode stage accepts the quantum instructions, decodes the Opcode and Payload and sends the data to the microcode unit. The microcode unit translates the quantum instructions to sets of micro-instructions using the micro-instructions available in the quantum control store. The data in the quantum control store is configurable through an external file that is uploaded along with the `.qisa` file. The micro-instruction are then communicated to the Instruction Router which, based on the qubit-mask routes the data to different SERDES units, each communicating to different waveform generation units via high-speed 3.3V LVDS links.

For the practical implementation, we are using simpler 8-bits parallel data at this step, but depending on the size of micro-instruction definition, we can increase the micro-instruction definition size to 64-bits and even 128-bits. The SERDES converts the incoming parallel data to serial data. Additionally, the SERDES encodes the data in 8b/10b encoding scheme. This is required because in cases when there is a long stream of 0's or 1's, it tends the transmitted signal to look like a DC signal in the communication channel. The 8b/10b encoder ensures that the encoded data stream has at least a certain amount of data transitions. Variant of the same, 64b/66b and 128b/130b can also be used. This is an essential component, as it helps in clock recovery and receiver side alignment of the data stream by providing enough state transitions. This portion of the quantum pipeline is implemented on the Master Controller SoC-FPGA and is not timing-dependent. The timing execution unit is implemented on FPGAs on the modular waveform generation units.

The Waveform Generation units can be implemented using different waveform generation techniques, such as, using AWGs (lookup table based, Numerically Controlled Oscillator-based etc.), DDS-based etc. The waveform generation units requires waveform parameters to generate waveform and these are provided by the Slave FPGAs [3]. The Slave FPGA's receive the LVDS data_in from the Master Controller SoC-FPGA, deserialize and decode using SERDES and 8b/10b decoder and send the data to Timing Precise Execution Unit (TPEU)[4]. The TPEU contains FIFO buffers, on which the output is triggered after precise number of cycle counts. From this point on, the instruction execution is time-deterministic. The FIFO output is an 8-bit codeword that triggers the memory controller on the AWGs to generate the control waveform. The final output of the waveform generation unit is IQ and DC waveform pulses with precise timing.

While testing the various sub-components of the quantum pipeline, we assume only the quantum operations (Q/C bit is set to 1) and an arbitrary payload. We do not assume timing information and code-translation to micro-code is disabled. Figure 3.8 shows the timing diagram of a simulation of the implemented quantum pipeline. In this example, `ch_mask` and `op_code` are decoded by the Instruction Decode unit and only the quantum instructions are routed to the appropriate qubit channel. The size of the channel mask (8) specifies the number of qubits that are addressed simultaneously. For example, at time 50 ns, the mask of channel 1, 4 and 6 are active and the payload is transmitted to those channels. The payload in this example need not necessarily be the payload specified in the QISA. It could be the translated micro-instruction from the microcode unit. The HDL-code also allows the flexibility to change payload sizes and number of channels by simply adjusting the parameter values.

[3] Slave with respect to Master Controller

[4] Timing information of each qubit channel is generated from a Time Stamp Generator implemented on the master controller (still needs to be developed).

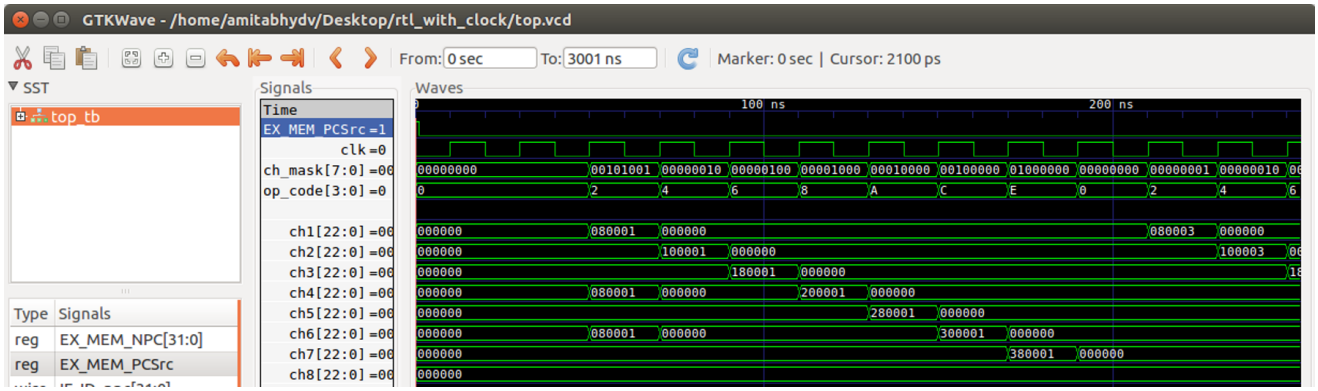


Figure 3.8: Simulation of Quantum Pipeline until Instruction Router

3.2.3. QUANTUM MICROCODE UNIT

The Microcode Unit is an on-hardware abstraction layer over the physical execution/sequencing units. Compared to microcode, the ISA can generate the 'macro-instructions'. An efficient microcode takes less memory space, helps in code optimization, minimizes execution time and allows an update mechanism through BIOS/UEFI or the OS. In this thesis, we develop a proof of concept hardware for using LVDS/MGT-based communication and waveform generation for control of qubits. Implementing the Microcode unit is an extension and hence, beyond the scope of the current thesis. For clarity reasons, we discuss the implementation strategy for Microcode Unit by means of an example.

Based on Instruction Register's contents, the Instruction Decode Unit (IDU) generates/routes control signals to the execution units. IDU can be *Hardwired* or *Micro-coded*.

`Hardwired Decode Unit` generates instruction-specific action sequences which is implemented on the hardware using sequential digital logic such as, Finite State Machines (FSM). Being hardwired, it is faster in terms of speed but it inhibits flexibility to make changes to the design.

In the quantum pipeline, we implement a `Micro-coded Decode Unit`. The microcode unit does not generate the control signals directly, but breaks down the QISA-instructions to micro-instructions. A 'microcode store' implemented as on-chip Read-Only Memory (ROM) is used from where the Micro-instructions are fetched. The Opcode of the decoded QISA instruction are used to generate an initial address, which becomes the entry point into microcode store. Each micro-instruction is followed by a sequence word, that holds the address to the next micro-instruction. Please note that, one QISA instruction can issue multiple micro-instruction. The microcode sequencer carries out the decoding process by successively selecting micro-instructions until decode-complete is indicated by a signal. Conditional microcode branches can also be handled by the microcode unit of the micro-architectures. Pre-computing and storing the control words helps make the micro-code unit design more flexible. Therefore, changes/adding new instructions can be added even at the later stages. This makes the design extension more simplified and feasible because now, to change the decode logic, we only need to adapt the microcode ROM content. The Microcode store in the quantum pipeline is referred to Quantum Control Store. Using this multi-level instruction decoding process, we can update the Quantum Control Store to include the decoding logic micro-instructions. This helps is adapting the the same micro-architecture for multiple qubit technologies.

Let us now take the example of the Quantum Microcode Unit from [43]. In the quantum pipeline, the microcode is responsible for translating the QISA-instructions to micro-instructions based on the Quantum Control Store contents. The micro-instructions contains the following components: *wait, pulse, measurement pulse* and *measurement result*. This is then converted to

micro-operations with timing labels in the Quantum Micro-instruction Buffer and sent to the timing queues, where appropriate codewords are triggered with precise timing. The same model can be used in the CC-Spin Microcode Unit for instruction decoding. Such a multilevel instruction decoding scheme allows development of a *qubit technology-independent* micro-architecture. The Quantum Control Store content can be modified using an external file that specifies the technology dependent parameters, such as timing information of pulse sequences; and the micro-architecture appropriately decodes the QISA instructions for any specified hardware.

3.2.4. TIMING PRECISE EXECUTION UNIT

In order to develop a trigger-based instruction issuing mechanism from FIFO buffers, we implemented and simulated (Figure 3.9) an experimental quantum pipeline [5] with two FIFOs. In this example, based on the trigger from the queue manager, the FIFO addressed by the channel mask sends `data_out`. Please note that, this implementation is a work in progress. The trigger implemented here simply fetches data from a test bench, but for Timing Control Unit implementation, the trigger is generated by counting number of clock cycles specified in the micro-instruction.

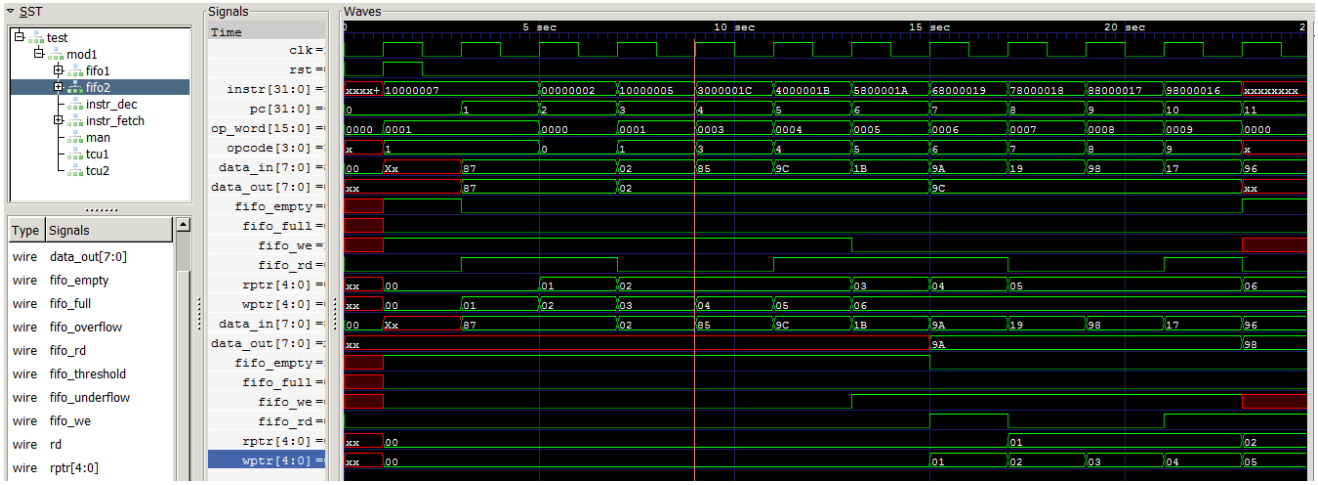


Figure 3.9: Simulation of Quantum Pipeline with two FIFOs in the Timing Control Unit

3.2.5. QUBIT MEASUREMENT

The quantum pipeline implements the Measurement Discrimination Unit to perform read-out and storage of the quantum states. A typical measurement is performed by sending long DC-pulses to the qubit chip. The current flow is detected and converted to voltage. A typical qubit measurement unit comprises of noise filtering, demodulation, integration and thresholding stages to determine the final qubit state. It is important to perform qubit readout within nanoseconds up to a few microseconds. An efficient readout system is essential for implementing comprehensive feedback control and for performing quantum error correction. The measurement discrimination unit takes the ADC output, performs Integration and Thresholding and record the readout state in the measurement register. From the registers, the execution unit can save the qubit state in external memory such as SDRAM. The value from the measurement register is used to for branching operations, which enables the micro-architecture to perform feed-forward circuit implementations.

[5] The code is accessible at my GitHub account: https://github.com/amitabhyadav/q-pipeline-experimental.

3.2.6. COMMENTS ON MICRO-ARCHITECTURE FOR NISQ-ERA

A typical example of NISQ-era algorithm for performing genome sequencing is shown in Figure 3.10.

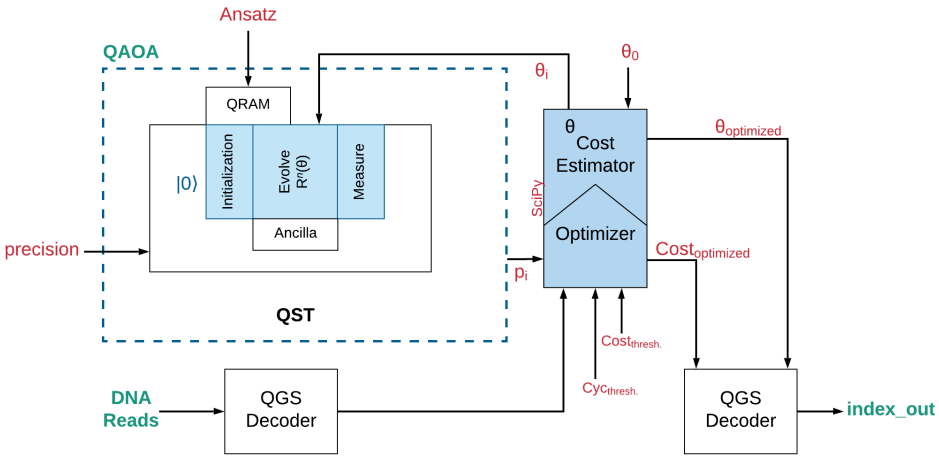


Figure 3.10: A general Quantum Approximate Optimization Algorithm architecture for Genome Sequencing Application (Credits: Aritra Sarkar, QCA/QuTech)

The quantum implementation of a typical Quantum Approximate Optimization Algorithm (QAOA)/Variational Quantum Algorithm comprises of a quantum circuit that is initialized with a trial state, the state is evolved on a quantum processor and final state is read-out by efficient measurement. This is represented by QAOA block. The readout after the quantum circuit execution is a histogram of probability of qubit states. This result is passed to matrix multiplication subroutine to generate a real value. This real-value is fed to the classical *optimizer* and the *Cost Estimator* that contains high-level optimization steps. The optimizer generates a new set of parameters that is used to adjust the rotation gate angles (parameters) in the Evolve-step of the quantum QAOA block. Since the optimization step is composed of SciPy libraries, it is not possible execute them in the same classical pipeline. To support the execution of variational quantum algorithms, the micro-architecture needs to fetch the compiled python binaries at every iteration. The CC-Spin micro-architecture can support the execution of these sub-routines by executing the python-optimization subroutines through the on-board ARM processor running Linux. The FPGA communicates with the ARM processor via AXI bus interface which allows fast ARM-FPGA communication. The optimized quantum circuit can be stored once on the FPGA memory which allows the user to bypass the openQL compiler for every iteration of the VQ algorithm. The evolution step of the Variational Quantum circuit stored in the memory contains rotation gates with arbitrary parameters that is required to be updates after each classical optimization step. The ARM processor running the python compiler can pass the optimized rotation-gate parameters that can be updated in the stored quantum circuit's QISA. This implementation therefore allows run-time tweaking of circuit parameters. Therefore, in order to implement the support for executing VQ algorithms, the CC-Spin micro-architecture needs to support a python compiler implemented on the on-board ARM processor and an AXI bus communication line with the quantum pipeline.

3.3. ENCLUSTRA HARDWARE

The micro-architecture is implemented on a Mercury+ SA2 SoC-FPGA board. The module has a Altera Cyclone V, Dual-core ARM processor-based SoC, with a fast DDR3L SDRAM, USB 3.0 controller, PCIe, Gigabit Ethernet, multi-gigabit (MGT) transceivers and high-speed LVDS I/O. This

makes it a powerful platform for implementing the high-speed designs. Enclustra hardware requires building it's own linux environment for accessing ARM-FPGA based system control.

The setup instructions and build files for Enclustra Cyclone-V SoC device is accessible at my GitHub account: https://github.com/amitabhyadav/enclustra_cyclone_5_soc. The compiled software such as, U-Boot, Linux, and BusyBox; and the necessary binaries, such as bitstream and preloader are accessible there. All necessary documentations to get started with Enclustra hardware can be accessed at: https://download.enclustra.com/.

Merits and Demerits The Mercury+ SA2 board is used in conjunction with the Enclustra PE-1 baseboard. The base board provides high speed interfaces for rapid-prototyping and has extensive features such as FMC connectors and MGT links. However there is no interface for direct FPGA programming. A separate USB Blaster II cable is required to get the 'only-FPGA' designs running. Further, there is no netlist for accessing internal clock directly from the FPGA. For sequential FPGA designs the board requires an external clock or, the design has to be compiled with an Hard Processing System (HPS) i.e. full HPS-FPGA design to access the clocks 50MHz and 25MHz from the HPS. Further, having a completely separate infrastructure and modified extensively for high-speed applications, little to none developer community exists hardware development which makes using the hardware very hard to use. Due to the steep learning curve, we switch to the Cyclone V hardware available from Terasic Inc. The hardware lacks some of the important features such as PCIe, MGT etc but, is good for high-speed applications and rapid-prototyping.

3.4. WAVEFORM GENERATION

In classical computers, input and output operations occur in the digital domain and expressed as a binary (expressed in Hexadecimal) representation for data and instructions. However qubit control is performed through analog pulses. Arbitrary Waveform Generators/Function Generators are programmed by writing an equation and/or uploading arrays containing waveform points in the AWG waveform memory. These waveform are applied to the qubits with precise timing using external (8-bit) codeword-based triggers. So, for each AWG channel, an external control micro-architecture can trigger $2^8 = 256$ different waveform from the channel's memory. A digital-to-analog converter (DAC) converts the waveform to analog domain, which are modulated with a microwave carrier signal and applied to the qubits. Similarly, the readout signal is obtained by measuring the (very weak) currents from the qubit chip in the analog domain and converted to the digital domain using a Digitizer card.

Operations such as initialization, control and readout, all require efficient waveform generation. The key role of the Arbitrary Waveform Generator is for generating IQ-waveform pulse envelopes and DC Pulses. However, AWGs are not a long-term solution for efficient quantum computation due to limited memory and long wait times for waveform uploads. Furthermore, they limit the number/types of operations that can be performed and do not support active qubit reset or feed-forwarding in quantum computation.

3.4.1. IQ SIGNALS

I/Q signals, also known as analytic signals, refers to two sinusoidal signals having the same frequency and a relative phase shift of 90°. I, "in-phase signal", is a cosine waveform and Q, "Quadrature-phase signal", is a sine waveform i.e. Q is shifted by 90° relative to I. By adding identical I and Q signals of equal amplitude, we get a sinusoidal signal with phase exactly between the phase of the I/Q signal. In I/Q modulation we can multiply I/Q waveforms by modulating signals having negative voltage values, and causing an "amplitude" modulation with a 180° phase shift. However, non-identical I/Q signals introduce a phase shift. Due to this, we can also achieve Phase modulation by varying the amplitude of the I/Q signal because, by increasing either of the

amplitudes will causes the signal sum to shift toward the higher-amplitude waveform.

We can also encode the two-dimensional I/Q signal onto the one-dimensional RF signal without losing data. The amplitude and the phase of the carrier of a given frequency can be modified. They are used to encode the I/Q data. We can encode it on a carrier RF of frequency f as follows:

$$\text{Modulated Carrier RF} = I.cos(2\pi f t) + Q.sin(2\pi f t)$$

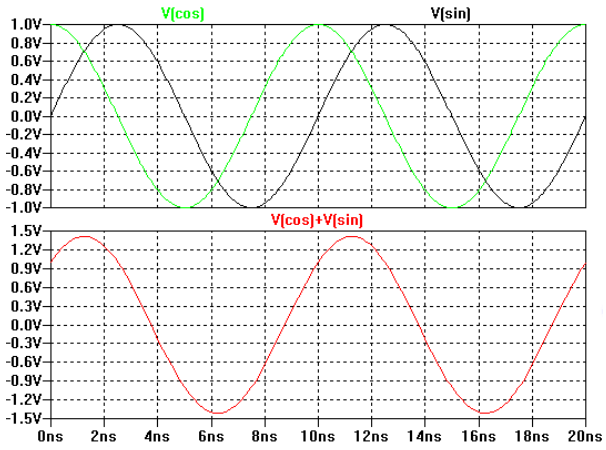


Figure 3.11: I and Q signals phase shifted by 45° and modulated waveform

By adding a cosine with corresponding sine component of the same frequency (as of carrier), we can change the phase and amplitude of the resulting RF signal. We can transform it back, as follows:

$$I = \text{LPF}(\text{RF}.cos(2\pi f t))$$

$$Q = \text{LPF}(\text{RF}.sin(2\pi f t))$$

Vector Signal Generators have inbuit quadrature modulator that accepts I(t) and Q(t) signals which is used to amplitude modulate a pair of quadrature sinusoidal signals and then added to get the modulated RF output.

3.4.2. AWG DESIGN PRINCIPLES AND REQUIREMENTS

AWGs are usually one of the four types — *True AWGs*, *DDS-based*, *Interpolating DAC* and *Interleaving DAC*. True AWGs read the waveform samples sequentially from the waveform memory and a DAC converts these samples to analog waveform at the set sampling rate. In the DDS-based AWG, the DAC works has a fixed sample rate whereas the repetition rate of the waveform stored in memory can be programmed. In the Interpolating DAC AWGs, signals are generated at a higher sampling rate than the sample-access rate. This is done by calculating the interpolated samples values in real-time using a processing block which lies between the the waveform memory and the DAC. Finally, the Interleaving DAC AWG allows a higher effective-sampling rate by combining the outputs of two DACs through switching or by simple summation.

Certain requirement complying with Arbitrary Waveform Generators[6] for quantum processor control are given in the Table 3.2, below:

[6]Phase Noise, Jitter, and Spurious-Free Dynamic Range (SFDR) are the performance metrics for waveform generators.

Pulse Length	Usually 10 to 100,000 ns
Codeword length	8-bits/AWG IQ pair channel
Delay from Trigger to Waveform Output	within 100 ns
Time between two consecutive waveform triggers on the same AWG channel	20 ns
Number of waveform of length 100 [µs] @ max sample rate that fit in memory	256 waveform IQ pair/AWG channel
Maximum output voltage (DC coupled)	1 V_{pp} 50Ω impedance
Number of channels required	2 for IQ signal pair and 14 for DC pulses (for few qubit experiments)

Table 3.2: Requirement compliance for Arbitrary Waveform Generators (AWGs)

3.4.3. USING PROGRAMMABLE AWG

Digital to Analog Converters (DAC) are devices that convert digital signals to analog signals. In this implementation, we use ADA (analog-to-digital-to-analog) converter available from Terasic. The board contains 2-channel, 14-Bit, 125 MSPS Digital-to-Analog Converter (AD9767). The DAC can be operated with two separate data ports, or with a single interleaved high speed port, which is specified through a mode select input pin.

The programmable AWG is basically an FPGA feeding 14-bit digital data to a DAC. The 14-bit resolution determines the output waveform. The FPGA receives micro-operations from the Master controller through high-speed LVDS links. The SERDES core deserializes the input data stream and stores the instructions in an Instruction FIFO buffer. Using the instruction specifications, the Timing Control Unit issues codeword specifying memory locations in the waveform memory which generates the output waveform. The implementation of Timing Control unit remains to be completed while we implemented the HDL design for generating simple sinusoidal waveform with the DAC based AWG. This implementation is given in section 4.3.

The architecture of the Slave FPGA driving a DAC based AWG is shown in figure 3.12. The slave FPGA buffers the operations in timing queues and executes triggers the memory controller for waveform generation.

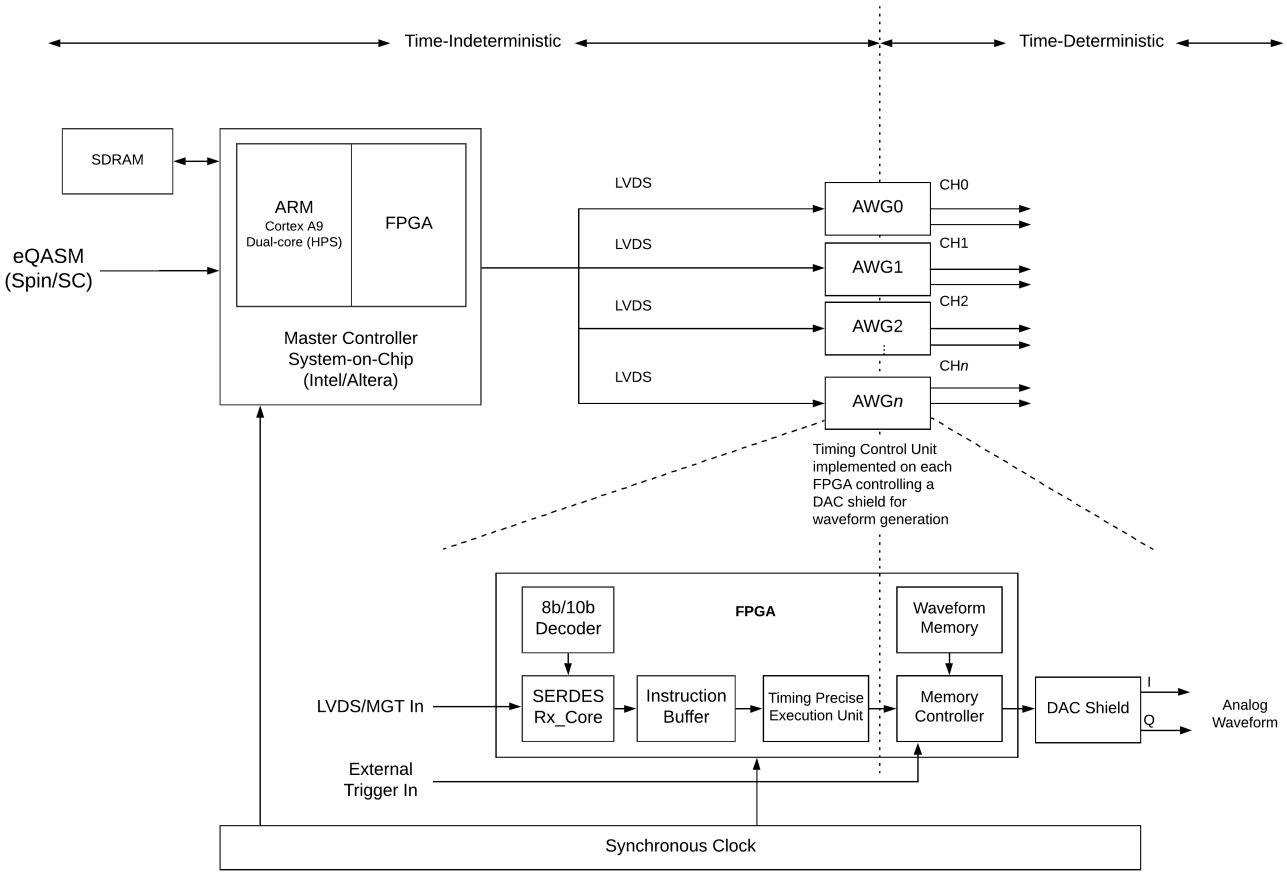


Figure 3.12: Waveform Generation using a True AWG Architecture

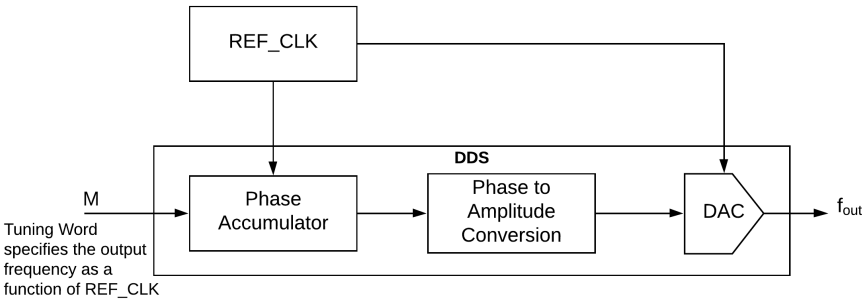
3.4.4. USING DDS BASED AWG ARCHITECTURE

Direct digital synthesis (DDS) hardware uses a method of producing an analog waveform (Square, Triangular, and sinusoidal waveform) by generating a digital time-varying signal and using a digital-to-analog converter to get the analog waveform. DDS allows fast switching between output frequencies, fine frequency resolution and allows a broad frequency spectrum for operation, as all internal operations are digital. The DDS are low-power, low-cost and can generate programmable analog output waveform with high resolution and accuracy. It also has low-phase-noise and variable frequency generation capability (<1 Hz up to 400 MHz, on a 1-GHz clock) for waveform generation. In this project, we use the AD9910 DDS-DAC based waveform generation. This DDS is programmable via a high-speed serial peripheral-interface (SPI) protocol that requires to be developed separately for the DDS-chip. The timing diagrams and instruction definition for designing the SPI interface is available on the AD9910 data-sheet.

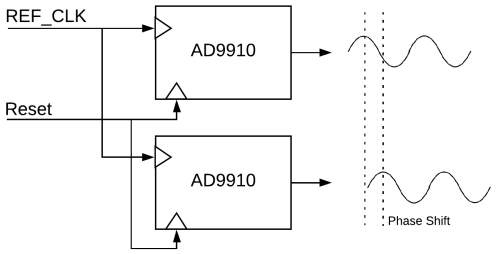
The main components of a DDS device are a *phase accumulator*, a means of *phase-to-amplitude conversion* (often a sine look-up table), and a DAC. The DDS internal circuitry is shown in figure 3.13a. In a DDS, the frequency depends on the reference clock and the tuning word.

The phase accumulator is a modulo-M counter. Every time it receives a clock pulse, it increments its stored number. The binary-coded input word (M) determines the magnitude of the incremented value. This word forms the phase step-size between reference-clock updates i.e. effectively sets how many points to skip around the 'phase wheel'. With larger jump size, the phase accumulator overflows faster and completes its equivalent of the sine-wave cycle. The resolution of the phase accumulator(n) (the tuning resolution of the DDS) determines the number of discrete phase points in the phase wheel.

A phase-to-amplitude look-up table is used to convert the phase-accumulator's instantaneous output value into the sine-wave amplitude information that is sent to the (10-bit) DAC. Since the sin wave is symmetrical, the DDS architecture uses only a quarter-cycle of data from the phase accumulator to generate a complete sin wave. The remaining data is generated by reading forward then back through the phase-to-amplitude look-up table.



(a) DDS components and signal flow



(b) Multiple synchronous DDS for IQ signal generation

Figure 3.13: Waveform generation using DDS

Because a DDS is digitally programmable, the phase and frequency of a waveform can be easily adjusted without the need to change the external components that would normally need to be changed when using traditional analog-programmed waveform generators. This offers an advantage in using a DDS. Further, DDS chip offers advantage for controlling quantum systems as it allows simple real-time adjustments in frequency to locate resonant frequencies or compensate for temperature drift.

IQ WAVEFORM GENERATION USING DDS-DAC

The chosen DDS solution for waveform generation is the AD9910 1 GSPS, 14-bit, 3.3V CMOS Direct Digital Synthesizer that can generate sinusoidal frequencies up to 400MHz using the 3 control parameters — amplitude, phase and frequency. Note that the REF_CLK frequencies must be 60-1000 MHz (with Clock Multiplier Disabled) and 3.2 - 60 MHz (Clock Multiplier Enabled); and Voltage levels must be 50 - 1000 mV (Single-Ended) 100 - 2000 mV (LVDS). The Detailed block diagram of the chip is shown in figure 3.14.

We can use two DDS units operating on the same REF_CLK to output the I/Q signals whose phase relationship can then be controlled. To do this, we use both AD9910 with the same REF_CLK and the same Reset that updates both. This setup to do I/Q waveform generation is shown in figure 3.13b. Note that the reset must be asserted after power-up and prior to transferring any data to the DDS. This sets the DDS output to a known phase, which serves as the common reference point that allows synchronization of multiple DDS devices. When new data is sent simultaneously to multiple DDS units, a coherent phase relationship can be maintained, and their relative phase offset can be predictably shifted by means of the phase-offset register.

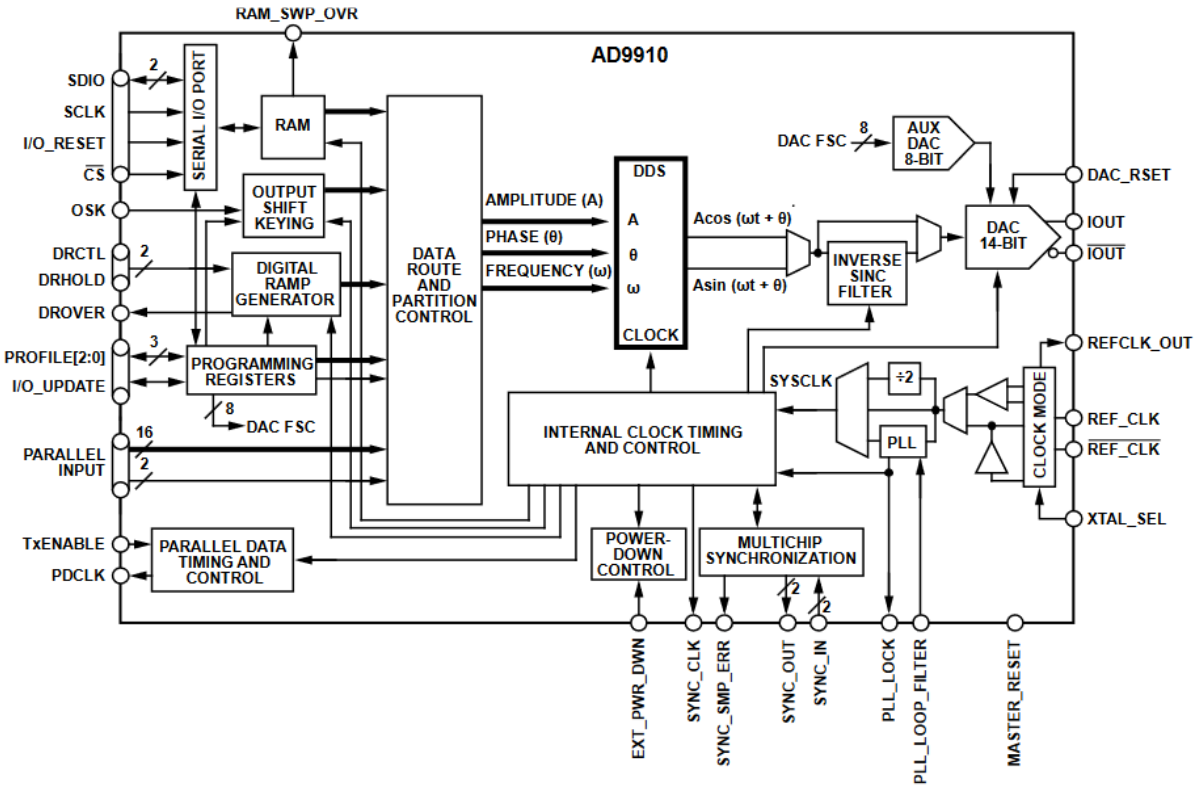


Figure 3.14: Detailed Block Diagram of AD9910 1GSPS, 14-bit, 3.3V CMOS DDS-DAC [3]

The micro-architecture developed around the Direct Digital Synthesis (DDS) based system is shown in figure 3.15. One DDS unit is controlled by a slave FPGA. The firmware for the FPGA is the same as that of DAC based programmable AWG, except an SPI Master interface is used to program the DDS-Slave device. The parameters for programming the DDS through SPI interface is extensive, and requires to be designed from scratch in VHDL/Verilog following the design parameters as provided in the AD9910 data-sheet. The DDS unit can generate control waveform in real-time using input from the Timing Precise Execution Unit (TPEU). Therefore, we can alternately use a simpler instruction set such as given in figure 3.3 to implement a simpler TPEU with simple registers. Besides this, the setup also uses a very low-noise AD9517 PLL clock synthesizer with an integrated VCO, clock dividers, and 14 outputs. Beside the DDS, we also have multi-channel DAC and ADC connected to the master controller. The DAC facilitates simple DC waveform generation and ADC converts the analog measurement of qubit states to digital data.

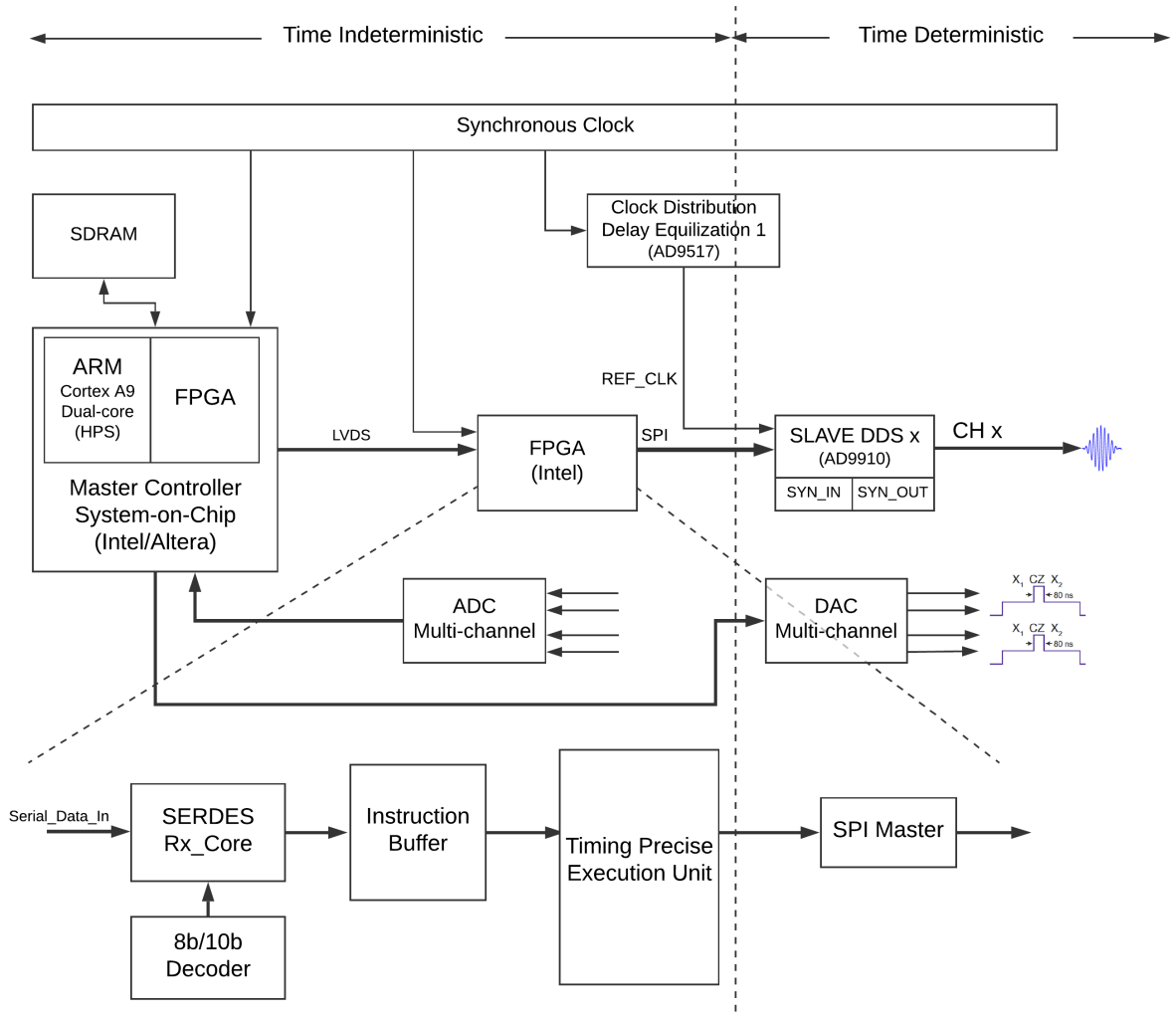


Figure 3.15: Waveform Generation using a Direct Digital Synthesis DDS-DAC

3.4.5. MULTI-DEVICE SYNCHRONIZATION

The AD9910 DDS-DAC chip facilitates synchronizing multiple channels for waveform generation. The setup implements multiple slave FPGAs driving multiple DDS units through the SPI-master controller interface. All FPGAs work synchronously with SYN_CLK of the DDS units, while DDS hardware allows synchronizing multiple DDS devices using the clock distribution and delay equilization board AD9517. The DDS chip has a sync generator block that makes one of the DDS unit as the master and remaining as the slaves. The master produces an LVDS SYN_OUT clock ($f_{SYN\_OUT} = f_{SYSCLK}/16$) which synchronizes the remaining boards with the rising or falling edge of $SYSCLK$. The detailed circuit for the setup is shown in figure 3.16.

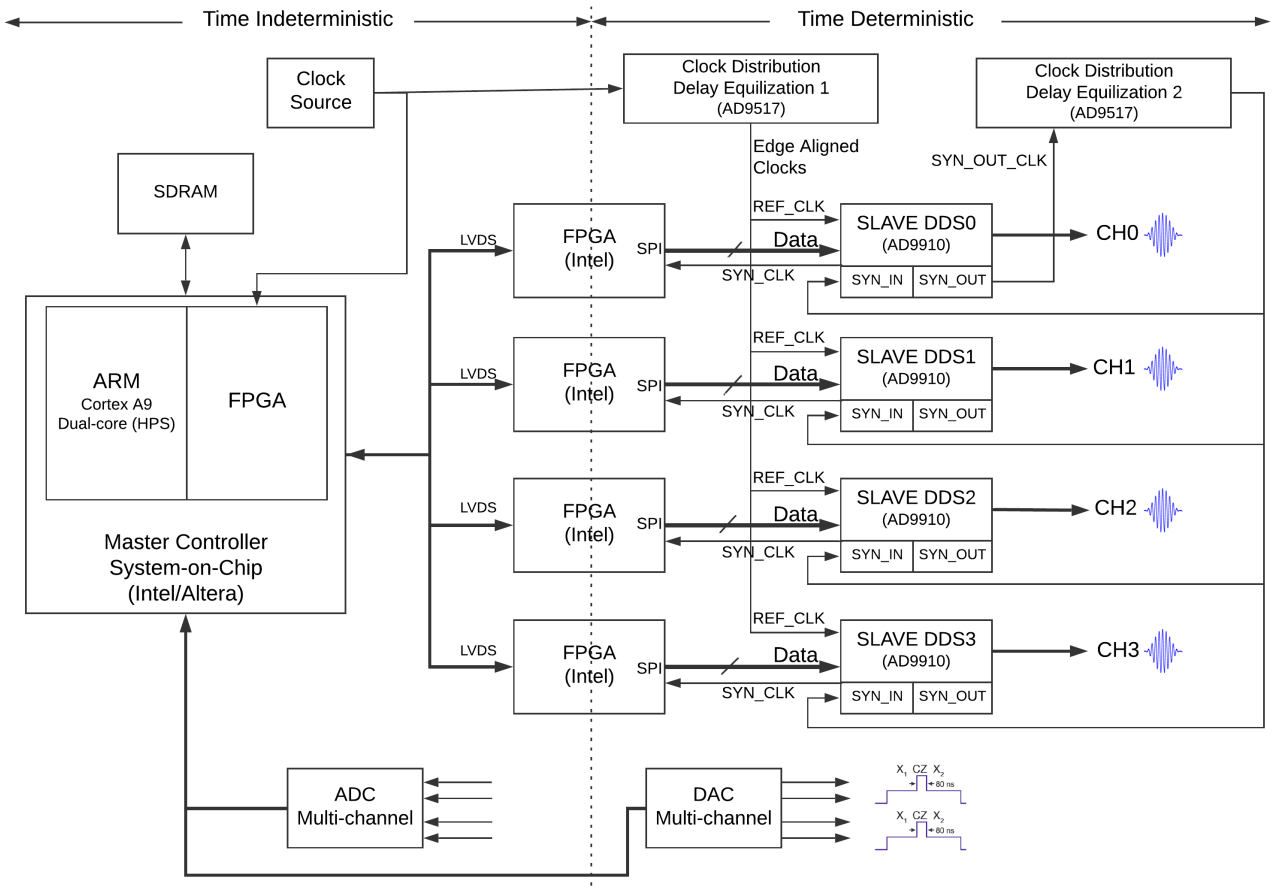


Figure 3.16: Implementation of Synchronized Waveform generation using AD9910 DDS-DAC

3.5. CONCLUSION

In this chapter, we highlighted the micro-architecture definition and it's functions. We discussed the detailed quantum pipeline and all it's components that facilitate quantum instruction execution flow. We then introduced two new QISA formats and discussed their scaling capability. In the later half of the chapter, we discussed Waveform Generation methodology using a DAC based AWG and DDS based AWG. In the last section, we presented a synchronized quantum control setup using multiple DDS and multi-channel DAC for qubit control and multi-channel ADC for measurement data acquisition. In the next chapter, we will discuss the FPGA implementation and results from the waveform generation hardware .

4

HARDWARE TESTING & RESULTS

We have described the microarchitecture design and various components of the same in the previous chapter. This chapter emphasizes on the testing methodology of these sub-components. An experimental realization of the CC-Spin control box is shown in Figure 4.1.

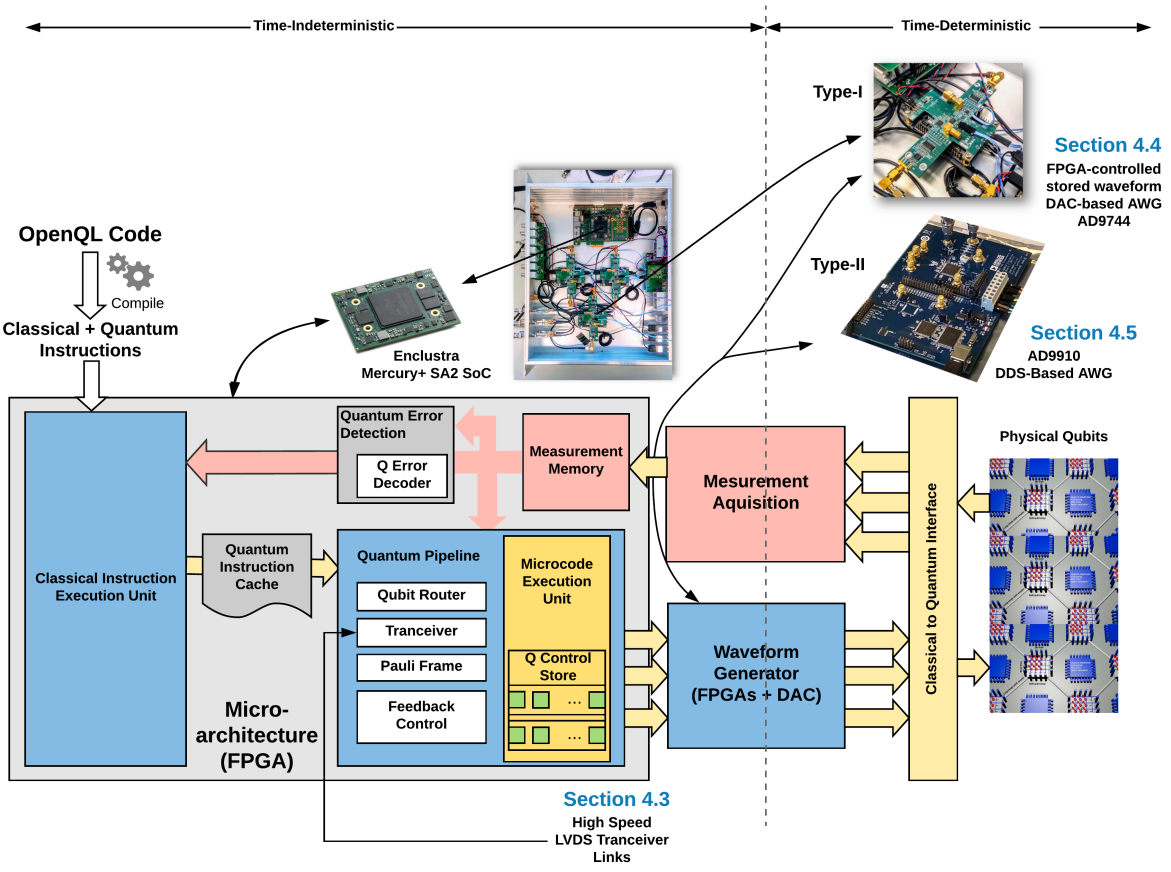


Figure 4.1: Schematic of Experimental Setup

The microarchitecture runs on a Master FPGA controller that issues commands for signal generation to different Waveform generators (DDS/DAC or Modular AWGs). The Master FPGA controller also hosts the Measurement Acquisition Unit that performs filtering, demodulation, integration and thresholding. The Master Controller Microarchitecture firmware is implemented

on Enclustra Mercury+ SA2 SoC-FPGA, and the FPGA-controlled DAC AWG is implemented on a Terasic DE10 Nano FPGA with AD9744 4-Bit, 210 MSPS DAC as a part of CBoxV3 hardware (Figure 4.1). While the complete implementation of the microarchitecture is complex and requires a longer time to develop, we implemented specific sub-components of the microarchitecture and obtained their results. Therefore, in the following sections, we have presented results from hardware testing of some of the components of the microarchitecture - the LVDS High Speed Transceivers and Waveform generation using two methods (DDS-based and DAC-based), also indicated in the figure 4.1.

4.1. IMPLEMENTATION OF FIFO BASED TIMING CONTROL UNIT

In this section, we first discuss the Timing Control Unit (TCU) of the microarchitecture, the relevance of FIFO buffers and the implementation and simulation of Asynchronous First-In First-Out (FIFO) buffers.

4.1.1. ASYNCHRONOUS FIFO BASED TIMING CONTROL UNIT

The microarchitecture implementation consists of a Queue-Based-Event- Timing Control Unit which is responsible for accurate and deterministic issue of code word based instructions to the waveform generators [43]. These queues are asynchronous FIFO buffers that are controlled by a *timing controller* that uses the instructions in the *timing queue* to activate *multiple event code word buffers*. This implementation usually consists of Mega functions from the vendors and hides away the implementation details of the the timing queues. This hinders further development in ASIC, or porting of platforms during microarchitecture design. Asynchronous FIFOs are nontrivial to design, therefore, here we present the implementation of a synthesizable asynchronous FIFO event-queue buffer that can handle clock domain crossing.

Asynchronous FIFOs use different clocks for reading and writing data. Asynchronous FIFOs are find-applications when data is transmitted from one clock domain to another. Clock domain crossing introduces asynchronous behaviour in the digital design and FIFOs help to overcome this through an efficient design. A hardware-synthesized FIFO consists of a data storage area, read and write pointers and a control logic. Usually, we use a dual-port SRAM (Static Random Access Memory) where one port is dedicated to reading data and the other port is dedicated to writing data.

In this chapter, we will present an Asynchronous FIFO design based on Gray Code counter read/write pointer generation which enables reliable flag generation for a stable operation [44].

In the FIFO design, we use read (`rptr`) and write (`wptr`) pointers. The pointers are associated to asynchronous clocks — `rclk` and `wclk`, respectively. If we are using simple binary bits as pointers, we see that in a situation where the pointer is changing from, for example FFFF to 0000, and every single bit goes metastable as the single-bit synchronization techniques (two flip-flip) does not hold here. This means that any value between 0000 and FFFF can be read, and the FIFO will not function correctly. In this case, single-bit synchronization techniques are not completely sufficient.[1]

However, if it is ensured that only 1 bit changes every time the pointer increments, we can avoid the problem of synchronizing multiple changing bits of the pointer on the clock signal. Therefore, we use Gray Code synchronization to safely pass `rptr` and `wptr` across asynchronous clock domains.

4.1.2. GRAY CODE COUNTER SYNCHRONIZATION

The gray-code counter used in our Asynchronous FIFO design is a Dual n-bit Gray code counter. Dual n-bit means that we generate an n-bit as well as (n-1)-bit gray-code. The n-bit gray codes helps

[1] Other techniques such as Handshaking can be used to synchronize multiple bits too, but it reduces bandwidth significantly.

to synchronize the pointers in the read and write clock domains. The (n-1)-bit gray code is used to generate the `addr` the FIFO buffer. The implementation of a gray-code counter is straightforward (such as given in appendix A.1), and for (n-1) bit gray code, we do a XOR gate on the two MSBs of n-bit Gray code to get the MSB of (n-1)-bit gray code. This is combined to (n-2) LSBs of the n-bit Gray Code to get the final (n-1)-bit Gray Code Counter.

4.1.3. IMPLEMENTATION AND SIMULATION

The structure of Asynchronous FIFO design implemented is shown in figure 4.2. The functional simulation is given in figure 4.3 and implementation of the same is given in appendix A.2.

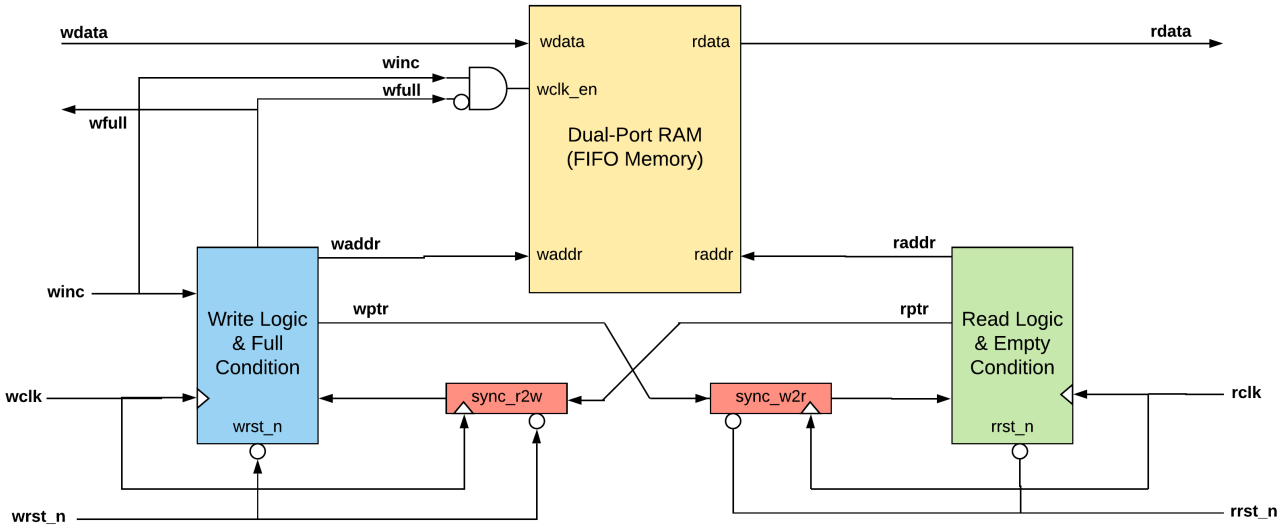


Figure 4.2: Block diagram of Asynchronous FIFO

The FIFO size is implemented using an on-chip dual-port SRAM. We use a write pointer (`wptr`) and a read pointer (`rptr`) to determine the write address and read address. Both pointers increment in the usual way 0x00 to 0xFF. As discussed above, here we use pointer represented in gray code.

The FIFO full condition occurs when the write pointer catches up with the read pointer and a FIFO empty condition occurs when the read pointer catches up with the write pointer. To generate full/empty conditions, the write pointer and read pointer need to pass values across the read and write clock domains.

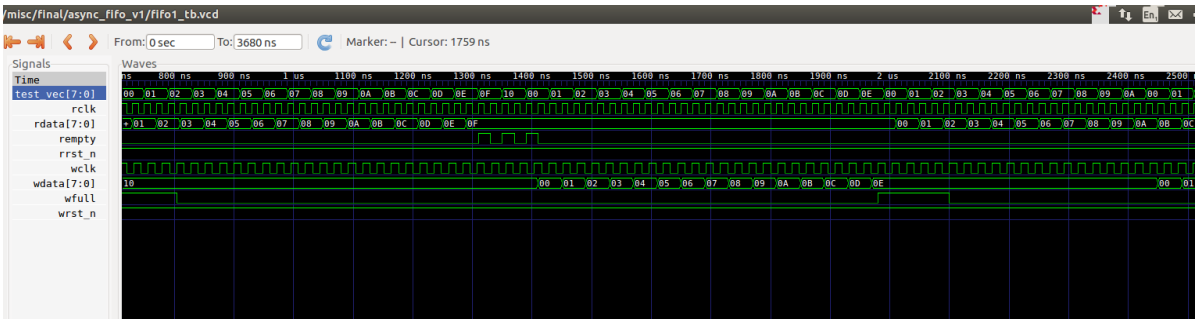


Figure 4.3: Functional Simulation of Asynchronous FIFO

FIFO EMPTY AND FIFO FULL CONDITIONS

We will now discuss the Empty and Full Conditions of Asynchronous FIFO design. In the implemented design, the Empty flag is implemented in the read logic and Full Flag is implemented

in the write logic.

FIFO Empty Condition : This is done trivially by comparing the n-bit read pointer to the synchronized n-bit write pointer. Pointers of size `raddr + 1` size is used. If the pointers of both, read pointer and synchronized write pointers are equal, it implies that the FIFO is empty.

FIFO Full Condition : Generating a Full condition is not trivial. For the Full condition, we synchronize the `rptr` with `wclk` and meet the following conditions: [44]:

1. The `wptr` and the synchronized `rptr` MSB's are not equal.
2. The `wptr` and the synchronized `rptr` 2nd MSB's are not equal.
3. All other `wptr` and synchronized `rptr` bits must be equal.

REMARKS

Asynchronous FIFO design, including Full flag generation is non-trivial and can easily deter the performance of the FIFO. Multi-bit synchronization can be accomplished using Gray Code pointers instead of binary. The maximum frequency of the clocks governs the timing condition for the operation of the FIFO. The clock frequency cannot be greater than what is necessary for the memory and MTBF needs to be taken into consideration. The design presented in here functions without error with both slow and fast clocks, however for faster applications, *Mesochronous synchronization* can be employed.

4.2. IMPLEMENTATION OF HIGH-SPEED LVDS LINKS

The distributed architecture of CC-Spin requires fast communication between the master controller and slave FPGAs. The slave FPGAs, by means of the received data, generate appropriate signals for for quantum control and readout. The communication can be established by any serial communication means such as, MGT (Multi-gigabit Transceivers), SFP (small form-factor pluggable) or mini-GBIC (gigabit interface converter), or LVDS (Low-voltage Differential Signalling). We implement the the high-speed LVDS transmission links for three reasons: High pin-count of LVDS pairs are available on the Master Controller for Data and Clock transmission, LVDS is good for short distance data transmission, is faster than single-ended transmission and is less prone to Electromagnetic Interference.

4.2.1. 8B/10B ENCODER/DECODER

The 8b/10b encoding mechanism is used for serial communication protocols such as SATA, PCIe and Optical fibre SFP. Using the encoder, we convert an 8-bit data to 10-bit data. The 8b/10b encoding helps to reduce the repetitive occurrences of 0 and 1, by distributing the bits and thus providing a DC balance to the overall signal. The 8b/10b encoding mechanism is shown in figure 4.4. Note that the encoded 10b-data also contains control characters that indicate for example, start-of-packet/end-of-packet etc. The same 8b/10b algorithm can be used for both encoding and decoding. Multiple bytes of data can also be encoded in a one clock cycle with the 8b/10b encoder by cascading the encoders. In order to avoid incurring delay in the circuit, we can also employ 'Disparity Select Scheme'.

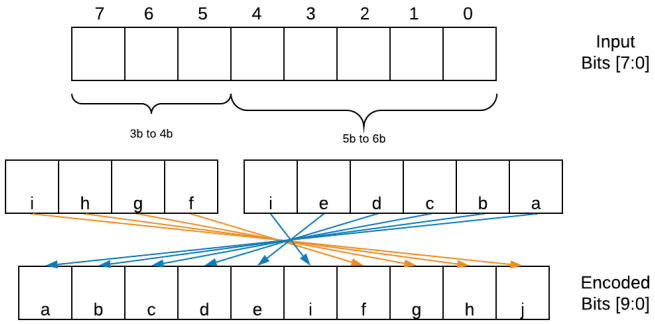


Figure 4.4: 8b/10b encoding

8b/10b encoding adds overhead to the transmitted data, as for every transmitted 10-bits only 8-bit data is actually transmitted. Therefore, we can also use 64b/66b or 128b/130b encoding schemes. The number of transitions are reduced but the overhead reduces too.

4.2.2. IMPLEMENTATION

The RTL schematic of the implemented circuit is shown in the Figure 4.5. In this simple example, we have a `data_generator` counter unit that outputs an 8-bit data in every clock cycle, the SERDES is an Intel TX IP Block that serializes the data and transmits via the LVDS TX pins. To check the protocol's correctness, we implement an RX SERDES block and a `data_checker` block on the same FPGA with input pins assigned to an RX-LVDS pair. The data_checker checks reception of particular data bits and blinks an FPGA LED when the data occurs.

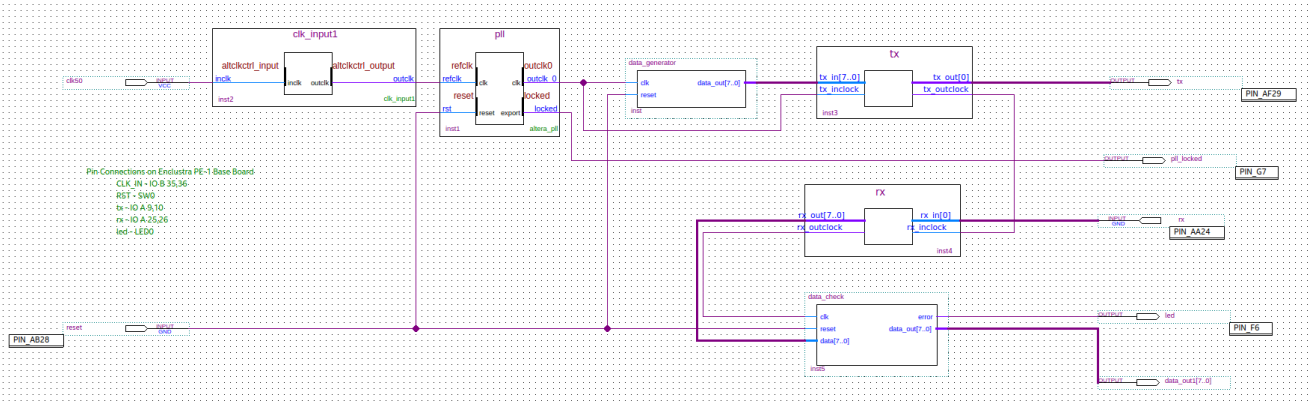
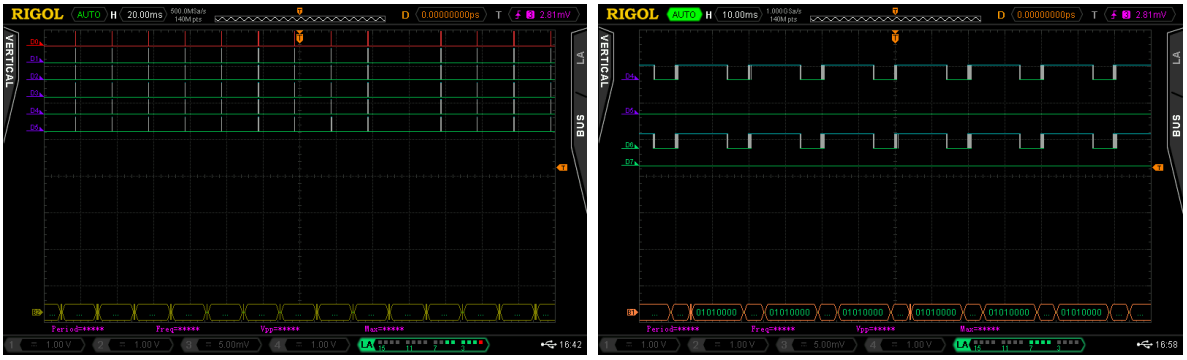


Figure 4.5: Schematic Serial Data Transmission firmware using ALT_TX and ALT_RX SERDES Megafunctions

The functioning of this circuit has not been completely verified yet. We tested data_out stream on the Logic Analyzer for simple set. The screenshot of is shown in Figure 4.6. However, in order to verify the correctness and performance of data transmission, we would proceed to analyze the the Eye Diagram of the serial data transmission. However due to limited feature on the RIGOL MSO4024 Oscilloscope, we did not proceed to do that. Note that while recording the output of LVDS, it is important to terminate the line with a 100Ω resistance. In an eye-diagram, a wider open eye indicates better transmission quality of data. We can also determine other parameter, such as, Bit Rate, Jitter etc.



(a) 6 bit data_out single ended signal (b) LVDS Clock and Data Signal

Figure 4.6: Using Logic Analyzer to access data for single-ended and LVDS signalling

4.3. IMPLEMENTATION OF FPGA CONTROLLED DAC-BASED AWG

Digital to Analog Converters (DAC) are devices that convert digital signals to analog signals. In this implementation, we use ADA (analog-to-digital-to-analog) converter available from Terasic. The board contains 2-channel, 14-Bit, 125 MSPS Digital-to-Analog Converters (AD9767). The DAC can be operated with two separate data ports, or with a single interleaved high speed port, which is specified through a mode-select input pin in the RTL design. The RTL design uses Numerically Controlled Oscillators with a 'Multiplier-based Architecture' to generate SIN and COS values [2]. The multiplier architecture implements a multiplier circuit in the logic elements of the FPGA and helps reduce memory usage. Alternately, we can also use 'Large ROM Architecture' with requires a larger internal memory but performs best for high speed applications. Our experience shows that large ROM architectures give the highest spectral purity and utilizes least number of logic elements on the FPGA.

Except CORDIC and multi-cycle multiplier-based architectures, all NCO-architectures output a sample every clock cycle. After asserting clock-enable, the oscillator outputs the sinusoidal samples at one sample per clock-cycle rate, following an initial latency of L-clock cycles. The exact value of L varies across architectures and parameters. The operation is shown in Figure 4.7. After the clock enable is asserted, the oscillator outputs the sinusoidal samples at a rate of one sample for every two clock-cycles, following an initial latency of L clock cycles.

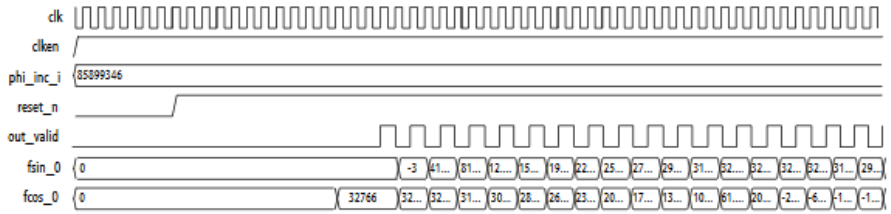


Figure 4.7: Two-Cycle Multiplier-Based Architecture Timing Diagram

4.3.1. USING SIGNALTAP

We have used the Quartus SignalTap II Embedded Logic Analyzer software which allows us to capture signals from internal RTL-design nodes from the FPGA in real-time. The tool is very useful to in debugging the design logic running in real-time on the FPGA hardware. The tool is available as a mega-function and can be integrated in the design schematic. We use the .stp file to define the

[2] The architecture can be Large ROM-based, Small ROM-based or CORDIC algorithm-based. All are used to generate SIN and COS values.

parameters we want to fetch. We setup the parameters by specifying their name and sample depth that we wish to 'tap'. The design is then compiled[3] and is uploaded to the hardware via the USB Blaster Interface. By running analysis on the interface, we can get the data. The real-time data obtained from ADC and DAC implementation is shown in Figure 4.8.

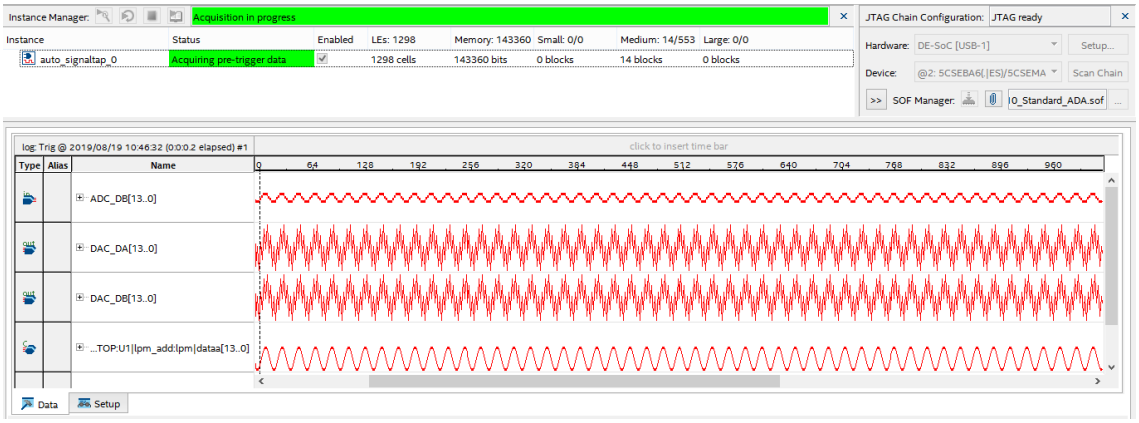
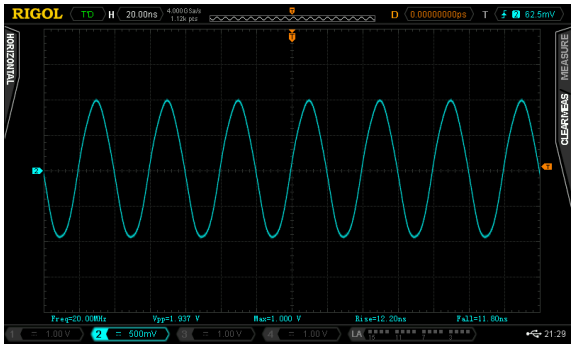
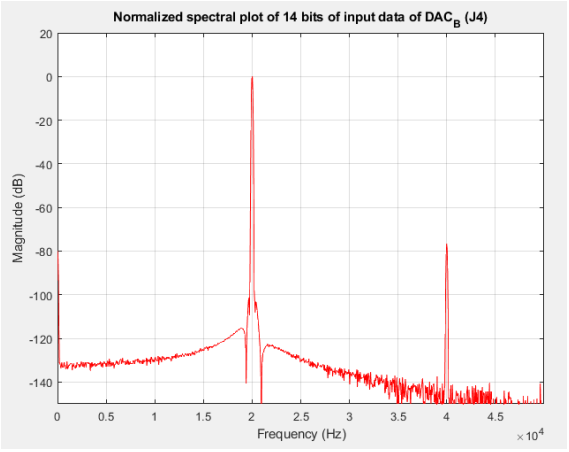


Figure 4.8: Real-time ADC and DAC waveform obtained through Signal Tap

Furthermore, we can record the data in a Signal Tap-II list file and get the Spectral plot by post-processing in MATLAB. In MATLAB, we take the DAC-input samples from the list file, perform normalization (subtract DC components), perform FFT transform and plot the data. The Normalized Spectral Plot of DAC performance is shown in Figure 4.9b. The main output frequency is at 20MHz and the 1st harmonic is at 40 MHz.



(a) SIN wave generated using DAC based AWG



(b) Normalized Spectral Plot of DAC

Figure 4.9: sine waveform and Normalized Spectral Plot of DAC, generated using Signal Tap and plotted in MATLAB

4.3.2. HARDWARE IMPLEMENTATION

The RTL-Netlist schematic of the DAC-based Waveform Generator is shown in Figure 4.10. The NCO generates SIN and COS waves based on the phase_increment value and an lpm_add_sub Megafunction adds the components together to produce a modulated wave. Using this circuit, we are producing a SIN-wave and 90° phase shifted wave (COS wave).

[3]Using Signal Tap incurs large area consumption in terms of Number of Logic Elements (LEs) occupied

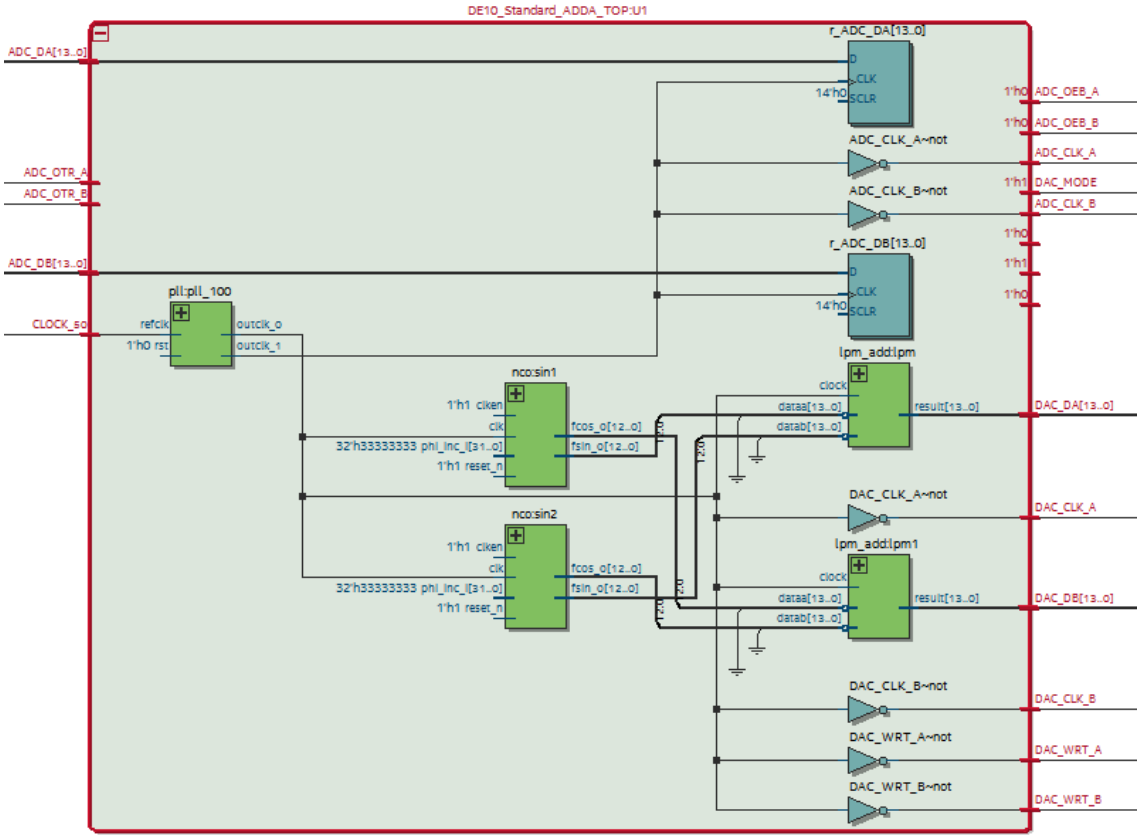


Figure 4.10: RTL Design of DAC-ADC implementation on Terasic DE10 Standard SoC-FPGA

The setup is shown in the Figure 4.11 and the analog signal out is shown in figure 4.12a. We further changed the values in the phase_increment to obtain the shape of a Gaussian Waveform (4.12b).

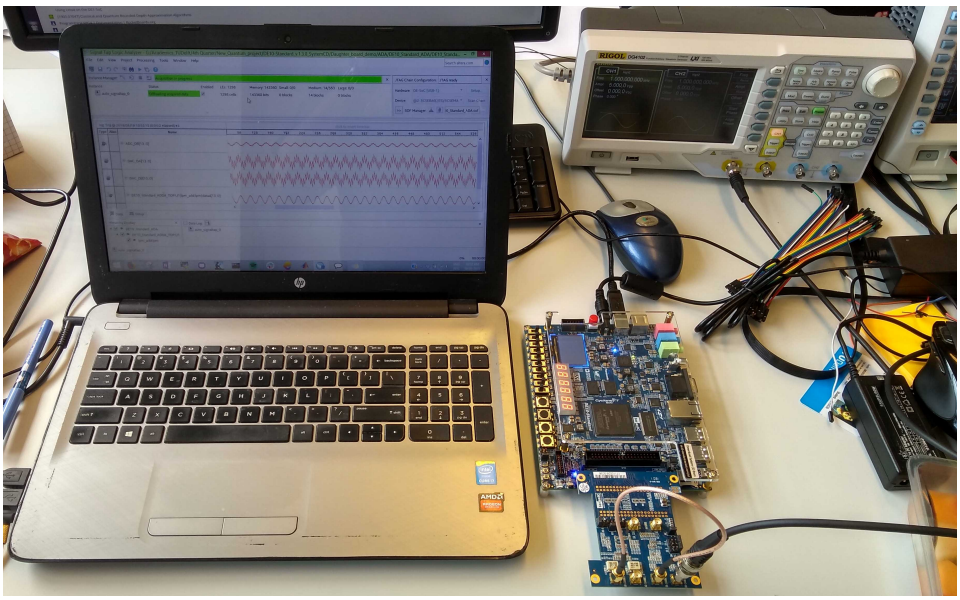
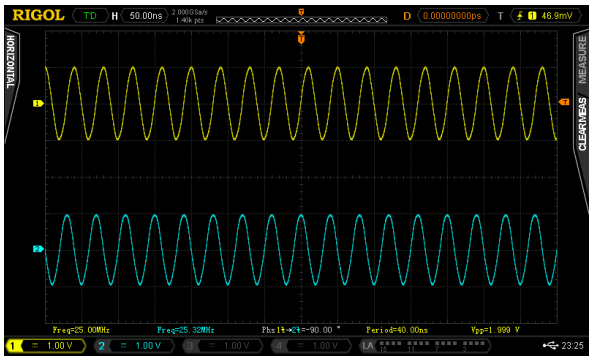
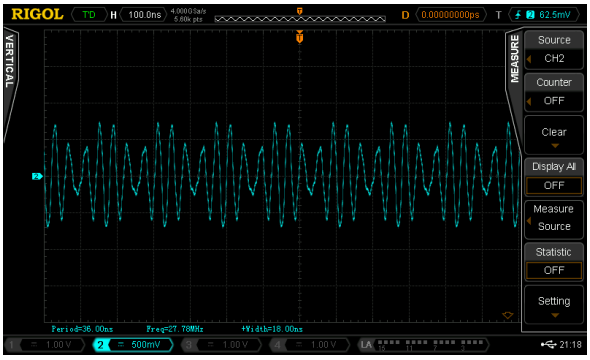


Figure 4.11: DAC based waveform generation Setup



(a) IQ SIN wave 25MHz, V_pp=2V from Terasic Cyclone V with DAC board



(b) Gaussian waveform (distorted) from DAC obtained by mixing waveforms on Terasic Cyclone V and converting to analog using DAC. These can be modified to obtain DRAG pulses.

Figure 4.12: Output waveforms using Terasic Cyclone V and DAC board

4.4. IMPLEMENTATION OF DDS-BASED AWG

In this section, we will discuss the demonstration of AD9910 Direct Digital Synthesis unit using the Evaluation Board. The block diagram of the evaluation board is shown in the Figure 4.13. There are two ways to program the DDS board: first, by using the USB interface, and second, by using the parallel port. The vendor, Analog Devices, provides a GUI-based evaluation software to facilitate easy communication through the USB interface. We program the board's internal control registers via either of the interfaces. The DDC-DAC board provides access to three parameters for controlling the DDS — frequency, amplitude and phase. We can tune the frequency using the 32-bit accumulator and at a sample-rate of 1GSPS, the tuning resolution is approximately, 0.23 Hz. The board also allows for fast phase and amplitude switching; and supports sinusoidal waveform outputs at frequencies up to 400 MHz.

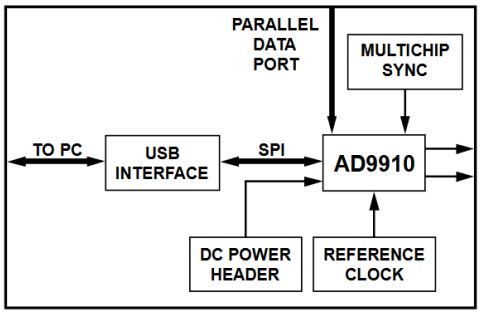


Figure 4.13: AD9910 DDS Evaluation Board Block Diagram

We can operate the AD9910 in one of the four modes: Single-tone, RAM modulation, Digital ramp modulation, and Parallel data port modulation.

In single-tone mode, the serial I/O port is used to program the internal registers. The registers provide the signal parameters to the DDS. Independent registers containing the DDS signal parameters are referred as 'profile'. There are 8 profile registers and are independently accessible using the external pin `PROFILE[2:0]`. Change in profile pin state is reflected with the next rising edge of SYNC_CLK and DDS output gets updated with next profile's parameter.

In the RAM-modulation mode, signal parameters are fetched from the internal RAM and played when triggered. The RAM has a depth of 1024 samples and consists of 32-bit words. Using the RAM-mode, we can generate arbitrary time-dependent waveform. The data fetch rate from the RAM is controlled through a programmable timer (value in μs). Further, programming the 8 RAM profile registers allows selecting specific DDS signal parameters that act as the destination for RAM samples.

In digital ramp modulation mode, a Digital Ramp Generator (DRG) provides the signal parameters; and the serial I/O port supplied the parameters for ramp generation. The digitally generated ramp has an output resolution of 32-bits that can be programmed to represent the frequency (32-bits), amplitude (14-bits), or phase (16-bits).

In parallel-data port modulation mode, the DDS signal parameters are driven into the 18-bit parallel port. The 16 MSB bits is the data-word (`D[15:0]`) represented in unsigned binary and 2 LSBs is the 2-bit destination word (`F[1:0]`). The 2 lsbs defines how the data-word is applied to the DDS signal parameters.

4.4.1. FUNCTIONING OF DDS-CORE

The DDS-core generates a SIN/COS-reference signal based on the value of the Control Function Register (CFR1). The frequency control, phase offset, and amplitude control takes the parameters - frequency, amplitude and phase - as input. This is shown in Figure 4.14

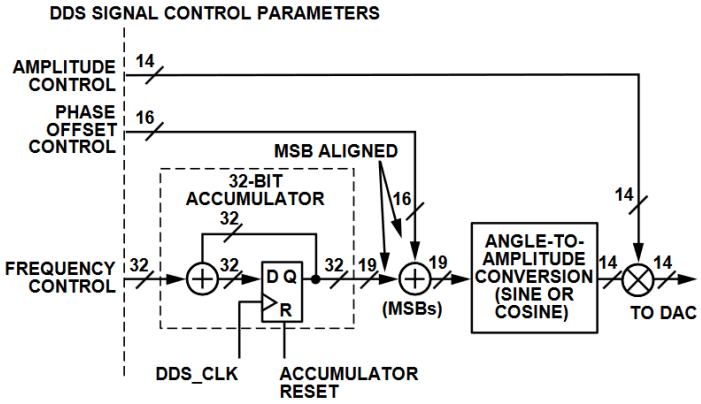


Figure 4.14: AD9910 DDS Core Functional Block Diagram (Credits: AD9910 Data-sheet)

The output frequency is a function of the 32-bit *Frequency Tuning Word (FTW)* and System Clock Frequency f_{SYSCLK}. The relationship is given as follows:

$$f_{OUT} = (\frac{FTW}{2^{32}})$$

For calculating FTW of desired f_{out}, we use the following relation[4]:

$$FTW = round(2^{32}(\frac{f_{OUT}}{f_{SYSCLK}}))$$

The relative phase of DDS is controlled by the 16-bit Phase Offset Word (POW). The DDS-core applies this before the Amplitude-to-Angle conversion step. The relative amplitude can also be scaled by the 14-bit Amplitude Scale Factor (ASF). This is done at the last step. The final output of the DDS core is then sent to the 14-bit DAC.

4.4.2. DDS SERIAL PROGRAMMING USING SLAVE FPGAS

The AD9910 can be programmed using the Serial Peripheral Interface (SPI) protocol. The Serial I/O port contains the following pins:

1. *SCLK* (Serial Clock) - used to synchronize read/write data and operate internal state machines.
2. $\overline{CS}$ (Chip Select) - By asserting an active low-on $\overline{CS}$ pin, we can choose between multiple SPI Slaves. Asserting high on the pin drives SDO and SDIO pins to high impedance.

[4] round to nearest integer

3. *SDIO* (Serial Data Input/Output) - Data is sent to DDS via the SDIO pin. However, it can also be used in bidirectional mode, depending on whether the SPI-protocol is used as a 2-wire interface or 3-wire interface. Bit-1 of the Control Function Register (CFR1) determines the configuration.

4. *SDO* (Serial Data Out) - Data is readout from this port by the SPI Master Device. It is not used when operating in single bidirectional I/O mode and is set to high impedance.

5. *I/O_RESET* - Resets and synchronizes the I/O port state machines. Is also used for generating I/Q signals as described in Section 3.4.4.

6. *I/O_UPDATE* - Updates the internal registers with the data available from the I/O port buffers.

The SPI-driver is used to control the DDS Evaluation Board for arbitrary waveform generation, which is complicated and a complete project, in itself. The bit representations and instruction used for operating the DDS-DAC can be referred from the AD9910 Data-Sheet.

4.4.3. IMPLEMENTATION

We used the Evaluation Board for simple sinusoidal waveform generation. The evaluation board has two different connectors for power supply, named TB1 and TB2. TB1 supplies power to the digital I/O interface, the digital core and USB circuit, whereas TB2 supplies power to the DAC and the clock_input circuit.

The `clk_in` can be provided in one of the 3 ways: either provide a high frequency input signal through the `SMA` port. The second option allows the user to connect using a lower input reference frequency, enabling the clock multiplier, and connects through J1. The third option allows the user to connect to the on-board crystal oscillator.

The detailed instructions for using the AD9910 Evaluation Board can be accessed from the Evaluation Board User Guide, accessible at `https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad9910.html`. The hardware setup is shown in Figure 4.15 and the the obtained experimental waveform are shown in Figure 4.16. The sinusoidal waveform can also obtained when using in the RAM mode. The RAM file values (in decimal) for constructing a sinusoidal waveform and square waveform are given in appendix A.3 [5].

[5]The square wave was prone to many sources of noise during hardware testing. A simple touch to the BNC-connectors leads to distortions in the waveform.

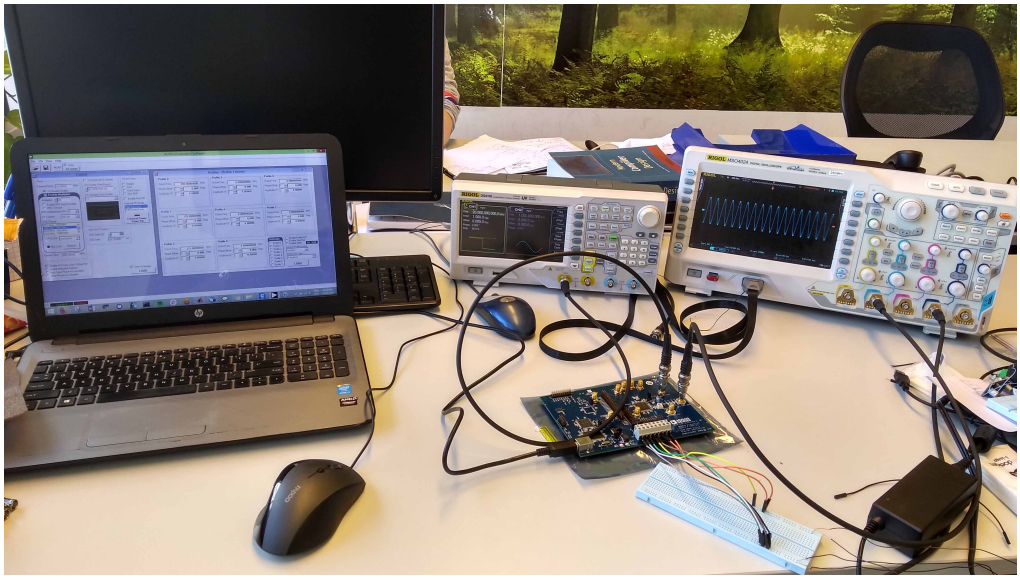
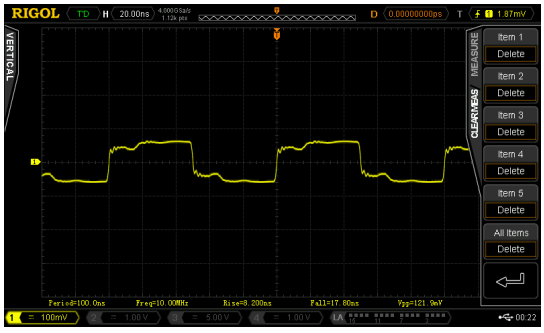
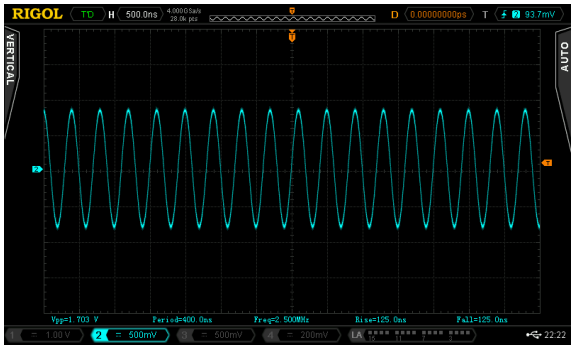


Figure 4.15: AD9910 DDS Evaluation Board Setup



(a) Square wave 10MHz, when operated in RAM mode from AD9910 DDS-DAC



(b) SIN wave 5.0MHz from AD9910 DDS-DAC in single-tone mode.

Figure 4.16: Filtered Waveform Output AD9910 DDS-DAC Evaluation Board

4.5. CONCLUSION

In this chapter, we presented the hardware prototyping for parts of the quantum pipeline and experimented with the presented two waveform generation methods viz. DAC-based and DDS-based. Operating each hardware element requires in-depth understanding of the elements from their data-sheets and requires a larger time-frame. We could not generate all the required waveform and trigger signals to completely integrate the micro-architecture to a real quantum hardware, however we implemented both the major components towards achieving that aim - design of a quantum pipeline communicating to waveform generation units in a distributed architecture and the waveform generation units themselves. The aim of hardware testing is to get familiar with the operating hardware and the necessary hardware/software tools used to analyze the functioning and performance of sub-components of the quantum pipeline.

The waveform generation methodology is essential for both - the Spin-qubit system and the Superconducting-Qubit system. Upon comparing DDS- and DAC-based approaches, DDS-based hardware allows for more flexibility in generating different waveform shapes, has many modes of operation and can be controlled with flexibility using a standard serial protocol. Whereas, FPGA-driven DAC-based AWG are good for Codeword-Triggered Pulse Generation (CTPG) control systems.

They can be utilized for generating standard control-pulses prerecorded in waveform memory and triggered using codewords issued by the Timing Precise Execution Unit. We used the Logic Analyzer to debug data_out from the master controller and Signal Tap II to debug internal signal transitions. More analysis on LVDS data_out signal e.g. getting the eye-diagram, can hint on it's effectiveness in the master-slave communication. We can also switch to multi-gigabit transceiver (MGTs) for faster and better signal quality transmission. All the digital logic was driven by the 50MHz internal FPGA clock. This can be pushed up to 200MHz for operating on the Enclustra SoC-FPGA. The results used in this chapter were generated using RIGOL DG4102 Function Generator, RIGOL MSO4024 Digital Oscilloscope and Logic Analyzer, Quartus Signal Tap II, GTKwave timing simulator and MATLAB R2018B.

5

Conclusion & Future Work

In this chapter, we summarize the micro-architecture developed during the course of the master thesis. We state the research contributions made with this work, present the limitations of the defined architecture and how we can extend the digital design to overcome them by implementing next steps. The master thesis titled "CC-Spin: A Micro-architecture design for scalable control of Spin-Qubit Quantum Processor" is intended to explore the design space for integrating the quantum control hardware to the OpenQL compiler tool-chain.

5.1. Implementation of CC Spin Micro-architecture

Micro-architecture development is an iterative process comprising of the following three parts:

1. **Partitioning the Chip:** Break down the idea into easier-to-understand functionality (Divide-and-Conquer).

2. **Datapath Design:** Identify the components required to design the functionality such as FIFOs, Multiplexers, Adders, Multipliers, ECC, CRC, Encoding/decoding, scrambling/encryption, Internal data-path size (32-/64-/128-bit) etc.

3. **Control Functions:** Design of one/multiple state-machines. Encoded State-Machines vs One-hot state machine. Full Handshake vs half-handshake in data transfer between clock crossing domains etc.

We approached the thesis with the same philosophy. While designing the QISA and Quantum Pipeline, we focus on the design's simplicity and modularity. We need the design to be able to meet all the functionality criteria and should be easy to understand, debug, port and extend (scale). To design the micro-architecture layer, we need to understand the requirements for the lower layers of the Full-Stack. We establish the requirement specifications for the Analog-Digital-Interface - the waveform generation layer - and design a QISA format for representing the quantum operations expressed in Quantum Assembly Language. The QISA directly affects the micro-architecture implementation, therefore an efficient QISA design is important.

We have designed a distributed quantum pipeline executing on a master controller FPGA (comprising of the Micro-coded Instruction Decoder and Router) and communicating via high-speed serial Links to the Slave FPGAs (with Timing Precise Execution Unit which is the waveform generation Trigger control). We implemented and tested parts of the pipeline for correct functionality and presented ways to synchronize the Waveform Generation across the distributed waveform generators via SYNC_CLK mechanism. Special care is taken while writing the HDL code for all the components of the quantum pipeline - the pipeline is implemented by setting standard

parameters for specifying variable elements in the code such as: bus_width, addr_width, memory_depth etc. This allows easy modification of the code for scaling of the same logic to accommodate hardware extensions. Further, all components are described in a *structural logic* which allows easy integration and continuous development on and around the pipeline.

Hardware architectures are prone to errors, such as timing mismatch in parallel lines, Metastability issues, etc. The development cycle of CC-Spin pipeline and waveform generation components faced similar challenges. In order to avoid the user to face the same issues, we have summarized our practical development notes in Appendix B.

The main contributions made by the thesis are as follows:

1. We have proposed two QISA design formats for specifying quantum and classical instruction execution on hardware. The QISA format can address from 32 up to 256 qubits. An extensive pipeline for realizing the QISA and achieve high instruction-issue rate is designed.

2. We proposed a Multi-stage Instruction Decode pipeline that helps make the micro-architecture design portable across multiple qubit-realization technologies. By updating the contents of the Quantum Control Store, necessary technology dependent constraints can be introduced.

3. We implemented FPGA-controlled DAC-based and DDS-based waveform generators to generate simple waveform used for one- and two- qubit operations. The flexibility of custom designed hardware to realize quantum operation can significantly bring down the operation costs. Moreover, a fully-integrated system for sequence generation and analog waveform generation using modular devices tailoring 'shaped' control pulses for qubit control, is the first step towards development of standardized unit-cells in quantum control.

4. A complete control architecture for quantum Control and Read-out of Spin-Qubits is proposed with all the hardware synchronized and achieving timing-precise waveform generation.

5.2. FUTURE WORK

The scope of the master thesis is 'wide' and therefore requires an understanding of quantum operations at all of the layers of the Full-Stack. A simple example of the same is presented in Section 3.2.6 where based on the type of Quantum Algorithm, by adding support for the algorithm, the micro-architecture layer can highly optimize the system for efficient execution. Similarly, an understanding the waveform generation units is required to make future control infrastructures for qubits, such as real-time waveform synthesizers, Parameterized Waveform Generation, TDM/FDM mechanism for signals etc.

Motivated by the challenges faced during the course of the thesis and getting an understanding of the requirements for future quantum control systems, we need to address a number of challenges to CC-Spin design in the future. These are as follows:

1. The Micro-code Decode Unit needs to be implemented that will allow for multi-stage Instruction Decoding and will help support different qubit-technologies.

2. Instruction Issue Rate needs to be increased and implementation of SIMQ (Single-Instruction-Multiple-Qubit) addressing is essential to parallelize the operation on the hardware. A dense encoding of QISA instructions is required along with VLIW or Super-scalar Architectures to implement the Quantum Pipeline.

3. For advanced quantum algorithms (such as, VQE, QLSA, QITE etc.) the Classical Pipeline requires many complex operations often comprising of classical optimization libraries

specified in high-level scientific packages. Therefore, implementation of Classical Pipeline on the Hard Processing System (HPS) is the right way to go.

4. Taking into consideration the near term Quantum Algorithms, local memory definitions are required for storing data and on hardware quantum circuit with tunable parameters.

5. A distributed architecture with ability to address large qubit counts also enables Quantum Error Correction (QEC). Development of complete, parameterized waveform generation units, synchronous with master-controller in a distributed architecture should be the next step in development of efficient hardware for quantum control.

A

HDL Codes

A.1. Binary-to-Gray Code Counter

An 8-bit binary to gray code counter

```
// TOP MODULE for Binary to Gray Counter

module binarytogray #(parameter PTR = 8) //change the value of parameter to
    generate variable size gray code counter. Odd gray code counters are not
    allowed.

  //output ports
  (output [PTR-1:0]   binary_val,
  //input ports
  input [PTR-1:0]   gray_val
  );

  wire  [PTR-1:0]   gray_val;

// Recommended to make 2^n deep gray code counters.
generate
  genvar i;
  for(i = 0; i < (PTR-1); i=i+1)
  begin
    assign gray_val[i] = binary_val[i] ^ binary_val[i+1];
  end
endgenerate

assign gray_val[PTR-1] = binary_val[PTR-1]

endmodule
```

A.2. ASYNCHRONOUS FIFO DESIGN

Top Module : This is the top-level module that encapsulates the complete asynchronous FIFO design and all its input/output signals.

```
// TOP MODULE for ASYNCHRONOUS FIFO with TWO CLOCK DOMAINS


`include "./fifomem.v"
`include "./sync_r2w.v"
`include "./sync_w2r.v"
`include "./rptr_empty.v"
`include "./wptr_full.v"

module fifo1 #( parameter      DSIZE = 8,
    parameter      ASIZE = 4)

  //output ports
  (output [DSIZE-1:0]   rdata,    //Data Output
  output       wfull,    //Full Flag
  output       rempty,   //Empty Flag
  //input ports
  input [DSIZE-1:0]   wdata,    //Data Input
  input        winc,    //Write Enable
  input      wclk,   //Write Clock
  input        wrst_n,   //Write Reset
  input        rinc,    //Read Enable
  input      rclk,   //Read Clock
  input      rrst_n    //Reset
  );

  //internal connections
  wire  [ASIZE-1:0]   waddr, raddr;
  wire  [ASIZE:0] wptr, rptr, wq2_rptr, rq2_wptr;

// component 1
// Module for synchronizing the read point from read to write domain

sync_r2w sync_r2w (
.wq2_rptr(wq2_rptr),
.rptr(rptr),
.wclk(wclk),
.wrst_n(wrst_n)
);

// component 2
// Module for synchronizing the write point from write to read domain

sync_w2r sync_w2r (
.rq2_wptr(rq2_wptr),
.wptr(wptr),
.rclk(rclk),
.rrst_n(rrst_n)
);

// component 3
// Memory Module

fifomem #(DSIZE, ASIZE) fifomem(
.rdata(rdata),
.wdata(wdata),
.waddr(waddr),
```

```
.raddr(raddr),
.wclken(winc),
.wfull(wfull),
.wclk(wclk)
);

// component 4
// Module for handling the read requests

rptr_empty #(ASIZE) rptr_empty(
.rempty(rempty),
.raddr(raddr),
.rptr(rptr),
.rq2_wptr(rq2_wptr),
.rinc(rinc),
.rclk(rclk),
.rrst_n(rrst_n)
);

// component 5
// Module for handling the write request

wptr_full #(ASIZE) wptr_full(
.wfull(wfull),
.waddr(waddr),
.wptr(wptr),
.wq2_rptr(wq2_rptr),
.winc(winc),
.wclk(wclk),
.wrst_n(wrst_n)
);



endmodule
```

FIFO Memory : This is a dual-port RAM where all the read from and write to operations take place.

```
module fifomem #(parameter DATASIZE = 8, // Memory data word width
parameter ADDRSIZE = 4) // Number of mem address bits

(output [DATASIZE-1:0] rdata,
input [DATASIZE-1:0] wdata,
input [ADDRSIZE-1:0] waddr, raddr,
input wclken, wfull, wclk);

`ifdef VENDORRAM
// instantiation of a vendor's dual-port RAM
vendor_ram mem (.dout(rdata), .din(wdata), .waddr(waddr), .raddr(raddr), .
    wclken(wclken), .wclken_n(wfull), .clk(wclk));
`else
// RTL Verilog memory model
localparam DEPTH = 1<<ADDRSIZE;
reg [DATASIZE-1:0] mem [0:DEPTH-1];

assign rdata = mem[raddr];

always @(posedge wclk)
if (wclken && !wfull) mem[waddr] <= wdata;
`endif

```

```
endmodule
```

Read-domain to Write-domain Synchronizer : This synchronizer module is used to pass the n-bit read pointer from the read clock domain to the write clock domain, through two registers clocked by `wclk` (the output is synchronized to the write clock). This synchronized n-bit read pointer will be used by `wptr_full` module to evaluate the FIFO Full condition.

```
module sync_r2w #(parameter ADDRSIZE = 4)

(output reg [ADDRSIZE:0] wq2_rptr,
input [ADDRSIZE:0] rptr,
input wclk, wrst_n);

reg [ADDRSIZE:0] wq1_rptr;

always @(posedge wclk or negedge wrst_n)
if (!wrst_n)  {wq2_rptr,wq1_rptr} <= 0;
else    {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};

endmodule
```

Write-domain to Read-domain Synchronizer : This synchronizer module is used to pass the n-bit write pointer from the write clock domain to the read-clock domain, through two registers clocked by `rclk` (the output is synchronized to the read clock). This synchronized n-bit write pointer will be used by `rptr_empty` module to evaluate the FIFO Empty condition.

```
module sync_w2r #(parameter ADDRSIZE = 4)
(output reg [ADDRSIZE:0] rq2_wptr,
input [ADDRSIZE:0] wptr,
input rclk, rrst_n);

reg [ADDRSIZE:0] rq1_wptr;

always @(posedge rclk or negedge rrst_n)
if (!rrst_n)  {rq2_wptr,rq1_wptr} <= 0;
else    {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};
endmodule
```

Read Pointer and Empty Generation Logic This module contain the FIFO logic associated to the read clock domain i.e. it is synchronous to `rclk`. The read pointer (`rptr`) is a dual n-bit Gray Code counter and (n-1)-bit pointer `raddr` is used as address. This module generates the FIFO Empty condition when the next `rptr` is equal to the synchronized `wptr`.

```
module rptr_empty #(parameter ADDRSIZE = 4)
(output reg rempty,
output [ADDRSIZE-1:0] raddr,
output reg [ADDRSIZE :0] rptr,
input [ADDRSIZE :0] rq2_wptr,
input rinc, rclk, rrst_n);

reg [ADDRSIZE:0] rbin;
wire [ADDRSIZE:0] rgraynext, rbinnext;


// GRAY code pointer

always @(posedge rclk or negedge rrst_n)
```

```
if (!rrst_n)  {rbin, rptr} <= 0;
else    {rbin, rptr} <= {rbinnext, rgraynext};
// Memory read-address pointer

assign raddr = rbin[ADDRSIZE-1:0];
assign rbinnext = rbin + (rinc & ~rempty);
assign rgraynext = (rbinnext>>1) ^ rbinnext;

// Empty condition
assign rempty_val = (rgraynext == rq2_wptr);

always @(posedge rclk or negedge rrst_n)
if (!rrst_n)  rempty <= 1'b1;
else    rempty <= rempty_val;
endmodule
```

Write Pointer and Full Generation Logic This module contain the FIFO logic associated to the write clock domain i.e. it is synchronous to `wclk`. The wite pointer (`wptr`) is also a dual n-bit Gray Code counter and (n-1)-bit pointer `waddr` is used as address. This module generates the FIFO Full condition.

```
module wptr_full #(parameter ADDRSIZE = 4)
(output reg wfull,
output [ADDRSIZE-1:0] waddr,
output reg [ADDRSIZE :0] wptr,
input [ADDRSIZE :0] wq2_rptr,
input winc, wclk, wrst_n);
reg [ADDRSIZE:0] wbin;
wire [ADDRSIZE:0] wgraynext, wbinnext;

// GRAY code pointer
always @(posedge wclk or negedge wrst_n)
if (!wrst_n)  {wbin, wptr} <= 0;
else    {wbin, wptr} <= {wbinnext, wgraynext};
// Memory write-address pointer

assign waddr = wbin[ADDRSIZE-1:0];
assign wbinnext = wbin + (winc & ~wfull);
assign wgraynext = (wbinnext>>1) ^ wbinnext;

//Full condition
assign wfull_val = (wgraynext=={~wq2_rptr[ADDRSIZE:ADDRSIZE-1], wq2_rptr[
    ADDRSIZE-2:0]});

always @(posedge wclk or negedge wrst_n)
if (!wrst_n)  wfull <= 1'b0;
else    wfull <= wfull_val;

endmodule
```

Testbench: Asynchronous FIFO Design The following testbench can be used with top module to perform functional simulation of asynchronous FIFO. See figure **??**.

```
// Testbench for Asynchronous FIFO
// Change everything below

`timescale 1 ns / 100 ps

// Define Module for Test Fixture
```

```
module fifo1_tb();

// Inputs
    reg [7:0] wdata;
    reg winc;
    reg wclk;
    reg wrst_n;
    reg rinc;
    reg rclk;
    reg rrst_n;


// Outputs
    wire [7:0] rdata;
    wire rempty;
    wire wfull;


// variable for test vectors
  reg [7:0] test_vec;
// Bidirs


// Instantiate the Unit Under Test (UUT)
// Please check and add your parameters manually
    fifo1 UUT (
      .rdata(rdata),
      .rempty(rempty),
      .wfull(wfull),
  .wdata(wdata),
      .winc(winc),
      .wclk(wclk),
      .wrst_n(wrst_n),
      .rinc(rinc),
      .rclk(rclk),
      .rrst_n(rrst_n)
        );


// Initialize Inputs
// You can add your stimulus here
    initial begin
      $dumpfile("fifo1_tb.vcd");
      $dumpvars(0,fifo1_tb);
      $display ($time, " << Starting the Simulation >> ");

      wclk = 1'b 0;
      rclk = 1'b 0;
                wdata = 24'b 0;
                winc = 1'b 0;
                rinc = 1'b 0;
      wrst_n = 1'b 0;
      rrst_n = 1'b 0;

    end



  always
      #10 rclk = ~rclk;    // every ten nanoseconds invert the clock

```

```
68  always
69      #12 wclk = ~wclk;     // every ten nanoseconds invert the clock
70
71
72
73  initial
74    begin
75
76      #30
77      wrst_n = 1'b 1;
78      rrst_n = 1'b 1;
79
80      // write to fifo
81      for(test_vec=0; test_vec < 17; test_vec = test_vec + 1)
82        begin
83          #20
84          winc = 1'b 1;
85          wdata = test_vec;
86          #20
87          winc = 1'b 0;
88        end
89
90      // read from fifo
91      for(test_vec=0; test_vec < 17; test_vec = test_vec + 1)
92        begin
93          #20
94          rinc = 1'b 1;
95          #20
96          rinc = 1'b 0;
97        end
98
99      // write to fifo
100     for(test_vec=0; test_vec < 15; test_vec = test_vec + 1)
101       begin
102         #20
103         winc = 1'b 1;
104         wdata = test_vec;
105         #20
106         winc = 1'b 0;
107       end
108
109     // read from fifo
110     for(test_vec=0; test_vec < 11; test_vec = test_vec + 1)
111       begin
112         #20
113         rinc = 1'b 1;
114         #20
115         rinc = 1'b 0;
116       end
117
118     // read and write to fifo
119     for(test_vec=0; test_vec < 11; test_vec = test_vec + 1)
120       begin
121         #20
122         rinc = 1'b 1;
123         winc = 1'b 1;
124         wdata = test_vec;
125         #20
126         rinc = 1'b 0;
127         winc = 1'b 0;
128       end
```

```
129
130       // read from fifo
131       for(test_vec=0; test_vec < 7; test_vec = test_vec + 1)
132         begin
133           #20
134           rinc = 1'b 1;
135           #20
136           rinc = 1'b 0;
137         end
138
139       // write to fifo
140       for(test_vec=0; test_vec < 13; test_vec = test_vec + 1)
141         begin
142           #20
143           winc = 1'b 1;
144           wdata = test_vec;
145           #20
146           winc = 1'b 0;
147         end
148
149       #10
150       wrst_n = 1'b 1;
151       rrst_n = 1'b 1;
152
153     $finish;
154
155     end
156
157
158
159
160
161 endmodule // FIFO_v_tf
```

A.3. QUANTUM PIPELINE

The code for **quantum pipeline** and the **timing control unit** are very large, and is therefore accessible at my GitHub account:

https://github.com/amitabhyadav/q-pipeline-experimental.

Quantum Program Execution Flow is shown in figure A.1.

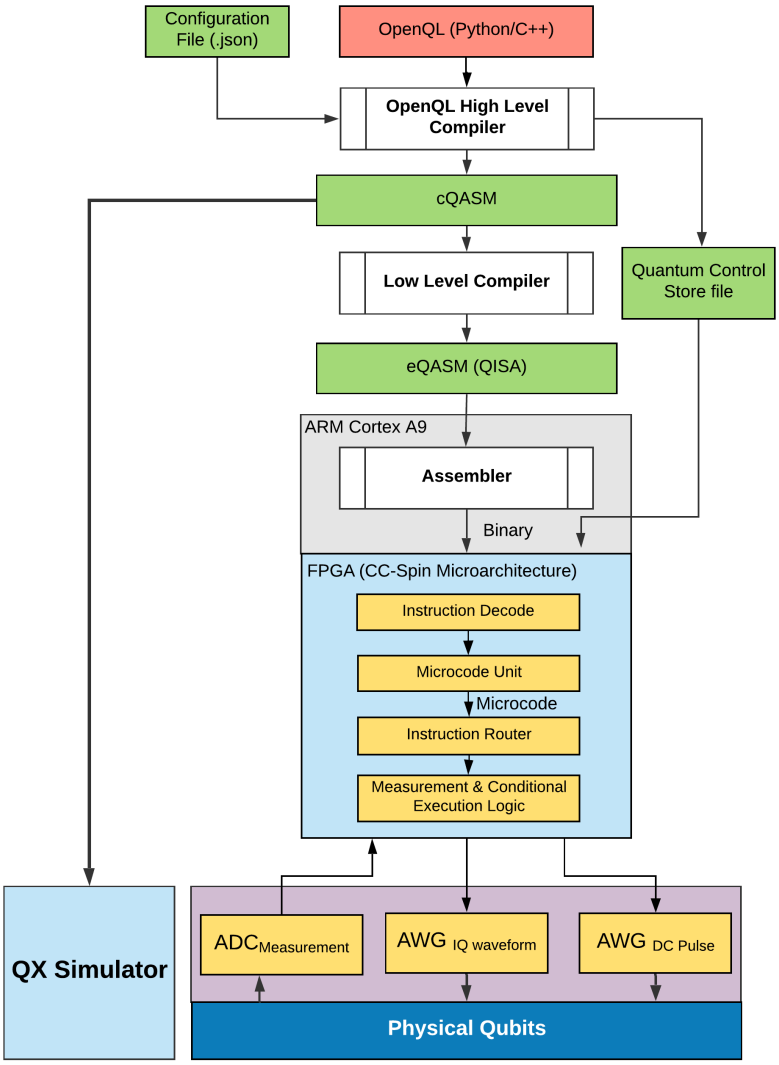


Figure A.1: Quantum Program Execution Flow

The different hardware used in the implementation of quantum pipeline and waveform generation unit are given below:

- Enclustra Mercury+ SA2 Cyclone V SE SoC-FPGA
- Enclustra Mercury PE1 Baseboard
- AD9517 Clock Distribution Board
- AD9910 DDS Evaluation Board
- SI5338-EVB-Quad Clock Generator
- Terasic DE10 Standard SoC-FPGA
- Terasic ADA Daughter Board
- Terasic HSMC-Pinout board for LVDS communication

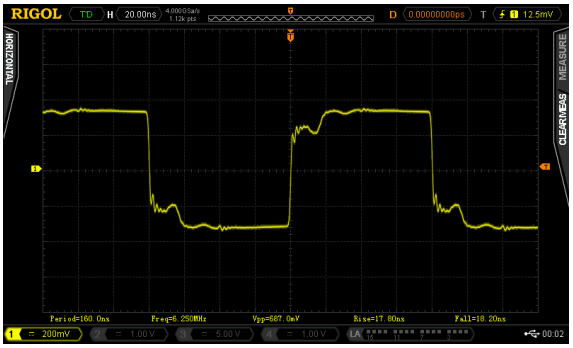
Sinusoidal Waveform: RAM file values (in decimal) for AD9910 Enter each value in a new line (.txt file) and remove the commas. Note that these are amplitude values.

1187, 2348, 3459, 4495, 5433, 6254, 6939, 7475, 7848, 8052, 8081, 7935, 7618, 7135, 6498, 5721, 4819, 3814, 2725, 1578, 397, 794, 1966, 3097, 4160, 5133, 5996, 6728, 7315, 7743, 8004, 8091, 8004, 7743, 7315, 6728, 5996, 5133, 4160, 3097, 1966, 794, 397, 1578, 2725, 3814, 4819, 5721, 6498, 7135, 7618, 7935, 8081, 8052, 7848, 7475, 6939, 6254, 5433, 4495, 3459, 2348, 1187, 0, 1188, 2349, 3460, 4496, 5434, 6255, 6940, 7476, 7849, 8053, 8082, 7936, 7619, 7136, 6499, 5722, 4820, 3815, 2726, 1579, 398, 793, 1965, 3096, 4159, 5132, 5995, 6727, 7314, 7742, 8003, 8091, 8003, 7742, 7314, 6727, 5995, 5132, 4159, 3096, 1965, 793, 398, 1579, 2726, 3815, 4820, 5722, 6499, 7136, 7619, 7936, 8082, 8053, 7849, 7476, 6940, 6255, 5434, 4496, 3460, 2349, 1188, 1

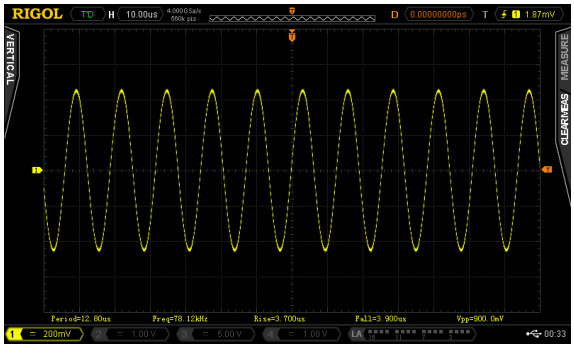
Square Waveform: RAM file values (in decimal) for AD9910 Enter each value in a new line (.txt file) and remove the commas. Note that these are amplitude values.

3435973837, 3403761583, 3371549328, 3339337073, 3307124818, 3274912564, 3242700309, 3210488054, 3178275800, 3146063545, 3113851290, 3081639035, 3049426781, 3017214526, 2985002271, 2952790016, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Output Waveform from RAM mode .

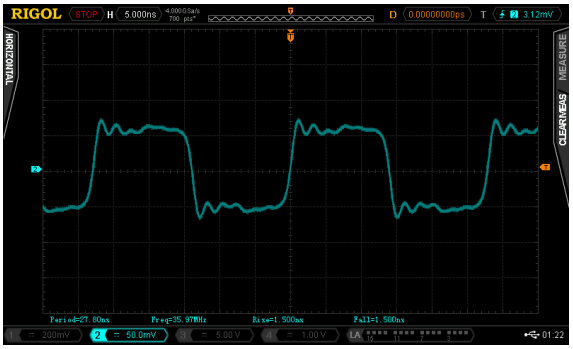


(a) Square wave 6.25MHz, when operated in RAM mode from AD9910 DDS-DAC

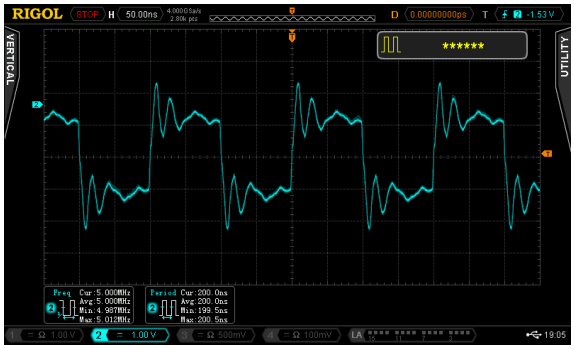


(b) Sin wave 80KHz, when operated in RAM mode from AD9910 DDS-DAC

Figure A.2: Filtered Waveform Output AD9910 DDS-DAC Evaluation Board when operated in RAM mode.



(a) Square wave 35MHz, when operated in RAM mode from AD9910 DDS-DAC



(b) Square Wave 5.0MHz, when operated in RAM mode from AD9910 DDS-DAC

Figure A.3: Filtered Waveform Output AD9910 DDS-DAC Evaluation Board when operated in RAM mode-II

A.4. TRUE DAC BASED WAVEFORM GENERATION DEMONSTRATION

The code uses a 14-bit input Numerically Controlled Oscillator (NCO) to generate a sin waveform.

```
module DE10_Standard_ADDA_TOP    (
                        CLOCK_50,
                        ADC_CLK_A,
                ADC_CLK_B,
                ADC_DA,
                ADC_DB,
                ADC_OEB_A,
                ADC_OEB_B,
                ADC_OTR_A,
                ADC_OTR_B,
                DAC_CLK_A,
                DAC_CLK_B,
                DAC_DA,
                DAC_DB,
                DAC_MODE,
                DAC_WRT_A,
                DAC_WRT_B,
                OSC_SMA_ADC4,
                POWER_ON,
                 SMA_DAC4
                        );


input                   CLOCK_50;

output                  ADC_CLK_A;
output                  ADC_CLK_B;
input        [13:0]     ADC_DA;
input        [13:0]     ADC_DB;
output                  ADC_OEB_A;
output                  ADC_OEB_B;
input                   ADC_OTR_A;
input                   ADC_OTR_B;
output                  DAC_CLK_A;
output                  DAC_CLK_B;
output      [13:0]    DAC_DA;
output      [13:0]    DAC_DB;
output                    DAC_MODE;
output                    DAC_WRT_A;
output                    DAC_WRT_B;
output                    POWER_ON;
output                      OSC_SMA_ADC4;
output                      SMA_DAC4;

//  REG/WIRE declarations

assign   DAC_WRT_B = ~CLK_125;       //Input write signal for PORT B
assign   DAC_WRT_A = ~CLK_125;       //Input write signal for PORT A

assign   DAC_MODE = 1;             //Mode Select. 1 = dual port, 0 = interleaved.

assign   DAC_CLK_B = ~CLK_125;         //PLL Clock to DAC_B
assign   DAC_CLK_A = ~CLK_125;         //PLL Clock to DAC_A

assign   ADC_CLK_B = ~CLK_65;          //PLL Clock to ADC_B
assign   ADC_CLK_A = ~CLK_65;          //PLL Clock to ADC_A

assign   ADC_OEB_A = 0;              //ADC_OEA
```

```
59 assign   ADC_OEB_B = 0;          //ADC_OEB
60
61
62 wire     [13:0]  sin10_out;
63 wire     [13:0]  sin_out;
64 wire     [13:0] cos_out;
65 wire     [13:0] cos10_out;
66
67 wire     [13:0]  comb1;
68 wire     [13:0] comb2;
69
70 wire     [31:0]  phasinc1;
71 wire     [31:0]  phasinc2;
72
73 wire     g = 0;
74 wire     v = 1;
75
76 assign   phasinc1 = {g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g
       ,v,v};
77 assign   phasinc2 = {g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g,v,v,g,g
       ,v,v};
78
79 assign   DAC_DA = comb1;//comb; //B
80 assign   DAC_DB = comb2;//comb; //A
81
82 assign   POWER_ON  = 1;            //Disable OSC_SMA
83
84 reg     [13:0]   r_ADC_DA/*synthesis noprune*/;
85 reg     [13:0]   r_ADC_DB/*synthesis noprune*/;
86
87 always @ (posedge CLK_65) r_ADC_DA <= ADC_DA;
88 always @ (posedge CLK_65) r_ADC_DB <= ADC_DB;
89
90 //  Structural coding
91
92 nco sin1          (
93             .phi_inc_i(phasinc1),
94             .clk      (CLK_125),
95             .reset_n  (v),
96             .clken    (v),
97             .fsin_o   (sin_out),
98             .fcos_o   (cos_out),
99             .out_valid(ovalid)
100             );
101
102 nco sin2          (
103               .phi_inc_i(phasinc2),
104               .clk      (CLK_125),
105               .reset_n  (v),
106              .clken   (v),
107             .fsin_o   (sin10_out),
108             .fcos_o   (cos10_out),
109             .out_valid(ovalid)
110               );
111
112
113 pll  pll_100   (
114              .refclk(CLOCK_50),
115             .rst(g),
116             .outclk_0   (CLK_125),
117             .outclk_1   (CLK_65)
```

```
118             );
119
120 lpm_add     lpm1  (
121                   .clock (CLK_125),
122                   .dataa ({g,~sin_out[12],sin_out[11:0]}),
123                   .datab ({g,~sin10_out[12],sin10_out[11:0]}),
124                   .result(comb1)
125                  );
126
127 lpm_add    lpm2 (
128               .clock (CLK_125),
129                   .dataa ({g,~cos_out[12],cos_out[11:0]}),
130                   .datab ({g,~cos10_out[12],cos10_out[11:0]}),
131                   .result(comb2)
132             );
133
134 endmodule
```

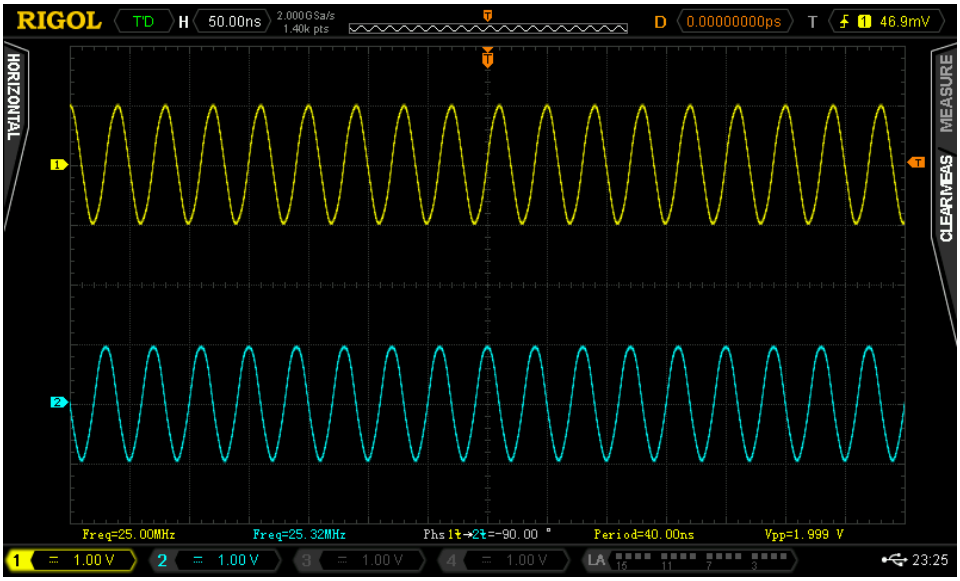


Figure A.4: IQ Sin wave 25MHz, V_{pp} = 2V from Terasic Cyclone V with DAC board

B

APPENDIX B: SET UP AND USE INSTRUCTIONS OF SOC-FPGA

B.1. ALTERA QUARTUS PRIME

Intel$^{\text{ő}}$*Quartus*$^{\text{ő}}$ Prime Standard Edition Design Software is a fully-integrated development tool featuring, Multiple design entry methods, Logic Synthesis, Place and Route, and Device Programming. Simulation tools include, Standard HDL simulation tools, and ModelSim - Intel FPGA Starter Edition tool.

The Design flow on an FPGA or SoC begins from an HDL Design in VHDL or Verilog. This is followed by compilation, simulation, programming and hardware verification.

Simulations There are two types of simulations: RTL (or, Functional) Simulation and Timing Simulation. RTL simulation is for functional verification of the HDL Design. Whereas, the Timing Simulation verifies that the design meets timing and functions appropriately on the Device. Timing Simulation is also called the post-Place & Route Simulation. When programming the hardware, Timing analysis is the process of evaluating the timing of logic in the device after it has been synthesized, placed and routed to ensure the all timing requirements are met. The goal is to achieve "timing closure" where all the timing constraints are met.

Some interesting things that can be done using Quartus:

- Simulation using the Quartus II Software
- Timing Analysis using the Quartus II Software
- Constraining and Analyzing Timing for Source Synchronous Circuits with TimeQuest
- Validating Performance with the TimeQuest Static Timing Analyzer

Programming the SoC/FPGA When we are programming the FPGA using an Altera Blaster, it is important to remove the SD card. When using the Altera USB Blaster, we use the Auto-Detect feature. This shows up two devices in the JTAG chain.
Why are there two devices found in the JTAG chain?
The Cyclone V SoC device has two JTAG chains, one dedicated to the FPGA and one dedicated to the hard processor system (HPS). On the DE10-Nano board, these JTAG chains are connected in serial so you only need one JTAG connection to communicate with both.

The FPGA JTAG chain is used to configure the FPGA logic, and for hardware debugging using one of several tools such as: 1. SignalTap II logic analyzer 2. System Console system-level debugger The

HPS JTAG chain is primarily used for software development using tools like the ARM Development Studio 5 (DS-5)[1].

We already have an FPGA programmed but in order to understand the full potential of SoC device. We need to get acquainted with the Hard-processing System (HPS). The HPS can be a single or dual-core processor and can boot from a memory (such as MicroSD card, or others.) The microSD card contains the embedded software needed to boot and run the board.

Knowing the Quartus Files . The following files are created when you start a new Quartus project:

- Quartus II Settings File (.qsf)
- Quartus II Project File (.qpf)

RTL Design Related Files:

- RTL specification (.v)(.vhd)
- Block Design File (.bdf) (Make this your top level entity if you're using block diagram schematic)
- Symbol File (.sym): when you work on block diagram schematic. In order to include a HDL code in it, we create it's symbol.

Simulation Related Files:

- Synopsis Design Constraints (.sdc) timing constraints file used during Place & Route. Without this, you would get warnings in your design.

Programming Files:

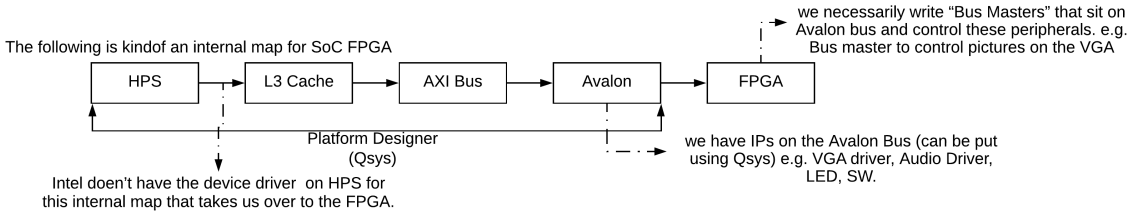
- SRAM Object Files (.sof files) are binary files containing data for configuring SRAM-based devices. The FPGA (in Cyclone V) is SRAM based.

B.2. SOC DESIGN GUIDELINES

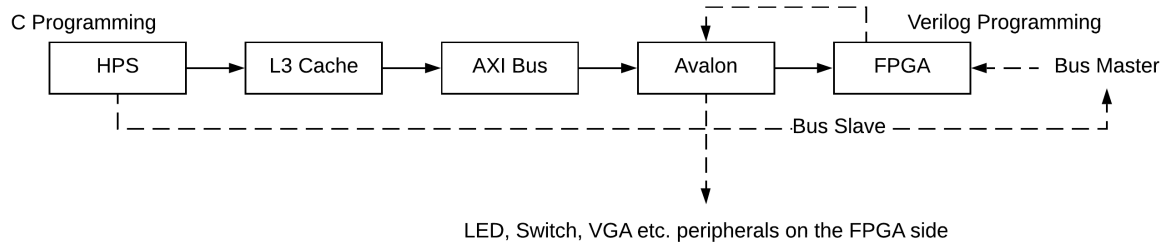
```
NOTE: This section contains my notes while I was learning how to use an SoC
FPGA to develop high speed digital design.  I find them very useful for
anyone to get started on SoC FPGA design.
The content is curated from numerous online and offline sources.  Major
sources being:  Bruce Land's lectures, Rocketboards.org, Enclustra
Documentations, Cyclone V documentation, EPFL documentation on DE-1 SoC,
Altera DE-10 SoC documentation etc.
```

Do all the reading for Linux. (Embedded Linux)
There must be a component of C running on ARM Cortex A9 and some FPGA fabric component.
Collaboratively write C, running on ARM; and write Verilog running on the FPGA and hooked together through some unbelievably complicated BUS systems which is hidden behind.
On the HPS, we run a Linux Distribution which has a native GCC compiler. We compile locally on the HPS.
Normally, write C program on notepad on the PC ⟶ SSH the program to the HPS through FTP and compile there.

[1] is an end-to-end suite of tools for embedded C/C++ software development on any Intel® SoC FPGAs. It has powerful FPGA-adaptive debugging capabilities, providing visibility and control of your SoC FPGA.



So, as a part of C program we do the direct memory mapping of the I/O units; and we run as root. We will be God for these hardware devices.
"We can map virtual memory to real memory and get real memory addresses; and manipulate them."
We can take these hardware at home and put it on our own local area network, it will support DHCP and we can do whatever we want. it will recognize the local router and connect with it.
HPS and FPGA have certain hardware components connected to them. Such as: Ethernet, USB, MicroSD etc are connected to the HPS; SW, Led, GPIOs etc. are connected to the FPGA. (This depends on the board.)
We write BUS MASTERS on the FPGA to control peripherals connected to the FPGA.
We essentially have to write BUS Slaves for the FPGA, so that HPS can communicate through the internal map (the stack, above) with the hardware; which would then communicate with the FPGA bus master hardware to control the peripherals on the FPGA.



e.g. Making an ODE solver on hardware. We get the parameters from the serial terminal on the HPS. Then we put the output (as waveform) on the VGA screen (through Bus master).
Qsys (platform designer) is a drag and drop interface that generates a ton on verilog and deposits it at the right place so that the gcc compiler can find it.

B.2.1. In Qsys

We have the Phase Locked Loops (PLLs) that sets up the Clock for the system. (A ref_clk is taken as input - is exported – and PLL generates the system clock). Then we have the ARM HPS system, that shows the BUS structure. If we double click it, we can add more I/O ports, like a 32-bit hardcoded IO port that looks like a GPIO port from the ARM side and looks like uncommitted 32-bits from the FPGA side. We can configure this in the HPS settings to expose that on the BUS.
We can write our own Bus master and connect it to the peripherals (i.e. those under Intel IP University program) in the Qsys platform.
note: We can write our own direct digital synthesis unit to generate say, sine wave. It's not hard.
In the "Export" column of the Qsys, we have the signals that have to appear (as physical connectios) in the Verilog module that Qsys generates.
Read more on ModelSim simulator.

A part of Quartus Prime is a module called Signal Tap; Signal Tap is a logic analyser that we build on the FPGA with our design, that uses on-chip memory to log data at Bus rate which we can read-out across the USB connection to application running on the PC (a GUI application to a hardware debugger). It is Verilog (so it takes space), and it uses on-chip memory (so it takes memory); so we need to be specific and cautious about what we would like to test this way; but we can expose signals that are not simulation signals but are real-time real hardware signals.

B.2.2. THE HARDWARE

The SoC board has a microSD port from where the HPS can load Linux. The boot-flow of Linux can be found on rocketboards.org

Through the miniUSB port we can access the linux environment using Putty (or any other serial terminal software); and once loaded we login as: root. Often there is no password. (For Enclustra board, the password was: root) Never connect a board to a network without resetting this password if you are using root.

Has inbuilt text editor. Such as vi, vim, emacs. *can be used to change the config files, for example to change the MAC address.

Secure Shell Login: Download the SSH server, using apt-get:

`sudo apt-get install openssh-server` ⟶T his helps use to do SSH into the linux box and get FTP access into the box.

Use passwd command to set up the password on the root.

Install Putty and PSFTP on the Windows machine.

Use Putty to SSH to ARM. – Open the IP address assigned by DHCP. (Can be found by using ifconfig command, on the onboard linux.).... Or we can use the static address assigned by ourselves, by configuring system files.

We use PSFTP to move files to/from the ARM.

Now we can open as many Putty/SSH terminals as we want.

GCC runs natively on the SoC board on Linux running on the HPS. We can add more utility softwares, mount USB drive etc.

pThreads on the HPS. ARM processor running on the linux is dual core and can do load balancing on the processes running on the HPS. Can utilize that.

B.2.3. PROGRAMMING ABSTRACTION ON HOW WE CAN GET TO THE FPGA

/dev/fpga ⟵ This device can be used for programming the FPGA via Linux; but it doesn't give the IO access. We would need to have (write by yourself) a serial device driver to do that.

We can use the /dev/mem and the linux utility called mmap which allows us to take physical addresses and drop them into virtual memory so that we can manipulate them as if they were part of our segment. If we don't do that, the system seg faults as soon as we touch the physical memory. If we mess up in any way with mmap, like, if we don't give it big-enough segment, we give the wrong start address, it seg faults. E.g. if we change the resolution of the VGA from say 320 x 240 (the default) to say, 640 x 480, it is seg fault.

⟶ Using on-board ADCs we can make simple oscilloscopes, AM receivers (500KHz – 1.25MHz) etc.

Caution: NEVER have the source code on the SD Card. Keep it on the PC, back it up. SD cards have chances of fail.

B.2.4. CYCLONE V AND MEMORY MAPPING

⟶ Using on-board ADCs we can make simple oscilloscopes, AM receivers (500KHz – 1.25MHz) etc. USB to UART is used to boot the linux operating system.

There are ports for communication with HPS/FPGA:

1. There is the control block that communicates via the FPGA manager. This is used to program the FPGA via the ARM processor.

2. There is FPGA-to-HPS 64-bit AXI bus. (FPGA is the master, in this case)

3. There is HPS-to-FPGA 64-bit AXI bus. (HPS is the master in this case.)

4. There is Lightweight HPS-to-FPGA 32-bit AXI bus (HPS is master, used for faster communication e.g. led light switching)

5. There is Master 1 – 6 connected to the SDRAM controller, connected to the HPS.

Most of the communication between the FPGA and the ARM goes through the L3 Main switch. Both cores of HPS are cache coherent.

Look at the examples to understand the QSYS. Know about the address mapping through the address header file. The address headers are defined in QSYS and they have to match the address headers in C.

HPS Program is the C-program:

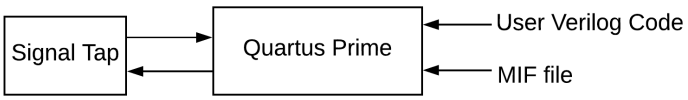
Most of the time, we map the real physical addresses to virtual addresses. Eg. The file sys/mman.h is used for that.

Ipc.h is the header file for inter-process communication.

mmap can be used to do the mapping from physical addresses to virtual address.

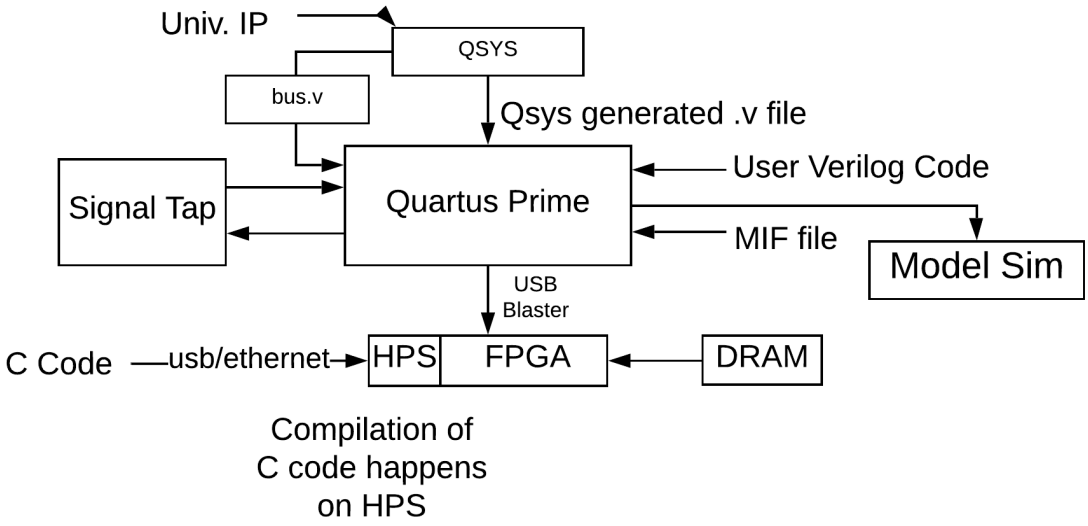
Figure out address of the peripherals and use those addresses to control operations.

B.2.5. Using Quartus and QSYS



Memory Image files (.mif files) can be used to convert text files into ROMs on the FPGA directly. Quartus Prime can help do that. We can use it to preload code on to the FPGA if we are using the ROM table if we're building a DDS (direct digital synthesis) unit.

How does it all connect?



Tutorial on using Model Sim: find on altera website

Tutorial on using Signal Tap: find on altera website

Signal Tap is very useful to view real-time signals from the hardware on the PC through Quartus Prime.

How to do SoC/FPGA: Pick up the hardware and start testing stuff. Simulation won't help in anything and waste your time. Just start writing code for the hardware.

Learning C for the SoC/FPGA is hard.

Note that the bus addresses generated in the QSYS must match the offset addresses in the C address file.

B.2.6. VERILOG — A QUICK RECAP

- No init statement.... In reset state, we reset the register.

- No system calls — no print statement

- No floats/no real values

- No delay/ no timing info – timing information is intrinsic to Hardware

- Loops are not sequential — it's copy of hardware. So, don't use loops.

- Module initiates a hardware. – it's a piece of reusable hardware

- Every 'assign' and 'always' happens all the time. — it's a combinational piece of h/w

- The only way to sequentialse an always block is to have a pos-edge/neg-edge assigned with it.

- Language is parallel

- There is no tri-states on cyclone-V except I/O i.e. we will very very very rarely use Z.

- X is don' t care.

- Incomplete 'case' or 'if-else' can infer a latch.

- Each wire must be assigned a value.

- Weakly typed.

- All sequentiality is asserted by the state machine that we write.

- Inferred latch: search warnings would show: inferred. Search that in warnings. Also search implicit (signals that we forgot to declare.) always@(*) begin

case(sel)

2'b11: d=a;

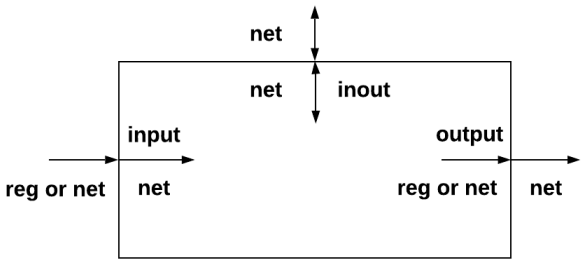
2'b10: d=b;

2'b00: d=c;
default: d=a; // if this is not defined here, it would place a latch here.
end case;
end; //always

- Blocking vs Non-Blocking assignments:
- Assign signal = 52; //blocking statement //here does not matter because assign statement happens all the time
combinatorial logic uses – blocking assignment i.e. =
sequential logic should use non-blocking assignment i.e. <=

- Never mix sequential and combinational blocks
- NOTE: if we assign a signal in 1 always block, we should not assign it anywhere else in the design because that will be like shorting two different voltage lines (which is a bad idea).
- E.g. writing a multiplexer:
1. Use if-else
2. Use case statement
3. Use combinatorial assign statements : assign out = (sel)? D1 : d0;

Wire and Reg .
- `wire` elements must be continuously driven by something. Therefore, they are assigned values using continuous assignment statements. They cannot store a value.
- `reg` can be used to create registers in procedural blocks. They can store value.
- `reg` elements can be used as output within a module declaration. But,`reg` elements cannot be connected to the output port of a module instantiation. (See figure **??**.)



- `wire` can only infer to combinational logic, while `reg` can infer to either combinational or sequential logic.
- A good place to get started with Verilog is: https://hdlbits.01xz.net/wiki/Step_one

FAQs related to SoC/FPGA design .
Ques. Suppose we have a blinky verilog design as shown below. How to determine the blink rate?

```
    // create module
module blink (
input wire clk, // 50MHz input clock
output wire LED // LED ouput
);
// create a binary counter
reg [31:0] cnt; // 32-bit counter
initial begin
```

```
cnt <= 32'h00000000; // start at zero
end
always @(posedge clk) begin
cnt <= cnt + 1; // count up
end
//assign LED to 25th bit of the counter to blink the LED at a few Hz
assign LED = cnt[24];
endmodule
```

Ans. Blinky Design, Clock and Counter Math
cnt[n] where n = the counter bit
2n; we've chosen n = 24 in our Verilog code sample
224 = 16,777,216
If the clock is 50 MHZ or 50,000,000
Clock / 2n = blinks per second
50,000,000 / 16,777,216 = 2.9802. That's about 3 blinks per second.

Ques. What is in the MicroSD card?

Ans. Below are the contents of the SD card (factory default) in the TerasicDE-10 Nano Board. Other SD card images contain other contents and could be partioned in other ways.
Partition 1:
- FPGA configuration bitstreams
- Linux* kernel and device trees for the board
- extlinux.conf used for boot configuration by U-Boot
Partition 2:
Angstrom* RootFS
- Arm toolchain
- XFCE Desktop
- Examples/demos
Partition 3:
U-Boot SPL* and U-Boot*

Ques. What would happen if you do not remove the SD card from the SoC Board while programming the device using Altera USB Blaster?

Ans. The processor boots, then configures the FPGA under software control. If you leave the SD Card plugged into the board and then reprogram the FPGA, a watchdog timer in the processor would eventually timeout since the FPGA can no longer respond to the processor because the image has changed. The timeout would force a warm reset which would cause the FPGA to be reprogrammed, overwriting the "blink" design you just downloaded.
This however also user reconfigurable. The user decides how the system responds to warm and cold reset. You may choose to leave the FPGA running "as is" when a processor reset condition occurs.

B.3. SOC FPGA DESIGN FLOW FOR ENCLUSTRA SOC-FPGA PLATFORM

The introduction of system-on-chip (SoC) FPGA devices.

Testing SoC/FPGA Platform Note: Don't use the FPGA until you have read Mercury+ SA2 SoC Module User Manual and Mercury+ PE-1 Base Board User Manual

1. The FPGA SoC CFGA and CFGB configurations should be accordingly for SD card boot settings viz.
   CFGA1 - OFF; CFGA2 - OFF; CFGA3 - OFF; CFGA4 - ON
   CFGB1 - OFF; CFGA2 - OFF; CFGA3 - OFF; CFGA4 - OFF

2. Once the SoC is ready configuration is set. Connect to PC through microUSB port; In device Manager, the SoC module is detected as Device Controller A and B. Go to Properties>Advanced>Enable VCP.
   Now the Board will be detectable as COM port. In the Serial Terminal, use the following settings:
   Baudrate = 115200, DataBits = 12, Parity = None, Stop = 1 Bit, FlowControl = None

3. Once the system is ready, it boot up process would be visible in the serial terminal. The Welcome prompt of Buildroot is now available. Username is *root* and Password is (also) *root*.
   The following command could be useful to get started from this point onward:
   `#/bin/busybox`

1. Learn using Buildroot (BusyBox[2]).

2. Configure the rootfs through Buildroot and figure things out from that point onward.

3. The default way to upload the FPGA program is using the USB Blaster (which is a JTAG interface for programming the FPGA). The reference design mentions the use of Blaster as a necessity; but perhaps there could be a way around it. Need to figure it out.
   SOLVED - The Quartus Prime 17.1 has option to support FPGA programming through USB. Check the software.

Information for building Linux Environment and common problems We are using the PE-1 Baseboard by Enclustra. Programming FPGA directly on Enclustra SoC board.

The detailed instructions along with necessary files are present in the Reference Design .zip file which can be downloaded from Enclustra Download Page (`https://download.enclustra.com/`).

Running Linux on Enclustra SoC board

The detailed documentation for building the linux distribution to run on the SoC is present at the Enclustra Build Environment page (`http://enclustra.github.io/ebe-docs/user-doc-altera/index_altera.html`). The instructions can be followed from there easily. We are going to build a linux environment for the Cyclone V SoC and boot it on the SD-MMC card.

Common Problems: .

Some common problems related to partitioning the SD card are given below:

1. During build process, the wizard would inquire which targets to fetch. It is important to fetch UBoot, Buildroot and Linux. RTLinux can also be fetched as it does not make a difference.

2. Before you begin to start partitioning the SD card, it is critical that the card's all partitions have been deleted such that everything is available as free space; and the card is unmounted. This can be achieved from terminal, or the easier way, by using Disks utility in the Applications menu of Linux (in our case, Ubuntu 16.04).

[2] also know as, The Swiss Army Knife of Embedded Linux

3. Once the SD card is prepared and loaded with Linux, the next stage is connecting the board to the system. Before that, the CFGA(OFF,OFF,OFF,ON) and CFGB(OFF,OFF,OFF,OFF) pins should be configured on the board. This instructs the board to load OS from the SD card.

4. In Windows the FTDI chip shows as Universal Serial Device Converter in the Device Manager. To get a COM port. You need to tick mark the enable VCP option in properties (for both Device Converters A B). Reconnect the board, the COM port will show up. Use the COM port with higher number out of the two, to connect on the serial terminal. Note: Flow Control Should be none, No handshaking, BAUD is 115200, Parity is None, Data Bits is 8 and Stop is 1 bit.

5. In Linux this can be a little tricky. The discussion and solution about this can be found here: Electrical Engineering Stack Exchange (https://electronics.stackexchange.com/questions/386407/how-to-open-up-serial-terminal-for-my-usb-device-converter-or-how-to-enable-vc). Again, you have to use ttyUSB1 (higher of the two) when communicating serially.

6. The initial .rbf (Raw Binary File) you upload after building the OS loads a simple program on the FPGA to blink LED3 on the SA2 board.

7. The Quartus generates .sof files that can be converted to .rbf (Raw Binary File) by a tool in Quartus prime, if there are no megafunctions used in the design. Otherwise a lisence is required for Quartus Prime.

8. Linux boots up and leaves you at the Buildroot Login. Username and Password are root

C

APPENDIX C: THEORY OF SPIN QUBITS IN QUANTUM DOTS

For a technology to be accepted as capable of performing quantum computation, it must satisfy the Di-Vincenzo's Criteria i.e. it must be (1) a scalable physical system with well characterized qubits, (2) the qubit states can be initialized to a simple state, such as $|000\rangle$, (3) coherence times should be much longer compared to the gate operation time, (4) there should be a "universal" set of quantum gates, and (5) the quantum state can be measured. These set of criteria dictate the qubit technology, however in order have a usable quantum system, in my opinion, we also require, 1) a scalable control methodology to manipulate the single quantum systems, 2) a high-level programming infrastructure and quantum/classical compiler toolchain, and 3) a hardware microarchitecture (close to the qubit) to control operations such as, branching, variational parameter updates and quantum error correction.

In case of Silicon-Spin Qubits in Quantum Dots, the electron spin-state in quantum dots is described by the Fermi-Hubbard model (Hamiltonian). The initialization and measurement of the same is performed by Elzerman method and the universal set of quantum gates can be performed by applying microwave pulses to these qubits.

A brief explanation of this is given as follows:

1. Spin qubits store information in the spin momentum of an electron. The spin of a single electron in a magnetic field (in the order of 1 Tesla) can be spin down (low energy) or spin up (high energy) (Energy Difference, approx. $100\mu eV$). This is due to the Zeeman Splitting. The g-factor gives a measure for the energy splitting corresponding to magnetic field strength.

2. The first step is to isolate a Single electron while preserving it from outside interactions. This can be done from a two dimensional electron gas (2DEG) layer. For this, we stack different materials and under certain precise conditions, we can get 2DEG plane that allows electrons to move freely. We can compare this with a very thin layer of metal.
   Gate electrodes are placed on top of 2DEG that attracts or repels the electrons and allows the formation of isolated electrons — one in each quantum dot. Each such electron in quantum dot acts as a qubit. Note that, in a 2DEG reservoir there is no difference between spin up and spin down of different electrons due to mutual interactions.

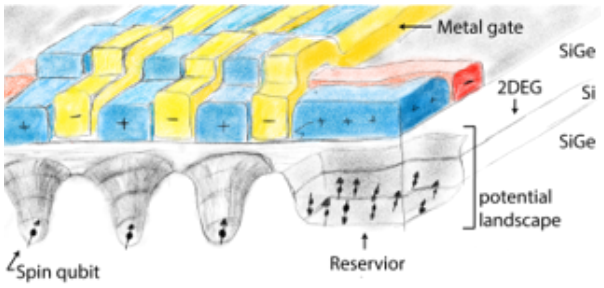


Figure C.1: 2DEG in a Spin-Qubit quantum dot device.

3. Initialization: To initialize the qubits, we align the energy level (electrochemical potential lines) of spin qubit with the reservoir such that only an electron with spin down can tunnel in.

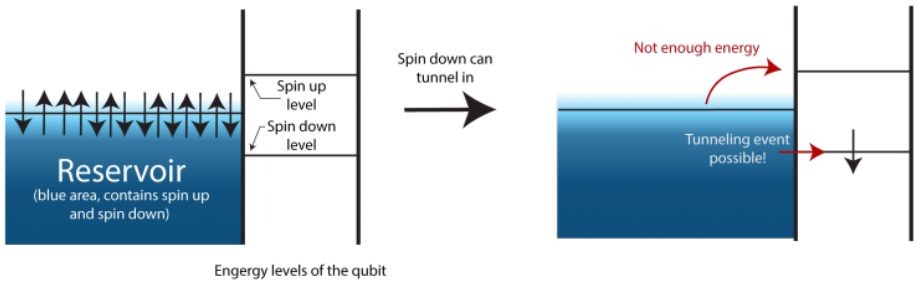


Figure C.2: Initialization of Spin Qubit

4. Gate Operations: Qubit control is done by introducing an alternating magnetic field that is perpendicular to the applied magnetic field, through a small strip near the quantum dot. The AC current generates electromagnetic AC wave and when frequency of the wave matches resonance frequency of the qubit, this leads to generation of photons with energy equal to the Zeeman split energy of spin states. This causes the spin state to flip. The spin starts to rotate as a function of time due to the alternating magnetic field. These are called *Rabi Oscillations* and are used to perform the single qubit gates.
   Two qubit gates are gates are used to cause interaction between two qubits and also to perform entanglement. This is achieved by pulsing the barrier between the two qubits. This causes the electrons to push close together, causing them to interact with each other, and their wavefunction to hybridize. Therefore, when the interaction is on, the resonance frequency of first qubit is determined by the state of other qubit. This allows us to control the state of one qubit depending on the state of the other qubit, and achieve for example, entanglement.

5. Measurement: Measurements is done by Elzerman method and is similar to initialization. We apply voltage to change the reservoir barrier potential and qubit (in quantum dot) such that such the Fermi energy of the reservoir lands in between the spin-up and spin-down states. When in spin up, the electron tunnels out into the reservoir. Due to this, a charge movement is registered. A charge detector detects a change in signal in the spin up case, no charge for spin-down state.
   The read-out graph shown below (figure C.3) is for a Gallium Arsenide (GaAs) where have negative g-factor. So, the bump corresponds to a spin-down.

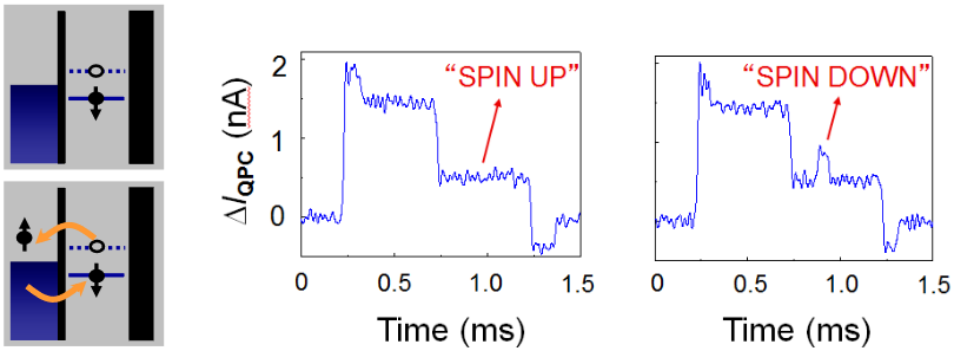


Figure C.3: Spin-Qubit Read-out.

The Spin-qubits can prove advantageous because:

1. The standard gate time in case of Spin-Qubits is within 10s of nanoseconds (*ns*) while the qubit decoherence time is usually between 100s of micro-seconds (which is nearly an eternity in quantum word). Decoherence is caused by electron's interaction with nuclear spin of the Si/SiGe substrate. Changing the substrate can allow further longer decoherence times up to 10s of Milli-seconds.

2. The Spin-qubits are one of the most promising candidates for long-term 'scalable' quantum computing. One-qubit occupies very less area on chip (70nm $times$ 70nm). These can be arranged in arrays and has potential to scale to millions of qubits for universal quantum computing (1 billion qubits in 1cm × 1 cm chip area).

3. Typically the spin-qubits are operated at temperatures of 10mK, but it has the potential to be operated at 1K to 4K. This is a very important point as classical CMOS can also be used at temperatures up to 4K which would allow manufacturing (in future) classical control and quantum-dot array units to be fabricated on the same silicon die, thus, eliminating complicated wiring for controlling these qubits (True Quantum Integrated Circuits). These cryo-electronics would have to be faster, less noisy and must have very low power dissipation.

4. Manufacturability of Spin-qubits is an important criteria. The current manufacturing is done by 300mm technology, which is an already perfected technology (such as, by Intel).

It is however, also important to remember that current quantum computing technologies are prone to high error rates and qubit decoherence. While the error-rates in spin-qubits are better than superconducting qubits; for the Spin-Qubit processor to function as a NISQ-era (Noisy Intermediate Scale Quantum) device (without quantum error correction); gate single-qubit gate error rates up to 10^{-5} or better (which currently, is not possible).

Another challenge to overcome is that each Spin qubit is different — Different gate volatges are required to control each qubit, Spin up and Spin down energies are different etc. This is due to imperfections in materials and in fabrication process. So, making a control infrastructure capable of addressing each qubit is hard and requires a collaborative effort of Electronics Engineering and Experimental Physics.

BIBLIOGRAPHY

[1] J. P. van Dijk, E. Charbon, and F. Sebastiano, *The electronic interface for quantum processors,* Microprocessors and Microsystems **66**, 90 (2019).

[2] X. Fu, L. Riesebos, M. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, V. Newsum, K. Loh, *et al., eqasm: An executable quantum instruction set architecture,* in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (IEEE, 2019) pp. 224–237.

[3] A. Devices, *Ad9910 datasheet,* https://www.analog.com/media/en/technical-documentation/data-sheets/AD9910.pdf, online Link.

[4] R. P. Feynman, *Simulating physics with computers,* International journal of theoretical physics **21**, 467 (1982).

[5] R. P. Feynman, *Quantum mechanical computers,* Foundations of physics **16**, 507 (1986).

[6] Y. I. Manin, *Vychislimoe i nevychislimoe (Computable and Non-Computable)* (Sov. radio, 1980).

[7] Y. I. Manin, *Classical computing, quantum computing, and shor's factoring algorithm,* arXiv preprint quant-ph/9903008 (1999).

[8] P. Benioff, *The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines,* Journal of statistical physics **22**, 563 (1980).

[9] P. Benioff, *Quantum mechanical hamiltonian models of turing machines,* Journal of Statistical Physics **29**, 515 (1982).

[10] D. Deutsch, *Quantum theory, the church–turing principle and the universal quantum computer,* Proc. R. Soc. Lond. A **400**, 97 (1985).

[11] D. Deutsch and R. Jozsa, *Rapid solution of problems by quantum computation,* Proc. R. Soc. Lond. A **439**, 553 (1992).

[12] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,* SIAM J. Comput. **26**, 1484 (1997).

[13] D. P. DiVincenzo, *The physical implementation of quantum computation,* Fortschritte der Physik: Progress of Physics **48**, 771 (2000).

[14] J. Preskill, *Quantum computing and the entanglement frontier,* arXiv preprint arXiv:1203.5813 (2012).

[15] J. Preskill, *Quantum computing in the nisq era and beyond,* Quantum **2**, 79 (2018).

[16] G. E. Moore *et al., Cramming more components onto integrated circuits,* (1965).

[17] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, *Design of ion-implanted mosfet's with very small physical dimensions,* IEEE Journal of Solid-State Circuits **9**, 256 (1974).

[18] N. Khammassi, G. Guerreschi, I. Ashraf, J. Hogaboam, C. Almudever, and K. Bertels, *cqasm v1. 0: Towards a common quantum assembly language,* arXiv preprint arXiv:1805.09607 (2018).

[19] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *Open quantum assembly language,* arXiv preprint arXiv:1707.03429 (2017).

[20] M. OCW, *Quantum complexity theory,* https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-845-quantum-complexity-theory-fall-2010/, oCW.

[21] D. Wecker, B. Bauer, B. K. Clark, M. B. Hastings, and M. Troyer, *Gate-count estimates for performing quantum chemistry on small quantum computers,* Physical Review A **90**, 022305 (2014).

[22] A. Sarkar, *Quantum algorithms: for pattern-matching in genomic sequences,* (2018).

[23] I. Shapoval and P. Calafiura, *Quantum associative memory in hep track pattern recognition,* arXiv preprint arXiv:1902.00498 (2019).

[24] J. Preskill, *Simulating quantum field theory with a quantum computer,* arXiv preprint arXiv:1811.10085 (2018).

[25] S. Wehner, C. Schaffner, and B. M. Terhal, *Cryptography from noisy storage,* Physical Review Letters **100**, 220502 (2008).

[26] E. Farhi, J. Goldstone, and S. Gutmann, *A quantum approximate optimization algorithm,* arXiv preprint arXiv:1411.4028 (2014).

[27] D. Loss and D. P. DiVincenzo, *Quantum computation with quantum dots,* Physical Review A **57**, 120 (1998).

[28] H. Bluhm, S. Foletti, I. Neder, M. Rudner, D. Mahalu, V. Umansky, and A. Yacoby, *Dephasing time of gaas electron-spin qubits coupled to a nuclear bath exceeding 200 μs,* Nature Physics **7**, 109 (2011).

[29] J. Medford, J. Beil, J. Taylor, S. Bartlett, A. Doherty, E. Rashba, D. DiVincenzo, H. Lu, A. Gossard, and C. M. Marcus, *Self-consistent measurement and state tomography of an exchange-only spin qubit,* Nature nanotechnology **8**, 654 (2013).

[30] J. R. Petta, A. C. Johnson, J. M. Taylor, E. A. Laird, A. Yacoby, M. D. Lukin, C. M. Marcus, M. P. Hanson, and A. C. Gossard, *Coherent manipulation of coupled electron spins in semiconductor quantum dots,* Science **309**, 2180 (2005).

[31] F. A. Zwanenburg, A. S. Dzurak, A. Morello, M. Y. Simmons, L. C. Hollenberg, G. Klimeck, S. Rogge, S. N. Coppersmith, and M. A. Eriksson, *Silicon quantum electronics,* Reviews of modern physics **85**, 961 (2013).

[32] A. M. Tyryshkin, S. Tojo, J. J. Morton, H. Riemann, N. V. Abrosimov, P. Becker, H.-J. Pohl, T. Schenkel, M. L. Thewalt, K. M. Itoh, *et al.*, *Electron spin coherence exceeding seconds in high-purity silicon,* Nature materials **11**, 143 (2012).

[33] T. Watson, S. Philips, E. Kawakami, D. Ward, P. Scarlino, M. Veldhorst, D. Savage, M. Lagally, M. Friesen, S. Coppersmith, *et al.*, *A programmable two-qubit quantum processor in silicon,* Nature **555**, 633 (2018).

[34] J. Yoneda, K. Takeda, T. Otsuka, T. Nakajima, M. R. Delbecq, G. Allison, T. Honda, T. Kodera, S. Oda, Y. Hoshi, *et al.*, *A quantum-dot spin qubit with coherence limited by charge noise and fidelity higher than 99.9%,* Nature nanotechnology **13**, 102 (2018).

[35] X. Xue, T. Watson, J. Helsen, D. Ward, D. Savage, M. Lagally, S. Coppersmith, M. Eriksson, S. Wehner, and L. Vandersypen, *Benchmarking gate fidelities in a si/sige two-qubit device,* Physical Review X **9**, 021011 (2019).

[36] D. Zajac, T. Hazard, X. Mi, E. Nielsen, and J. R. Petta, *Scalable gate architecture for a one-dimensional array of semiconductor spin qubits,* Physical Review Applied **6**, 054013 (2016).

[37] C. Volk, A. Zwerver, U. Mukhopadhyay, P. Eendebak, C. van Diepen, J. Dehollain, T. Hensgens, T. Fujita, C. Reichl, W. Wegscheider, *et al.*, *Loading a quantum-dot based" qubyte" register,* arXiv preprint arXiv:1901.00426 (2019).

[38] U. Mukhopadhyay, J. P. Dehollain, C. Reichl, W. Wegscheider, and L. M. Vandersypen, *A 2× 2 quantum dot array with controllable inter-dot tunnel couplings,* Applied Physics Letters **112**, 183505 (2018).

[39] P.-A. Mortemousque, E. Chanrion, B. Jadot, H. Flentje, A. Ludwig, A. D. Wieck, M. Urdampilleta, C. Bauerle, and T. Meunier, *Coherent control of individual electron spins in a two dimensional array of quantum dots,* arXiv preprint arXiv:1808.06180 (2018).

[40] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, *A quantum engineer's guide to superconducting qubits,* Applied Physics Reviews **6**, 021318 (2019).

[41] M. S. Q, *Qcodes,* http://qcodes.github.io/Qcodes/ (), gitHub.

[42] M. S. Q, *Pycqed_py3,* https://github.com/DiCarloLab-Delft/PycQED_py3 (), gitHub.

[43] X. Fu, M. Rol, C. Bultink, J. Van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, J. De Sterke, W. Vlothuizen, R. Schouten, *et al.*, *An experimental microarchitecture for a superconducting quantum processor,* in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (ACM, 2017) pp. 813–825.

[44] C. E. Cummings, *Simulation and synthesis techniques for asynchronous fifo design,* in *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers* (2002).