Scalable Control Architectures for Quantum Computing

by

Bradley Barrier Bondurant

Department of Electrical and Computer Engineering
Duke University

Date: May 30, 2024

Approved:

Kenneth R Brown, Supervisor

Jungsang Kim, Co-Chair

Iman Marvian

Michael Gehm

Henry Pfister

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University
2024

Abstract

Scalable Control Architectures for Quantum Computing

by

Bradley Barrier Bondurant

Department of Electrical and Computer Engineering
Duke University

Date: May 30, 2024

Approved:

Kenneth R Brown, Supervisor

Jungsang Kim, Co-Chair

Iman Marvian

Michael Gehm

Henry Pfister

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University
2024

Abstract

As performance gains in classical CPUs have slowed over time, the field of computing has looked to accelerators like the GPU as a solution. One such emerging technology is the quantum computer, leveraging the exotic properties of quantum mechanical systems to perform calculations that are intractable on any classical hardware. While there is still a long road ahead until some of the most influential algorithms can be realized, small quantum computers are now a reality, with larger systems on the horizon. As these systems scale to larger sizes, the complexity of the classical hardware and software that control them must also scale. This dissertation seeks to address some of engineering challenges of scaling those real-time control systems to keep up with the demand of present and future quantum computers. We show that well-architected control software can improve performance and portability even between distinct quantum hardware systems, allowing faster development of more robust software for the next generation of systems. Due to the extreme precision required for high-fidelity control of quantum systems, routine calibration of the analog hardware components is required for the operation of a quantum computer. We present a framework for automating and optimizing that calibration using a graph-based approach. Finally, this dissertation presents progress toward a heterogeneous, system-on-chip based hardware platform for quantum control and its potential significance for the modular scaling of quantum hardware.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

1D one-dimensional

2D two-dimensional

3D three-dimensional

ACP accelerator coherency port

ADC analog to digital converter

AMD Advanced Micro Devices, Inc.

AMP asymmetric multi-processing

AOM acousto-optic modulator

API application programming interface

APU application processing unit

ARTIQ advanced real-time infrastructure for quantum physics

ASIC application-specific integrated circuit

AWG arbitrary waveform generator

AXI advanced extensible interface

CCI cache-coherent interconnect

CPU central processing unit

CRC cyclic redundancy check

CW continuous wave

DAC digital to analog converter

DAG directed acyclic graph

DAX Duke ARTIQ extensions

DDR double data rate

DDS direct digital synthesizer

DIMM dual inline memory module

DMA direct memory access

DSL domain-specific language

EEPROM electrically erasable programmable read-only memory

EL exception level

ENIAC electronic numerical integrator and computer

EPIC explicitly parallel instruction computing

FAT file allocation table

FFI foreign function interface

FIFO first in, first out

FPD full-power domain

FPGA field-programmable gate array

FPU floating point unit

FSBL first-stage boot loader

GALS globally asynchronous, locally synchronous

GEM gigabit ethernet MAC

GIC generic interrupt controller

GMII gigabit MII

GPIO general purpose input/output

GPU graphics processing unit

GSPS gigasamples per second

GST gate set tomography

HDL hardware definition language

I^2C inter-integrated circuit

IC integrated circuit

I/O input/output

IRQ interrupt request

ISA instruction set architecture

LPD low-power domain

MAC media access control/controller

MII media-independent interface

MIO multiplexed input/output

MMU memory management unit

MOSFET metal–oxide–silicon field-effect transistor

MPSoC multi-processing system-on-chip

MS Mølmer-Sørensen

MTU maximum transmission unit

MW microwave

NISQ noisy intermediate-scale quantum

OCM on-chip memory

OS operating system

PA physical address

PC personal computer

PCAP processor configuration access port

PL programmable logic

PLL phase-locked loop

PMT photomultiplier tube

PS processing system

PUB PHY utility block

QICK Quantum Instrumentation Control Kit

QoS quality of service

QPU quantum processing unit

QSCOUT quantum scientific computing open user testbed

qubit quantum bit

RA read allocate

RB randomized benchmarking

RC red chamber

RF radio-frequency

RFMC radio-frequency mezzanine card

RFSoC radio-frequency system-on-chip

RGMII reduced GMII

RPC remote procedure call

RPU real-time processing unit

RTIO real-time I/O

RX receive

SAWG smart AWG

SD Secure Digital

SDRAM synchronous dynamic random-access memory

SIMD single instruction, multiple data

SMMU system memory management unit

SoC system-on-chip

SODIMM small-outline DIMM

SPAM state preparation and measurement

SPD serial presence detect

SPI serial peripheral interface

SQST single-qubit state tomography

STAQ software-tailored architecture for quantum co-design

TCM tightly coupled memory

TCP transmission control protocol

TLB translation lookaside buffer

TPU tensor processing unit

TX transmit

UART universal asynchronous receiver-transmitter

UDIMM unbuffered DIMM

VA virtual address

VLIW very long instruction word

VMSA virtual memory system architecture

WA write allocate

WB write-back

WT write-through

$^{171}\mathbf{Yb}^{+}$ Ytterbium 171

1

Introduction

Computers are pretty neat. Less than a century has passed since Alan Turing's seminal work in 1936 [1] describing what would come to be known as a universal Turing machine and introducing defining criteria for what we now think of as a "computer." The first such general purpose computer, electronic numerical integrator and computer (ENIAC), was completed in 1945 and used vacuum tubes to represent its data. Only two years later the transistor was invented, followed in 1959 by the metal–oxide–silicon field-effect transistor (MOSFET). The MOSFET was the first transistor that was able to be miniaturized and mass-produced, and would go on to form the basis for all modern computing. Using MOSFET technology, Intel produced the world's first single-chip microprocessor, the Intel 4004, in 1971. The time since then has seen continuous advancement in microprocessor technology, with Moore's law [2] accurately predicting a near doubling of transistor counts every two years and Dennard scaling [3] implying that such doubling would not result in an increase in power consumption provided the circuit area remained the same.

Both of those laws, however, have begun to break down in recent years. After all, transistors can only be made so small, clock speeds can only increase so much,

and Silicon can only dissipate so much heat. So, what comes next? After clock speeds began to plateau around 2005 [4], we saw the advent of multi-core processors, allowing processing power to continue to scale without solving any of the fundamental problems. Finally, someone remembered that CPUs are *general purpose* machines – if something is computable, they can compute it, but that does not mean they are the most efficient machines for doing so. Thus, the *accelerator* was born: hardware that specializes in a certain class of problems it can compute more efficiently than a traditional CPU. Now ubiquitous, GPUs – capable of simultaneously performing the same operation on large data sets – were the first accelerator to gain popularity. This capability offers a significant speedup for the matrix arithmetic involved in not only graphics processing, but many scientific computations as well. Accelerator development continues today, with inventions such as the tensor processing unit (TPU) offering even more specialized speedups targeted toward machine learning applications.

The most capable accelerator of all, however, might be one that is still in its infancy: the quantum computer, or quantum processing unit (QPU). The QPU presents a fundamental divergence from classical computing methods, exploiting the exotic properties of quantum mechanics to perform calculations in a way that classical CPUs and accelerators physically cannot. The notion of using a quantum mechanical system to simulate more complex quantum systems was first proposed by Richard Feynman in 1981 [5], but the idea of more general computation using quantum systems only started to gain traction in the 1990s with the development of quantum algorithms that offered significant speedups over classical methods in solving practical problems. Such algorithms include Shor's factoring algorithm [6], Grover's search algorithm [7], and Kitaev's phase estimation algorithm [8]. In the years since, the QPU has gone from an abstract idea to a burgeoning commercial product, with some modern systems having over 100 qubits [9, 10] and being able to

out-perform classical computers on certain contrived tasks [11].

Despite the boom in commercial interest over the last decade or so, even the most advanced QPUs are still not much more than over-engineered physics experiments – operating them requires careful calibration and precise control over dozens of state-of-the-art devices. As the number of qubits scales, so too does the complexity of this device network and the software that controls it. In industry, that control problem is approached by delegating large teams of experienced engineers to perform thousands of man-hours of work. In academia, we do not have that luxury. A typical academic experiment (read: QPU under construction) might, at best, have 3 or 4 PhD students and 1 or 2 postdoctoral scholars working on it, most of whom are probably coming from a physics background. And though academic research tends to focus on fundamental technologies rather than creating large, commercial systems, scalability is integral to the long-term viability of any of those technologies, and as such is still very much a topic of interest. So, with the resources and personnel available in an academic setting, how can we approach the problem of scaling up QPUs and more specifically the software and hardware that control them? That is the question I seek to answer in this dissertation.

Chapter 2 provides the necessary background for understanding the rest of my work: the general requirements for a QPU control system, the specific requirements of the physical qubit realization that my work focuses on, and the control framework that my work builds upon. Chapter 3 discusses an approach to control software that achieves a more robust and portable product, lowering the barrier to scalability from a software design perspective. Chapter 4 builds upon that to automate and optimize the calibrations necessary to keep a QPU running, again lowering the barrier to scalability. Chapter 5 takes a dive down to the hardware level, discussing my work toward supporting a new quantum control platform and the architectural implications it has for scaling up quantum control systems. Finally, in Chapter 6 I

conclude the dissertation and provide an outlook for the field of quantum control

2

Background

2.1 What Is a Qubit and How Do We Do Stuff With It?

To understand what a quantum control system needs to do, we must first understand the basics of the quantum system being controlled: the quantum bit (qubit). Abstractly speaking, a qubit is any two-level quantum system – that can take the form of a "true" two-level system such as the spin of an electron (up or down), or two levels of a larger system, provided we can reasonably isolate them from the rest of it. Analogous to bits and voltages in a transistor, we pick one of these levels and call it $|0\rangle$, and we call the other level $|1\rangle$. Where the analogy breaks down is in the description of the state of the qubit in terms of these levels. While the state of a bit is either 0 or 1, the state of a qubit, $|\psi\rangle$, is described by a *superposition* of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,,$$

where α and β are complex valued and

$$|\alpha|^2 + |\beta|^2 = 1.$$

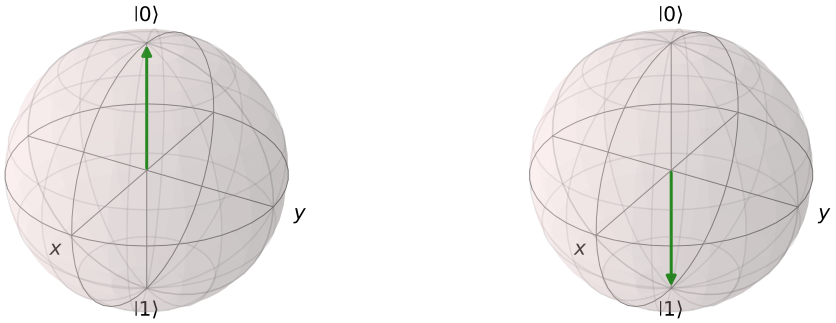
Together, those equations allow us to express the state geometrically as

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

where γ, θ, and ϕ are all real valued. It turns out that we can disregard the leading term [12], giving us

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle.$$

Now we can visualize the state of our qubit as a vector on the complex three-dimensional (3D) unit sphere, called the *Bloch* sphere [12]. Figure 2.1 shows what the $|0\rangle$ and $|1\rangle$ states look like on the Bloch sphere. Similar to the digital electronics



(a) $|0\rangle$ on the Bloch sphere. (b) $|1\rangle$ on the Bloch sphere.

Figure 2.1: Example states on the Bloch sphere.

that make up a classical computer, we can perform logic gates on our qubits. Unlike their digital counterparts, however, quantum gates must continuously *evolve* the qubit from initial to final state. Take, for example, the Pauli X gate, the quantum analog of a digital NOT gate. If our qubit is initially in the $|0\rangle$ state, the Pauli X gate will evolve the state over time to the $|1\rangle$ state, rotating it around the Bloch sphere as shown in Figure 2.2.

On a physical quantum processing unit (QPU), that evolution is achieved by applying some sort of signal – typically an electromagnetic field – to the qubit for

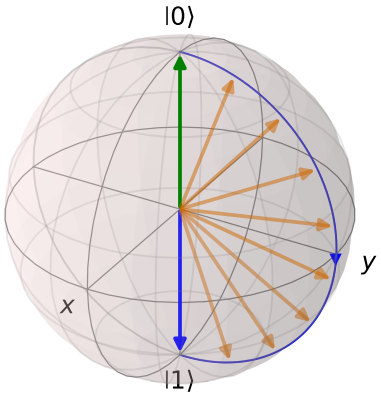


Figure 2.2: A visualization of the Pauli X gate applied to a qubit initially in the $|0\rangle$ state.

some duration. Any error in the timing will result in an imperfect (not quite $|1\rangle$) final state – an error. We now have the first requirement of a quantum control system: it must be able to perform input and output with precise, deterministic timing. These real-time I/O (RTIO) operations are integral to operating any QPU, regardless of the physical implementation. The definition of "precise," however, *is* platform-dependent. Continuing with the same example, the Pauli X gate might take 10 microseconds on one platform and 10 nanoseconds on another, so a precision of 1 nanosecond can mean very different things depending on the physical qubit. Any error in the signal we apply (e.g., the frequency) will also result in some error on the qubit, but the type of signal required is platform-dependent and its quality is more a characteristic of the electronics being controlled than the controller itself.

Gate durations are far from the only factor placing timing restraints on the quantum control system. When a qubit is subject to environmental noise (as is the case for any real-world qubit), the information held by that qubit deteriorates over time, a process called *decoherence*. The *coherence time* of a qubit – how quickly it decoheres – is another critical timing constraint and can vary just as widely as gate durations between different physical qubits. To make matters worse, there are actually two

types of decoherence – *amplitude* and *phase* – both with their own characteristic times.

Moving beyond the example of performing a single gate, our controller must also support *feedback*: changing its output in real time based on some input. This requirement comes from the fact that much of a QPU's behavior is nondeterministic, requiring a repeat-until-success approach. Furthermore, many quantum algorithms require feedback, performing different gates based on the result of measuring a qubit's state. In principle, this feedback could be performed by a specialized circuit: an application-specific integrated circuit (ASIC). At present, however, that is not a realistic option for two main reasons:

1. ASICs are prohibitively expensive to make, only achieving cost-effectiveness at high production volumes.

2. QPUs are immature and unstable – it's not feasible for every small change to a system to require designing and producing a new circuit.

The second reason gives us our third control system requirement: *programmability*.

We now have the three core requirements of a quantum control system:

1. RTIO capabilities

2. Feedback support

3. Programmability

The device that satisfies all three is a field-programmable gate array (FPGA), a device that allows its user to program digital circuits into it – essentially a programmable ASIC. FPGAs are programmed with *gateware* written in a hardware definition language (HDL) such as Verilog, not a particularly easy task. Especially given the complexity of controlling a QPU – complexity which scales with the size

of the system – writing new gateware and reprogramming the FPGA still isn't an ideal solution. You know what *is* easy to program? A regular old CPU. That's kind of their whole thing. Wouldn't it be nice to have the programmability of a CPU alongside the RTIO capabilities of an FPGA? The quantum field's answer to that question is a resounding "yes" [13–16]. After all, a CPU is just a digital circuit, fully implementable on an FPGA. In order to talk about the specific options for such a system, it is first necessary to discuss a few details of the target qubit: in the case of my work, trapped atomic ions.

2.2 Trapped Ions

In their neutral form, the atoms that make for good qubits have two valence electrons. Once we ionize one of those electrons, there is one left, giving us a Hydrogen-like system that is relatively easy to reason about and control. The fact that we now have an electrically charged species also makes it simple to suspend or *trap* the ion in a vacuum using oscillating electric fields [17]. To avoid an ultimately superfluous discussion of the different options for such an ion and the different ways we can use them to represent a qubit, I will focus on the ion that my work at Duke has most often been applied to: Ytterbium 171 ($^{171}Yb^{+}$).

$^{171}Yb^{+}$ is a *hyperfine* qubit, meaning that the two energy levels we choose to represent $|0\rangle$ and $|1\rangle$ arise from the hyperfine splitting [18] of the ground electronic state. These states are long-lived, with coherence times as high as ten minutes [19] (although more typically on the order of seconds [20–22]). The frequency splitting between these two energy levels is roughly 12.6 GHz [20]. If we apply microwave radiation on resonance with that frequency, we can induce transitions of the electron between the two states as illustrated in Figure 2.3, physically implementing the Pauli X gate depicted in Figure 2.2. Another option is to induce the same transition using two laser beams with a frequency difference of 12.6 GHz [23], something called a

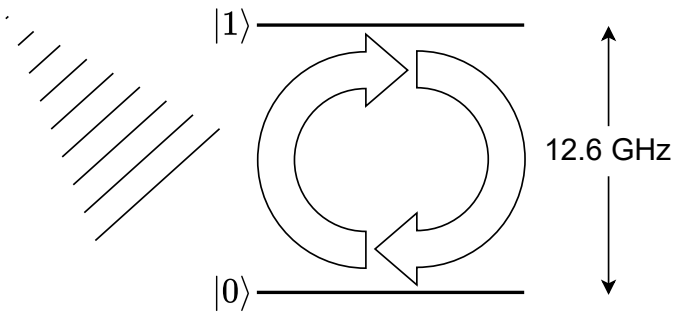


Figure 2.3: Illustration of a single-qubit microwave gate in $^{171}Yb^{+}$.

stimulated Raman transition. The benefit of using lasers is that they can be focused down tightly enough to address individual ions in a multi-ion chain, which is not possible at microwave frequencies. With the lasers and electronics used in our labs [24], the only fine tuning that needs to be applied to the laser frequencies is at roughly 200 MHz. The fine tuning is applied via an acousto-optic modulator (AOM), which modulates the laser beam's frequency, phase, and amplitude using a radio-frequency (RF) source. With these mechanisms in place, single-qubit Raman gates have a duration on the order of microseconds [21, 22, 25–27].

Two-qubit gates are a different matter, taking hundreds of microseconds to complete [21, 22, 25]. They are significantly more complicated as well, involving not only the electronic states of the two ions, but also the quantized states of motion that arise from having the charged particles trapped in a chain together. This type of interaction is called a Mølmer-Sørensen (MS) gate, with the energy levels involved shown in Figure 2.4. The complex nature of this interaction along with the relatively long time it requires make it particularly susceptible to environmental noise. Progress toward making MS gates more robust to that noise has been made via the introduction of complex modulation schemes [29–31]. Excellent as that work may be, it places additional strain on our quantum control system, with one possible solution explored in Chapter 5.

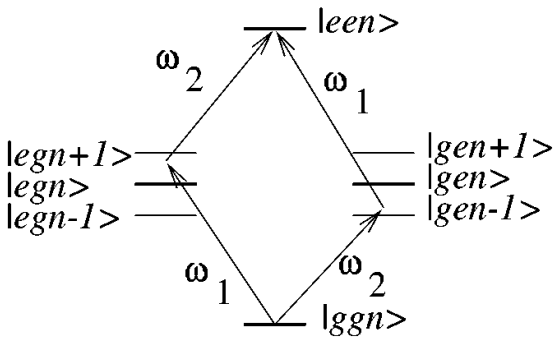


Figure 2.4: Energy levels and laser frequencies involved in an MS gate. g and e represent our qubit's $|0\rangle$ and $|1\rangle$ states, respectively. n represents the motional state of the ions. ω_1 and ω_2 are the laser frequencies. Figure from [28].

2.3 ARTIQ

The core of what ARTIQ provides is a layer of abstraction over FPGA and RTIO hardware so that the average user does not need to be highly proficient in digital electronics to operate that hardware. Programs in ARTIQ are written in a Python-like domain-specific language (DSL) (often called ARTIQ-Python, for lack of a better name) that gets compiled to machine code and run on the soft-core CPU in the FPGA (the *core device*). ARTIQ follows the accelerator model, where a classical *host* computer offloads tasks to the quantum accelerator via timing-critical *kernels* running on the core device. Communication between the host process and the kernel is supported via remote procedure calls (RPCs), allowing for flexible and sophisticated control schemes.

There is a cost that comes along with ARTIQ's abstraction: it is not as performant as a custom FPGA design. That being said, ARTIQ achieves nanosecond timing resolution and sub-microsecond latency [13], more than satisfying the requirements for "slower" qubits such as those based on trapped atomic ions. Controlling trapped-ion QPUs using ARTIQ is in fact the subject of most of the work I have done, however many of the principles of that work extend beyond trapped ions and

ARTIQ. Further details of the ARTIQ ecosystem will be introduced as needed in Chapters 3, 4, and 5.

3

Modular Software for Real-Time Quantum Control Systems

This chapter is based on the following paper: L. Riesebos, B. Bondurant, J. Whitlow, J. Kim, M. Kuzyk, T. Chen, S. Phiri, Y. Wang, C. Fang, A. V. Horn, J. Kim, and K. R. Brown, "Modular software for real-time quantum control systems," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 545–555. DOI: 10.1109/QCE53715.2022.00077. It also appears in Leon Riesebos' thesis: L. Riesebos, "Software architectures for real-time quantum control systems," Ph.D. dissertation, Duke University, 2022.

My contributions to the work presented in the paper were in the design of the software architecture and the implementation of some of the software-tailored architecture for quantum co-design (STAQ) and red chamber (RC) system code.

3.1 Introduction

The field of quantum computing is rapidly evolving in the areas of software and hardware. On the software side, quantum programming languages and compilers

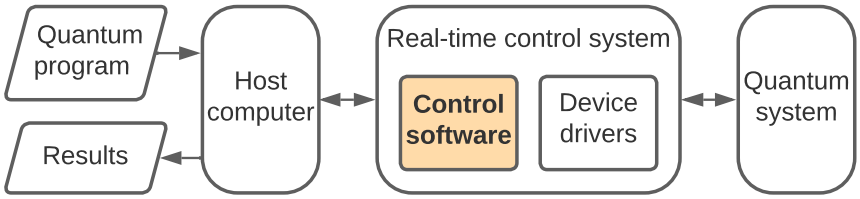


Figure 3.1: A real-time control system bridging the gap between the quantum program and the quantum system.

are becoming more available and feature-rich [34–39]. At the same time, quantum hardware is becoming increasingly powerful with recent systems demonstrating computations on tens of qubits [27, 40–45]. An often underexposed area in the field of quantum computing is the control software and hardware that bridges the gap between the quantum program and the targeted quantum system. Recent papers [24, 40, 43, 46] have shown that current state-of-the-art quantum systems already require tens to hundreds of devices to be controlled with high precision and strict real-time requirements, proving to be a significant challenge for the control system. Existing control hardware as described in [13, 47–50] provides the required real-time control of devices, but it is up to the real-time control software to close the remaining gap between device-level control hardware and quantum programs as illustrated in Figure 3.1.

Control software for quantum systems that runs on the real-time controller is similar to high-performance and resource-constrained embedded software. The software is responsible for real-time control of a set of devices while capturing and processing data simultaneously. Real-time controlled devices include direct digital synthesizer (DDS) devices, digital I/O, and digital to analog converters (DACs). Additionally, the real-time control system for a quantum system functions as a coprocessor and maintains a connection with a host computer. Due to the performance requirements and the strong dependence between the software and hardware, real-time control software is often tailored for a specific quantum system at the cost of flexibility and

portability. Since quantum computing is still an emerging technology, most quantum systems are unique. As a result, real-time control software is often redeveloped for each system which causes significant development overhead.

In this paper, we propose a systematic design strategy for real-time quantum control software. We present an open-source software framework for the advanced real-time infrastructure for quantum physics (ARTIQ) open-source software and hardware ecosystem [13, 51] to apply our design concepts to real-time quantum control software. Our framework supports the development of modular control software to enhance flexibility and portability on the level of real-time system code. Portability on the application level is achieved by introducing software abstractions. We show that modular control software developed with our framework can reduce the execution time overhead of real-time software and achieve high degrees of code portability. Reduced execution time overhead is achieved by fine-grained timing management and data offloading features of the modular software. We demonstrate the capabilities of our software framework by running a portable randomized benchmarking experiment on two different ion-trap quantum systems that are fully controlled by software based on our framework.

The remainder of this paper is structured as follows. Related work is discussed in Section 3.2. In Section 3.3 we present our design strategy and modular software architecture for real-time quantum control software. Our performance analysis and code portability analysis can be found in Section 3.4 and 3.5, respectively. In Section 3.6 we present experimental results from two ion-trap quantum systems. We conclude our paper in Section 3.7.

3.2 Related Work

Control software for ARTIQ systems can be developed without the use of our proposed framework. Programmers will have access to a classically complete program-

ming environment, basic data storage utilities, and device drivers to program real-time devices. However, ARTIQ is set up as a fully generic control system and no utilities are provided to support modular software development. As a result, control software is often tightly coupled to the hardware and needs to be completely redeveloped for each system. Especially the real-time timing of the software is often highly dependent on the controlled devices. The tight coupling of the hardware and software makes it difficult to change devices in existing systems since any modification likely introduces timing issues. At this moment, we are not aware of any other frameworks or libraries to support the development of modular control software for ARTIQ. Other real-time control systems similar to ARTIQ, such as M-ACTION [43, 47] and IonControl [48], suffer from the same limitations.

QCoDeS [52] is a modular data acquisition framework mainly intended to orchestrate the setup and data collection of instruments and devices part of a quantum control system. Some of these instruments can have real-time features, but QCoDeS does not control instruments in real-time while an experiment runs. Instead, all code involving QCoDeS runs on the host computer in a Python environment. Real-time instruments are also much more coarse compared to ARTIQ. A single ARTIQ real-time controller with many real-time I/O devices would correspond to a single QCoDeS instrument. The concepts behind QCoDeS show some similarities with the non-real-time components of the ARTIQ host environment. What sets ARTIQ apart from QCoDeS is its seamless integration and combination of real-time software within the host environment. ARTIQ puts the real-time controller in the center based on the principles of the accelerator model, while QCoDeS behaves more as a hypervisor for devices. The concepts presented in this paper apply to low-level real-time control software and its interaction with the host, and QCoDeS is not involved in the former. QCoDeS does have features for software modularity but these are limited to the instrument level without introducing any form of hierarchy. Other software based on

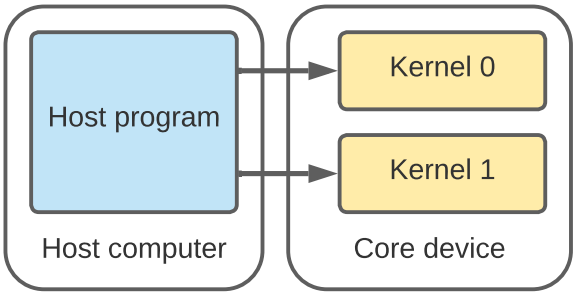


Figure 3.2: Schematic overview of the accelerator model with a host program and one or more kernels.

or derived from QCoDeS, such as PycQED [53], is built on the same principles and has the same limitations.

Qiskit [36] is an open-source library for creating, compiling, and executing quantum programs at the gate level. While it does allow users to execute quantum programs, Qiskit itself does not transparently connect to any device-level drivers and is not directly involved in the real-time control of devices when the compiled quantum program runs. Hence, Qiskit merely describes a circuit and is not directly part of the real-time control software that runs the circuit on hardware. Even the pulse-level control provided in Qiskit is an opaque abstraction over the device-level drivers. The same holds for related and similar tools such as OpenQASM [35], OpenPulse [54], Q#[34], and Cirq [39].

3.3 Software Architecture

Our software architecture targets the ARTIQ open-source software and hardware ecosystem [13, 51] which is used by dozens of research groups and has deployed over 200 real-time control systems worldwide. ARTIQ follows the principles of the *accelerator model* [34, 37, 49, 55–59] where a program consists of a host program and one or more *kernels*. The host program executes on a host machine and can offload the execution of kernels to an accelerator. For ARTIQ, the host program runs on a classical computer while kernels run on the real-time control hardware as illustrated

in Figure 3.2. ARTIQ kernels are classically complete and have access to real-time devices that interact with the quantum system (e.g. DDS devices, digital I/O, and DACs). The real-time control hardware, referred to as the *core device*, contains a classical CPU and a real-time I/O (RTIO) subsystem that schedules events for real-time devices on a timeline using a timeline cursor as described in [49, 60]. The ARTIQ system is programmed in a Python host environment and kernels are functions written in the ARTIQ domain-specific language (DSL), which is a Python-like language containing additional constructs for manipulating the timeline cursor and inserting events. ARTIQ allows host and kernel code to be combined in a single file, and kernels are functions or methods decorated with the `@kernel` decorator. When the host calls a kernel function, the host will invoke the ARTIQ compiler that compiles the kernel to a binary. The resulting binary is uploaded to and executed by the core device. While the core device executes the kernel, the host serves any synchronous or asynchronous remote procedure calls (RPCs) from the core device. RPCs are often used to stream real-time data to the host, access peripheral (i.e. non-real-time) devices, and offload computationally heavy tasks to the host. Once the kernel finishes execution, the host program resumes execution. The ARTIQ programming environment is set up generically and does not provide additional utilities for organizing real-time control software. In this section, we will present our design principles for real-time control software written for an ARTIQ control system.

The first step towards code organization is to separate common functionality of the system (i.e. *system code*) from experiment-specific routines (i.e. *experiment code*). System code can be collected in a base class, and each experiment can inherit from such a class and add experiment-specific routines. This approach is already common practice for most ARTIQ experiments. Our software architecture focuses on the development of modular system code. We propose to break the system code into two components: device organization with *modules* and extensible system-wide

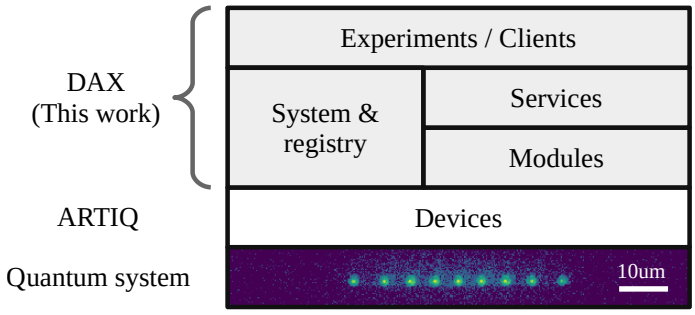


Figure 3.3: Schematic overview of the software components in our modular architecture controlling a quantum system, in this case, trapped atomic ions.

functionality using *services*. Additionally, we will introduce the notion of a central and searchable *registry* in which all modules and services of a system are registered. To improve code portability, we introduce abstractions with *interfaces* and *clients*. Figure 3.3 shows an architectural overview of the different components, which we will describe in the remainder of this section.

3.3.1 Modules, Services, and the Registry

While most experiments require a large set of real-time devices to collaborate closely, subsets of devices that perform basic procedures often have a tighter relation from a control perspective. A subset of devices might have strict control or safety requirements independent from other devices in the system. For example, two devices might always need to be switched simultaneously to achieve some desired functionality. We introduce the concept of modules to group such a logical collection of devices.

A module is self-contained and controls zero or more devices that depend on each other to perform basic procedures. For example, a detection module can contain devices required to apply a readout signal to the system together with the input devices that read the state of the qubits during detection. To guarantee that modules are *independent* from a control perspective, a device can only be assigned to a single module. Each module has access to its own persistent data storage to store configuration and calibration data related to its operation. The collective behavior

of devices in a module can be described using module functions. For example, a detection module could have a function that controls the readout signal and the input devices in parallel to perform a detection procedure. Module functions are not solely used for collective device behavior and can also be used to manipulate devices separately, read or write configuration data, or update calibration parameters.

A system contains one or more modules that are organized in a tree structure. Every module in the system can contain zero or more sub-modules, and the root module is known as the *system module*. Parent modules can access features of all their child modules, allowing hierarchical and transparent structuring of devices and functionality. Because each device can only be assigned to a single module, modules in non-overlapping sub-trees of the system hierarchy are *device-independent*. Two independent modules can be controlled in parallel, which means they can both add events to the RTIO event timeline without device conflicts. Hence, the width of the module tree represents the amount of control- and device independence between different parts of the system. Device dependencies are encoded in the tree structure, and more independent modules lead to more available control parallelism in the system. All modules are added to the central registry of the system such they can be easily found later. Modules form the first level of system organization and introduce fundamental abstractions for device control and dependencies.

Modules introduce a straightforward device and system organization, but each separate module has limited power due to its local scope. Only the system module (i.e. the root module) can control all devices in the system, a requirement for most meaningful operations on the quantum system. With only modules, all system-wide functionality would have to be implemented in the system module, reducing the modularity of the software architecture. To overcome this issue, we introduce services as a technique to organize system-wide functionality.

A service is a component that can control multiple modules or even the whole

system if desired. Through the registry, a service can obtain any modules required for its functionality. A single module in the system can be accessed by any number of services. The functions of services usually describe the collective behavior of multiple modules, and functionality can vary from short operations to lengthy procedures. If modules are device-independent, a service is allowed to control them in parallel. Just as modules, services have access to their own persistent data storage, and every service is added to the central registry of the system.

Services are not limited to calling just modules. Through the registry, services can also find other services to use their functionality. Building services on top of other services allows transparent layering of increasingly complex system behavior. Hence, services are organized in a directed acyclic graph (DAG). Services contain powerful functions and enable the organization of system-wide functionality, but because of their system-wide control, services can not operate on the system in parallel with any other module or service as this could potentially lead to device-control conflicts.

3.3.2 Interfaces and Clients

System code can be organized with the concepts described in Section 3.3.1, but because most quantum systems are unique, the majority of modules and services are still developed and optimized for a specific system. To abstract system code, we introduce standardized interfaces. Interfaces describe a set of functions that must be implemented by a module or service. A single module or service can implement multiple interfaces and a single interface can be implemented by multiple modules or services in a system. For example, a gate interface could expose a set of functions that implement operations to perform quantum gates. Multiple instances of a gate interface within a single system could represent different gate implementations. To prevent device- and control conflicts, interfaces should be considered as implemented by a service. Hence, an interface can not operate on a system parallel with any other

module, service, or interface.

System code that implements one or more standardized interfaces can run portable experiments, which we call clients. Clients exclusively control a system through interfaces. A client is instantiated against a system, and the necessary interfaces will be obtained at runtime using the system registry. With clients, we can develop portable experiments that can run on different systems or different implementations of an interface in a single system. The system code functions as middleware between the generic experiment described in the client and the system-specific implementation of the utilized interfaces.

3.3.3 Implementation

We have implemented a software framework to support the development of real-time control software based on the presented concepts. The framework is part of our open-source library Duke ARTIQ extensions (DAX) [61] , which integrates tightly with the ARTIQ open-source software and hardware ecosystem. DAX implements a set of base classes that developers must inherit when defining modules and services. All these base classes provide direct access to data storage functions and the central registry of the system. When modules or services are instantiated, a unique hierarchical key and data storage location is assigned to the object based on the module tree or service DAG. In addition, the DAX library contains standard modules and services with portable functionality that can be used by any system. Such modules and services include functionality for processing measurement data, device-safety control, and common device control. Additionally, we have developed a generic class that defines the standard control flow of a single- or multi-dimensional scanning-type calibration experiment. Our DAX framework relies heavily on multiple inheritance to combine features of multiple classes, and fortunately, the Python host environment supports this well.

DAX defines various interfaces that can be implemented by modules and services. The two interfaces of interest for this paper are the *operation interface* and the *data-context interface*. The operation interface contains functions for common gate-level quantum operations, including single- and two-qubit Clifford operations, arbitrary rotation gates, and qubit state preparation/measurement. The data-context interface is used to store and process obtained measurement results. We developed clients to perform RB [62–64] and gate set tomography (GST) [65] which use the operation interface and the data-context interface to execute benchmark circuits. The RB and GST clients work with every system that uses DAX-based control software and implements the required interfaces. Both clients are based on the open-source pyGSTi library [66] which is used to generate benchmarking circuits and analyze results. Finally, we have defined an API that can be used to write portable quantum programs in an ARTIQ environment given an operation interface and a data-context interface. Using this API, we implemented a program to perform single-qubit state tomography (SQST) [67]. Such portable programs can be executed by dynamically linking the program to the interfaces of a system using the program loader client we developed.

3.4 Performance Evaluation

To evaluate the benefits and overhead of modular control software developed with the DAX framework, we re-implemented the control software for the STAQ system, an experimental trapped-ion quantum processor [24]. The real-time control hardware of STAQ is based on a Kasli 2.0 controller [51], which is part of the ARTIQ hardware ecosystem. The old ARTIQ control software for STAQ is designed with a system-specific and monolithic architecture while the new modular control software is developed using our DAX framework. In this section, we will compare the old control software with the new DAX-based control software.

The old STAQ control software separates system code from experiment code, but the system code has a monolithic architecture and is highly hardware dependent. For example, code related to RTIO timing is highly dependent on the latencies introduced by the programming of real-time devices. Any changes to the devices will likely cause RTIO timing constraints violations throughout the code. Hence, the old control software is not modular or portable. We did make modifications to the old control software to optimize its performance and make the comparison to the new control software fair. All delays inserted on the event timeline used to compensate for device programming times larger than 200 us were reduced to 200 us or replaced by other efficient solutions that satisfy timing constraints. Such delays are often necessary to not violate any timing constraints of the RTIO system. The new control software also uses 200 us delays to compensate for device programming time, which has been found empirically to be sufficient. Minor bugs found in the old code were also fixed to ensure the old and new experiments are functionally equivalent.

The new DAX-based system code for STAQ is modular and organized in 11 modules and 11 services. Most modules and services are system-specific, but two services use portable DAX data-processing modules, and one module extends a DAX module with safety-related functionality. The DAX-based system code implements various DAX interfaces, including the data-context interface and four implementations of the operation interface. The new system code packs more features and complexity, including control over more real-time devices (increased from 23 to 35) and external devices (increased from 4 to 8). Figure 3.4 shows a subset of the STAQ modules and services relevant for the microwave operation service, which implements the DAX operation interface. Solid arrows show the tree structure and DAG dependencies for modules and services, respectively. The dashed arrows indicate modules that are directly used by services. The microwave module controls a DDS to apply microwave

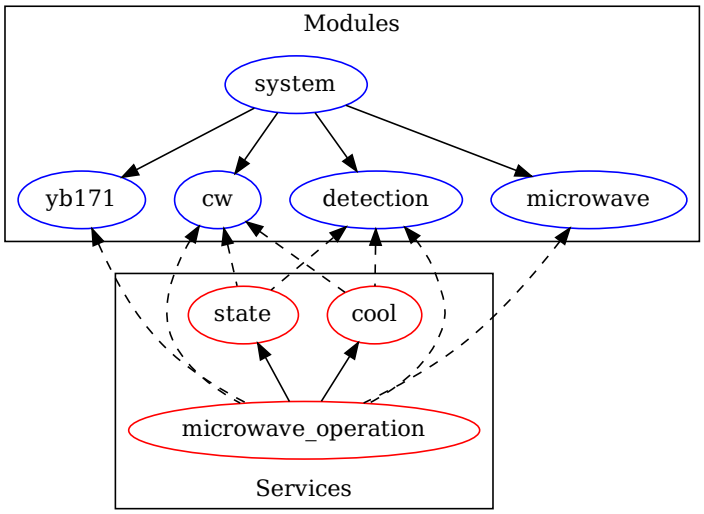


Figure 3.4: Subset of the STAQ modules and services relevant for the microwave operation service.

pulses to the ions. The photomultiplier tubes (PMTs) and lasers used for detection are controlled by the detection module. The continuous wave (CW) module controls various other lasers for cooling and pumping while the Yb171 module stores ion calibration data. The cool service contains various subroutines for cooling ions while the state service implements the data-context interface and is used for state initialization and detection. Finally, the microwave operation service uses all the mentioned modules and services to perform microwave gates, qubit state preparation, and measurements.

We chose five relevant experiments with a single real-time kernel available in both the new and the old STAQ control software for comparison. The selected experiments include two microwave (MW) experiments (MW freq/time), a qubit initialization experiment (qubit init), a tickle experiment (tickle), and an Ytterbium spectroscopy experiment (Yb spec). All experiments are one-dimensional (1D) scanning-type experiments and scan over 20 data points. The new control software utilizes the generic DAX scanning infrastructure while the old control software has defined scanning control-flow procedures as part of the system code. Each experiment

takes 100 samples per point except for Yb spec, which takes 30 samples per point. We ran each experiment with the same configuration using the old and new control software. Additionally, we run each experiment using the new control software with buffering enabled. Buffering allows the real-time control software to schedule the operations for the next samples while the incoming data of earlier samples are kept temporally in hardware buffers. ARTIQ supports such hardware buffers, but the real-time software must be designed appropriately to utilize them. Buffering can further increase the throughput and performance of kernels by reducing stalling time at the cost of increased latency between receiving and processing input events. None of the mentioned experiments are sensitive to the increased latency and will benefit from increased throughput. We configure a buffer size of 16 samples, which should be large enough to get the maximum performance gain achievable with buffering. The old control software does not include features for buffering. We measured the execution time of the kernel with nanosecond precision using the real-time clock available in the Kasli controller (i.e. the core device). An execution time measurement starts when the kernel starts execution, after the kernel binary is compiled and uploaded to the core device, and stops when the kernel finishes execution. Any RPCs from the core device to the host are included in the execution time measurement. We will use the execution time measurements to calculate the overhead of the real-time software. The kernel binary size is measured on the host at the output of the ARTIQ compiler and is used to calculate any binary-size overhead caused by our software framework. All our measurements are performed with ARTIQ version 6.7659.c6a7b8a8 and the results are presented in Figure 3.5 and 3.6.

3.4.1 Execution Time Overhead

The results in Figure 3.5 show the execution time overhead of the kernel for each experiment using the old and new DAX-based control software. For each experiment,

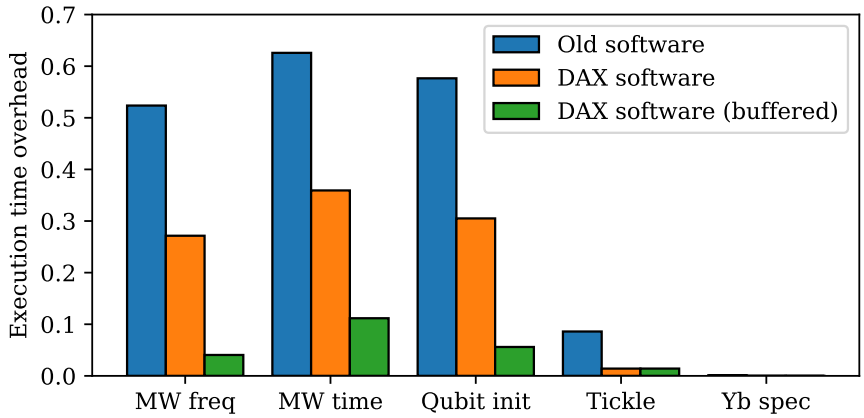


Figure 3.5: Kernel execution time overhead for the old control software and new DAX-based control software of the STAQ system.

we calculate the minimal execution time t_{min} based on the pulse lengths, detection times, and intentional wait times of the experiment. Given the measured execution time of an experiment t_{exe}, the execution time overhead is defined as $(t_{exe} - t_{min})/t_{min}$. Figure 3.5 shows that the old control software has an execution time overhead between 52.4% and 62.6% for the two MW and the qubit init experiments. These experiments consist of relatively short and quick operations which increase the operation density and induce more strain on the RTIO subsystem. Any inter-sample execution overhead introduced by the real-time control software will quickly increase the total execution overhead. The tickle experiment consists of slower operations resulting in a measured overhead of 8.6% for the old control software. Any software overhead will be less significant due to the longer total duration of the experiment. The Yb spec experiment has very slow operations and includes a 500 ms wait time for each sample. Any overhead introduced by the real-time control software will be negligible on the timescale of the experiment.

If we look at the results of the DAX-based control software (without buffering) in Figure 3.5, we see that the new control software significantly reduces the execution time overhead compared to the old control software. On average, the new control software reduces the execution time overhead by 63.3% compared to the old

control software. This average overhead reduction also includes the Yb spec experiment which already has a negligible execution time overhead for the old control software. When not including the Yb spec experiment, the average execution time overhead reduction is 55.4%. When we include buffering, we see that experiments with short and quick operations benefit the most (i.e. the two MW and the qubit init experiments). Buffering reduces inter-sample overhead by scheduling multiple samples ahead before retrieving results. Hence, the experiments most affected by inter-sample overhead benefit the most from buffering. The execution time overhead of the tickle experiment is not further reduced by buffering. The new control software already reduced its overhead to 1.4%, and inter-sample overhead does not appear to be a significant part of that. Compared to the old control software, the DAX-based control software with buffering enabled reduces execution time overhead by 88.7% and 87.1% on average with and without the Yb spec experiment, respectively.

We further analyzed our measurements to understand why the DAX-based control software performs better than the old control software. We attribute the reduced overhead to two main sources: timing management and data offloading. As mentioned earlier, real-time control software often inserts some delays on the event timeline to compensate for device programming times. The new control software groups devices in modules which in turn provides functions to manipulate those devices. The inserted delays can be optimized for each function which reduces the overhead. The old control software is less structured which often leads to larger worst-case delays or redundant delays to be inserted. Modular and well-designed real-time software allows us to insert more fine-grained delays, which reduces the total execution time overhead. Modular software design also leads to code that is more flexible and robust to changes. When a module has any modifications to its real-time devices or their behavior, its function might need to be optimized again, but other modules and services are not affected by the change. If devices in a module completely change, a

module might need to be redeveloped. Fortunately, if the function signatures of the new module are compatible with the old one, the modules could be swapped without affecting other parts of the system.

The second major contributor to overhead reduction is data offloading. Measurement data for an experiment is often offloaded to the host using asynchronous RPCs while the kernel is running. Such offloading can be very efficient and transfers parts of computational tasks from the kernel to the host while also reducing memory usage on the core device. The new control software uses portable DAX data-processing modules which are highly optimized to maximize the benefit of the data offloading. As a result, the complexity and execution time overhead of the kernel is reduced.

We would like to mention that better timing management and data offloading is also achievable with monolithic control software, but modular software makes it much easier. Devices will always be addressed through the functions of the module it is part of, making it easy to optimize the inserted delays for each scenario and improve timing management. For data offloading, the DAX data-processing module is portable between systems (see Section 3.5) and only has to be developed and optimized once thanks to the modular software architecture.

3.4.2 Kernel Binary Size

The results in Figure 3.6 show the kernel binary size of the new control software normalized to the kernel size of the old control software. We see that for the two MW experiments the kernel size is increased by 43.0% to 43.8% while for the other experiments the kernel size changed less than 1%. While all experiments are 1D scanning-type experiments, the two MW experiments merged into a single two-dimensional (2D) scanning experiment in the new control software. For our tests, we configured one dimension to be static to reduce the experiment to a 1D scan. While this will result in a functional 1D scan, the DAX scanning infrastructure still stores data for

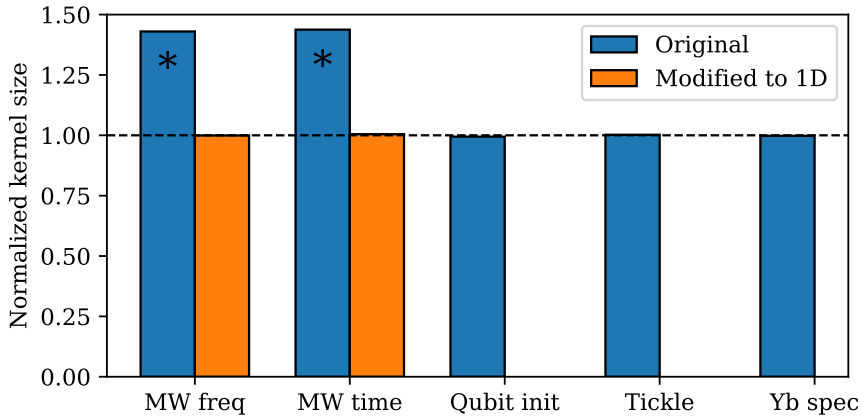


Figure 3.6: Kernel binary size of the new control software normalized to the kernel binary size of the old control software.
∗ In the new control software, the two microwave (MW) scan experiments merged into a single 2D scan experiment causing an increased kernel size.

the static dimension for each point in the scan. Hence, the kernel binary size increases. We manually modified the new MW experiment to a 1D scan for frequency and time storing the fixed value of the other dimension as a constant. When we measure the kernel binary size again, the difference with the old control software is less than 1%. From our results, we can conclude that the ARTIQ compiler works well with modular control software, and modular real-time software does not cause extra overhead that increases the kernel binary size.

3.5 Code Portability

A principal benefit of modular control software is the potential for code portability between different quantum systems. The ARTIQ ecosystem already successfully abstracts real-time hardware with drivers and gateware to hide differences between hardware configurations or even hardware platforms. The DAX framework tries to achieve portability and abstraction on a higher level, more specifically the system-level and application-level software. On the system-level, generic DAX modules, services, and scanning infrastructure (see Section 3.3.3) allow portability of real-time

control code between systems. Portability for application-level software is achieved by using interfaces and clients.

To evaluate the amount of code portability between two different systems, we have implemented DAX-based control software for a second experimental trapped-ion quantum system known as the RC system [22]. The real-time control hardware of RC is based on a KC705 [68] evaluation board with custom breakout boards that contain digital I/O and DDS devices. The control software for the RC system consists of 20 modules and 7 services. Two services use portable DAX data-processing modules and one module extends the DAX module for safety-related functionality. The RC system code implements multiple DAX interfaces, including the data-context interface and two implementations of the operation interface. The system code controls 30 real-time devices and 1 external device. Notable is that the RC system has more modules than the STAQ system even though there are fewer real-time devices. The software of the RC system was developed after that of the STAQ system, and we learned it was better for modularity and portability to have a deeper system tree with more and smaller modules.

Figure 3.7 shows a subset of the RC modules and services relevant for the microwave operation service, which implements the DAX operation interface. The graph looks very similar to the one for the STAQ system shown in Figure 3.4 despite the real-time devices and controlled hardware being significantly different. The Yb171 and microwave modules are similar to their STAQ counterparts while the main differences are found in the CW and PMT modules. One key difference between the systems is that the detection laser shares an upstream master switch with other continuous wave lasers. Due to the master switch, it is impossible to control the detection laser independently from other lasers in the CW module without potential conflicts. Hence, the detection laser is controlled by the CW module and the PMTs are contained in an independent module. A detection subroutine now

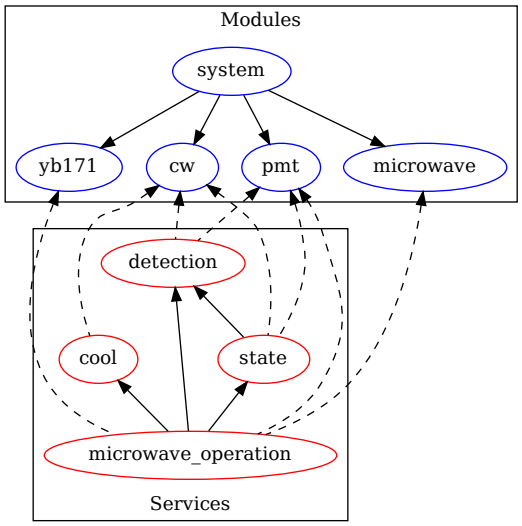


Figure 3.7: Subset of the RC modules and services relevant for the microwave operation service.

requires the CW and PMT module to work in parallel which is captured in the detection service. The remaining services in the RC system are similar to their STAQ equivalents. Figure 3.4 and 3.7 show that two systems with significantly different real-time control systems and devices can still have real-time control software with similar architectures. Modules and services can successfully abstract such differences.

To evaluate code portability between the STAQ and RC system, we disabled modules and services not relevant for microwave operations. We then run a set of six experiments on each system. Three are MW calibration experiments and include a MW frequency calibration, a MW Ramsey frequency calibration, and a MW gate experiment that executes a sequence of X rotations to fine-tune the microwave Rabi gate time. These calibration experiments are hardware-specific and therefore have system-specific implementations. Two other experiments are the DAX clients for RB and GST that use the operation interface and the data-context interface. The last experiment is the portable SQST quantum program that is dynamically linked to the system using the program loader client. The mentioned clients and portable experiments are described in Section 3.3.3.

All ARTIQ experiments have four execution phases: build, prepare, run, and analyze. The build phase is used to instantiate objects and process arguments. The prepare phase is the first moment where code directly relevant to the experiment can execute. This phase allows experiments to execute code on the host without accessing any devices or data storage. The run phase is the only moment where the experiment has access to devices and data storage. Kernels can only execute in the run phase of the experiment and any data analysis for calibration purposes should also be done here. Finally, the analysis phase is used for the post-experiment analysis of data. The run, prepare, and analyze phases are separated to pipeline experiments and maximize usage of the real-time control system. Our code portability evaluation focuses on the prepare and run phase, which are the two phases directly relevant to the functionality of the experiment. We decided not to add the build and analysis phase to not give ourselves a potentially unfair advantage by including more code in the coverage analysis.

For our code portability evaluation, we will run the six mentioned experiments on both systems while keeping track of the statement coverage of the prepare and run phase for each experiment. Statement coverage is a technique often used for testing and keeps track of code statements evaluated at least once during program execution. The resulting data gives insight into the quantity of code that is used during program execution and does not provide information about the execution time spent for each statement. We measure coverage by simulating our kernel code using the DAX simulator [69] in conjunction with Coverage.py [70]. Our statement coverage data includes statements executed as part of host code, kernels, and RPCs. For our analysis, we are interested in the coverage of four categories of code: experiment code, system code, DAX library code, and *application code*. The first two categories are already defined in Section 3.3 and the third category is self-explanatory. We define application code as high-level and portable code that extends or utilizes the

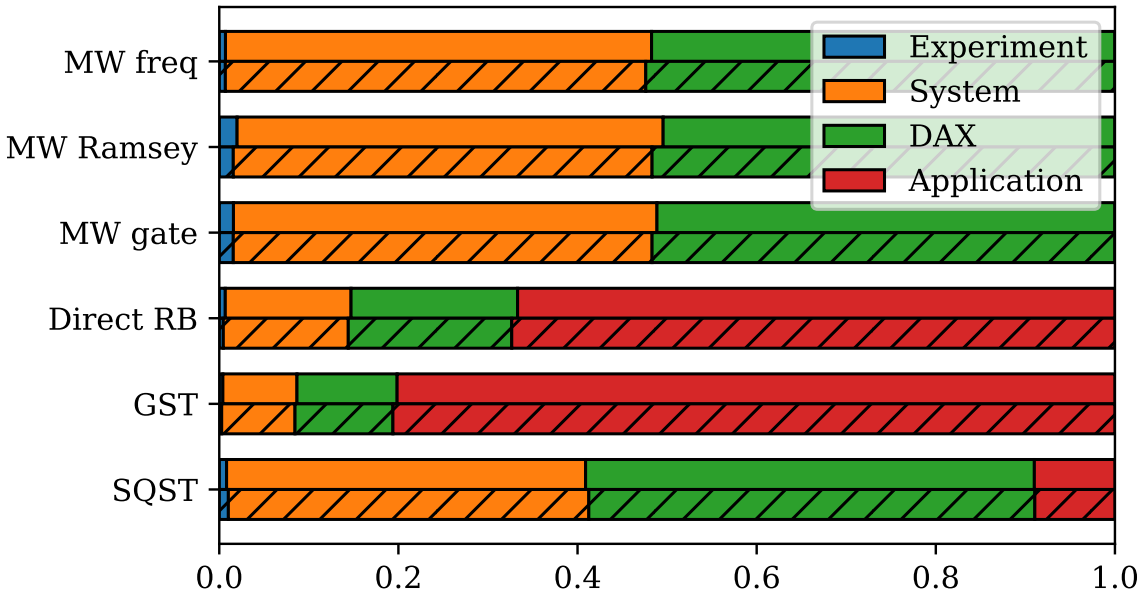


Figure 3.8: Categorized and normalized proportions of covered statements for the STAQ (solid bars) and RC (hatched bars) control software.

real-time control software to achieve its functionality. For the experiments we chose, application code includes the pyGSTi library and the SQST program. Coverage in other supporting libraries, such as ARTIQ or the standard library, is not included in this analysis. The coverage results are shown in Figure 3.8.

The results in Figure 3.8 show that for all experiments, the proportions of code in each category do not differ much between the STAQ and RC system. What can not be seen from the figure is that the total number of statements covered for each experiment does not differ more than 1.8% between the two systems. Figure 3.8 shows that the three MW calibration experiments have very similar results with 2.0% or less experiment code, between 46.8% and 47.6% system code, and 50.4% to 52.4% of DAX code. The covered statements of DAX library can mainly be found in its data-processing module, scanning infrastructure, and system initialization-related code. The Direct RB and GST experiments both have a large proportion of application code that covers parts of the pyGSTi library. These are procedures used to generate the benchmarking circuits. Both experiments also use pyGSTi for measurement data

analysis, but those procedures are not included in our coverage data because they are part of the analysis phase of the experiment. For the remaining portion of covered statements for the Direct RB and GST experiments, more than half is DAX library code which includes the code of the client itself and the data processing module. Finally, the SQST program contains 9.0% and 8.9% application code, which is the portable SQST code itself, for the STAQ and RC system, respectively. The DAX library code mainly includes statements from the program loader client and the data-processing module in addition to the initialization code used by the loader to create and dynamically link the portable program to the system.

The results in Figure 3.8 show that with a modular software architecture, large portions of covered statements do not have to be system-specific and can be shared as application code or as part of a shared library for system code, such as DAX. For each unique quantum system, only the experiment code and system code would have to be developed which would significantly reduce the development time. For the code that does need to be developed, most of it is part of the system code which is shared between experiments for a single system and reduces development time even further.

Only covered statements in the DAX and application categories in Figure 3.8 are potentially portable between the STAQ and RC systems. We took the coverage data for each experiment and compared how many statements in the DAX and application categories were covered by both systems. These are the statements that are directly shared between the two systems. The results, which are normalized to the total number of covered statements for STAQ, are shown in Figure 3.9.

The results in Figure 3.9 show that even the system-specific MW calibration experiments consists of 49.8% to 51.7% of shared statements. Data-processing modules and scanning infrastructure add a significant number of covered statements during an experiment and are relatively easy to make portable. By sharing portable modules and services, we achieve portability on the system-code level. The Direct RB

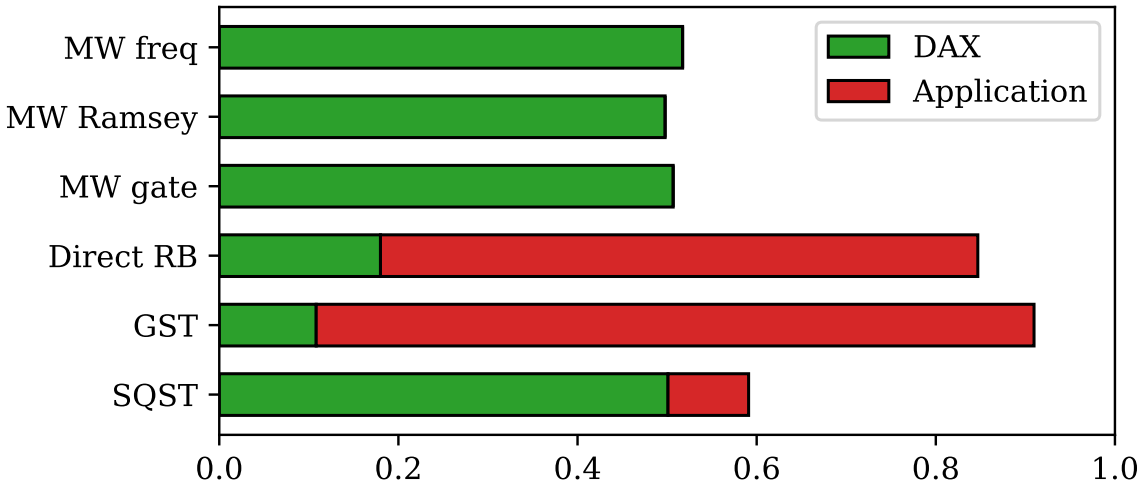


Figure 3.9: Categorized and normalized proportions of covered statements from STAQ that are shared with RC.

and GST experiments consist of 84.7% and 91.0% of shared statements, respectively. The application code is a major contributor to the proportion of shared statements but portable system code also contributes a significant part. Application code on the quantum operation level is inherently portable and we show that by introducing interfaces and clients, we can successfully connect to application-level software. Finally, the SQST program contains 59.1% of shared statements of which 9.0% is application code. The SQST program is small and therefore does not contribute a lot of covered statements. The remaining shared statements all originate from the system code.

To further increase the amount of shared code between the STAQ and RC systems, we could generalize more modules and services. For example, the microwave module, the state service, and the microwave operation service of both systems contain very similar code and could probably be converted to a portable DAX module or service. So far, we have not done that in favor of flexibility and code simplicity. A module or service part of the system code can easily be modified for testing or mitigating device-related issues. Especially in an academic setting, flexibility can

sometimes be more important than code portability. Additionally, portable modules and services are often required to have many configuration and customization capabilities to function correctly for different systems. Hence, portable code is often more complex which might not always be desired. Instead, we keep such modules and services part of the system code and manually "port" them to new systems. Overall development time can still be reduced by porting modules and services while full flexibility is preserved.

3.6 Experiments

To demonstrate the capabilities of modular DAX-based control software and the portability of clients, we used the RB client presented in Section 3.3.3 to perform benchmarking on two experimental quantum systems. The RB client uses the operation interface which is implemented on both systems by their respective MW operation services. These services utilize a microwave horn to excite a dipole transition between the hyperfine states of Ytterbium 171 ($^{171}Yb^{+}$), which is where the qubit is encoded. This performs X and Y rotations on the Bloch Sphere. To perform cooling, state preparation, and measurement, the two systems use a 370nm laser to excite the dipole transition between $^{2}S_{1/2}$ and $^{2}P_{1/2}$ [20].

Before performing coherent operations, very accurate knowledge of the hyperfine frequency difference between the qubit states is needed, along with the Rabi frequency corresponding to oscillations between these states. The hyperfine splitting between the states is very well known [71]. However, a strong magnetic field is installed in our systems, slightly altering this value. Assuming we have some knowledge of what the frequency change should approximately be, three calibration experiments are still needed. The first experiment performs a sequence of timed microwave pulses near the qubit transition frequency to get Rabi oscillations. We then fit this data to get a rough estimate of the Rabi frequency. The second experiment uses Ramsey

interferometry to fine-tune the qubit transition frequency. This experiment can be done with incrementally smaller frequency ranges to get greater precision of the resonance. Lastly, we perform increasingly longer sequences of π-pulse rotations designed to end with the qubit in the ground state in order to fine-tune the Rabi frequency.

After calibration, we perform Direct RB, with circuit lengths starting at 1 and scaling up exponentially to 1024. Direct RB is a modification of the original RB proposal which implements randomized circuits by sampling a system's native gates from a user-provided distribution Ω [63]. For microwave gates on ion trapping systems, the native gate sets are X and Y rotations, and we chose Ω to be uniform. The sequences are provided by the pyGSTi library [66] using the DAX RB client. For each circuit length, we performed 10 different circuits with 100 samples for each. The circuits were designed such that output was randomized to avoid skewed data because of bias toward a particular outcome. For example, the detection process in this experimental setup is designed such that the ground state is dark when shining the 370nm laser on the ion. Thus, losing the ion during an experiment would lead to always measuring the ground state.

To demonstrate the flexibility of DAX and the portability of the RB client, we perform Direct RB with two different experimental setups: the STAQ and RC systems. Besides the different real-time control systems and devices, the main difference between these two setups is that STAQ is at cryogenic temperatures while RC is at room temperature. However, this shouldn't have any drastic effect on microwave operations and the data between the two systems should be quite comparable. The results from this experiment can be found in Figure 3.10. Here, the error per gate r is estimated to be $r = 4(1-p)/3 = 1.45 \times 10^{-4} \pm 2.58 \times 10^{-5}$ for the STAQ system and $r = 2.28 \times 10^{-4} \pm 1.94 \times 10^{-5}$ for the RC system, where p is calculated from fitting to the function $P(m) = 0.5 + Bp^m$. This number can also be interpreted as $1 - F_g$, where F_g is the average gate fidelity of the system. The difference in errors at

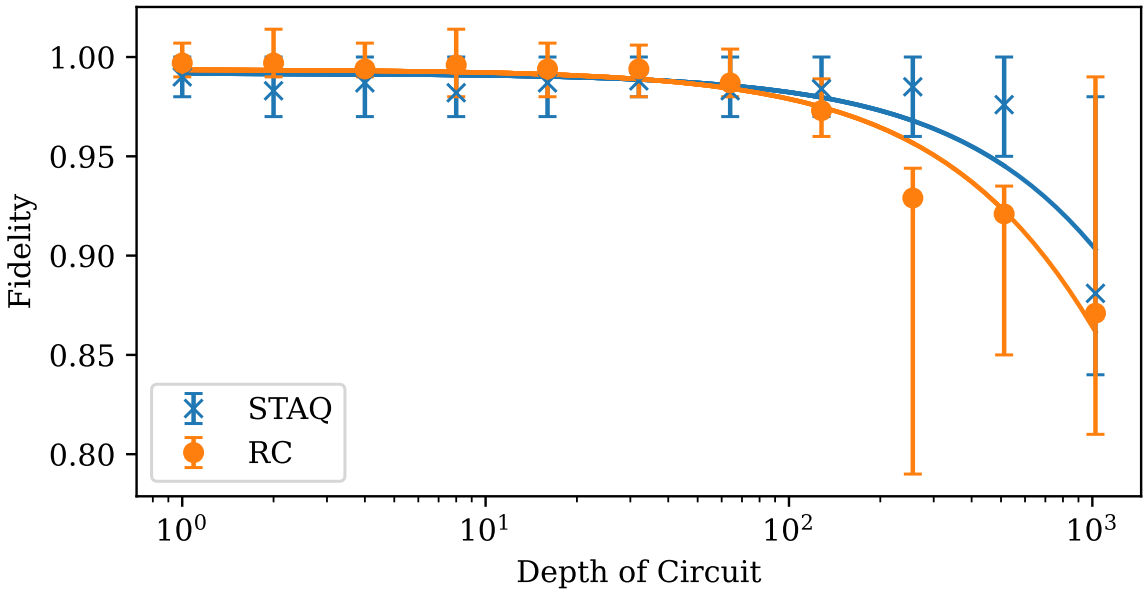


Figure 3.10: Single-qubit Direct randomized benchmarking fidelity results for the STAQ and RC system using microwave gates. Error bars are calculated using the 10th to 90th percentile as boundaries. RC starts at higher fidelity due to better SPAM but decays quicker due to lower gate fidelity.

low circuit depth is simply a result of different state preparation and measurement (SPAM) error, while the STAQ system can be seen to overtake RC at higher circuit depth due to better gate calibration.

3.7 Conclusion

We have presented a systematic design strategy and a modular architecture for real-time quantum control software that organizes devices and system-wide functionality in modules and services, respectively. Our architecture supports the development of modular control software and enhances the flexibility and portability of real-time control software. We implemented a software framework to develop real-time control software based on our proposed architecture, which is part of our open-source library DAX. Our evaluation shows that modular control software can reduce the execution time overhead of kernels by 63.3% on average while not increasing the binary size. Software portability is achieved on the system level by introducing portable modules,

services, and scanning control flow. We achieve application-level portability using interfaces and clients. Our analysis shows that modular control software for two distinctly different systems can share between 49.8% and 91.0% of covered code statements. Finally, we have shown that we can run a portable Direct RB experiment on two different ion-trap quantum systems that are fully controlled and calibrated by software based on our framework.

4

Universal Graph-Based Scheduling for Quantum Systems

This chapter is based on the following paper: L. Riesebos, B. Bondurant, and K. R. Brown, "Universal graph-based scheduling for quantum systems," *IEEE Micro*, vol. 41, no. 5, pp. 57–65, 2021. DOI: 10.1109/MM.2021.3094968. It also appears in Leon Riesebos' thesis: L. Riesebos, "Software architectures for real-time quantum control systems," Ph.D. dissertation, Duke University, 2022.

My primary contribution was the implementation and benchmarking of the Optimus algorithm as described in Sections 4.4, 4.5, and 4.6

4.1 Introduction

Operations on quantum systems are realized by applying analog signals to various components of the system containing the quantum bits (qubits). To perform useful computations with such systems, it is critical that analog signals are precisely tuned to ensure high fidelity operations on the qubits [40, 73, 74]. The calibration of all different operations often consists of running complex series of dependent experiments

in a specific order. Full system calibration can require over 40 calibration experiments and take up to tens of hours to complete (see supplementary information of [40]), and the complexity of this time-consuming process will further increase when scaling the number of qubits in the system. Due to the analog nature of the operations, optimal control parameters drift over time, requiring continuous monitoring and frequent re-calibration of the system. To keep quantum systems continuously operational we need automated background processes that can be simple periodic tasks such as system monitoring as well as complex sets of dependent experiments for system calibration.

In this paper, we present an open-source scheduling toolkit that can manage generic background processes for quantum systems. 1. Our scheduling toolkit schedules experiments based on a directed acyclic graph (DAG) using a configurable traversal algorithm. 2. We introduce triggers to support universal feedback for our scheduler. 3. We implement a complex system calibration algorithm based on our scheduling infrastructure to demonstrate the capabilities of our system.

Our paper is organized as follows. We first introduce the execution model of our system in Section 4.2. In Section 4.3 we present the scheduling graph and the traversal algorithm that is at the core of our scheduling toolkit. The implementation of the Optimus scheduling algorithm for system calibration is outlined in Section 4.4. The implementation of our scheduling toolkit is discussed in Section 4.5 and our simulation results are presented in Section 4.6. We conclude our paper in Section 4.7.

4.2 Execution Model

Our goal is to make a practical scheduling toolkit and therefore we need to understand the execution model of our quantum system. We envision the quantum system to work according to the *accelerator model* as described in [34, 49, 55, 58, 75]. In such a model, quantum programs are considered hybrid programs that consist of a classical

host program combined with one or more quantum kernels that can be mapped to a quantum co-processor. Note that experiments or calibrations are quantum programs in the context of the accelerator model and the terms can be used interchangeably. An analysis of the accelerator model itself is outside the scope of this paper and will not be further discussed.

We aim to run the scheduling algorithm as a classical host-only process that submits experiments asynchronously to a *pipeline*. Experiments submitted to the pipeline are sequentially executed by a separate process which allows them to have exclusive access over the quantum co-processor. The pipeline functions as a priority queue where higher priority experiments run first and experiments with equal priority run in the order in which they were submitted.

4.3 Scheduling Graph

The calibration of a quantum system consists of a set of dependent calibration experiments. Consider a universal gate set consisting of H, T, and CNOT on n qubits that are fully connected. There are $n^2 + n$ gates that need to be tuned-up. Experimental quantum information scientists do not first directly tune these gates, but instead tune the underlying electromagnetic signals that drive these gates. Additionally, the gates themselves cannot be fully calibrated in isolation, and benchmarking experiments are needed to check that the gates perform well together. A common structure is to first carefully calibrate the radio frequency signals, then optimize individual gates, and finally test with a high-level calibration. Such a set of dependent *jobs*, where jobs represent calibration experiments or other periodic tasks, can be represented by a graph where the jobs are nodes and their dependencies are directed edges. Based on such a graph, which we will call a *scheduling graph*, it is possible to derive an execution schedule for the jobs. Circular dependencies are not allowed because they lead to impossible schedules. Hence, the scheduling graph is a DAG.

Table 4.1: Job specification and state.

Name	Type	Description
Experiment	Spec	The experiment to submit
Arguments	Spec	Experiment arguments
Interval	Spec	Submit interval time (optional)
Dependencies	Spec	Set of nodes (optional)
Last submit time	State	The last submit time

DAGs have been used previously to schedule single-qubit calibrations [76] — here we extend that to multi-qubit and global calibrations as well as any other periodic tasks. A scheduling graph does not need to be weakly connected and can therefore contain multiple components. To schedule jobs, we will traverse over the scheduling graph and submit the experiments represented by the jobs to the pipeline. In the remainder of this section, we will lay out the components of the scheduling graph and the traversal algorithm.

4.3.1 Jobs

A job is a node in the scheduling graph that represents a specific experiment with a set of fixed arguments. Additionally, a job specification optionally contains a submit interval time and a set of dependencies. Every node in the graph has its persistent state and stores the last time it was submitted. A list of job specs and states is shown in Table 4.1

It is possible to perform a few functions on jobs. If a job is submitted, its specified experiment will be submitted to the pipeline, the last submit time of the job state will be updated, and the function returns. Our scheduling process *visits* a node while traversing the graph, which will return a *node action*. A node action is an enumeration and if a node has expired (i.e. the current time minus the last submit time is greater than the submit interval), visiting that node will return node

Table 4.2: Node action enumeration.

Action	Description
`PASS`	Pass node
`RUN`	Run node
`FORCE`	Force to run node

action `RUN`. If a node is not expired or has no submit interval, node action `PASS` will be returned. At the start of the graph traversal it can be useful to force a run irrespective of the node state. This third node action `FORCE`, will be discussed in Section 4.3.2. All node actions are listed in Table 4.2.

4.3.2 Wave Algorithm

The *wave algorithm* is a recursive algorithm used to traverse over the scheduling graph depth-first. The goal is to have a simple yet highly configurable algorithm that can be used to traverse over parts of the scheduling graph. Regardless of its configuration, the wave algorithm will always submit visited nodes, if any, in an order that satisfies the dependencies of the scheduling graph.

The wave algorithm uses node actions and a *scheduling policy* to determine if nodes need to be submitted or not. A scheduling policy is a map from two node actions to a single node action. During the graph traversal, the scheduling policy determines how the state of a node influences its dependants. We define two scheduling policies, `LAZY` and `GREEDY`. The `LAZY` scheduling policy will only submit expired and forced nodes while the `GREEDY` policy will additionally submit the dependencies of those nodes. The complete definitions of the scheduling policies are shown in Table 4.3.

Given a scheduling graph, the wave algorithm can be configured by providing a root node (or nodes), a root action, and a scheduling policy. The wave algorithm

Table 4.3: Maps of the scheduling policies. Entries in parenthesis are not reachable and therefore undefined, but included for completeness.

Previous, current action	`LAZY` policy	`GREEDY` policy
`PASS`, `PASS`	`PASS`	`PASS`
`PASS`, `RUN`	`RUN`	`RUN`
(`PASS`, `FORCE`)	-	-
`RUN`, `PASS`	`PASS`	`RUN`
`RUN`, `RUN`	`RUN`	`RUN`
(`RUN`, `FORCE`)	-	-
`FORCE`, `PASS`	`RUN`	`RUN`
`FORCE`, `RUN`	`RUN`	`RUN`
(`FORCE`, `FORCE`)	-	-

visits the current (root) node to obtain the current node action. Using the provided scheduling policy, the previous (root) action and current action are mapped to a new action. The wave algorithm recursively calls the dependents of the current node while passing the new action. When all dependents are visited, the current node will be submitted if the new action for the node equals `RUN`. The algorithm keeps a set of submitted nodes to make sure every node is not submitted more than once during a traversal. A single run of the wave algorithm is an atomic operation that will update the state of the nodes and submits zero or more experiments to the pipeline. The basic wave algorithm is shown in Listing 4.1.

For a given scheduling graph, different wave configurations lead to different results based on the current state of the nodes. Figure 4.1 shows a simple scheduling graph with four nodes. Assuming job C is expired, the submitted jobs for various wave configurations are shown in Table 4.4. We can see that node action `FORCE` can be used as a root action to force one or more nodes to be submitted.

Configuration options for the wave algorithm not shown in Listing 4.1 are *depth*, *start depth*, and *priority*. The depth parameter is an integer that can limit the

```python
def wave(node, prev_action,
         policy, submitted=None):
    """
    :param node: The current node
    :param prev_action: Previous node action
    :param policy: The scheduling policy
    :param submitted: Set of submitted nodes
    """
    if submitted is None:
        submitted = set()

    # Visit node to obtain current action
    curr_action = node.visit()
    # Compute new action based on policy
    new_action = policy(prev_action,
                        curr_action)

    # Recursion
    for n in node.dependents():
        submitted = wave(n, new_action,
                         policy, submitted)

    if new_action is NodeAction.RUN:
        if node not in submitted:
            # Submit node
            node.submit()
            submitted.add(node)

    return submitted
```

Listing 4.1: The basic wave algorithm.

Table 4.4: Submitted jobs for different wave configurations for the scheduling graph shown in Figure 4.1 with job C expired.

Root node	Root action	Policy	Submitted jobs
Job A	PASS / RUN	LAZY	C
Job A	FORCE	LAZY	C, A
Job A	PASS	GREEDY	D, C
Job A	RUN / FORCE	GREEDY	D, C, B, A

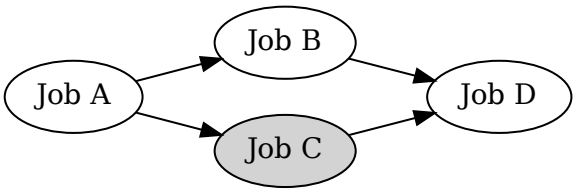


Figure 4.1: A simple scheduling graph with four jobs. For our example, we assume that job C is expired.

recursion depth of the wave algorithm. With the start depth parameter, it is possible to only visit and submit nodes beyond a given recursion depth. By default, nodes are visited and submitted starting from depth zero and the recursion depth is not limited. The priority parameter is passed to the pipeline as the experiment priority when submitting a job. A default priority can be configured in the scheduler, and if a trigger is submitted without the specification of a priority, the default will be used.

4.3.3 Scheduler Process

As mentioned in Section 4.2, we envision running our scheduler as a host process. At startup, the scheduler is given a scheduling graph and a *trigger queue* will be created. Any process on the host can submit triggers to this queue through the scheduler process where a trigger is defined as a tuple containing a list of root nodes, a root action, and a scheduling policy. The scheduler process waits for a trigger to be submitted after which it will read the request and run the wave algorithm using the parameters given by the trigger. Note that a running experiment is also a host process that can trigger the scheduler and an experiment could for example request a full system calibration on demand by triggering the scheduler.

When time passes, nodes will expire, and to detect such events, the scheduler will periodically submit pre-configured triggers to its queue. The desired system for background tasks is realized by these periodic triggers.

4.3.4 Universal Feedback

Some scheduling scenarios can not be conveniently expressed by a dependency graph and require universal classical logic to determine how the scheduler should proceed. For example, based on the results of job J, sub-graph G_A or G_B needs to be evaluated by the scheduler. Such scenarios can still be part of our scheduling infrastructure by using the schedulers' trigger feature to feedback a decision to the scheduler. We can create an experiment for job J which decides based on its results what sub-graph needs to be evaluated next. Once the decision is made, the experiment submits a trigger to the scheduler with the root nodes of the sub-graph that needs to be evaluated. When the scheduler runs job J it will receive feedback from the experiment through a trigger and the scheduler will proceed by evaluating the appropriate sub-graph.

4.4 Optimus Scheduling Algorithm

As mentioned earlier, a use case of particular interest for the scheduler is that of automated system calibration. In [76], Kelly et al. propose the "Optimus" algorithm for the intelligent calibration of qubit systems. Much like the approach we have outlined so far, they use a DAG to represent the calibration experiments and their dependencies. In this section, we will explain how the Optimus algorithm can be implemented using our scheduling infrastructure.

4.4.1 Algorithm Summary

The goal of the Optimus algorithm is to maintain accurate system control parameters while spending as little time as possible on calibrations. They achieve this goal by introducing three levels of interaction with each calibration in the graph: `check_state`, `check_data`, and `calibrate`. `check_state` relies solely on the specification of a timeout period (roughly corresponding to the drift timescale of the

parameter in question) and previous system knowledge — namely the last time the parameter in question was calibrated or verified by `check_data` and the last time any dependencies were calibrated. Based on that knowledge, `check_state` will either report a pass or a failure. `check_data` is intended to be a minimal experiment to determine whether a parameter value is still valid — referred to as *in-spec* or *out-of-spec*. If `check_state` reports a failure, the `check_data` experiment will run and report either in-spec, out-of-spec, or *bad-data* (explained in further detail below).

Each level of interaction is increasingly time-consuming and is designed to execute only if the previous level fails. If `check_data` reports out-of-spec, then `calibrate` will run the full calibration and update the value of the parameter in question. The primary graph traversal routine used by Optimus (dubbed `maintain` by the authors) is a simple greedy, depth-first traversal. While the calibration interactions are executed lazily, the `maintain` traversal is greedy in the sense that each calibration job must be submitted in order for `check_state` to run. There is also a secondary traversal routine called `diagnose`, which only runs in the special case that `check_data` reports bad-data. In that case, the assumption is that some part of our knowledge of the system is inaccurate. So, `diagnose` triggers a traversal of the sub-graph containing the immediate dependencies of the calibration in question, in which `check_state` is skipped and each calibration is forced to run (at least) `check_data`. If any of the dependencies in question also report bad-data, then another `diagnose` sub-graph traversal will be recursively triggered on that node. The execution model for a single calibration is illustrated in Figure 4.2.

4.4.2 Algorithm Implementation

To integrate the Optimus algorithm into our scheduling toolkit, we introduce a new type of job called a *calibration job*. The implementation of the calibration job builds upon the scheduling infrastructure presented in Section 4.3 and adds the traversal

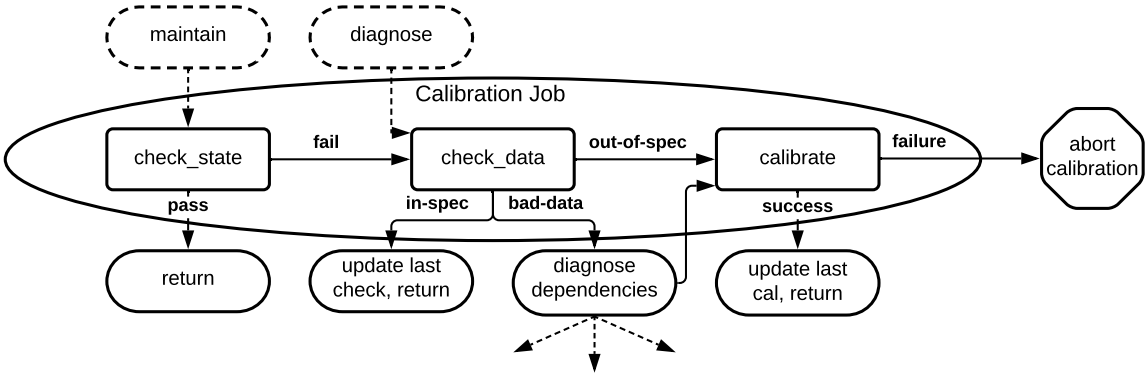


Figure 4.2: A state machine representation of the behavior of a single calibration in our implementation of the Optimus algorithm. The dashed lines represent entry and exit points for the different triggers. Returning from a calibration continues the current wave.

logic specific to the Optimus algorithm. While a job represents a single experiment to be run, a calibration job represents two experiments — one for `check_data` and one for `calibrate`. The `check_state` logic is incorporated into the calibration job definition via the specification of a timeout. The full list of calibration job specs and states is shown in Table 4.5.

In our framework, the `maintain` traversal is initiated by a periodic trigger and the `diagnose` traversals are initiated by triggers submitted by any calibration jobs that report bad-data from the `check_data` experiment. The root nodes of the `maintain` trigger are all calibrations with no incoming dependencies, and the trigger has a `GREEDY` scheduling policy with root action `FORCE` to ensure that the `check_state` logic is executed for every calibration. The `diagnose` triggers are `GREEDY` (root action `FORCE`), with the root node specified as the calibration job representing the current experiment, and a depth and start depth of one so that only the immediate dependents are submitted. Additionally, it has a priority that is one level higher than the priority of the current experiment to ensure that the diagnose wave takes precedence over previously submitted experiments.

In implementing the Optimus algorithm, we also had to address a question

Table 4.5: Calibration job specification and state.

Name	Type	Description
Check experiment	Spec	`check_data` experiment
Calibration experiment	Spec	`calibrate` experiment
Check arguments	Spec	Check experiment arguments
Calibration arguments	Spec	Calibration experiment arguments
Calibration timeout	Spec	Calibration interval (optional)
Dependencies	Spec	Set of nodes (optional)
Last calibration time	State	The last successful calibration
Last check time	State	The last successful check

that was only partially addressed in [76]: what do we want to do if the algorithm fails to successfully calibrate a parameter? In [76], the authors mention that they raise a "`DiagnoseError`" in the case that a `diagnose` wave is triggered without resolution. We take a slightly different approach and introduce a generic `FailedCalibrationError`, which is to be raised whenever a calibration runs but fails to achieve the desired accuracy. Then we execute the `diagnose` waves blindly, without checking whether or not any dependencies were re-calibrated as a result. After the `diagnose` wave, we simply run the calibration and allow it to pass or fail independently of the result of `diagnose`. If a calibration does fail for any reason, we assume that external intervention is required and halt all execution indefinitely until it is manually resumed.

4.5 Implementation

Our scheduling toolkit is implemented in Python as part of our open-source library Duke ARTIQ extensions (DAX) [61] which is tightly integrated with the ARTIQ open-source software and hardware ecosystem [13, 51]. The ARTIQ control infrastructure provides real-time control of our quantum systems and is used by dozens of

research groups with over 200 systems deployed worldwide. Our software can schedule any ARTIQ experiment and is therefore fully compatible with existing systems that use ARTIQ.

The DAX scheduling toolkit allows users to define jobs and calibration jobs as classes that inherit the DAX `Job` and `CalibrationJob` base classes respectively. The job specifications shown in Table 4.1 and 4.5 are provided as class variables. The Optimus algorithm is implemented as part of the `CalibrationJob` class and the implementation builds upon the implementation of the `Job` class. Users can define a scheduler by creating a class that inherits the `DaxScheduler` base class, which is the class that contains the implementation of the wave algorithm as defined in Section 4.3.2. The set of nodes in the scheduling graph and the specifications for the periodic triggers as described in 4.3.3 are provided as class variables of the user-defined scheduler class. Additionally, a user-defined scheduler class inherits the ARTIQ `Experiment` class which marks it as an executable experiment for the ARTIQ runtime. Once the user-defined scheduler is started, the scheduling graph will be constructed based on the specified nodes, a timer will be set for periodic triggers, and the process will start a Python asyncio TCP server to receive incoming trigger requests. When the running scheduler receives a trigger, the scheduler will run the wave algorithm using the parameters given by the trigger.

ARTIQ includes a management system, the ARTIQ master, that queues experiments, handles data storage, and supervises the quantum co-processor. The pipeline for experiments, as described in 4.2, is already implemented by the ARTIQ master, and experiments submitted to the pipeline by a scheduler will be executed by the ARTIQ master process. The ARTIQ master manages a centralized database which we use to store the state of the nodes. The same database also stores system configuration and calibration results that need to be communicated from one experiment to the next. The part of our DAX library dedicated to system organization, real-time

data processing, and data organization will be covered in an upcoming paper.

All our scheduling code is tested thoroughly using static test cases as well as random testing. We tested our implementation of the Optimus algorithm by generating random DAGs containing calibration jobs whose interaction methods return random results. The DAGs are generated by randomly populating the upper triangle of an n by n adjacency matrix (where n is the number of nodes in the graph) with ones with some probability p, resulting in a random partial order over the set of nodes from which a DAG can be derived. We run our implementation over a single call to `maintain` (including any sub-calls to `diagnose`), and verify that our traversal matches the specification in [76]. Due to the existence of multiple valid traversals for a single call to `maintain`, we developed a recursive matching algorithm to determine if a traversal is valid given the graph structure and the calibration states/results.

4.6 Results

To benchmark the efficiency of the Optimus algorithm compared to a naive full calibration, we chose to simulate our implementation over a variety of conditions. For our simulations, we choose a fixed number of nodes and run our implementation of the `maintain` traversal (and any `diagnose` sub-traversals) over randomly generated DAGs. To reflect realistic conditions, there must be a cause-effect relationship for a `check_data` experiment to return bad-data. The purpose of the bad-data condition is to indicate that there are dependencies that are out-of-spec but did not time out. In other words, bad-data indicates an unresolved out-of-spec dependency. Hence, in the initial graph construction we completely randomize the results of `check_state` (whether or not a node has timed out), but we only randomize the results of `check_data` to select between in-spec and out-of-spec. At run time, if a node reaches the `check_data` stage and any of its dependencies (or sub-dependencies) remain out-of-spec, the node will return bad-data. Additionally, in

order to create a fair comparison between the Optimus algorithm and a naive full calibration of the system, we must guarantee that the system is fully calibrated by the end of the `maintain` wave. To achieve that, we force any root nodes to always time out, so that any unresolved out-of-spec nodes will be properly diagnosed and calibrated via the above bad-data logic.

The parameters for our simulation are 1. the probability that a node is out-of-spec in our initial graph construction and 2. the probability that a node has timed out (excluding the forced timeouts on the root nodes). We sweep each of these probabilities from 0 to 1 in intervals of 0.2, conducting 20 iterations at each point, each over a different random DAG. We choose the number of nodes to be $n = 20$ as a middle ground between the number calibrations required for ion-trap systems in our lab ($n \approx 10$) and current superconducting systems ($n \approx 40$) [40]. The DAG is generated via the method described in Section 4.5, with $p = 0.5$, resulting in a moderately connected graph.

The results of our simulation can be found in Figure 4.3. Figure 4.3-a shows the total number of calibrations performed normalized to the number of nodes versus the timeout and out-of-spec probability. We normalize the number of nodes in the graph to represent the relative efficiency of the Optimus algorithm compared to a full calibration. We can see that for out-of-spec probability 0 and 1 the proportion of calibrations performed is 0 and 1 respectively, as expected. For other out-of-spec probabilities, we see that the number of performed calibrations reduces when the timeout probability increases. This result is explained by the behavior resulting from a bad-data return value in the `check_data` stage. A higher likelihood of timeouts makes it more likely that any out-of-spec nodes are detected at or near the source. Conversely, a lower timeout probability means out-of-spec nodes are more likely to go unresolved, increasing the likelihood of bad-data results (and subsequent calibrations) in the parent nodes. In the worst-case scenario, the out-of-spec node

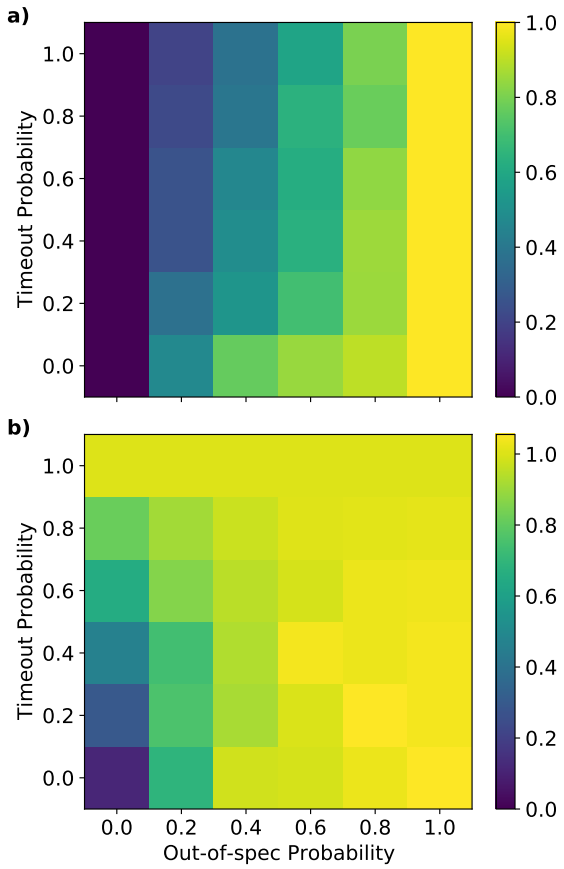


Figure 4.3: a) Proportion of calibrations performed and b) proportion of check experiments performed vs. timeout and out-of-spec probability.

goes undetected until the root nodes of the graph are reached, triggering a cascade of bad-data results and calibrations until the out-of-spec node is reached and resolved.

In Figure 4.3-b we see a graph with the number of check experiments performed relative to the number of nodes. The relative number of executed check experiments increases when the timeout probability or the out-of-spec probability increases. More timed-out experiments result in more check experiments to run while an increased number of out-of-spec nodes will cause more bad-data conditions, triggering more check experiments to run. It is possible for a calibration to be checked more than once throughout a traversal, in the case that it times out, gets checked, and then a parent calibration triggers a `diagnose` traversal due to an unresolved out-of-spec on

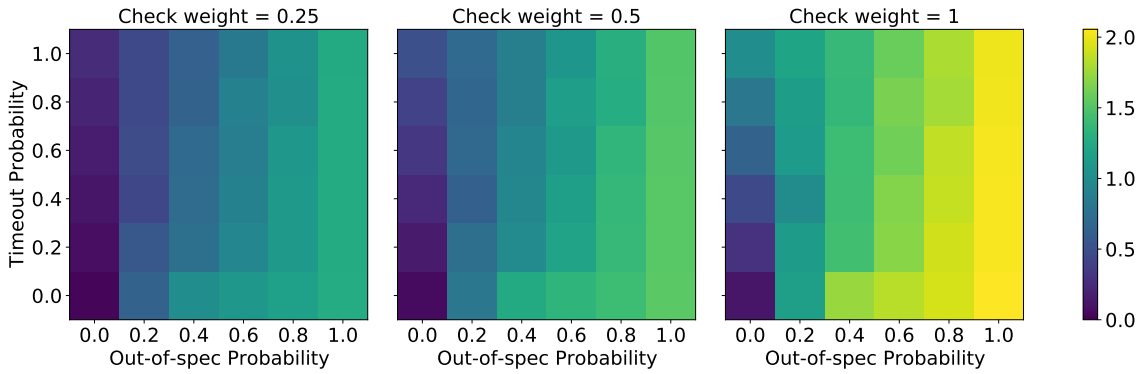


Figure 4.4: The relative efficiency of the full Optimus algorithm (checks and calibrations) with the cost of the check experiments weighted relative to the calibrations.

another node. The result is that in some cases — particularly when the out-of-spec probability is high and the timeout probability is low — the total number of checks can exceed the number of nodes in the graph.

To understand the overall efficiency of the Optimus algorithm, we need to take into account both the number of calibrations as well as the number of check experiments that are executed. As described in Section 4.4.1, the check experiment is a quick experiment to check if a parameter is still calibrated which normally takes significantly less time compared to a full calibration of the same parameter. We have estimated the overall efficiency of the Optimus algorithm for different weights of the check experiments compared to the calibration experiments and the results are shown in Figure 4.4. The weights represent how long the check experiment takes relative to the calibration itself. While lower-cost check experiments are ideal, we can see in Figure 4.4 that the Optimus algorithm can be beneficial even for more costly checks, depending on the out-of-spec probability. Specifically, the Optimus algorithm is more efficient than a naive full calibration for out-of-spec probabilities ≤ 0.4 and ≤ 0.6 with check weights 0.5 and 0.25, respectively. In the case that the check experiment takes the same amount of time as the calibration (i.e. check weight 1), Figure 4.4 shows that the Optimus algorithm is essentially always detrimental to

performance, as expected.

In terms of the limitations of our simulations, Figure 4.4 likely represents a lower bound of the efficiency of the Optimus algorithm. In a more realistic scenario, the timeouts would reflect the relative stability of the calibrations (i.e. how often a calibration goes out-of-spec), rather than the two results being drawn from independent probability distributions. Careful selection of timeouts would result in fewer bad-data results, and thus fewer unnecessary calibrations. Hence, we expect the Optimus algorithm to perform better in scenarios where timeouts are chosen carefully.

4.7 Conclusion

We have presented an open-source scheduling toolkit that schedules jobs based on a directed acyclic graph. Our scheduler is driven by a configurable wave algorithm and is capable of running simple periodic tasks as well as complex system calibration routines as background processes. We enable universal feedback between the scheduler and any running experiment or process by using triggers that will start new traversals on the scheduling graph. To demonstrate the capabilities of our toolkit, we have implemented the Optimus system calibration algorithm using our scheduling infrastructure and used our implementation to benchmark the efficiency of the algorithm.

5

Toward ARTIQ on the RFSoC

5.1 Introduction

When designing a quantum control system, one of the chief concerns is the mechanism for performing quantum gates. With both trapped ions and superconductors – the two leading qubit platforms – gates are driven by electromagnetic radiation. For ions, that takes the form of either direct microwave radiation [77] or laser light modulated by a radio-frequency (RF) signal [20]. For superconductors, it is typically microwave radiation [78]. In all cases, it is desirable to have high quality signal generation hardware. Furthermore, not only is it necessary to have a feedback mechanism for gate selection as discussed in Section 2.1, it is often advantageous to modify the waveforms that drive our gates in real time to compensate for things like heating or the drifting frequencies of other electronics [78, 79].

At the modulation frequencies required for trapped ions (discussed in Section 2.2), one candidate for this real-time waveform generation is the direct digital synthesizer (DDS): an integrated circuit (IC) that outputs high-fidelity sine waves based on an input frequency, phase, and amplitude. The DDS is an example of a *parametric*

waveform generator, where only a small amount of data (the inputs) is required in order to generate a continuous stream of voltages (the sine wave). Parametric waveform generators are particularly well suited to the problem of real-time waveform definition, placing less strain on the producer of those definitions: the controller. For trapped ions, all that is required for single-quantum bit (qubit) gates is a sine wave to fine tune the laser frequency [20], perfect for a DDS. Two-qubit gates however require a minimum of four signals to be generated, with additional requirements on the relative phase between them. While these signals can be multiplexed into a pair of two-tone sine waves applied to each ion, DDSes can only generate a single tone per channel, meaning two channels must be mixed together and presenting the first obstacle to scalability for DDSes. The other obstacle is a result of work toward making two-qubit gates more robust to noise, with researchers discovering that carefully varying the frequency, phase, and/or amplitude of the applied modulation over the course of a single gate can help achieve that robustness [29–31]. The issue is that the controller and DDS are loosely coupled, requiring communication between them to go off-chip or off-board over relatively low-bandwidth inter-integrated circuit (I^2C) or serial peripheral interface (SPI) buses. As these two-qubit modulation schemes, or pulse designs, have become more complex, that bus has become unable to keep up with bandwidth demands.

In order to use modern pulse designs, many experimenters turned to the arbitrary waveform generator (AWG) [80, 81]. In terms of waveform generation capabilities alone, AWGs are the gold standard – after all, they can generate *arbitrary waveforms*. Of course, that comes at a cost: they are not parametric. Outputs from an AWG are specified as a series of *waypoints*: voltages that may be stepped or interpolated between. Waveform definitions are created (perhaps parametrically) on a host computer, compiled into these waypoints and some metadata, then uploaded into memory on the AWG – almost completely decoupled from the real-time controller.

Compilation is an expensive process, taking minutes on a desktop computer for even a moderately sized circuit on tens of qubits. In other words, there is no chance of achieving real-time waveform definition using an AWG. Additionally, the fact that waveforms must be stored in the AWG memory means that gate and circuit complexity are bounded by the size of that memory, a significant obstacle to scalability. While there has been work to allow real-time waveform *selection* with an AWG or AWG-like architecture [60, 80], allowing for gate-level feedback and easing memory requirements, the problem of waveform-level feedback remains. Another factor to consider is that *arbitrary* waveforms are rarely – if ever – necessary for performing quantum gates, regardless of the physical platform. The physical equations governing a qubit are parametric, as are any modulation schemes derived from those equations. There is no reason the waveform generation hardware cannot be parametric as well. Lastly, AWGs are prohibitively expensive.

Researchers across the domains of quantum computing have recognized the need for a better solution [15, 16, 82–84]. Parametric waveform definitions together with tight coupling between the controller and a sufficiently sophisticated waveform generator present a significant step toward scaling up quantum control systems. One option is to create a custom (platform-specific) waveform engine in field-programmable gate array (FPGA) logic alongside the controller and connect it to an on-board digital to analog converter (DAC). A custom engine allows researchers to find the right per-platform balance between waveform complexity and compression of the waveform definition via parametrization. When connected to a high-quality DAC, that engine can then create the required pulses in real time, allowing for arbitrary feedback schemes over circuits of any length.

Even better than a DAC on the same board as the controller FPGA is a DAC on the same chip – the tightest coupling possible. First introduced in 2018, the AMD Zynq UltraScale+ RFSoC family of chips provides just that. In addition to its

industry-leading FPGA and up to 16 channels of 14-bit, 9.85 GSPS DACs, the RFSoC hosts up to 16 channels of 14-bit, 2.5 GSPS (or 8 channels of 5 GSPS) analog to digital converters (ADCs) along with a quad-core Arm Cortex-A53 application processing unit (APU) and a dual-core Arm Cortex-R5 real-time processing unit (RPU). The platform is particularly appealing to superconducting platforms [15, 16], which can utilize the tight coupling between the ADCs (used for qubit measurement) and DACs (used for gates) to create low-latency feedback loops. Trapped ions, which use digital input/output (I/O) for qubit measurement and have much longer lifetimes than superconducting qubits, do not require that particular feedback loop and have much more relaxed latency requirements. The real-time definition of complex waveforms, however, is very much of interest for trapped ions for the reasons detailed above.

In this chapter I will discuss my work toward supporting an existing control platform, ARTIQ, natively on the RFSoC. In Section 5.2 I will introduce existing work using the RFSoC for superconducting qubits [15, 16] as well as for trapped ions [83, 84]. I will then discuss the drawbacks of those designs and how my work presents an architectural improvement over them for the control of trapped ions, along with previous work using ARTIQ on a similar platform, which serves as evidence for the viability of this architecture. In Section 5.3 I give an overview of the RFSoC platform and how it naturally leads to the software architecture of my implementation. Section 5.4 and Section 5.5 contain the implementation details of my work relating to that hardware. Section 5.6 gives an overview of the higher-level software support built upon those implementations. Finally, in Section 5.7, I discuss the remaining work required to fully support ARTIQ on the RFSoC and conclude with the architectural improvements it will enable in quantum control systems and how those improvements could facilitate scaling of those control systems.

5.2 Related Work

Two of the superconducting control solutions, the Quantum Instrumentation Control Kit (QICK) [15] and QiCells [16], feature soft-core implementations of the RISC-V instruction set architecture (ISA) alongside pulse generation and qubit measurement logic in the RFSoC's FPGA. Each work features quantum-specific extensions to the ISA along with differing implementations thereof, however the fundamental architecture remains the same. Both of the implementations are heavily FPGA-focused, using the APU (running embedded Linux) only to initialize the system and communicate between the host PC and the real-time RISC-V core. The advantage of the soft-core architecture is that it avoids the relatively slow and unpredictable operation of transferring data over a shared system bus between a hardened CPU and the real-time I/O (RTIO) logic in the FPGA. However, it comes with two distinct disadvantages. The first is the hardware itself – FPGAs are inherently less performant than the equivalent application-specific integrated circuits (ASICs). Taking the QICK implementation as an example, the RFSoC's APU can be clocked at up to 1.33 GHz [85], while the soft-core "tProcessor" in the QICK operates at 384 MHz [15]. The second disadvantage comes from the microarchitectural limitations of a soft-core processor. Modern CPUs have a plethora of specialized hardware and microarchitectural optimizations, most of which are difficult (due to timing constraints) or impossible (due to lack of specialized hardware) to implement in an FPGA. Combined, those two disadvantages result in a soft-core processor that is significantly handicapped compared to its ASIC counterpart.

The only (non-commercial) use of the RFSoC with trapped ions comes from Sandia National Laboratories under the quantum scientific computing open user testbed (QSCOUT) program. An integral part of the control system, the RF generation for QSCOUT's gates is based on the Xilinx RFSoC platform [83]. Dubbed *Octet*, the

gateware design is specially tailored to trapped ions. Each physical channel supports two tones, with each of the parameters able to be driven by cubic spline interpolators, resulting in state-of-the pulse shaping abilities with a compressed, parametric definition. In addition, the Octet design features integrated crosstalk cancellation and frequency feedback to compensate for laser drift. At the time of this writing there have been 20 publications associated with QSCOUT, including collaborations in quantum chemistry [86], characterization and validation [87, 88], software optimization [89], and robust gate design [31]. The QSCOUT program, with Octet and the RFSoC at the center of its controls, has been a great success.

However, the Octet architecture has its limitations. In its original design [83], Octet was similar to an AWG: pulses, and the circuit as a whole, had to be defined and compiled on a host computer before being sent to the RFSoC over a network connection and loaded into memory. The parametric waveform representation and custom engine did much to ease the memory requirements of the system, but the fundamental architecture was the same. Recent improvements on the design have been substantial, allowing for branching (real-time waveform selection) based on signals from the main controller, leveraging the APU to perform inter-shot mutation of waveform definitions, and more. The result of these optimizations is that the Octet framework will be able to keep up with the demands of trapped-ion hardware for the foreseeable future. I would argue, however, that the *need* for those optimizations is a symptom of a sub-optimal architecture that will eventually fail to scale.

The path to arbitrarily-scalable control systems, in my view, lies in a combination of the two approaches I have presented: tight coupling between controller and the pulse engine as in the superconducting platforms while also leveraging the superior capabilities of a hardened CPU not only for waveform generation, as in Octet, but as the controller. For superconducting platforms, that may not be possible with current hardware: typical system-on-chip (SoC) communication buses simply are

not targeted for such low-latency, deterministic interfaces. That being said, it is not difficult to imagine a private, ultra-low-latency path between CPU and FPGA (though I make no claims about the difficulty of implementation). Trapped ions, on the other hand, *can* afford the latency and throughput restrictions of modern, more general-purpose SoC buses.

In 2021, researchers from M-Labs and the University of Oxford demonstrated just that, porting ARTIQ to the Zynq-7000 family of SoCs [90]. The Zynq-7000 features a dual-core Arm Cortex-A9 processor coupled with an Artix 7 or Kintex 7 FPGA [91]. In their implementation, the Cortex-A9 hosts the ARTIQ runtime: a bare-metal runtime for real-time control in an asymmetric multi-processing (AMP) configuration. In the ARTIQ AMP configuration, one core – the *Kernel CPU* is responsible for running the real-time kernels that communicate with the RTIO logic in the FPGA. The other core, the *Comms CPU*, takes care of general housekeeping: network communication with the host PC, starting and stopping the Kernel CPU, and other miscellaneous tasks. This control flow is shown in Figure 5.1. Prior to the Zynq-7000 work, ARTIQ's Comms and Kernel CPUs were soft-core implementations alongside the real-time core in an FPGA – very similar to the above superconducting platforms. In those implementations, the Kernel CPU and the real-time core communicated through a wishbone bus: a simple, resource efficient bus that allowed for the required low-latency communication. Whether or not acceptable latency and throughput could be achieved between the Zynq-7000's hardened CPUs and its FPGA was a significant unknown. Through creative use of the accelerator coherency port (ACP) and the Kernel CPU's `SEV` signal, their team achieved a sustained RTIO event interval of 376 nanoseconds compared to 418 nanoseconds with a soft-core design on a Kintex-7 FPGA. Making use of both the CPU and FPGA, this work from M-Labs demonstrated that a truly heterogeneous solution to real-time control was indeed viable.

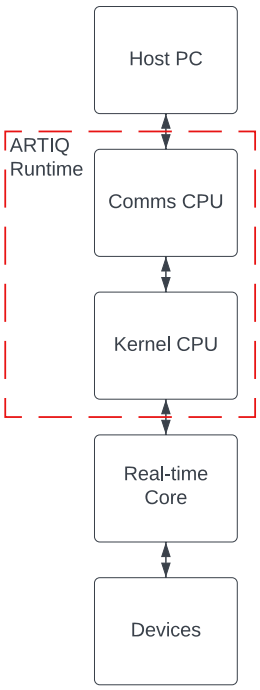


Figure 5.1: ARTIQ control flow, outlining the components that make up the runtime.

5.3 System Overview and Software Architecture

The work presented in Section 5.2 provides motivation for porting ARTIQ to the RFSoC as well as evidence for the success of such a project. The ARTIQ runtime, written in the Rust language, requires bare-metal control of the entire platform: the APU and its caches, various I/O peripherals, the FPGA gateware, and more. Since the provided bare-metal libraries for the RFSoC [92] are written in C, enabling that control means either writing thousands of foreign function interfaces (FFIs) between Rust and C or implementing the libraries from scratch using pure Rust. Similar to the authors of the Zynq-7000 work [90], I opted for the latter, deciding that the safety and robustness of avoiding the unsafe C libraries outweighed the potential ease of development from using them. The structure of the libraries I wrote arises directly from the RFSoC hardware.

5.3.1 *Hardware Overview*

The Xilinx RFSoC, at its core, is a powerful heterogeneous computing platform. It features a quad-core Arm Cortex-A53 APU alongside a dual-core Arm Cortex-R5 RPU, an Arm Mali-400 GPU, and an industry-leading FPGA [93]. Figure 5.2 shows a high-level block diagram of the RFSoC components. The components are

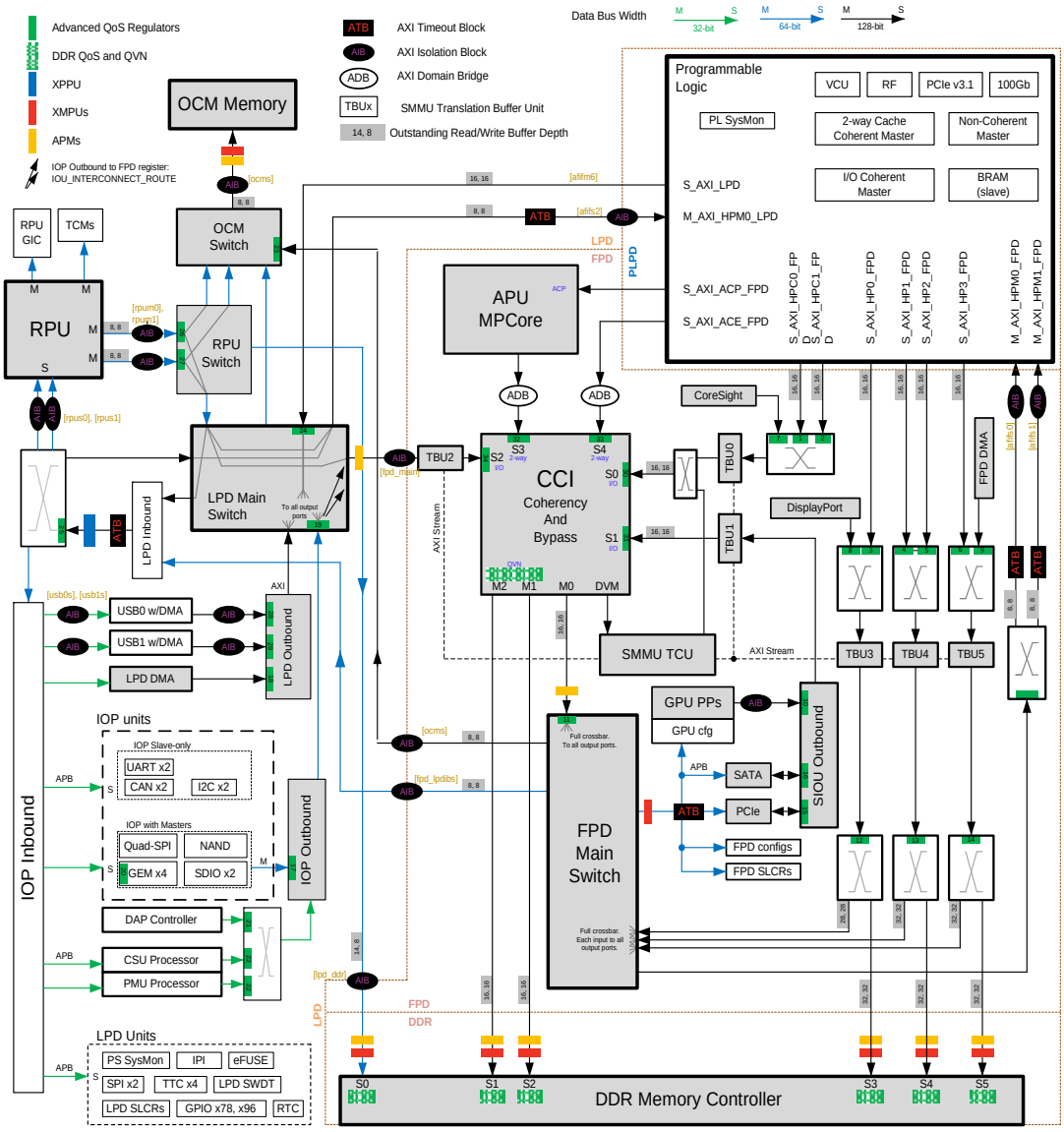


Figure 5.2: RFSoC overview and interconnect. Figure from Xilinx UG1085 with minor modifications [93].

broadly divided into two subsystems: the processing system (PS), which contains the APU, RPU, GPU, and related peripheral controllers, and the programmable logic (PL), which contains the FPGA, the RF DACs and ADCs, and other FPGA-side peripherals.

5.3.2 Software Architecture

While the existing RFSoC-based quantum control solutions strongly emphasize the PL as described in Section 5.2, my approach centers around the PS, using the powerful classical processors as the "brain" of the control system. The PL, of course, still has an important role to play: in the deterministic timing of RTIO events, the necessary logic for interfacing with the real-time devices, and let us not forget the RF DACs we will use to drive our quantum gates. Since control of the APU is an integral component of the software and is largely independent from the details of the board that houses the RFSoC, I put that in its own library: `libcortex_a53`. The peripheral devices, on the other hand, are board-specific. So, the drivers for the RFSoC's device controllers are in the library `libboard_zynqmp`. Since the Comms CPU is responsible for maintaining several I/O channels (network connections, messages from the Kernel CPU, etc.) at any given time, it is helpful to have language-level support for concurrency. The library `libasync` provides just that, allowing the use of Rust's `async`/`await` concurrency. Finally, higher-level (requiring both the APU and board drivers) software support for things like dynamic memory allocation is contained in `libsupport_zynqmp`. I have used those libraries to write a boot loader for the device, the final piece for which will be to load the user's runtime from an SD card into memory and then hand off to that runtime. All components are open source, contained in the GitLab repository ZynqMP-rs [94].

5.4 The APU: `libcortex_a53`

The heart of the RFSoC PS is the APU, with four Arm Cortex A53 cores implementing the 64-bit Armv8-A architecture. Each core has separate 32 KB L1 instruction and data caches, and they share a unified 1 MB L2 cache with hardware coherency to other system components through the cache-coherent interconnect (CCI). Figure 5.3 shows a high-level diagram of the APU components. Furthermore, the cores

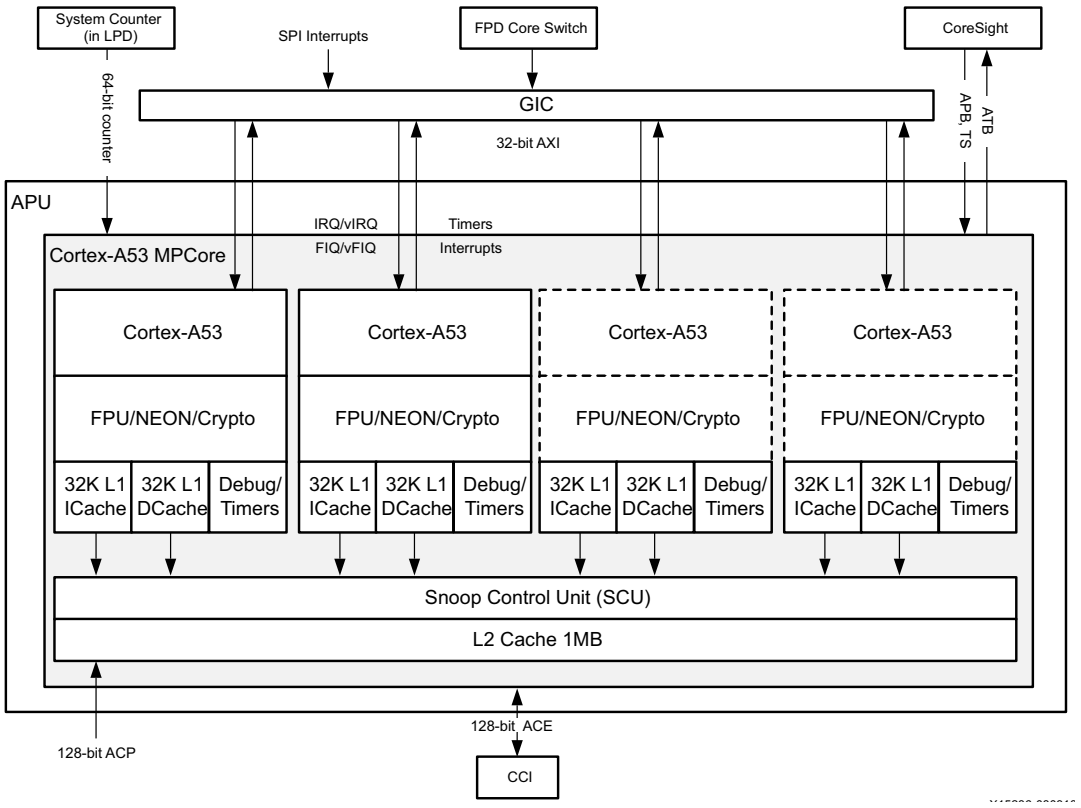


Figure 5.3: High-level APU diagram. Figure from Xilinx UG1085 [93].

feature some hardware performance extensions, most notably the advanced single instruction, multiple data (SIMD), VFPv4 floating-point, and cryptography extensions. On the ZCU111 the APU can be clocked at up to 1333 MHz [85], and on other multi-processing system-on-chips (MPSoCs) at up to 1500 MHz [95]. While the Cortex A53s and associated system components (system memory management

unit (SMMU), generic interrupt controller (GIC), etc.) have extensive virtualization support, I do not want any operating system, let alone two operating systems stacked on top of each other wearing a hypervisor trench coat, so I will not discuss those features. I will also omit discussions of Armv8's AArch32 mode and the details of the architecture's exception levels (ELs) – I want full 64-bit instructions and I have no reason to limit the execution privileges of my own code. In other words, all of the following assumes the cores are running in AArch64 mode at EL3.

5.4.1 Memory Virtualization – The MMU

In introductory computer architecture, memory virtualization is typically taught in the context of user-space processes in an operating system: each process gets its own virtual address (VA) space for sandboxing and abstraction purposes. The VAs used by that process are translated into physical addresss (PAs) via page tables and the corresponding location in memory is what is actually accessed. The operating system keeps track of an ever-evolving collage of page tables, creating maps for each process, moving pages in and out of memory when necessary, and so on. The other simplification often made at the introductory level is that our system consists of a CPU and memory, perhaps with the vague notion of external storage existing. In reality, any useful computing system contains a plethora of elements – many of which interact with the outside world – connected by some shared communication bus. At the hardware level, the CPU has no idea whether a PA points to system memory (more accurately, the memory controller) or a general purpose input/output (GPIO) pin, for example. So what happens when you try to cache a GPIO pin? (Hint: nothing good.) Without a way to distinguish between what *type* of device the CPU is talking to at a given PA, the only safe option is to cache nothing, disabling what is arguably the single most important performance accelerator a CPU has. That is why, even at the level of bare-metal software, the MMU is a crucial component in

terms of both performance and correctness.

Chapter D8 of the Arm A-Profile Architecture Reference Manual [96] defines the Arm virtual memory system architecture (VMSA): the specification of how memory virtualization should be handled on CPUs implementing the A-profile architecture. Figure 5.4 shows a (very) rough sketch of how addresses are translated with the MMU. Translation of a VA returns not only the corresponding PA, but also a set of *attributes* which tell the CPU exactly how that location should be treated. Section 5.4.1 explores those attributes in more detail.

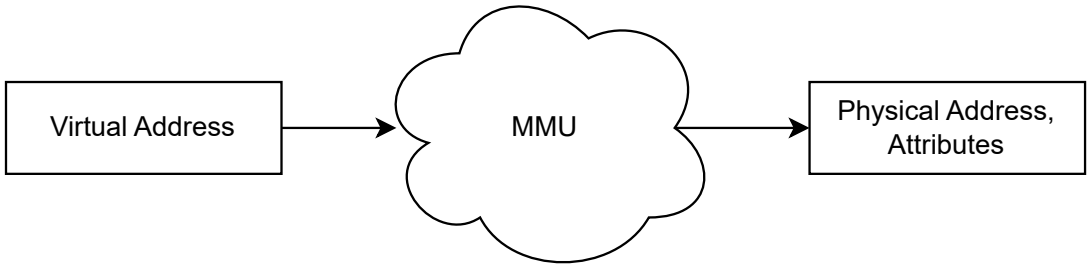


Figure 5.4: Rough sketch of the MMU translation process.

Memory Attributes in the VMSA

The VMSA defines two types of memory: *Normal* and *Device*.[1] Normal memory, as the name suggests, describes typical memory devices. Specifically, it indicates that the CPU can perform speculative reads to that location. Device memory, on the other hand, is typically used for peripheral devices. It indicates that accesses to that location can have side effects and, among other things, cannot be speculatively accessed. There are additional attributes associated with Device memory – *Gathering*, *Reordering*, and *Early Write Acknowledgment* – which can be used to impose or relax additional constraints on how those locations are accessed. For the sake of brevity and since section B2.15.2 of the A-Profile Architecture Manual [96] explores

[1] In this context, "memory" refers to any address the CPU can read from or write to.

them in depth, I will not discuss the details of those constraints. I will, however, give an overview of the attributes associated with Normal memory, as those become relevant in Section 5.5.7.

Shareability Shareability defines whether a memory location is expected to be accessed by multiple observers and needs to be kept coherent between them. If a region is non-shareable, that means we only expect a single observer to access it, and it can be cached without concern. If a region is shareable, that complicates things a bit. In a system without hardware coherency, memory being shareable means it cannot be safely cached. Luckily for us, the Cortex-A53 implements coherency between the L1 data caches of its four cores. That means that memory can be shared between the cores while still being cacheable. The RFSoC also contains a CCI, which provides coherency between other elements of the system and the Cortex-A53's shared L2 cache. These two "levels" of coherency are reflected in what Arm calls shareability *domains*. The Arm architecture defines two such domains: inner and outer shareable. The division between those domains, however, is implementation-defined, so I will focus on what they look like in the RFSoC. Figure 5.5 shows some of the observers in the RFSoC and their shareability domains from the perspective of a core in the Cortex-A53. The four cores, with their L1 data cache hardware coherency, make up the inner shareable domain. Any other observers in the system, such as the GPU or direct memory access (DMA) devices, with coherency via the CCI, make up the outer shareable domain.

Cacheability The cacheability attribute defines if and how a region of Normal memory may be cached. In broad terms, a region can be non-cacheable, write-through (WT) cacheable, or write-back (WB) cacheable. The Cortex-A53 does not support WT cacheability, so our options are non-cacheable and WB cacheable. The Cortex-

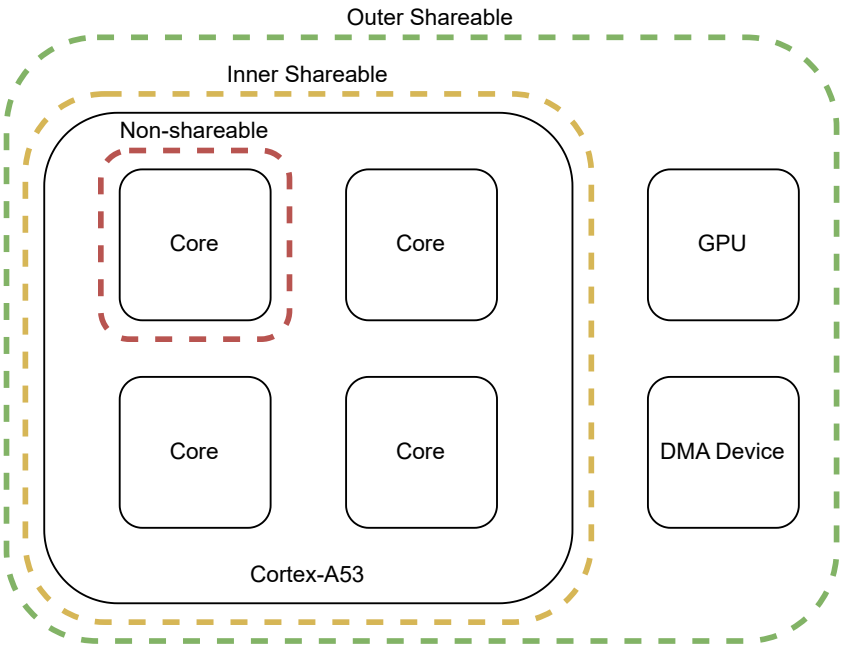


Figure 5.5: RFSoC shareability domains from the perspective of a core in the Cortex-A53.

A53 does, however, support cache allocation hints for WB cacheable memory: read allocate (RA) and write allocate (WA). RA means a block is brought into the cache when it is read from main memory (in other words the read was a cache miss) and WA does the same when it is written to main memory. For typical computation under a WB policy, we want both the RA and WA allocation hints in order to make the best use of our cache.

RFSoC System Addresses

Figure 5.6 shows the system addresses of the various components of the RFSoC PS. The SMMU (not to be confused with the Cortex-A53-specific MMU) supports physical address spaces of 40, 36, or 32 bits, shown by the vertical dashed lines in Figure 5.6. Unlike in the case of user-space programs running on an operating system, we do not want to use the MMU to abstract away the physical addresses. We want a direct map (VA = PA), maintaining readability while using the aforementioned

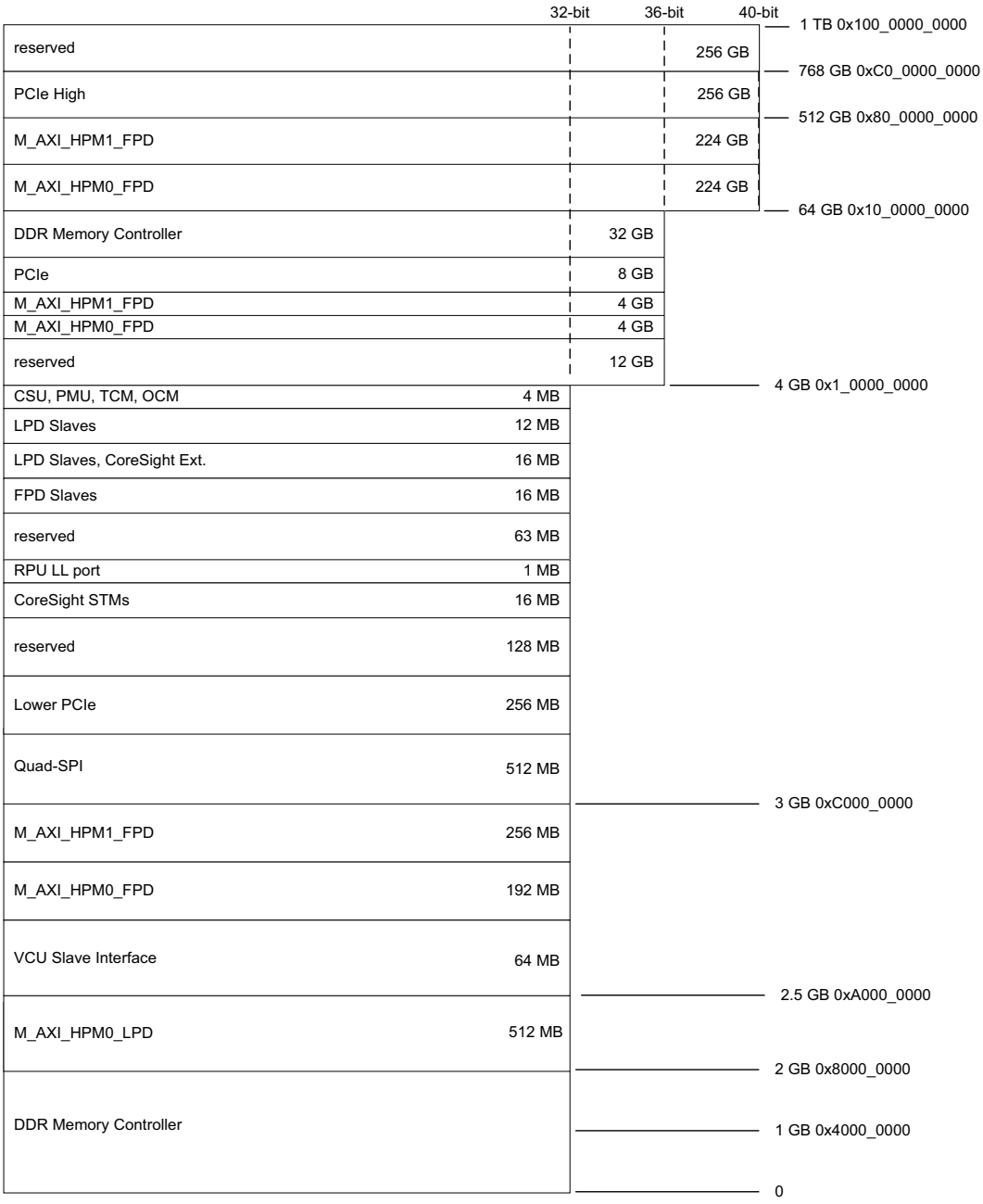


Figure 5.6: The RFSoC system address map. Figure from Xilinx UG1085 [93].

attributes to safely cache what we can. To avoid discrepancies between the Cortex-A53s and the SMMU, the choices for the MMU address space are also 40, 36, and 32 bits. A general-purpose boot loader like the first-stage boot loader (FSBL) has no choice but to implement the full 40-bit address space in the MMU. I, however, have the freedom to look at the address map and realize that we do not need any of the upper 936 GB for experiment control – the 36-bit address space will do just fine.

You might notice that Figure 5.6 shows some address ranges as *reserved*, which at the bare-metal level simply means that there is nothing there. Translation tables in the VMSA can directly express this nothingness with a third "type" of memory, aptly called Reserved or Invalid memory. If an address is reserved, the CPU will not perform speculative accesses to it and any direct access via user code will result in a Translation Fault exception.

The Translation Process

Before getting into the consequences of choosing a 36-bit address space over the full 40-bit space, we must dig into a few details of the VMSAv8 translation process. The VMSA supports up to 5 levels of nested translation tables for 3 different *granules* (page sizes): 4, 16, and 64 KB. The number of levels required depends on both the size of the VA space and the granule size. For the 40-bit address space, the FSBL uses a 4 KB granule resulting in the three-level table structure shown in Table 5.1. Note that the final lookup level (4 KB pages) is not used – for the purposes of the boot loader such fine-grained control is not necessary. Another important detail is that the level 0 table is indexed by a single bit, and to my knowledge the upper 512 GB address space is never accessed during boot. While it may seem wasteful to spend an entire level of indirection on a single bit, it turns out that for a 40-bit space and the granularity required for the RFSoC's address map, you will always end up with at least 3 levels of translation tables. With our 36-bit space, however, we can

Table 5.1: FSBL translation table structure, 40-bit address space with 4 KB granule.

Lookup Level	Block Size	VA Range Used to Index
0	512 GB	VA[39]
1	1 GB	VA[38:30]
2	2 MB	VA[29:21]

Table 5.2: Translation table structure for the ZynqMP-rs memory map, 36-bit space with a 4 KB granule.

Lookup Level	Block Size	VA Range Used to Index
1	1 GB	VA[35:30]
2	2 MB	VA[29:21]

get rid of the level 0 table, resulting in the structure shown in Table 5.2.

Effect of Implementation Details

The FSBL sets up the translation tables and enables the MMU as one of the first steps upon booting, before branching to any C code. That is to say: the tables are defined and enabled entirely in AArch64 assembly. While setting up the MMU as early as possible does speed up the rest of the boot loader by enabling the caches, there is not much reason for that setup to be done in pure assembly. In addition to reducing the maximum depth of translation table walks from 3 to 2, my 36-bit map is created entirely in Rust, providing a much more user-friendly interface along with type safety and the host of other guarantees that Rust provides. As a result, it was easy to implement a mechanism for making runtime modifications to the translation tables. My use case for that mechanism, as detailed in Section 5.5.7, is to change the cacheability attributes of a block.

Writing the MMU setup in a high-level language also makes it easier to set things

up in a hardware-specific way. For example, the RFSoC memory map has space for up to 34 GB of synchronous dynamic random-access memory (SDRAM), but the ZCU111 has only 4 GB of memory. The FSBL relies on hard-coded, board-specific macros to determine how much of that 34 GB should be marked as Normal memory in the translation tables and how much should be marked as Reserved. By having both the SDRAM setup (which detects the size of the physical memory present) and the MMU setup written in the same language, it is easy to pass parameters between the two.

It is also easy to run into some very strange bugs. I found that in certain situations, enabling the MMU with a complete memory map before the SDRAM setup results in a (seemingly) random assortment of exceptions. My best guess is that something in my code was causing a speculative access to the (yet to be initialized) SDRAM, though I do not know why that would show up in different ways. Regardless of the cause, the solution was to initially mark the SDRAM memory space as Reserved in the translation tables, setting it to Normal cacheable memory only after it is initialized and can actually function as such. Thanks to the interfaces I created for programming and modifying the translation tables, implementing that solution only required a few lines of code, and is really the correct way to set things up as the SDRAM address space truly is invalid until the memory has been initialized.

Mistakes and Opportunities for Improvements

While my implementation of the translation tables and MMU setup provides some valuable optimizations, it is not the best from a software architecture standpoint. The whole endeavour was a learning process for me, and when I started writing the code I did not have a clear picture of where it would end up. As a result, I made some bad design choices that created strong coupling between `libcortex_a53` and `libboard_zynqmp`. Simply put, I made some assumptions in programming the trans-

lation tables that depend on the RFSoC memory map, making the `libcortex_a53::mmu` module of limited use on any other system. By the time I realized my mistake, the effort to refactor was significant and I decided I had bigger fish to fry, so that code remains.

Another optimization that could be made is to utilize the *contiguous* attribute, which allows 16 consecutive translation table entries to be cached as one in the translation lookaside buffers (TLBs), provided the entries meet certain conditions [96]. Less crowding in the TLBs means fewer entries will have to be evicted, and thus fewer (costly) table walks required. Take, for example, the first 2 GB of SDRAM, which is translated in 2 MB blocks via the level 2 table. What would be 1024 TLB entries to cache translations for the entirety of that 2 GB turns into 64 with proper use of the contiguous attribute. That is a significant difference, especially when you consider that the Cortex-A53's main TLB has a capacity of 512 entries [97]. The problem comes when modifying the cacheability of individual blocks as required for the DMA detailed in Section 5.5.7. At that point we need to modify not only the cacheability attributes of the block in question, but its contiguous attribute and that of the other 15 blocks in its group. It is a fairly straightforward improvement in principle, but again, bigger fish.

5.4.2 Caches

Now that we have done all this work to safely enable the caches, what comes next? Not a whole lot, really – for typical operations, the hardware takes care of everything. However there are a few situations that require doing some cache maintenance. Cache maintenance operations in the Arm architecture [96] come in two flavors: *clean* and *invalidate*. There are many nuances, but the basic functions of the two operations boil down to this: *clean* operations make sure that changes we (the core our code is running on) have made to a memory location are visible to other observers; *invalidate*

operations make sure that changes others have made to a memory location are visible to us. The operations are made available in several forms via Arm instructions. For convenience and a bit of added safety I wrote Rust wrappers over the instructions, along with functions that use those wrappers to do things like clean/invalidate an arbitrary type `T` or a `slice[T]`.

5.4.3 Synchronization Primitives

The Armv8 instruction set offers several types of atomic operations, referred to as *exclusive* operations in the Arm lexicon. These exclusive operations can be used to craft higher level synchronization primitives to allow for shared memory and message passing between the cores in our Cortex-A53. The way the primitives are made is similar to the Armv7 implementation, which allowed me to reuse the code M-Labs developed in the `libcortex_a9` library [98] with minimal adaptations required. The final synchronization primitives supported for the Cortex-A53 are `libcortex_a53::mutex`, `semaphore`, and `sync_channel`.

5.5 Device Drivers: `libboard_zynqmp`

The RFSoC is a very complex heterogeneous computing platform, featuring controllers for dozens of types of peripheral devices [93]. The system initialization performed by the boot loader requires interacting with several of these devices, and supporting the ARTIQ runtime requires several more. This section discusses the details of those devices as they relate to my implementation of the Rust drivers, however it is not meant to be fully comprehensive documentation, as that is accomplished by Xilinx UG1085. As the scope of my current work is limited to the PS, only those peripherals are detailed here.

5.5.1 Power Domains

The PS contains two main power domains, the low-power domain (LPD) and the full-power domain (FPD). An important aspect of the design is that the system can function with only the LPD. However, the APU, along with some important peripherals such as Ethernet, lies in the FPD, so we will always power up both domains. Figure 5.7 shows the various power supplies and domains. The power domains themselves are not of particular interest, but a basic understanding is good as I will need to refer to them later on.

5.5.2 Clocks

The PS features 5 system phase-locked loops (PLLs), detailed in Table 5.3, from which all PS clocks are derived. Each PLL, in turn, can use one of 5 reference clocks,

Table 5.3: RFSoC system PLLs, the power domains in which they lie, and their recommended usage.

PLL	Power Domain	Usage
I/O PLL (`IOPLL`)	LPD	Low-speed peripherals, interconnect
RPU PLL (`RPLL`)	LPD	RPU, OCM, interconnect
APU PLL (`APLL`)	FPD	APU, interconnect
Video PLL (`VPLL`)	FPD	Video I/O
DDR PLL (`DPLL`)	FPD	DDR controller, interconnect

listed in Table 5.4. The `PS_REF_CLK` is the default clock source for all PLLs and is the one used in the FSBL [93]. As it is the only on-board clock out of the 5 options, it was also my choice.

The system PLLs can generate output frequencies from 750 MHz to 1600 MHz [85]. Figure 5.8 shows an overview of the RFSoC clock domains and the PLLs that drive them. Each power domain has access to its own PLLs as well as PLLs from the other domain via cross-domain clock generators. Each clocking subsystem (every

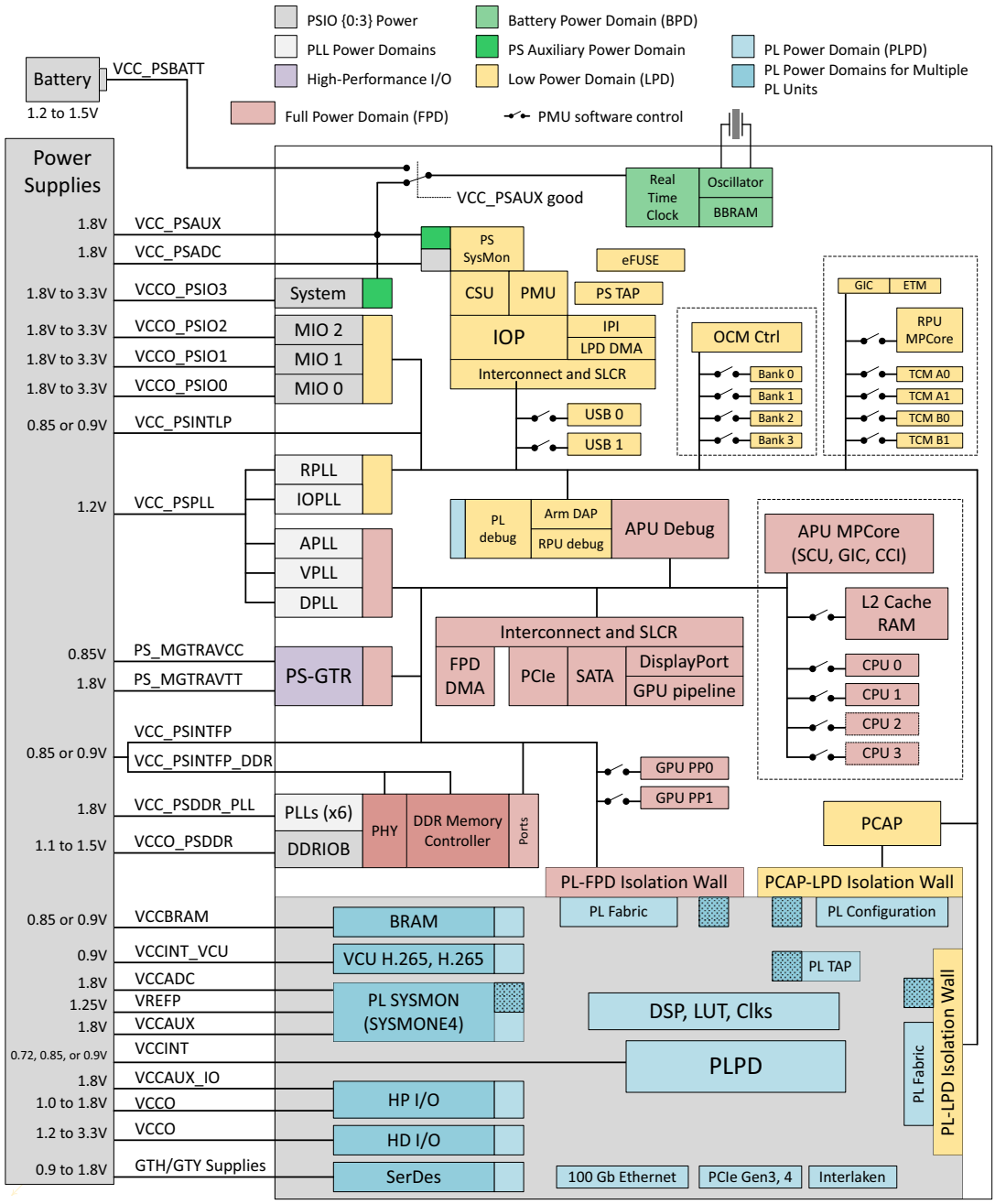


Figure 5.7: RFSoC power domains. Figure from Xilinx UG1085 [93].

Table 5.4: RFSoC system PLL reference clocks, their sources, and their frequencies on the ZCU111 [99].

Reference Clock	Source	Frequency (MHz)
PS_REF_CLK	On Board	33.33
ALT_REF_CLK	MIO pin	—
VIDEO_REF_CLK	MIO pin	—
AUX_REF_CLK	PL fabric	—
GTR_REF_CLK	GTR serial unit	—

individual block in Figure 5.8) has a strict maximum frequency, requiring careful programming of the clock generators.

The 5 system PLLs, being at the center of nearly all clocking, are a critical component of the RFSoC. One shortcoming of the FSBL is the lack of configurability of the clocks. The code that programs both the PLLs and the clock generators is auto-generated and hard-coded [92], with no library interface available for changing those parameters. One example of how this hinders performance is the APU clocking. Depending on speed grade, the APU can be clocked at up to either 1200 MHz or 1333 MHz [85], however, the clock in the FSBL is hard-coded to 1200 MHz [92]. In my implementation, all of the PLL parameters are calculated at runtime based on the desired output frequency alone, resulting in much more flexible clocking, and as a result, opportunities to improve performance.

5.5.3 Input/Output

The PS I/O peripherals are connected to the PS via a wide multiplexer/de-multiplexer structure, allowing for flexibility in board designs. Each of the 78 MIO pins has 4 levels of select bits, along with programmable pull up/down resistors, tri-state enable, drive strength, and slew rate. The pins are divided into 3 banks of 26, with each bank having its own (hardware-determined) voltage. The select bits for each

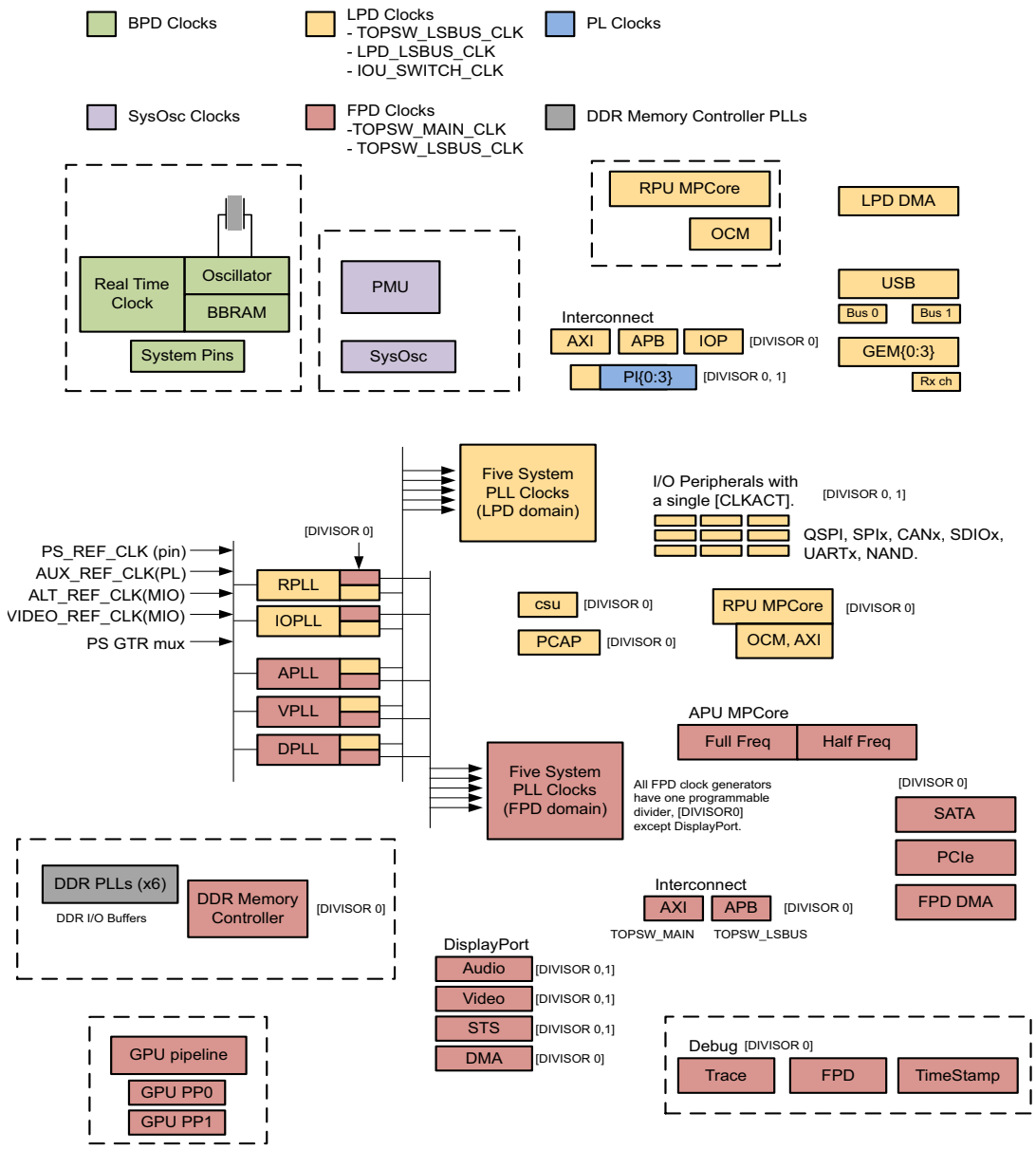


Figure 5.8: Clock domains in the RFSoC. Figure from Xilinx UG1085 [93].

pin determine which peripheral hardware controller the signals are routed to based on the connectivity of the board. For the ZCU111, the full MIO pinout can be found in Xilinx UG1271 [99].

5.5.4 UART

Even when developing a boot loader, the first step is to write a "Hello, world!" program.[2] Of course, in the case of a boot loader, that takes a bit more than a single `print` statement – first we need to configure the UART controller. The RFSoC has 2 independent PS UART controllers, however on the ZCU111 only one of them (UART0) is used, connected to a micro-USB output via the FTDI FT4232HL chip. Using the I/O PLL (1500 MHz) as the source, I set UART to have a 50 MHz clock – half of the maximum allowed frequency [85]. Reference frequency does not matter much for UART since the controller contains further dividers to reach the desired baud rate, shown in Figure 5.9. The controller also has configurable word length,

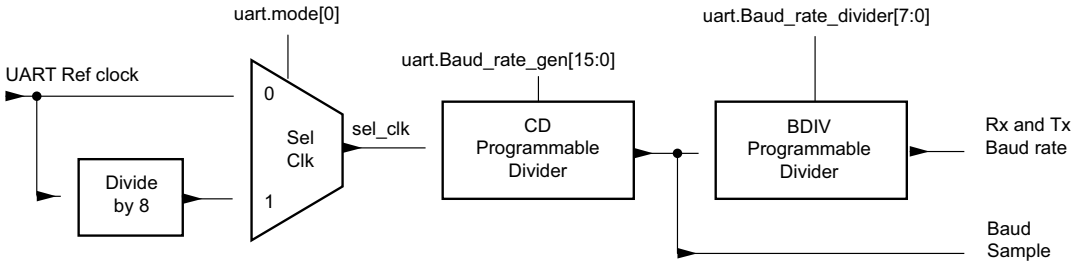


Figure 5.9: UART baud rate generator. Figure from Xilinx UG1085 [93].

stop bit(s), and parity bits. By default, I configure the UART for a baud rate of 115,200 Hz, word length of 8 bits, 1 stop bit, and no parity bits – common settings in my experience. One small improvement over the FSBL is that I calculate the optimal dividers for baud rate generation in code, rather than hard-coding anything, allowing users to easily change the baud rate. Any further configuration of the UART

[2] It's the law.

parameters can be achieved with a single register write, making full reconfiguration possible in two lines of code.

The final step of our "Hello, world!" – actually writing output via UART – is almost upon us. The UART controller contains 64-byte first in, first outs (FIFOs) for both transmit and receive data, accessible via writing or reading a memory-mapped register. I wrapped writes to that register along with the necessary safety checks in an implementation of Rust's `core::fmt::Write` trait, converting strings into a list of bytes to write. From there, I wrote `print!` and `println!` macros to emulate those in Rust's standard library. So, in the end, the program actually does come down to a simple `print` statement[3], I just had to implement the function myself.

5.5.5 I^2C Bus

The RFSoC contains 2 independently controlled I^2C buses, configurable in master or slave mode, with maskable interrupts. On the ZCU111, both buses are used to communicate with various onboard devices, including critical components such as the radio-frequency mezzanine card (RFMC) and the PS-side DDR4 small-outline DIMM (SODIMM). The board also contains an I^2C-controllable GPIO expander to supplement the MIO offerings. The full I^2C connection overview for the ZCU111 is shown in Figure 5.10. After configuring the MIO appropriately for the ZCU111's I^2C, I set the reference clocks for both to 100 MHz, using the I/O PLL as the source. According to the I^2C specification, clocking glitches of 50 nanoseconds must be filtered. The ZCU111 provides a digital glitch filter to achieve that requirement, which must be programmed in terms of reference clock cycles. In this case, a reference clock of 100 MHz results in $100\ MHz * 50\ ns = 5\ cycles$. As with all the calculations

[3] After writing the OpenOCD script to flash the compiled binary to the correct address, writing the linker script to dictate the binary layout, booting the cores and configuring per-core stack pointers, setting up the MIO pins for UART0, initializing the I/O PLL, and configuring the UART0 reference clock.

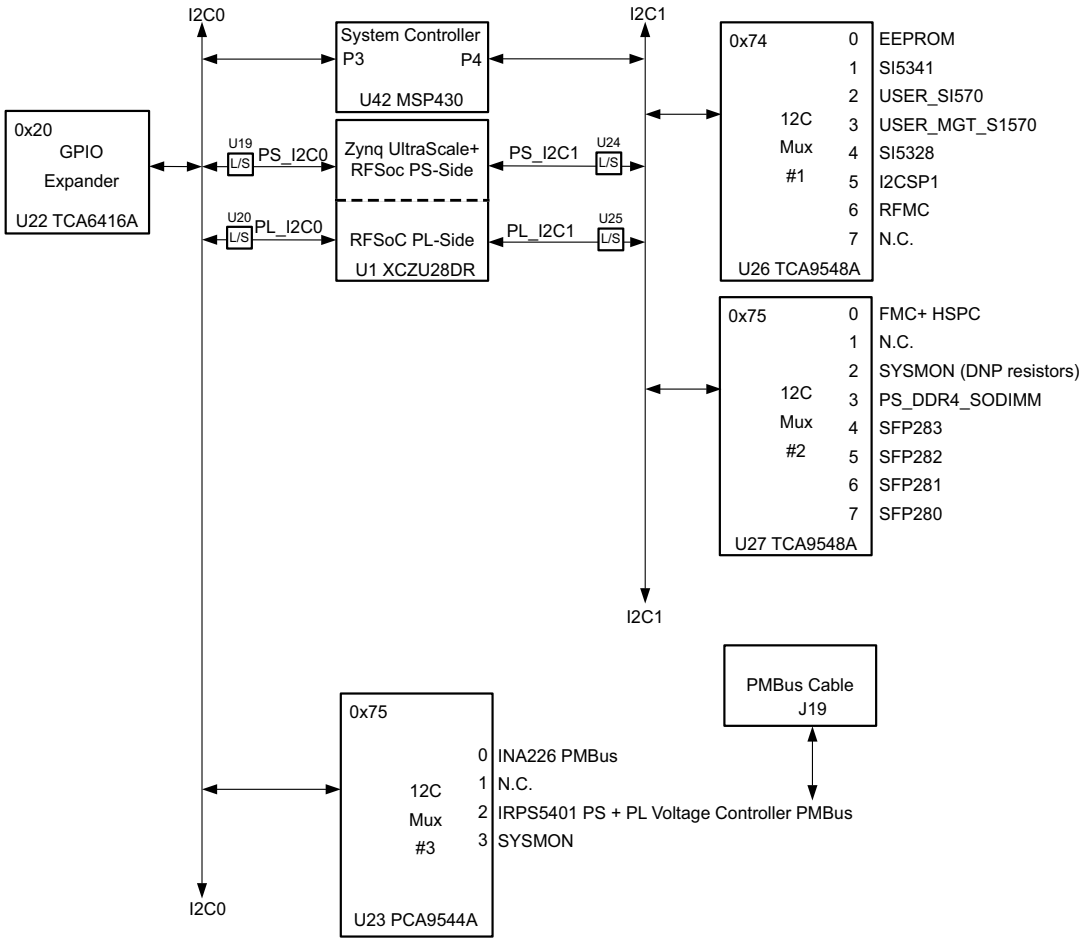


Figure 5.10: I^{2}C bus overview. Figure from Xilinx UG1271 [99].

I perform in the boot loader, however, that value is not hard-coded and is determined automatically from the reference clock.

UG1085 provides reasonably detailed and accurate instructions for using the I^{2}C interfaces, however, there are some key omissions. For example, the following is from a comment in the Xilinx I^{2}C driver [92] discussing setting up the I^{2}C clock:

```
/*
* If frequency 400KHz is selected, 384.6KHz should be set.
* If frequency 100KHz is selected, 90KHz should be set.
* This is due to a hardware limitation.
```

*/

Said hardware limitation is not mentioned in UG1085. I did my best to document any such discrepancy that I came across.

5.5.6 Memory

The PS memory on the ZCU111 is a DDR4 SODIMM socket containing a 4 GB Micron MTA4ATF51264HZ-2G6E1 memory module [99]. Although the memory itself can run at up to 2666 Mb/s [99], the RFSoC limits the speed of a single-rank DDR4 DIMM to 2133 Mb/s [85]. Figure 5.11 is a block diagram of the functional units involved in memory control on the RFSoC.

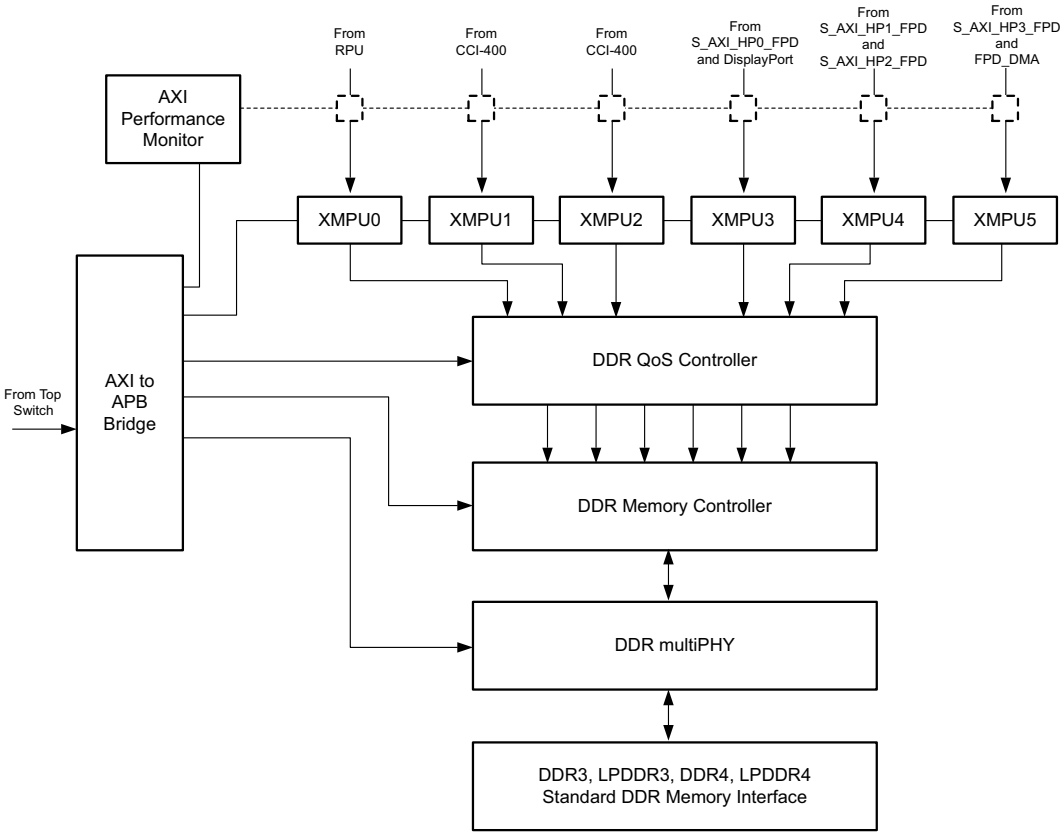


Figure 5.11: DDR subsystem block diagram. Figure from Xilinx UG1085.

The SPD EEPROM

While for a single board and a single memory component, it would be easier to hard-code the memory parameters from the device datasheet, they can be determined automatically from the SPD EEPROM. Using the SPD protocol allows for much easier extension of this work to other boards, as the memory configuration is one of the most complicated parts of the boot loader. To put the complexity in numerical terms, there are 669 32-bit registers defined between the `DDRC` (DDR controller) and `DDR_PHY` modules [100], most of which need to be precisely programmed for the operation of the DDR4 SDRAM.

For DDR4, the SPD layout is defined in JESD 21-C Annex L (henceforth referred to as the "SPD specification") [101]. The SPD specification defines 512 bytes of data, partitioned as shown in Table 5.5. The SODIMM on the ZCU111 is a type

Table 5.5: Top-level SPD layout [101].

Block	Address Range	Description
0	0-127	Base Configuration and DRAM Parameters
1	128-191	Standard Module Parameters
	192-255	Hybrid Module Parameters
2	256-319	Hybrid Module Extended Function Parameters
	320-383	Manufacturing Information
3	384-511	End User Programmable

of unbuffered DIMM (UDIMM), which leaves us with the much simpler SPD mapping shown in Table 5.6. In fact, the manufacturer information is unnecessary for operating the module, so we only need block 0 and the used parts of block 1.

On the ZCU111, the SPD EEPROM is accessed via the TCA9548A I^2C multiplexer at address `0x75` on I^2C1 [99]. Each block defined in the SPD specification

Table 5.6: Top-level SPD layout for UDIMM [101].

Block	Address Range	Description
0	0-127	Base Configuration and DRAM Parameters
1	128-191	Unbuffered Module Parameters
	192-253	Unused
	254-255	CRC Code, Block 1
2	256-319	Reserved
	320-383	Manufacturing Information
3	384-511	End User Programmable

contains a CRC code to allow for verification of the data. One improvement made in my work over the Xilinx FSBL is the use of said code for said verification [92]. The parameters defined in the SPD specification are not the full set of parameters required for configuring the memory – rather, they provide a base from which the full set of parameters can be calculated when paired with the technical standard for the type of memory involved. So, after reading the EEPROM and decoding the bytes according to the SPD specification, there is still work to do.

Configuration, Initialization, and Training

As mentioned above, there are two main register blocks for configuring the memory: `DDRC`, which contains high level control information, and `DDR_PHY`, which contains the multitude of timing parameters required to operate a specific SDRAM module. In order to save memory, I chose to calculate the individual timing parameters only when required for register programming rather than having a large, long-lived array or struct of pre-calculated values. Beyond that, there were not many "design choices" to make, per se – calculate the required value, put it in the register, move on.

After entering all the parameters into the required registers, we can commence

with initialization and training. An overview of the steps required is shown in Figure 5.12 and Figure 5.13. As one might have guessed from the number of timing

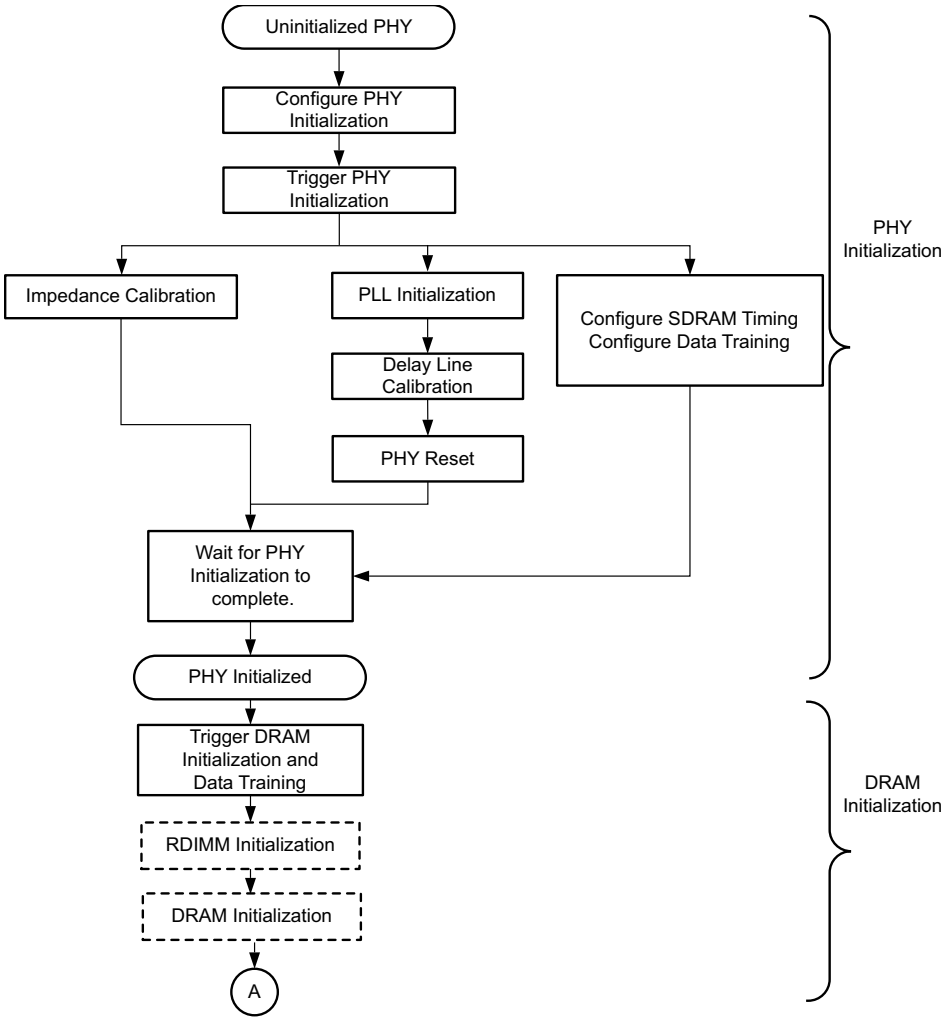


Figure 5.12: SDRAM initialization and training process. Figure from Xilinx UG1085 [93].

parameters involved, clocking is an important component of the DDR SDRAM subsystem. The reference clock comes from the DDR PLL in the FPD and is used for the controller as well as the 6 PLLs contained in the DDR subsystem, shown in Figure 5.14. In order to operate the ZCU111 DDR4 at the maximum rate of 2133 MT/s [85], we need a reference clock of about 533 MHz. Since that is below the

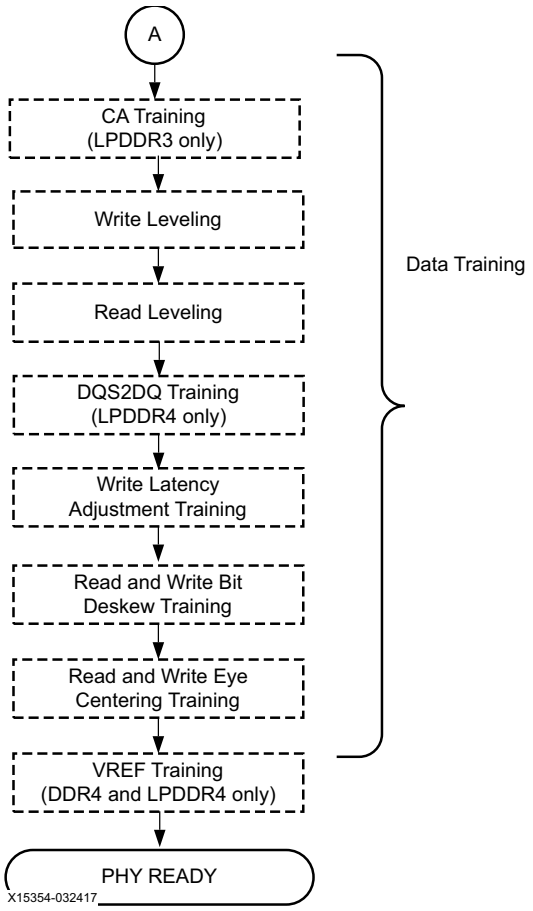


Figure 5.13: SDRAM initialization and training process (continued). Figure from Xilinx UG1085 [93].

range achievable with the system PLLs [85], I had to set the system DDR PLL to 1066 MHz and divide by 2 for the DDR reference clock. From there, the PLL initialization step was as simple as setting a flag in one of the `DDRC` registers and waiting for completion. In fact, most of the training is simply triggered with a control flag and then executes on its own via the PHY utility block (PUB).[4] UG1085 covers the training steps in more detail [93].

[4] Assuming you have not made any mistakes when programming the 668 other registers.

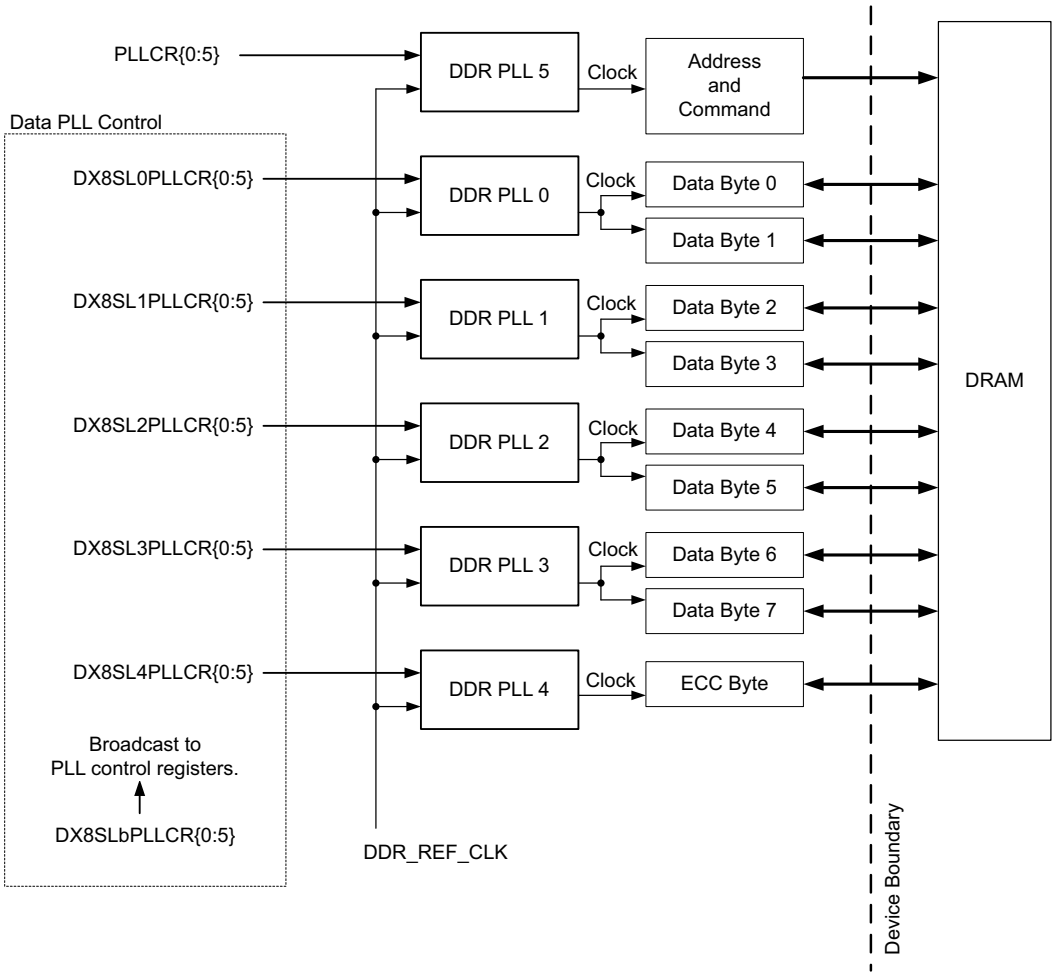


Figure 5.14: SDRAM clocking subsystem. Figure from Xilinx UG1085 [93].

The QoS Controller

As shown in Figure 5.11, the DDR subsystem contains a QoS controller to allow prioritizing certain types of memory traffic over others. The QoS controller defines 3 types of traffic: high priority (low latency), low priority (best effort), and video/isochronous (bounded latency). The FSBL opaquely divides the ports in a balanced way between the types of traffic [92], which is, of course, a good general-purpose approach. Real-time control of a quantum computer, however, is not a *general* purpose. My "architecture" was very simple: requests from the kernel CPU(s)

are high priority; everything else is low priority (and there is no video involved).

5.5.7 GEM Ethernet

The RFSoC features 4 ethernet media access control/controllers (MACs) capable of supporting 10/100/1000 operation via several types of PHYs. The 4 GEMs are denoted `GEM{0:3}`. Figure 5.15 shows a top-level block diagram of the ethernet controller. The ZCU111 uses `GEM3` over MIO with a TI DP83867IRPAP reduced

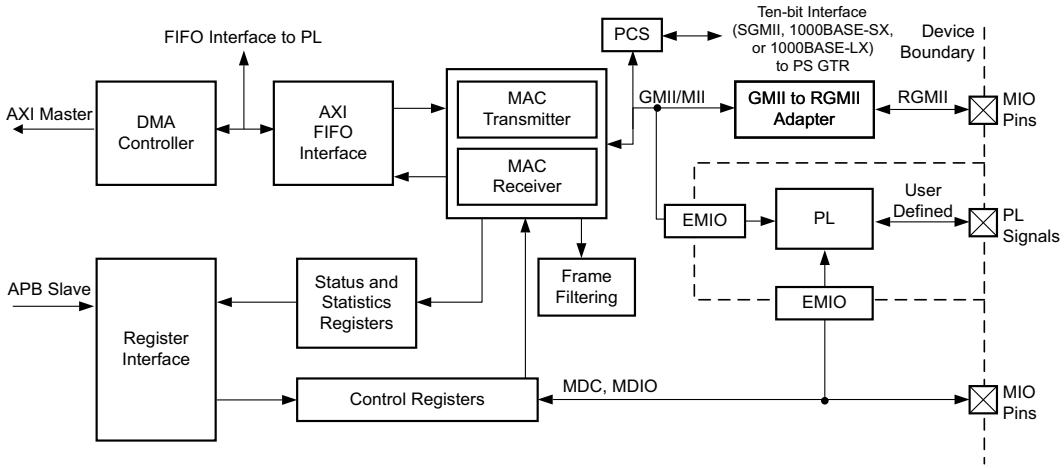


Figure 5.15: GEM block diagram. Figure from Xilinx UG1085 [93].

GMII (RGMII) PHY connected to an RJ45 ethernet connector [99].

Clocking

For most of the drivers written so far, the reference clock does not require runtime changes. However, due to the nature of the GEM clocking subsystem, changing between 10, 100, and 1000 Mb/s link speeds requires automatically adjusting the reference clock in `CRL_APB` [100].

The PHY Layer

The RGMII PHY contains its own set of registers, which we primarily need to access for starting link autonegotiation and reading back the negotiated speed once com-

plete. But, of course, nothing is that simple. It turns out that there is a "quirk" with the strap pins (hardware signals used to configure the PHY on power up) causing the PHY to start in one of its many self-test modes. Returning the PHY to its normal operating mode requires writing to one of the extended PHY registers, a slightly more complicated process than writing to the base set.

DMA

The main challenge with the ethernet driver was the DMA. Not only is it relatively complex to set up, it needs to be highly performant in order to accommodate a gigabit of data per second. The DMA controller provides a scatter-gather type functionality, where the packet buffers are accompanied by *descriptors* containing metadata about the location of the buffer and the data it contains. Scatter-gather DMA allows one packet to be spread across multiple buffers, however I found it more practical to create buffers the same size as the maximum transmission unit (MTU) (1536 bytes) so that there is a one-to-one relationship between packet and buffer. For 64-bit addressing, the details of the RX buffer descriptors can be found in UG1085 Tables 34-5 and 34-6, with the TX descriptors in Tables 34-8 and 34-9. The highlights are that each descriptor contains a) a pointer to its corresponding buffer and b) an *ownership* bit, indicating whether the CPU or the ethernet DMA controller is in charge of the buffer. Figure 5.16 illustrates the packet receive process. TX works similarly, with the CPU writing a packet into a buffer and then transferring control to the ethernet DMA engine by changing the ownership bit in the buffer descriptor. In a polling (non-interrupt-based) scheme, the CPU loops, waiting for the DMA controller to change the ownership bit, indicating that a packet has been received and written to the buffer in memory. In order for the change to be visible to the CPU, we have two options: either invalidate the cache line containing the descriptor before each read, or do not cache the descriptors in the first place. Invalidation is time-

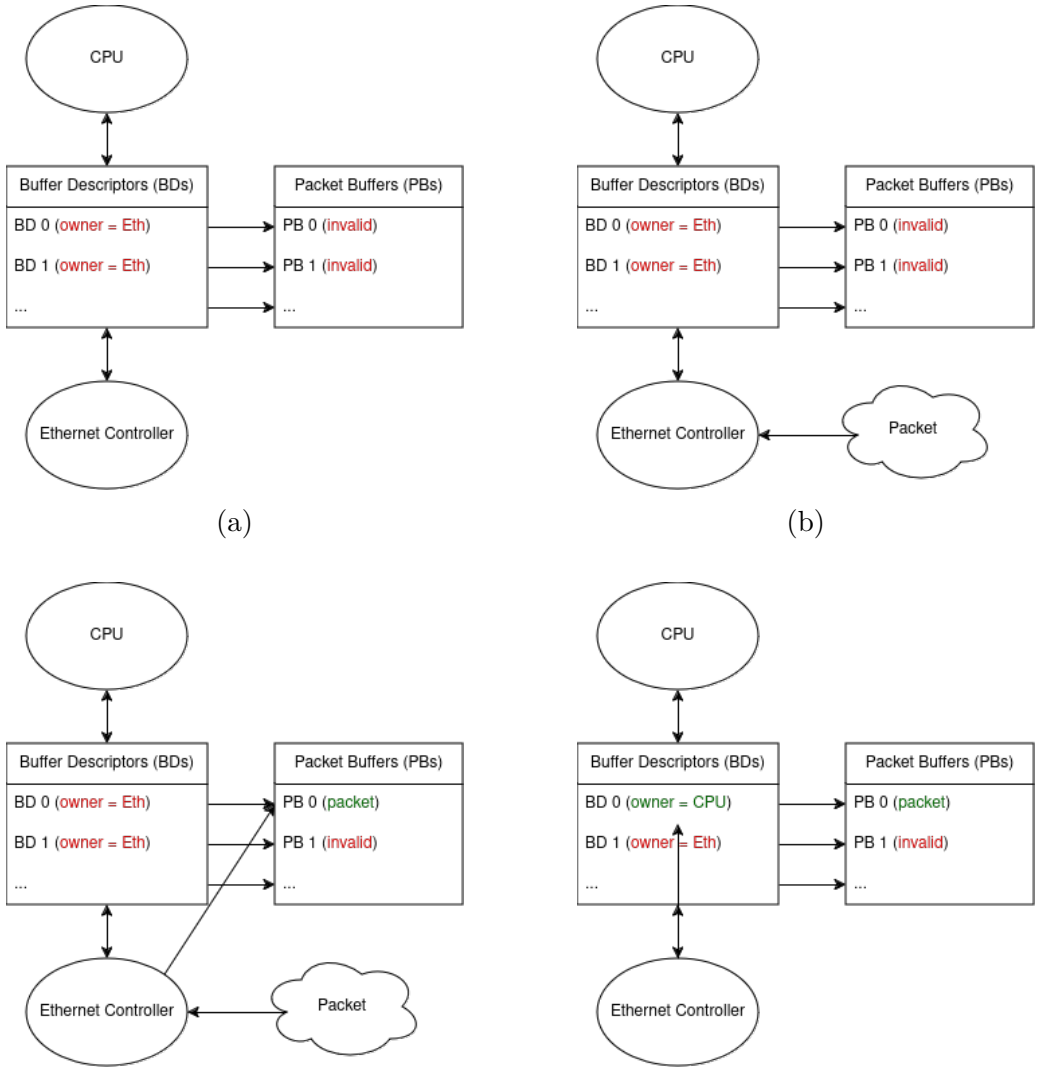


Figure 5.16: Example packet receive process. Figure 5.16a shows the initial state of the buffers and their corresponding descriptors. In Figures 5.16b and 5.16c, an incoming packet is written to memory by the ethernet DMA controller. Then in Figure 5.16d, the DMA controller sets the ownership field of the buffer descriptor to transfer control to the CPU.

Table 5.7: TX throughput statistics.

	Average (Mb/s)	**St. Dev. (Mb/s)**
1 KiB Payload	928.8	4.5
1 MiB Payload	941.7	6.6

consuming and – more importantly – unsafe: since the descriptors do not occupy a full cache line, invalidation could erroneously discard changes to other memory contained in the line. So, not caching the descriptors is the only real choice. As the number of buffers is configurable, it makes sense for them and their descriptors to be dynamically allocated, requiring both the memory allocator detailed in Section 5.6.2 and runtime changes to the MMU translation tables (Section 5.4.1) to indicate that the descriptors should not be cached.

Performance

Benchmarking was performed using the `async` runtime discussed in Section 5.6.1 with 3 concurrent tasks: an RX server, a TX server, and a statistics tracker counting the number of data bytes sent or received each second. While all 3 tasks ran concurrently, I only benchmarked the connections in one direction at a time. From the host side, I used the `netcat` Linux command line utility. For both TX and RX, I tested with payloads of 1 KiB and 1 MiB, collecting data for 100 seconds and then calculating the average throughput and the standard deviation. Gathering the TX data was straightforward: I ran the command `netcat -I 1000000 <device IP address> <TX port> > /dev/null`, resulting in the data shown in Table 5.7. The RX data on the other hand initially displayed some puzzling characteristics. Running `dd if=/dev/zero bs=1M count=1M | netcat -N <device IP address> <RX port>` resulted in an average of 822 Mb/s and a standard deviation of 214 Mb/s – very high compared to the TX standard deviation. Such a high variance seemed

Table 5.8: RX throughput statistics.

	Average (Mb/s)	St. Dev. (Mb/s)
1 KiB Payload	871.1	83.8
1 MiB Payload	896.4	11.4

more likely to be caused by OS scheduling than by anything on the RFSoC, so I focused my efforts on tweaking the command on the host side. Eventually I arrived at the final command used for testing: `sudo nice -n -20 dd if=/dev/zero bs=1M count=1M | netcat -O 1000000 -N <device IP address> <RX port>`. The resulting benchmark statistics are shown in Table 5.8.

5.6 Higher-Level Libraries: `libasync` and `libsupport_zynqmp`

The final pieces of the puzzle build upon the hardware interfaces and abstractions introduced in Sections 5.4 and 5.5 to create necessary utilities for a workable bare-metal ecosystem. While the hardware differences between the Zynq-7000 SoCs and the RFSoCs are significant enough that there was not much overlap between Zynq-rs [98] and my work in the low-level libraries, the same (fortunately) does not hold true here. While developing `libcortex_a53` and `libboard_zynqmp`, I attempted to maintain as much symmetry as possible with the interfaces defined in the Zynq-rs counterparts `libcortex_a9` and `libboard_zynq`. The result of that effort is that much of the code contained in `libasync` and `libsupport_zynqmp` was just a simple refactor of the original code in Zynq-rs. In this section I will give a brief description of the functionality in the two libraries and any changes I made.

5.6.1 `libasync`

The core functionality of `libasync` is a single-threaded `async` runtime. For the uninitiated, an `async` runtime in Rust is a state machine, allowing any number

of procedures to take or yield control of one or more threads, so "single-threaded `async`" is not in fact an oxymoron. In the single-threaded case, this mechanism essentially provides the illusion of concurrency. "Why bother?" you might ask. As a relevant example, let's think about listening for TCP connections on a particular port. With `libboard_zynqmp`'s ethernet and `smoltcp`, that basically boils down to executing an infinite loop. Now, say we want to listen for connections on multiple ports – perhaps while also executing some other code. Would we rather dive into `smoltcp`'s internals and figure out how to weave together multiple complex state machines with our own or simply write a few `async` wrappers? I thought so. The `libasync::smoltcp` module contains wrappers for that exact purpose and is the only place I made changes, updating `smoltcp` from v0.7.0 to v0.10.0 – not an entirely trivial effort due to breaking changes.

5.6.2 `libsupport_zynqmp`

From an API perspective, the primary functionality currently provided by `libsupport` is the memory allocator. The actual allocation logic is contained in its dependency `linked_list_allocator` [102]: `libsupport_zynqmp` simply initializes the heap in SDRAM and protects it with a `libcortex_a53::mutex`. The allocator can be configured with a single heap – safely shareable between cores thanks to the mutex lock – or private heaps for each core. Rust's `alloc::GlobalAllocator` trait is implemented and a default allocator defined via the `#[global_allocator]` attribute. The effect is that all types requiring dynamic memory allocation (e.g., `alloc::vec::Vec`) are linked to `libsupport`'s allocator and can be used just as in a hosted environment.

From a boot loader perspective, `libsupport` contains several key components. The most important is the boot code – the very first instructions executed by the core after it powers on or is reset. The first few instructions are (and must be) assembly to take care of the things (such as initializing the stack pointer) that a higher-level

language like Rust requires. Then the boot code take steps to put the processor in a known state: clean the caches, set some architectural configuration registers, enable the floating point unit (FPU), and ensure the `.bss` memory section is zeroed out. After that, it sets up the MMU as described in Section 5.4.1 and jumps to the boot loader's `main` function. Other noteworthy inclusions in `libsupport` include default exception and interrupt request (IRQ) handlers, a default `panic` handler, and utilities for starting and stopping the secondary (non-boot) cores.

5.7 Conclusion

5.7.1 Future Work

On the PS side, the last remaining piece is the SD card. The low-level driver in `libboard_zynqmp` will need to be finished and then built upon in `libsupport_zynqmp` to mount the FAT filesystem. From there the boot loader can read the runtime program into memory along with reading the FPGA bitstream and writing it to the PL via the processor configuration access port (PCAP). Those features, along with a bit of wrapping and a bow on top, would form a complete boot loader.

Next is the matter of the ARTIQ runtime. I would like to believe that my efforts to mirror the Zynq-rs APIs as much as possible will make that port a relatively painless process, but I have learned that such a thing does not exist at this level of programming. Porting the ARTIQ runtime takes care of the PS side. On the PL side, gateware and an ARTIQ driver will need to be written for producing waveforms with the RF DACs. Some of the groundwork for that has been laid by ARTIQ's smart AWG (SAWG) modules, but much still needs to be done. The final, crucial component for ARTIQ is the communication between PS and PL. In the paper that describes the port of ARTIQ to the Zynq-7000, *Combining processing throughput, low latency and timing accuracy in experiment control*, the authors discuss two channels they explored for that communication: the general-purpose advanced extensible

interface (AXI) ports and the ACP. It turned out that the recommended method – the general-purpose AXI ports – was far slower than previous, pure-FPGA implementations of ARTIQ. They discovered that through creative use of the ACP they could significantly reduce the latency, even to the point of beating the pure-FPGA designs. The same two mechanisms are available on the RFSoC although their latencies could differ from those on the Zynq-7000 platform. Another potential option, not present on the Zynq-7000, is PL-mastered DMA to the RPU's tightly coupled memory (TCM). With a kernel running on the APU, however, that would incur the extra latency between APU and RPU, making it unlikely to beat the ACP method. UG1085 Table 35-5 [93] provides a helpful, albeit qualitative, comparison of the available PS-PL interfaces.

5.7.2 Architectural Outlook

Beyond the immediate goal (and reward) of a fully functioning RFSoC-based ARTIQ controller, the RFSoC presents an opportunity to explore a new real-time control paradigm: real-time multiprocessing. Recall Figure 5.1 and the two classical cores required by ARTIQ: the Comms CPU and the Kernel CPU. Also recall that the RFSoC's APU is a *quad*-core device. What do we do with those extra cores? We put them to work!

When exploring the topic of utility-scale quantum computing, a modular approach often comes up [103–105]. The basic motivation of the modular approach is that qubit and gate quality is difficult to maintain when you put a lot of qubits together, be it ions in a trap of superconducting qubits on a chip. They interfere with each other. The modular solution involves dividing the qubits into more manageable chunks and then connecting those chunks with a sort of quantum communication bus. Each of those modules requires its own controller, with the same requirements as for a single-module system in addition to the burden of managing that inter-module

communication.

A proposed solution for trapped ions involves physically separate traps with a photonic interconnect [103]. With photonic interconnects, one or more ions in each trap are designated as *communication qubits* with the rest designated for typical computation. When desired, these communication qubits can be entangled via the photonic interconnects and the resulting entanglement then *swapped* to entangle the two chains of computation qubits. The process of remotely entangling ions using photons, however, is probabilistic, resulting in a repeat-until-success process of unknown duration. Once entanglement succeeds (as measured by a third device), the two real-time controllers in question receive a signal indicating the success. For a single-process real-time controller, that means polling the signal until completion. Want to do other things with the computation qubits in the meantime? Too bad. With even one more real-time process, we could start attempting to entangle the communication qubits while the prerequisite computation on the other qubits is *still running*, drastically reducing the time cost of remote entanglement and improving overall performance. This example presents a critical use case for real-time multiprocessing, with principles applicable across platforms and specific schemes for modularity.

6

Conclusion

We are at a point where many of the proposed architectures for scaling up quantum systems are within reach. As quantum bit (qubit) counts rise from the hundreds into the thousands and beyond, the problem of controlling them will become increasingly complex. In order for control software and hardware systems to keep up, they must be carefully designed – robust enough to build upon while also flexible enough to adapt to changing demands. Furthermore, computing in the noisy intermediate-scale quantum (NISQ) era requires a mix of high- and low-level control in order to achieve optimal performance. In other words, practical quantum advantage in the near term will not be achieved by opaque layers of abstraction, compiling a hardware-agnostic quantum circuit down to optimal machine code, but rather by careful, device-specific optimizations made at each level of the stack. The work presented in Chapter 3 helps to address the robustness and flexibility of control software while also enabling transparent abstractions to higher levels in the stack. The work in Chapter 4 addressed the automation and optimization of repetitive hardware maintenance operations, easing the load on hardware operators even as systems scale in size. Finally, in Chapter 5, I introduced open-source work toward a

new hardware control platform for trapped ions, and proposed a model for how that platform can be used as a node in a distributed control architecture. In total, this dissertation has addressed some of the hardware and software architectural challenges to scaling up quantum control systems.

6.1 Outlook

Based on current and proposed technologies, it seems likely that near-term scaling of quantum computers will be achieved through the development of modular systems. These systems in turn require distributed, modular control architectures. Even in distributed classical systems there is ongoing research into how best to architect and use them. Distributed real-time control systems come with a host of additional requirements and progress in the field is of critical importance to the future of quantum computing.

On the hardware side of things, I see a distinct need for more specialized system-on-chipsystems-on-chip (SoCs). While the radio-frequency system-on-chip (RFSoC) product family has gained popularity in the field of quantum control, it is not designed for such an application. To my knowledge, no SoC on the market today is. The soft-core architectures researches have come up with are a symptom of that deficiency, *not* a fundamental design requirement for quantum control systems. Trapped ions and other longer-lived qubits can afford the time cost of using a complex, shared bus to communicate between the control logic on a hardened CPU and the real-time I/O (RTIO) logic in a field-programmable gate array (FPGA), which means those platforms can reap the benefits of a drastically more capable processor to implement more sophisticated control schemes. Superconducting qubits – the platform with the strictest instruction latency and throughput constraints – meanwhile are stuck with the least capable control units. *All* quantum computing platforms would benefit from an SoC designed for dedicated, ultra-low-latency communication between

a high-performance, multi-core application processing unit (APU) or real-time processing unit (RPU) and an FPGA.

Of course, as I mentioned in Chapter 2, application-specific integrated circuits (ASICs) are only cost-effective when produced at large scales, so for now such an SoC may be nothing more than a dream. The competing, sometimes customized, architectures (RISC-V, ARM, various very long instruction word (VLIW) designs) and microarchitectures may also be a deterrent for a product designer entertaining the notion of creating such an SoC. I would argue however that something is much better than nothing. The use of advanced real-time infrastructure for quantum physics (ARTIQ) for real-time control from a stock Arm Cortex-A9 demonstrates that architectural extensions are not strictly necessary for quantum control. With a capable enough multi-core system, even the perceived need for VLIW architectures, popular among superconducting platforms for their throughput capabilities, could be dispelled. Or perhaps we will see the renaissance of hardened VLIW or explicitly parallel instruction computing (EPIC) architectures.[1] There is much uncertainty in the future of quantum control hardware, but the one certainty is that progress in classical computing systems is required to enable the long-term scaling of quantum systems.

Beyond the level of a single controller, progress in distributed control architectures will need to be made as well. Near-term distributed real-time control will likely have a star topology, with one central controller orchestrating interactions between a handful of local module controllers. Distribution of control will likely require a globally asynchronous, locally synchronous (GALS) model, wherein each local controller operates independently until communication with another module is required, then asynchronously signals a communication request to the central controller and waits for completion. As discussed in Section 5.7.2, such a model naturally lends

[1] R.I.P. Intel Itanium

itself to a real-time multiprocessing approach. Low-latency, high-bandwidth classical communcation buses between the devices will be required, potentially needing new bus and communication protocols beyond modern capabilities. As quantum systems scale to even larger sizes, the processing bandwidth of the central controller will hit a limit, necessitating a heirarchical structure and further increasing demand on the communication buses. There is reason to believe that this area of research will benefit distributed classical computing, which often experiences bottlenecks in communication, as well.

On the software side, exploration of new concurrency paradigms may also be necessary. If you have ever worked on a concurrent classical application, you know what a headache it can be: dealing with race conditions, resource contention, etc. Now imagine that 1 nanosecond can make or break that application. For the time being, the `async/await` concurrency model, for example as in the Rust language, seems to appropriately capture the GALS principles and is relatively easy for application programmers to reason about. However, as quantum control systems grow and concurrency becomes increasingly complex, the need for a new paradigm may very well arise. In particular, the ease of programming with such a model must be considered. In classical computing, the need for memory consistency models and cache coherency protocols arose once multi-core systems increased the complexity of data coherency to the point where it was no longer tenable to put that burden on software. We may eventually see the need for an analagous development of those models and protocols for quantum control hardware in order to enable reasonable software control of these systems.

Regardless of the form future control hardware and software takes, it must be co-designed alongside the quantum systems themselves. Each layer of abstraction built upon the control software must be carefully crafted to support current *and* future devices, both quantum and classical. Even then, unforeseen developments

may occur, in which case system architects must be willing to pivot their designs, even if it means breaking backwards compatibility.[2] It is very possible that the quantum analog of the metal–oxide–silicon field-effect transistor (MOSFET) has not even been discovered yet, and a rush to standardization without carefully considering the consequences could ultimately backfire.

[2] Let's not have another x86 situation on our hands, please.

Bibliography

[1] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. DOI: https://doi.org/10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230. [Online]. Available: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, 1965.

[3] R. Dennard, F. Gaensslen, H.-N. Yu, *et al.*, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/JSSC.1974.1050511.

[4] A. Danowitz, K. Kelley, J. Mao, *et al.*, "Cpu db: Recording microprocessor history," *Commun. ACM*, vol. 55, no. 4, pp. 55–63, Apr. 2012, ISSN: 0001-0782. DOI: 10.1145/2133806.2133822. [Online]. Available: https://doi.org/10.1145/2133806.2133822.

[5] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, no. 6, pp. 467–488, Jun. 1, 1982, ISSN: 1572-9575. DOI: 10.1007/BF02650179. [Online]. Available: https://doi.org/10.1007/BF02650179.

[6] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[7] L. K. Grover, *A fast quantum mechanical algorithm for database search*, 1996. arXiv: quant-ph/9605043 [quant-ph].

[8] A. Y. Kitaev, *Quantum measurements and the abelian stabilizer problem*, 1995. arXiv: quant-ph/9511026 [quant-ph].

[9] Y. Kim, A. Eddins, S. Anand, *et al.*, "Evidence for the utility of quantum computing before fault tolerance," *Nature*, vol. 618, no. 7965, pp. 500–505, Jun. 1, 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-06096-3. [Online]. Available: https://doi.org/10.1038/s41586-023-06096-3.

[10] S. Ebadi, T. T. Wang, H. Levine, *et al.*, "Quantum phases of matter on a 256-atom programmable quantum simulator," *Nature*, vol. 595, no. 7866, pp. 227–232, Jul. 1, 2021, ISSN: 1476-4687. DOI: 10.1038/s41586-021-03582-4. [Online]. Available: https://doi.org/10.1038/s41586-021-03582-4.

[11] F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 1, 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5.

[12] M. A. Nielsen and I. Chuang, *Quantum computation and quantum information*, 2002.

[13] S. Bourdeauducq, R. Jördens, P. Zotov, *et al.*, *Artiq 1.0*, version 1.0, May 2016. DOI: 10.5281/zenodo.51303. [Online]. Available: https://doi.org/10.5281/zenodo.51303.

[14] K. H. Park, Y. S. Yap, Y. P. Tan, *et al.*, "Icarus-q: Integrated control and readout unit for scalable quantum processors," *Review of Scientific Instruments*, vol. 93, no. 10, 2022.

[15] L. Stefanazzi, K. Treptow, N. Wilcer, *et al.*, "The qick (quantum instrumentation control kit): Readout and control for qubits and detectors," *Review of Scientific Instruments*, vol. 93, no. 4, 2022.

[16] R. Gebauer, N. Karcher, M. Güler, *et al.*, "Qicells: A modular rfsoc-based approach to interface superconducting quantum bits," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, pp. 1–23, 2023.

[17] W. Paul and H. Steinwedel, "Notizen: Ein neues massenspektrometer ohne magnetfeld," *Zeitschrift für Naturforschung A*, vol. 8, no. 7, pp. 448–450, 1953. DOI: doi:10.1515/zna-1953-0710. [Online]. Available: https://doi.org/10.1515/zna-1953-0710.

[18] E. Fermi, "Über die magnetischen Momente der Atomkerne," *Zeitschrift für Physik*, vol. 60, no. 5, pp. 320–333, May 1, 1930, ISSN: 0044-3328. DOI: 10.1007/BF01339933. [Online]. Available: https://doi.org/10.1007/BF01339933.

[19] Y. Wang, M. Um, J. Zhang, *et al.*, "Single-qubit quantum memory exceeding ten-minute coherence time," *Nature Photonics*, vol. 11, no. 10, pp. 646–650, Oct. 1, 2017, ISSN: 1749-4893. DOI: 10.1038/s41566-017-0007-1. [Online]. Available: https://doi.org/10.1038/s41566-017-0007-1.

[20] S. Olmschenk, K. C. Younge, D. L. Moehring, *et al.*, "Manipulation and detection of a trapped yb+ hyperfine qubit," *Physical Review A*, vol. 76, no. 5, p. 052 314, 2007.

[21] N. M. Linke, D. Maslov, M. Roetteler, *et al.*, "Experimental comparison of two quantum computing architectures," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3305–3310, Mar. 28, 2017. DOI: 10.1073/pnas.1618020114. [Online]. Available: https://doi.org/10.1073/pnas.1618020114.

[22] Y. Wang, S. Crain, C. Fang, *et al.*, "High-fidelity two-qubit gates using a microelectromechanical-system-based beam steering system for individual qubit addressing," *Phys. Rev. Lett.*, vol. 125, p. 150 505, 15 Oct. 2020. DOI: 10.1103/PhysRevLett.125.150505. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.125.150505.

[23] C. Monroe, D. M. Meekhof, B. E. King, *et al.*, "Demonstration of a Fundamental Quantum Logic Gate," *Physical Review Letters*, vol. 75, no. 25, pp. 4714–4717, Dec. 18, 1995. DOI: 10.1103/PhysRevLett.75.4714. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.75.4714.

[24] J. Kim, T. Chen, J. Whitlow, *et al.*, "Hardware design of a trapped-ion quantum computer for software-tailored architecture for quantum co-design (staq) project," in *Quantum 2.0*, Optical Society of America, 2020, QM6A–2.

[25] S. Debnath, N. M. Linke, C. Figgatt, *et al.*, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, no. 7614, pp. 63–66, Aug. 1, 2016, ISSN: 1476-4687. DOI: 10.1038/nature18648. [Online]. Available: https://doi.org/10.1038/nature18648.

[26] B. Lekitsch, S. Weidt, A. G. Fowler, *et al.*, "Blueprint for a microwave trapped ion quantum computer," *Science Advances*, vol. 3, no. 2, e1601540, DOI: 10.1126/sciadv.1601540. [Online]. Available: https://doi.org/10.1126/sciadv.1601540.

[27] L. Postler, S. Heußen, I. Pogorelov, *et al.*, *Demonstration of fault-tolerant universal quantum gate operations*, 2021. DOI: 10.48550/ARXIV.2111.12654. [Online]. Available: https://arxiv.org/abs/2111.12654.

[28] A. Sørensen and K. Mølmer, "Quantum Computation with Ions in Thermal Motion," *Physical Review Letters*, vol. 82, no. 9, pp. 1971–1974, Mar. 1, 1999. DOI: 10.1103/PhysRevLett.82.1971. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.82.1971.

[29] Z. Jia, S. Huang, M. Kang, *et al.*, "Angle-robust two-qubit gates in a linear ion crystal," *Physical Review A*, vol. 107, no. 3, p. 032 617, Mar. 23, 2023. DOI: 10.1103/PhysRevA.107.032617. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.107.032617.

[30] P. H. Leung, K. A. Landsman, C. Figgatt, *et al.*, "Robust 2-qubit gates in a linear ion crystal using a frequency-modulated driving force," *Phys. Rev. Lett.*, vol. 120, p. 020 501, 2 Jan. 2018. DOI: 10.1103/PhysRevLett.120.020501. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.120.020501.

[31] B. P. Ruzic, M. N. H. Chow, A. D. Burch, *et al.* "Frequency-robust Mølmer-Sørensen gates via balanced contributions of multiple motional modes." arXiv: 2210.02372 [quant-ph]. (Oct. 5, 2022), [Online]. Available: http://arxiv.org/abs/2210.02372, preprint.

[32] L. Riesebos, B. Bondurant, J. Whitlow, *et al.*, "Modular software for real-time quantum control systems," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 545–555. DOI: 10.1109/QCE53715.2022.00077.

[33] L. Riesebos, "Software architectures for real-time quantum control systems," Ph.D. dissertation, Duke University, 2022.

[34] K. Svore, A. Geller, M. Troyer, *et al.*, "Q#: Enabling scalable quantum computing and development with a high-level dsl," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, ser. RWDSL2018, Vienna, Austria: Association for Computing Machinery, 2018, ISBN: 9781450363556. DOI: 10.1145/3183895.3183901. [Online]. Available: https://doi.org/10.1145/3183895.3183901.

[35] A. W. Cross, A. Javadi-Abhari, T. Alexander, *et al.*, *Openqasm 3: A broader and deeper quantum assembly language*, 2021. arXiv: 2104.14722 [quant-ph].

[36] M. S. ANIS, H. Abraham, AduOffei, *et al.*, *Qiskit: An open-source framework for quantum computing*, 2021. DOI: 10.5281/zenodo.2573505.

[37] X. Fu, J. Yu, X. Su, *et al.*, "Quingo: A programming framework for heterogeneous quantum-classical computing with nisq features," *arXiv preprint arXiv:2009.01686*, 2020.

[38] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018, ISSN: 2521-327X. DOI: 10.22331/q-2018-01-31-49. [Online]. Available: https://doi.org/10.22331/q-2018-01-31-49.

[39] C. Developers, *Cirq*, version v0.10.0, See full list of authors on Github: https://github.com/qu
Mar. 2021. DOI: 10.5281/zenodo.4586899. [Online]. Available: https://doi.org/10.5281/zenodo.4586899.

[40] F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[41] C. Ryan-Anderson, J. G. Bohnet, K. Lee, *et al.*, "Realization of real-time fault-tolerant quantum error correction," *Phys. Rev. X*, vol. 11, p. 041 058, 4 Dec. 2021. DOI: 10.1103/PhysRevX.11.041058. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.11.041058.

[42] Y. Wang, Y. Li, Z.-q. Yin, *et al.*, "16-qubit ibm universal quantum computer can be fully entangled," *npj Quantum information*, vol. 4, no. 1, pp. 1–6, 2018.

[43] I. Pogorelov, T. Feldker, C. D. Marciniak, *et al.*, "Compact ion-trap quantum computing demonstrator," *PRX Quantum*, vol. 2, p. 020 343, 2 Jun. 2021. DOI: 10.1103/PRXQuantum.2.020343. [Online]. Available: https://link.aps.org/doi/10.1103/PRXQuantum.2.020343.

[44] R. Acharya, I. Aleiner, R. Allen, *et al.*, *Suppressing quantum errors by scaling a surface code logical qubit*, 2022. DOI: 10.48550/ARXIV.2207.06431. [Online]. Available: https://arxiv.org/abs/2207.06431.

[45] G. Pagano, A. Bapat, P. Becker, *et al.*, "A quantum approximate optimization algorithm in a trapped-ion quantum simulator," en, Oct. 2020. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=928237.

[46] M. Blok, V. Ramasesh, T. Schuster, *et al.*, "Quantum information scrambling in a superconducting qutrit processor," *arXiv preprint arXiv:2003.03307*, 2020.

[47] V. Negnevitsky, "Feedback-stabilised quantum states in a mixed-species ion system," Ph.D. dissertation, ETH Zurich, 2018.

[48] P. Maunz, J. Mizrahi, and J. Goldberg, *Ioncontrol v. 1.0, version 00*, Jul. 2016. [Online]. Available: https://www.osti.gov/biblio/1326630.

[49] X. Fu, L. Riesebos, M. Rol, *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.

[50] C. A. Ryan, B. R. Johnson, D. Ristè, *et al.*, "Hardware for dynamic quantum computing," *Review of Scientific Instruments*, vol. 88, no. 10, p. 104 703, 2017.

[51] G. Kasprowicz, P. Kulik, M. Gaska, *et al.*, "Artiq and sinara: Open software and hardware stacks for quantum physics," in *OSA Quantum 2.0 Conference*, Optical Society of America, 2020, QTu8B.14. DOI: 10.1364/QUANTUM.2020.QTu8B.14. [Online]. Available: http://www.osapublishing.org/abstract.cfm?URI=QUANTUM-2020-QTu8B.14.

[52] J. H. Nielsen, M. Astafev, W. H. Nielsen, *et al.*, *Qcodes/qcodes: V0.30.0.dev0*, version v0.30.0.dev0, Oct. 2021. DOI: 10.5281/zenodo.5595929. [Online]. Available: https://doi.org/10.5281/zenodo.5595929.

[53] M. Rol, C. Dickel, S.Asaad, *et al.*, *Pycqed_py3*, version v0.2, Dec. 2019. DOI: 10.5281/zenodo.3574563. [Online]. Available: https://doi.org/10.5281/zenodo.3574563.

[54] D. C. McKay, T. Alexander, L. Bello, *et al.*, *Qiskit backend specifications for openqasm and openpulse experiments*, 2018. arXiv: 1809.03452 [quant-ph].

[55] L. Riesebos, X. Fu, A. Moueddenne, *et al.*, "Quantum accelerated computer architectures," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2019, pp. 1–4.

[56] T. Nguyen, A. Santana, T. Kharazi, *et al.*, "Extending c++ for heterogeneous quantum-classical computing," *arXiv preprint arXiv:2010.03935*, 2020.

[57] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," *arXiv preprint arXiv:1608.03355*, 2016.

[58] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[59] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[60] X. Fu, M. A. Rol, C. C. Bultink, *et al.*, "An experimental microarchitecture for a superconducting quantum processor," in *Proceedings of the 50th Annual*

IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-50 '17, Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 813–825, ISBN: 9781450349529. DOI: 10.1145/3123939.3123952. [Online]. Available: https://doi.org/10.1145/3123939.3123952.

[61] L. Riesebos, B. Bondurant, and K. R. Brown, *Duke artiq extensions (dax)*, https://gitlab.com/duke-artiq/dax, 2021. [Online]. Available: https://gitlab.com/duke-artiq/dax.

[62] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical review letters*, vol. 106, no. 18, p. 180 504, 2011.

[63] T. J. Proctor, A. Carignan-Dugas, K. Rudinger, *et al.*, "Direct randomized benchmarking for multiqubit devices," *Phys. Rev. Lett.*, vol. 123, p. 030 503, 3 Jul. 2019. DOI: 10.1103/PhysRevLett.123.030503. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.123.030503.

[64] J. M. Epstein, A. W. Cross, E. Magesan, *et al.*, "Investigating the limits of randomized benchmarking protocols," *Physical Review A*, vol. 89, no. 6, p. 062 321, 2014.

[65] R. Blume-Kohout, J. K. Gamble, E. Nielsen, *et al.*, *Robust, self-consistent, closed-form tomography of quantum logic gates on a trapped ion qubit*, 2013. DOI: 10.48550/ARXIV.1310.4492. [Online]. Available: https://arxiv.org/abs/1310.4492.

[66] Erik, L. Saldyt, Rob, *et al.*, *Pygstio/pygsti: Version 0.9.10*, version v0.9.10, Oct. 2021. DOI: 10.5281/zenodo.5546759. [Online]. Available: https://doi.org/10.5281/zenodo.5546759.

[67] R. Schmied, "Quantum state tomography of a single qubit: Comparison of methods," *Journal of Modern Optics*, vol. 63, no. 18, pp. 1744–1758, 2016.

[68] *Xilinx kc705*. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[69] L. Riesebos and K. R. Brown, "Functional simulation of real-time quantum control software," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022.

[70] *Coverage.py*. [Online]. Available: `https://github.com/nedbat/coveragepy`.

[71] P. T. Fisk, M. J. Sellars, M. A. Lawn, *et al.*, "Accurate measurement of the 12.6 ghz" clock" transition in trapped/sup 171/yb/sup+/ions," *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 44, no. 2, pp. 344–354, 1997.

[72] L. Riesebos, B. Bondurant, and K. R. Brown, "Universal graph-based scheduling for quantum systems," *IEEE Micro*, vol. 41, no. 5, pp. 57–65, 2021. DOI: `10.1109/MM.2021.3094968`.

[73] V. Schäfer, C. Ballance, K. Thirumalai, *et al.*, "Fast quantum logic gates with trapped-ion qubits," *Nature*, vol. 555, no. 7694, pp. 75–78, 2018.

[74] N. Wittler, F. Roy, K. Pack, *et al.*, "Integrated tool set for control, calibration, and characterization of quantum devices applied to superconducting qubits," *Phys. Rev. Applied*, vol. 15, p. 034 080, 3 Mar. 2021. DOI: `10.1103/PhysRevApplied.15.034080`. [Online]. Available: `https://link.aps.org/doi/10.1103/PhysRevApplied.15.034080`.

[75] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, 2017. arXiv: `1608.03355 [quant-ph]`.

[76] J. Kelly, P. O'Malley, M. Neeley, *et al.*, *Physical qubit calibration on a directed acyclic graph*, 2018. arXiv: `1803.03226 [quant-ph]`.

[77] T. P. Harty, D. T. C. Allcock, C. J. Ballance, *et al.*, "High-Fidelity Preparation, Gates, Memory, and Readout of a Trapped-Ion Quantum Bit," *Physical Review Letters*, vol. 113, no. 22, p. 220 501, Nov. 24, 2014. DOI: `10.1103/PhysRevLett.113.220501`. [Online]. Available: `https://link.aps.org/doi/10.1103/PhysRevLett.113.220501`.

[78] P. Krantz, M. Kjaergaard, F. Yan, *et al.*, "A Quantum Engineer's Guide to Superconducting Qubits," *Applied Physics Reviews*, vol. 6, no. 2, p. 021 318, Jun. 1, 2019, ISSN: 1931-9401. DOI: 10.1063/1.5089550. arXiv: 1904.06560 [cond-mat, physics:physics, physics:quant-ph]. [Online]. Available: http://arxiv.org/abs/1904.06560.

[79] K. Singh, C. E. Bradley, S. Anand, *et al.*, "Mid-circuit correction of correlated phase errors using an array of spectator qubits," *Science*, vol. 380, no. 6651, pp. 1265–1269, Jun. 23, 2023, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.ade5337. arXiv: 2208.11716 [cond-mat, physics:physics, physics:quant-ph] [Online]. Available: http://arxiv.org/abs/2208.11716.

[80] R. Bowler, U. Warring, J. W. Britton, *et al.*, "Arbitrary waveform generator for quantum information processing with trapped ions," *Review of Scientific Instruments*, vol. 84, no. 3, p. 033 108, Mar. 20, 2013, ISSN: 0034-6748. DOI: 10.1063/1.4795552. [Online]. Available: https://doi.org/10.1063/1.4795552.

[81] L. Egan, D. M. Debroy, C. Noel, *et al.*, "Fault-tolerant control of an error-corrected qubit," *Nature*, vol. 598, no. 7880, pp. 281–286, Oct. 1, 2021, ISSN: 1476-4687. DOI: 10.1038/s41586-021-03928-y. [Online]. Available: https://doi.org/10.1038/s41586-021-03928-y.

[82] N. Khammassi, R. W. Morris, S. Premaratne, *et al.*, *A scalable microarchitecture for efficient instruction-driven signal synthesis and coherent qubit control*, 2022. arXiv: 2205.06851 [quant-ph].

[83] S. M. Clark, D. Lobser, M. C. Revelle, *et al.*, "Engineering the quantum scientific computing open user testbed," *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–32, 2021. DOI: 10.1109/TQE.2021.3096480.

[84] D. S. Lobser, J. W. Van Der Wall, and J. D. Goldberg, "Performant coherent control: Bridging the gap between high- and low-level operations on hardware," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 320–330. DOI: 10.1109/QCE53715.2022.00053.

[85] *Zynq ultrascale+ rfsoc data sheet: Dc and ac switching characteristics*, DS926, Rev. 1.12, Advanced Micro Devices, Inc., May 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ds926-zynq-ultrascale-plus-rfsoc.

[86] P. Richerme, M. C. Revelle, C. G. Yale, *et al.*, "Quantum Computation of Hydrogen Bond Dynamics and Vibrational Spectra," *The Journal of Physical Chemistry Letters*, vol. 14, no. 32, pp. 7256–7263, Aug. 17, 2023. DOI: 10.1021/acs.jpclett.3c01601. [Online]. Available: https://doi.org/10.1021/acs.jpclett.3c01601.

[87] S. Majumder, C. G. Yale, T. D. Morris, *et al.*, "Characterizing and mitigating coherent errors in a trapped ion quantum processor using hidden inverses," *Quantum*, vol. 7, p. 1006, May 15, 2023. DOI: 10.22331/q-2023-05-15-1006. [Online]. Available: https://quantum-journal.org/papers/q-2023-05-15-1006/.

[88] R. Shaffer, H. Ren, E. Dyrenkova, *et al.*, "Sample-efficient verification of continuously-parameterized quantum gates for small quantum processors," *Quantum*, vol. 7, p. 997, May 4, 2023. DOI: 10.22331/q-2023-05-04-997. [Online]. Available: https://quantum-journal.org/papers/q-2023-05-04-997/.

[89] C. Campbell, F. T. Chong, D. Dahl, *et al.*, *Superstaq: Deep optimization of quantum programs*, 2023. arXiv: 2309.05157 [quant-ph].

[90] C. K. Lam, S. Maka, D. Nadlinger, *et al.*, *Combining processing throughput, low latency and timing accuracy in experiment control*, 2021. arXiv: 2111.15290 [physics.ins-det].

[91] *Zynq 7000 soc technical reference manual*, UG585, Rev. 1.14, Advanced Micro Devices, Inc., Jun. 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug585-zynq-7000-SoC-TRM.

[92] *Embeddedsw*, Xilinx. [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/3728f546f178a1bcd91cf6efc9f8921447846cec/lib/sw_apps/zynqmp_fsbl.

[93] *Zynq ultrascale+ device technical reference manual*, UG1085, Rev. 2.4, Advanced Micro Devices, Inc., Dec. 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm.

[94] B. Bondurant, *Bare-metal rust on zynqmp*. [Online]. Available: https://gitlab.com/duke-artiq/zynqmp-rs.

[95] *Zynq ultrascale+ mpsoc data sheet: Dc and ac switching characteristics*, DS925, Rev. 1.25, Advanced Micro Devices, Inc., Aug. 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ds925-zynq-ultrascale-plus.

[96] *Arm architecture reference manual for a-profile architecture*, ARM DDI 0487K.a, 2024.

[97] *Arm cortex-a53 mpcore processor, technical reference manual*, ARM DDI 0500J, Revision r0p4, 2024.

[98] *Zynq-rs*. [Online]. Available: https://git.m-labs.hk/M-Labs/zynq-rs.

[99] *Zcu111 evaluation board user guide*, UG1271, Rev. 1.4, Advanced Micro Devices, Inc., Apr. 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1271-zcu111-eval-bd.

[100] *Zynq ultrascale+ devices register reference*, UG1087, Rev. 1.9, Advanced Micro Devices, Inc., May 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1087-zynq-ultrascale-registers.

[101] *Jesd 21-c section title: Annex l: Serial presence detect (spd) for ddr4 sdram modules*, JEDEC, Nov. 2020. [Online]. Available: https://www.jedec.org/standards-documents/docs/spd412l-6?destination=node/9007.

[102] rust-osdev, *Linked-list-allocator*. [Online]. Available: https://github.com/rust-osdev/linked-list-allocator.

[103] C. Monroe, R. Raussendorf, A. Ruthven, *et al.*, "Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects," *Phys-*

ical Review A, vol. 89, no. 2, p. 022 317, Feb. 13, 2014. DOI: 10.1103/PhysRevA.89.022317. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.89.022317.

[104] D. Kielpinski, C. Monroe, and D. J. Wineland, "Architecture for a large-scale ion-trap quantum computer," *Nature*, vol. 417, no. 6890, pp. 709–711, Jun. 1, 2002, ISSN: 1476-4687. DOI: 10.1038/nature00784. [Online]. Available: https://doi.org/10.1038/nature00784.

[105] C. Zhou, P. Lu, M. Praquin, *et al.*, "Realizing all-to-all couplings among detachable quantum modules using a microwave quantum state router," *npj Quantum Information*, vol. 9, no. 1, p. 54, Jun. 6, 2023, ISSN: 2056-6387. DOI: 10.1038/s41534-023-00723-7. [Online]. Available: https://doi.org/10.1038/s41534-023-00723-7.