qRV32

RISC-V 32-bit instruction set extension to address the control of diamond qubits

Master's Thesis Report

Jacopo Costantini

Delft University of Technology

TU Delft

qRV32

RISC-V 32-bit instruction set extension to address the control of diamond qubits

by

Jacopo Costantini

To obtain the degree of Master of Science at the Delft University of Technology.
To be defended publicly on Thursday, August 24th, at 1.30 PM

Student number:	5610893	
Project Duration:	September 2022 - July 2023	
Thesis committee:	Prof. dr. ir. J.S.S.M. Wong	TU Delft, supervisor
	Dr. ir. F. Sebastiano	Tu Delft, QuTech

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

Cover: Dall-E generated picture

TU Delft

Preface

This work is the end of my academic journey, at least for now. Five years of struggles and happy moments that changed completely who I am.

During this project, I had the luck to work with amazing people, like Stephan Wong, Matti Dreef, and Folkert de Ronde. I wish you all the best.

Most importantly, I would like to thank my family, which has always made sacrifices for me, and my friends, that have supported me.

Jacopo Costantini
Delft, August 2023

Abstract

RISC-V is an open-source Instruction Set Architecture that offers a simple, modular, and scalable design. Its extensions allow for customization and optimization based on specific execution workloads. One of these workloads could be quantum computing, which exploits the concepts of superposition and entanglement to manipulate qubits and perform computations that would be infeasible for classical computers. The customization offered by RISC-V presents a remarkable opportunity to develop specialized architectures that can efficiently address the execution of quantum algorithms, bridging the gap between classical and quantum computation.

In this thesis work, a RISC-V 32-bit instruction set extension called qRV32 is developed to address the control of diamond qubits, based on an existing QISA. The architecture defines the encoding syntax for the machine-level instructions and the exchange protocol for control and data in the system. Accordingly to this specification, the hardware of a control core processing the ISE has been designed. Custom functional units and necessary peripherals have been added to the base core CV32E40P in order to implement the desired control functionalities.

The thesis also proposed additional work to ease the complete design and functionality of the system. In particular, an assembler targeting qRV32 has been developed, enabling the automated translation of assembly instructions to machine-level code. Furthermore, an experimental model is developed to evaluate the parallelism of the system.

The resulting architecture is eventually tested and evaluated. Software simulations are used to test the functionality of the control core and the custom components. Eventually, a simplified version of the model is used to estimate the parallelism of the core, which can control 23605 network nodes when operating at $f_{clk} = 55MHz$.

Contents

1 Introduction

Quantum computing, a revolutionary field of study at the intersection of physics and computer science, has garnered significant attention and excitement in recent years. Unlike classical computers that operate based on bits, quantum computers harness the fundamental principles of quantum mechanics to process information using quantum bits, or *qubits*.

The journey of quantum computing can be traced back to the early 1980s when physicist Richard Feynman envisioned the possibility of harnessing quantum systems to perform calculations more efficiently than classical computers [17]. In the mid-1990s, Peter Shor developed a groundbreaking quantum algorithm for integer factorization [41], highlighting the immense computational advantage of quantum computers in certain areas, such as cryptography. In 1998, Isaac Chuang and Neil Gershenfeld at the IBM Almaden Research Center demonstrated the first 2-qubit quantum computer [7]. IBM's 127-qubit Eagle [23], Rigetti's 80-qubit Aspen-M-2 [8], and Google's 53-qubits Sycamore [27] are the three most advanced quantum computers currently available.

1.1. Thesis Project

Despite the fact that these computers employ qubits to perform quantum operations, control electronics play a vital role in manipulating and maintaining the delicate quantum states, ensuring coherence and enabling complex quantum operations. A collection of analog-operating classical electronics, such as lasers and microwave generators, and a layer of digital control logic, that manages the electronics, need to be included in a quantum computer.

This thesis project aims to build a **prototype of the global control electronics** required by diamond-based quantum computers, guaranteeing flexibility through the use of an instruction-set architecture (ISA) that captures the actions of the digital controller without putting any restrictions on the underlying hardware. To ensure compatibility with the most cutting-edge open hardware currently on the market, the Global Controller (GC) control core was chosen to be implemented as a **RISC-V** 32-bit core.

1.2. Research Question

The research question can be formulated as the following:

> **Can the control of a diamond-based quantum computer be addressed using the inherent extendable architecture structure of the RISC-V ISA?**

In order to answer the question, the following goals in the development of the global controller have been set:

1. Define custom RISC-V instructions based on the available Quantum ISA (QISA)
2. Define a communication protocol between GC and the rest of the control plane
3. Implement Functional Units to address the execution the said instructions

1.3. Methodology

A number of actions must be carried out in order to accomplish the objectives. This will guarantee a thorough understanding of the functionality of quantum computers based on diamond NV qubits, which the core must control, and of the scalability of the RISC-V ISA, which the core will implement.

1. **Understanding of the existing system definition and requirements**
   A control plane has already been initially defined. At the moment of starting the thesis, a system simulator and a compiler to address the Global Controller have been developed.
2. **Profiling of the existing QISA and creation of RISC-V-compliant encoding**
   QISA instructions, defined at assembly-level, model the operations to be performed by the quantum computer. They are then mapped to machine-level following the RISC-V guidelines.
3. **Design and implementation of the Global Controller core and SoC**
   From an existing RV 32-bit core, functional units and communication interface must be designed to create the GC hardware.
4. **Development of a custom assembler**
   An assembler must be developed to translate the outputs of the compiler into the novel qRV32 ISA format.
5. **Design validation and evaluation**
   Simulations must be used to verify the design of the controller. Its control ability are then evaluated.

1.4. Report Overview

The structure of this thesis is now presented. Chapter 2 delves into Quantum Information Theory, NV Center Quantum Computers, and the RISC-V Instruction Set Architecture, providing crucial background knowledge to support our research. Transitioning to Chapter 3, our focus shifts to the modeling of the control plane, where we thoroughly explore the concept of parallelism. Advancing further, Chapter 4 takes a closer look at the qRV32 Instruction Set Extension. As we proceed to Chapter 5, we showcase the hardware implementation, which includes conducting a survey of 32-bit RISC-V cores, designing the Global Controller and the System on Chip. Within Chapter 6, we delve into the intricacies of the qRV32 Assembler and the development steps involved. Subsequently, Chapter 7 unfolds the results, encompassing software simulations and model evaluations. Ultimately, Chapter 8 brings the thesis to a conclusion, summarizing key findings and presenting potential future research directions.

2

Background information

The ideas of quantum computing and qubits were introduced in the preceding chapter, along with a mention of the system specification of the quantum computer stack. Given that they are essential for fully comprehending the thesis work, these topics are presented in this chapter.

Section 2.1 presents an overview of Quantum Information Theory. Section 2.2 focuses on NV center quantum computers, discussing their fundamental principles and the analog (Sec.2.2.2) and digital (Sec.2.2.3) control required. Eventually, Section 2.3 delves into the essential aspects of RISC-V, including its features and properties.

2.1. Quantum Information Theory

Quantum information theory investigates the principles and limitations of transmitting classical and quantum information over quantum channels, utilizing concepts from quantum mechanics and information theory to develop advanced communication protocols and quantum technologies [14].

A brief review of quantum information theory will be provided in this section. It will introduce quantum bits (qubits) and describe how they differ from conventional bits. The definition of Diamond Quantum Computers and their purpose will also be covered in this section. The notions of quantum algorithms and quantum gates will be introduced at the end of the section.

2.1.1. Quantum bit

Qubits, the fundamental units of quantum information, lie at the heart of quantum computing. In contrast to traditional bits, which represent information as either 0 or 1, qubits exploit the principles of quantum mechanics to exist in a linear superposition of states [26]. Qubits are represented by the *braket* notation, introduced by P. Dirac and commonly used in quantum mechanics.

The two orthogonal states are represented as $|0\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, while a superposition state ψ is described by Equation 2.1.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \equiv \begin{pmatrix} \alpha \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.1}$$

In the equation, α and β are complex numbers called *probabilistic amplitudes* [26]. They represent the probability of the state collapsing into one of the orthogonal states of the measurement basis. In particular, during measurement, the qubit always collapses into the $|0\rangle$ or the $|1\rangle$ state, with probabilities $|\alpha|^2$ and $|\beta|^2$ respectively [26]. Consequently $|\alpha|^2 + |\beta|^2 = 1$.

Moreover, the state of a qubit can be visually represented using the concept of the *Bloch sphere* [6], depicted in Figure 2.1. The state of the qubit, characterized by the polar angle $\theta \in [0, \pi]$ and the azimuthal angle $\phi \in [0, 2\pi]$, can be mathematically expressed by Equation 2.2.

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\cos\frac{\theta}{2}|1\rangle \tag{2.2}$$

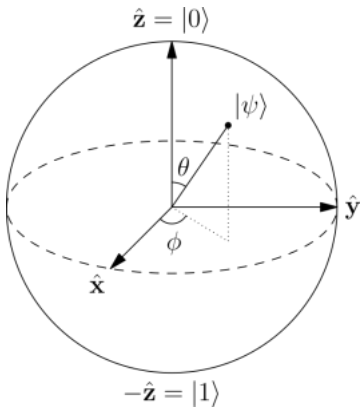


Figure 2.1: Bloch phere [6] of a qubit.

This visualization is particularly useful to grasp the concept of single-qubit operations, as they can be modeled as rotations along the sphere over a specified axis.

2.1.2. Quantum gates

Qubits are the basic unit of information in a quantum system. The manipulation of qubits states is performed by *quantum gates* (**unitary, reversible operations**) on any number of qubits. Using the vector notation, quantum gates can be represented by a matrix. These matrixes can be used to calculate the state of the qubit after the quantum gate by multiplying them with the quantum state vector, producing an output vector.

Single Qubits

A Bloch vector is a unit vector $[cos(\phi)sin(\theta), sin(\phi)sin(\theta), cos(\theta)]$ used in quantum mechanics to represent the state of a two-level quantum system, such as a qubit, on a Bloch sphere [29]. A quantum gate applied on a single qubit 'rotates' the Bloch vector at some angle along a specified axis.

For example, the commonly used **Pauli-X**, **-Y** and **-Z** gates (Figure 2.2) perform a rotation of π radians over the specified axis, equivalent to flipping the bit around the sphere over their respective axis [31]. The possibility for intricate and interconnected quantum computations increases as we move from the domain of single-qubit quantum gates to multiple-qubit quantum gates.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

(a) Pauli-X **(b)** Pauli-Y **(c)** Pauli-Z

Figure 2.2: Pauli-X, Pauli-Y and Pauli-Z gates matrices

Multiple Qubits

While a single-qubit rotation is completely independent of any other external condition, it is also possible to create rotations based on the state of a second qubit. These multi-qubit gates are called *controlled gates*. The prototypical controlled operation is the **controlled-NOT** (Figure 2.3a), a quantum gate with two input qubits, known as the control qubit and target qubit [32].

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

(a) Controlled-X **(b)** Controlled-y **(c)** Controlled-Z

Figure 2.3: Controlled-X (CNOT), Controlled-Y and Controlled-Z gates matrices

Entanglement
Quantum computing's fundamental qubit entanglement phenomenon is crucial to achieving computational advantages over classical systems [1]. When two or more qubits exhibit a quantum correlation that makes their states interdependent and entangled, this is referred to as entanglement.

Entangled qubits have a combined state that **can not be broken down into the individual states** of the individual qubits. The concurrence is an entanglement measure [9] of a multi-qubit state (Eq. 2.3), and it is useful as a separability criterion. The value of the concurrence $C \in [0, 1]$ can be used to determine if two qubits are maximally entangled and can be calculated with equation 2.4. Maximal entanglement is reached when the concurrence is 1. The probability amplitudes are indicated by the parameters $\alpha_{00}, \alpha_{01}, \alpha_{10}, \alpha_{11}$.

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{2.3}$$

$$C(|\psi\rangle) \equiv 2|\alpha_{00}\alpha_{11} - \alpha_{01}\alpha_{10}| \tag{2.4}$$

In particular, characteristics of a maximally entangled state, such as $|\psi\rangle = |01\rangle + |10\rangle$, include nonlocality, which allows measurements on one qubit to instantly change the state of another, regardless of their spatial separation. In fact, measuring the first qubit will cause it to collapse onto either the $|0\rangle$ or $|1\rangle$ state; however, if the second qubit is measured afterward, the outcome will be identical to the first.

2.1.3. Qubits operations

A quantum system can change the state of a qubit or a group of qubits by employing either unitary or non-unitary gates. Unitary gates are reversible and preserve the information in the quantum system. On the contrary, non-unitary gates are irreversible and destroy a portion of the current qubit state. This section will present operations that involve non-unitary gates as they allow us to measure and reset qubits.

Initialization
An essential step in quantum computing is qubit initialization, which involves setting up qubits in specific quantum states to facilitate subsequent quantum operations. The goal of the qubit initialization process is to set up the quantum system so that the qubits are in a superposition of the known states $|0\rangle$ and $|1\rangle$. Due to the fact that initialization is an irreversible process that forces the qubit into a specific state regardless of its previous state, it is not a unitary quantum gate.

For qubit initialization, a number of methods have been developed, including electron spin resonance, microwave-based approaches and optical pumping [18]. These techniques use specific gate sequences and meticulously controlled external fields to manipulate the qubits into the desired initial state. For performing trustworthy quantum computations and obtaining precise results, the capacity to initialize qubits accurately is crucial.

Measurement
The measurement operation is a fundamental component of quantum computing that allows the extraction of information from qubits. Qubit measurement entails performing a measurement in a specific basis, typically the computational basis represented by $|0\rangle$ and $|1\rangle$, in order to determine the state of a qubit. With a probability determined by the coefficients in the superposition, the measurement process collapses the entangled or superposition state of the qubit into one of the basis states. The state of the qubit can be inferred probabilistically from various measurement results.

Different measurement methods have been developed, including non-projective methods like weak measurement [25] and quantum non-demolition measurement [43] as well as projective methods [42] using quantum logic gates. For obtaining trustworthy results and validating the outcomes of quantum computations, accurate qubit measurement is essential.

2.2. NV Center Quantum Computers

Different technology for quantum computations have been developed over the past decades, such as silicon's phosphorous qubits [20] or superconducting qubits. The latter technology was made recently scalable by Google [27], however, it shows relatively short coherence times [13], while **nitrogen-**

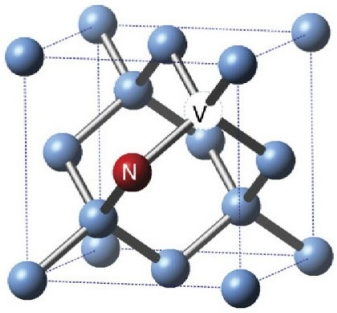


Figure 2.4: NV center representation [28]. N identifies the nitrogen atom and V the vacancy.

vacancy (NV) centers are known for their longer coherence times [33]. This technology is the focus of this thesis, and it will be presented in this section.

2.2.1. Qubits

The optimal condition to build a quantum computer is to ensure system isolation and individual qubit control, a scenario that happens in an atom trap [24]. A diamond Nitrogen-Vacancy center is formed in a diamond crystal from which two carbon atoms are missing, 1 of them replaced with a Nitrogen atom (Figure 2.4). This point defect center can be compared to an atom in a trap because the defect-free isotopic 12C diamond crystal, with its large bandgap of 5.4 eV, has properties resembling those of a vacuum [35]. The negatively charged NV− state can be artificially created by capturing an additional electron from the environment using lasers.
Specifically, one of the qubit states is associated with the **electron spin** of the NV center, while the additional **Carbon-13 isotopes** can also be used as qubits. The e^- qubit is easily controllable by electromagnetic fields and photons, but it is also susceptible to noise; by contrast, ^{13}C qubits are less susceptible to noise and decoherence but cannot be controlled directly. Therefore, from an architectural point of view, the first ones are suitable to perform operations, while the latter model data memory.

2.2.2. Analog control

Direct control of the electron spin qubit in an NV center can be achieved using **electromagnetic fields and photons** [40], providing a pathway to realize the quantum operations described in Sections 2.1.2 and 2.1.3.

Manipulation of the NV center's single-qubit rotations, particularly the e^- qubit, can be achieved through the magnetic field. The ^{13}C qubit can also be rotated in controlled and uncontrolled ways, as demonstrated by [4]. Initialization of the qubit is achieved through charge pumping [18] in which the NV center is continuously hit with photons. Measurement is achieved in a process where the absorption of photons indicates the qubit state. Moreover, photons can be employed to entangle two distinct NV centers by using electron-photon pair creation [15].

This plethora of ways to interface with the qubits creates the foundation of the *Quantum-to-Classical* layer, which employs various analog actuators for its implementation. One such actuator is the Microwave Arbitrary Waveform Generator, which generates micro- or radio-frequency waves for the rotation of electron and carbon nuclei. Additionally, lasers and switches are utilized to illuminate the diamond and deliver photons, while a photon detector counts the emitted photons and stores the count value in a register.

These analog actuators pave the way for exploring digital control techniques, unlocking new possibilities for enhanced precision and scalability in qubit operations.

2.2.3. Digital control

The analog control hardware described must be coordinated in order to ensure the correct execution of a quantum program. In order to achieve this, the layered digital control architecture proposed in [40] and refined in [16] is employed in this work. This architecture consists of two layers: the lower layer comprising *Local Controllers* (LCs), dedicated to individual NV centers, and the upper layer comprising of single a *Global Controller* (GC) responsible for the entire system. Figure 2.5 provides a visual

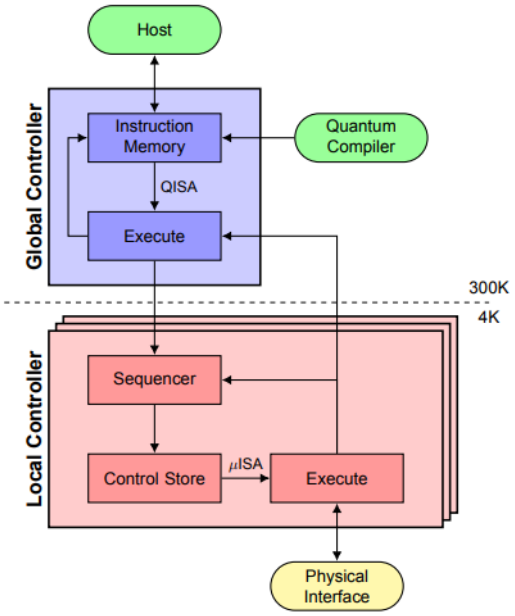


Figure 2.5: Overview of the envisioned hardware architecture [16].

depiction of this hierarchical structure.

The **local controller** plays a vital role in activating and controlling the underlying analog electronics used for NV center operations. It employs a *Local Micro-instruction Set Architecture* (uQISA) [16] and comprehends registers and ports used to control the underlying Quantum-to-Classical layer. A detailed design of such controllers is out of the scope of this work.

By leveraging the localized control, NV centers across the network have the potential to operate in parallel when addressed by the Global Controller. Furthermore, their entanglement scheme based on photons facilitates on-chip interaction between neighboring NV centers.

Global Controller

Object of this thesis work and main component of the control plane, the Global Controller is responsible for the execution of quantum algorithms.

The Global Controller utilizes a *Quantum Instruction Set Architecture* (QISA) to comprehensively handle algorithm execution. This architecture incorporates custom instructions that explicitly capture all required parameters. An example of such a structured QISA can be found in [47]. Moreover, further advancements will be explored in the current study.

In addition to program execution, the GC plays a crucial role in coordinating local controllers to achieve parallelism and managing data transfers to and from the lower layers. Efficiently addressing the local controllers is essential for achieving parallel processing and ensuring seamless data flow.

2.3. RISC-V Instruction Set Architecture

RISC-V is an open and extensible instruction set architecture that has gained significant popularity in recent years due to its simplicity, modularity, and scalability. By adopting the RISC-V ISA, the project aims to leverage its flexibility and robustness to design and implement the GC, enabling efficient and effective control over the lower layers of the system. This section provides an introduction to the RISC-V ISA and its relevance in the context of this work.

2.3.1. History of RISC-V

The RISC-V instruction set architecture (ISA) has emerged as a prominent and influential open-source standard in the field of computer architecture. Its history dates back to the early 2010s when researchers at the University of Berkeley, recognized the need for an open and customizable ISA to drive innovation in processor design. The RISC-V project was initiated under the leadership of *Prof.*

Krste Asanović and *Prof. David Patterson*, resulting in the development of an ISA that offers simplicity, modularity, and extensibility [45].

The aim of creating RISC-V was to address the drawbacks of closed and proprietary ISAs, which limit access to the underlying architecture and obstruct experimentation and research. By offering an **open-source ISA**, RISC-V fosters industry-wide innovation, academic research, and collaboration. It provides a **flexible platform** for academia, business, and individuals to create, implement, and modify processors in accordance with their unique needs, enabling novel approaches and ideas to be explored without proprietary constraints. The implications are far-reaching: a wide range of applications, from embedded systems to high-performance computing, have quickly adopted and used the RISC-V ISA. Notably, the open-source nature of RISC-V has led to the establishment of the RISC-V Foundation, a consortium of organizations driving the development and standardization of the architecture [39].

Importantly, the RISC-V ISA has seen the creation of a number of illustrious processors. The University of Berkeley developed the first-ever RISC-V processor, known as the *Rocket* core, which served as the model for later improvements to the architecture [2]. On the other hand, the *SweRV EH1* core, developed by Western Digital, is recognized as one of the most performant RISC-V processors, delivering high clock frequencies and efficient execution [3].

2.3.2. Characteristics

The RISC-V ISA is designed with a modular and extensible structure, offering flexibility and customization options to meet diverse computing requirements. The base RISC-V ISA provides a **minimal set of instructions** necessary for building a functional processor. This simplicity allows for easy adoption and implementation of the architecture.

With a base integer instruction set that is subsequently expanded by optional *standard extensions*, the RISC-V ISA adopts a layered design strategy. The base set is available in four variants: 32-bit (**RV32I**), 64-bit (RV64I), 128-bit (RV128I), and embedded 32-bit (RV32E). The standard extensions include modules for integer multiplication and division [*M*], atomic instructions [*A*], floating-point operations [*F*], and more. By choosing the proper extensions, designers can modify the RISC-V instruction set to meet their unique requirements. This encourages a scalable method of processor design, where extra capability can be added as needed, without extra complexity or overhead.

One of the most intriguing characteristics of the ISA is its modularity. The architecture allows for the creation of unique extensions in addition to the standard ones, providing unequaled flexibility for particular computing workloads. The *RVV* vector extension is one such addition that adds vector processing capabilities for effective parallel data processing tasks, which are frequently utilized in scientific computing and multimedia applications [37]. The *RVC* cryptography extension is another noteworthy addition. It offers hardware acceleration for cryptographic operations, improving security-related computations in embedded devices and communication [38]. Accordingly, the goal of this thesis project is to create a RISC-V Instruction Set Extension (ISE) to handle the control of NV center diamond quantum computers.

2.4. Conclusion

In conclusion, this chapter presented the background knowledge required to comprehend the thesis work. Beginning with an explanation of quantum computing and qubits, it detailed the physics behind this technology. Quantum bits, quantum gates, and qubit operations were some of the subjects covered in the discussion of quantum information theory. Furthermore, the chapter looked into NV Center Quantum Computers, exploring the key components of qubits and the control mechanisms involved, including analog and digital control. Finally, a summary of the RISC-V Instruction Set Architecture was given, emphasizing its background and salient features. This foundation lays the groundwork for the subsequent chapters, which explore in depth how control electronics are implemented in contemporary quantum computers.

3

Modeling the Control Plane

In this chapter, we present a mathematical model aimed at determining the allocation of local controllers to global controllers within the Digital Control Layer. By developing this model, we can establish guidelines for achieving an optimized and scalable control structure that attempts to minimize idle states and maximize the overall system efficiency.

This chapter explores the modeling of the control plane, starting with the execution of instructions (Section 3.1). Time diagrams are then discussed, considering ideal cases (Section 3.2.1) and accounting for processing (Section 3.2.2) and communication (Section 3.2.3). The chapter concludes with the definition of the model (Section 3.3).

3.1. Instruction Execution

In order to develop an accurate mathematical model, the initial step involves identifying the comprehensive sequence of actions required for a complete quantum instruction execution within the layer, spanning from the Global Controller to the Local Controller. To accomplish this, a high-level flow chart illustrating the execution process is presented in this section, providing a visual roadmap. Following the flow chart, a more precise definition and profiling of each step are presented, allowing for a thorough understanding of the intricate processes involved in the execution sequence.

3.1.1. Flowchart

Figure 3.1 visualize a high-level flow chart of the execution. This representation aims to offer a **functional and hardware-agnostic** perspective, providing insights into the elaboration and communication data flow.

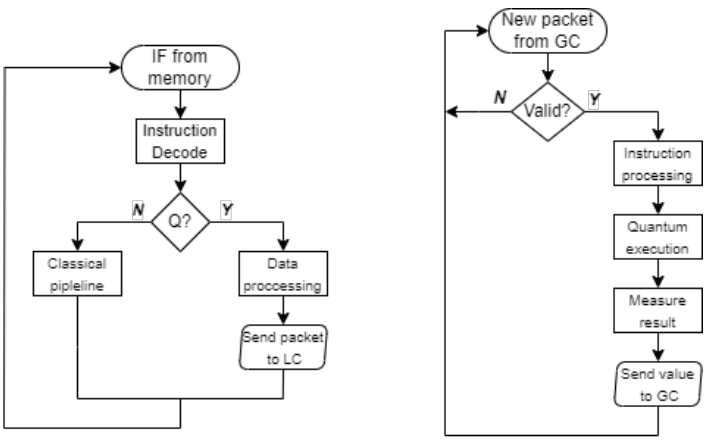


Figure 3.1: The high-level flowchart of the digital control plane. Here are represented the execution steps: on the left in the GC, while on the right in the LC

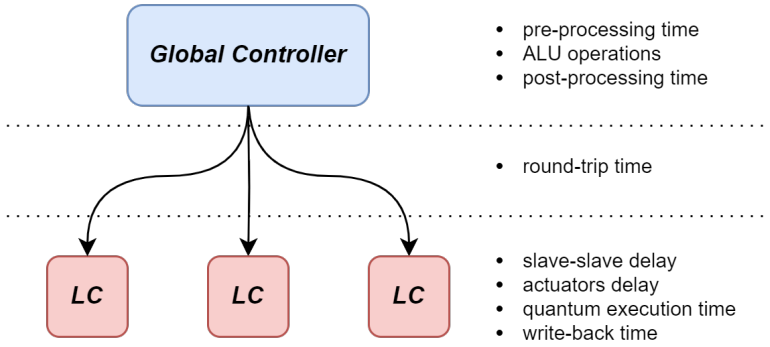


Figure 3.2: Classification of the parameters based on the origin of latency generation

3.1.2. Profiling

The provided flowchart can be used as a basis for starting a more detailed investigation of the system's execution steps. In particular, in this section, the emphasis is on determining each latency interval needed and parameterizing them uniquely. Table 3.1 presents a description and defines a symbol for each identified **parameter**, while Figure 3.2 classifies them based on the hardware components responsible for introducing latency.

Name	Description	Symbol
Pre-processing latency	Data conversion and management needed to execute the instruction(*i*)	$\alpha \cdot Tpe_G(i)$
Calculation latency	Calculation of actuator control values, based on the parameters	$\beta \cdot Tc_L(i)$
Round Trip Time (RTT)	Latency that models the master-slave round communication (to node *j*)	$\gamma \cdot RTT_C(j)$
Propagation delay	Latency that models the intra-slaves communication (from node *j* to node *k*)	$\delta \cdot Tpd_C(j,k)$
Phisical layer latency	Latency needed by the actuators (mw generator, lasers, etc.) to get activated	$\varepsilon \cdot Tact_L(i)$
Quantum execution latency	Quantum gate or qubit operations execution (measured experimentally)	$Tqex_L(i)$
Write back latency	Writing the results from the physical sensors to the appropriate registers	$\xi \cdot Twb_L$
Post-processing latency	Data management to correctly store the execution results	$\zeta \cdot Tpo_G(i)$

Table 3.1: Profiling of the execution parameters

In the above table, particular notations have been used to define the symbols:

- The coefficients $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta \in 0, 1$ are added to exclude specific variables, making the module simpler if needed.
- The index (*i*) indicates the influence of the executed instruction on the parameter value.
- The indexes (*j*,*k*) indicate the influence of the positions of the nodes on the parameter value.
- The subscripts "$_G$", "$_L$" and "$_C$" indicate the origin of latency generation, the GC, the LC, or the communication infrastructure.

Having analyzed the execution profiling of the system, we now turn our attention to the generation of time diagrams. Those are helpful to correlate the presented parameters and formulate an accurate model that captures the execution dynamics effectively.

3.2. Time Diagrams

In this section, time diagrams are generated and analyzed using the previously identified parameters. By considering different levels of idealities, incorporating additional parameters, and varying the workload of controllers with send, receive and mixed operations, we can gain valuable insights into the temporal dynamics of the system. These time diagrams provide a comprehensive and intuitive representation of the system's behavior, enabling us to optimize performance and make informed decisions for system design and resource allocation.

The aim of this analysis is to determine the maximum achievable parallelism of Local Controllers for a specific instruction (*i*). To estimate this, we create and analyze workload scenarios designed to present limitations and extreme cases, which help in identifying the boundaries of the parallelism. The focus is on the relative values of the latency parameters rather than on their representativity of real workloads.

Note that, in the study proposed, the Global Controller is constrained to address Local Controller LC#1 immediately upon completing its execution, thereby preventing any idle state. By doing this, the maximum parallelism of LCs can be estimated.

3.2.1. Ideal case

The first case accounts only for the processing time: namely send and receive processing on the GC and execution time on the LCs. Table 3.2 sums up the latency values for a workload dominated by the local execution time and with equal communication delays, expressed in clock cycles t_{clk}.

GC	*t_send*	$1 \cdot t_{clk}$
GC	*t_receive*	$1 \cdot t_{clk}$
LC	*t_execute*	$5 \cdot t_{clk}$

Table 3.2: Parameters used in the ideal case.

- **Scenario 1**: The GC sends data and control to as many LCs until the first needs to send its results back (Figure 3.3). The parallelism $\#LCs = 5$ can be calculated as:

$$\#LCs = \frac{t\_execute}{t\_send} = \frac{5}{1} = 5 \tag{3.1}$$

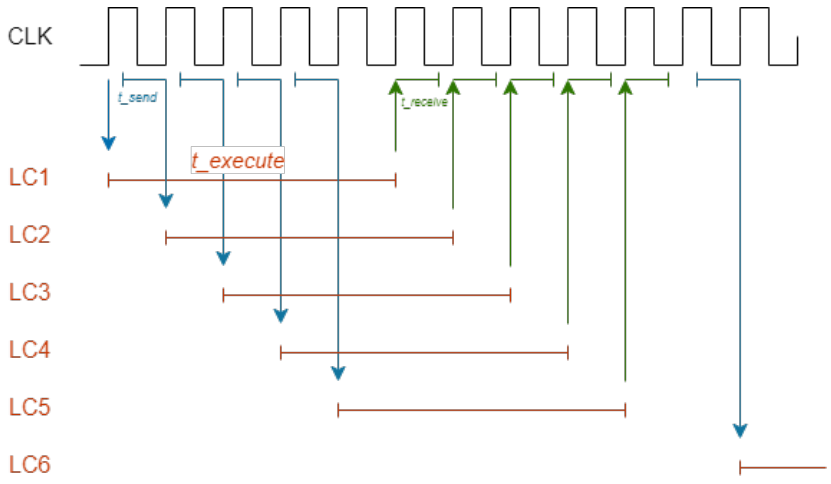


Figure 3.3

- **Scenario 2**: The GC sends data and control to LC#1 and addresses as many LCs until LC#1 needs to send its results back (Figure 3.4). The parallelism $\#LCs = 5$ can be calculated as:

$$\#LCs = \frac{t\_execute}{t\_receive} = \frac{5}{1} = 5 \tag{3.2}$$

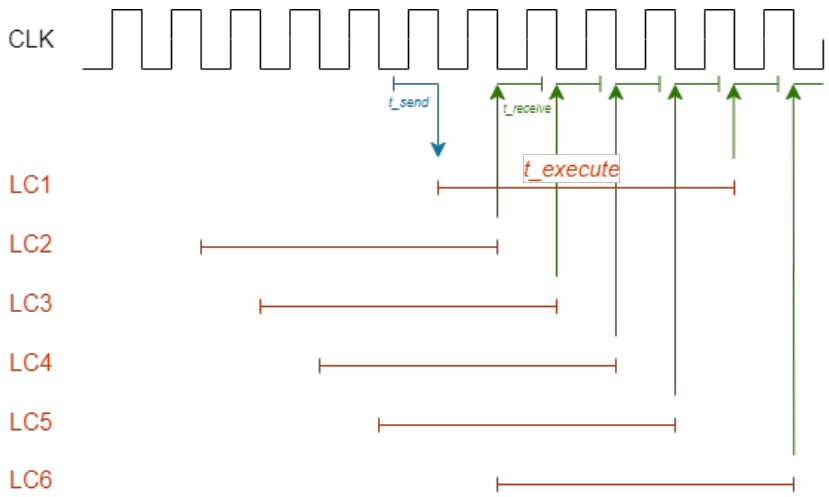


Figure 3.4

In this simplified scenario, the parallelism is solely determined by the execution latencies of the controllers. Furthermore, both workloads yield the same parallelism value since the latencies for sending and receiving processing are equal. However, this simplistic representation does not accurately model real-world scenarios, and thus, additional flexibility is introduced in the subsequent case.

3.2.2. Accounting for processing

The second case presents a difference in the processing times in the GC, as described in Table 3.3. In this scenario, *t_send* is different than *t_receive*, in order to study any difference in parallelism created by this inequality. To highlight this, a mixed workload is also analyzed.

GC	*t_send*	$2 \cdot t_{clk}$
GC	*t_receive*	$1 \cdot t_{clk}$
LC	*t_execute*	$5 \cdot t_{clk}$

Table 3.3: Parameters used in the second case.

- **Scenario 1**: The GC sends data and control to as many LCs until the first needs to send its results back (Figure 3.5). The parallelism $\#LCs = 2$ can be calculated as:

$$\#LCs = \lfloor \frac{t\_execute}{t\_send} \rfloor = \lfloor \frac{5}{2} \rfloor = 2 \tag{3.3}$$

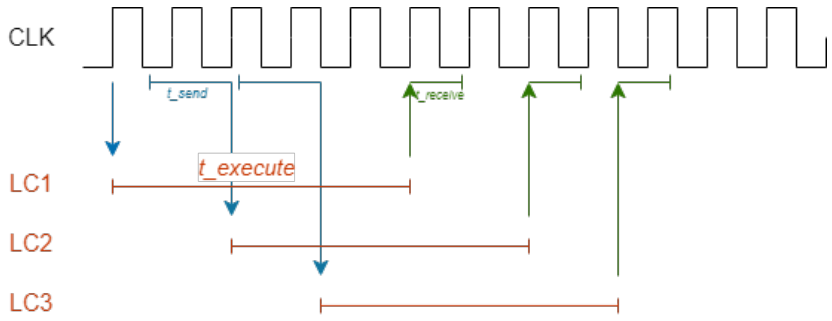


Figure 3.5

- **Scenario 2**: The GC sends data and control to LC#1 and addresses as many results until LC#1 needs to send its results back (Figure 3.6). The parallelism #$LCs = 5$ can be calculated as:

$$\#LCs = \frac{t\_execute}{t\_receive} = \frac{5}{1} = 5 \tag{3.4}$$

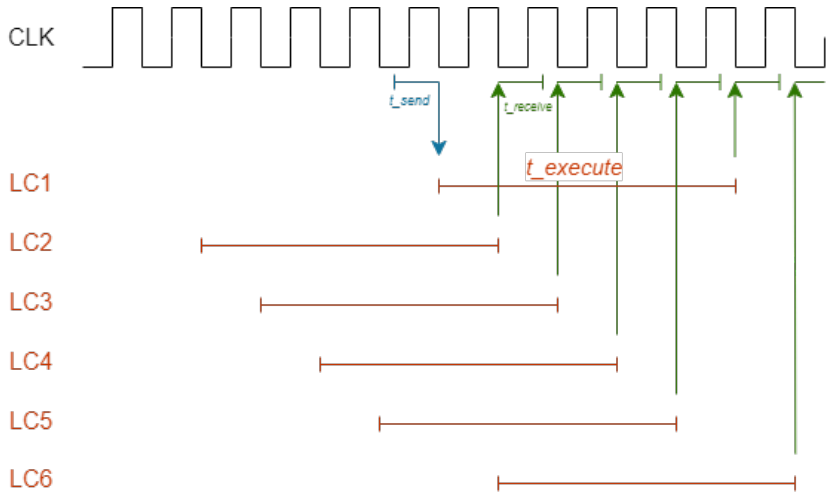


Figure 3.6

- **Scenario 3**: The GC sends data and control to LC#1 and addresses as many LCs until LC#1 needs to send its results back (Figure 3.7). The parallelism is measured as: #$LCs = 4$.

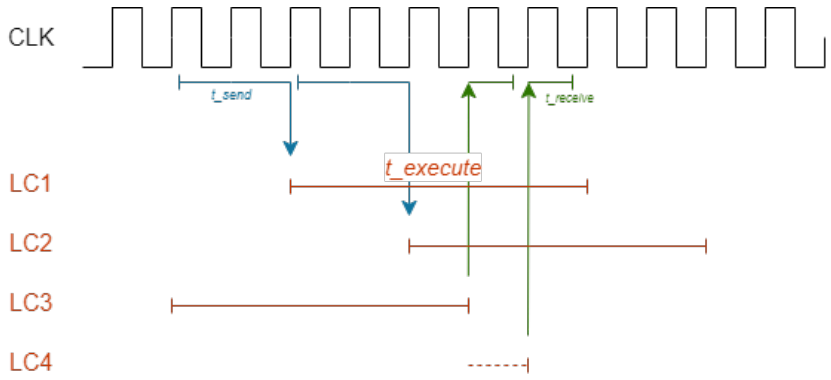


Figure 3.7

This study case offers interesting insights about the boundaries of parallelism. In particular, the slower send processing time corresponds to the lower boundary, while the faster receive processing time identifies the upper boundary. Accordingly, the mixed workload corresponds to a parallelism value within the boundaries.

3.2.3. Accounting for communication

By incorporating communication delays, a more general expression can be formulated to calculate the parallelism, considering latencies beyond the Global and Local Controllers. Despite the small value of the delay, its impact is already significant and greatly influences the overall result. This highlights the importance of accurately accounting for external latencies when evaluating the system's parallelism. Figure 3.8 presents the same scenario as in the ideal case, with the addition of the $t\_latency$ parameter, specified in Table 3.4. The parallelism is now increased to #$LCs = 6$ due to the communication delay, which gives more time to the GB to address additional LCs. It can be calculated as:

$$\#LCs = \frac{t\_execute + 2 \cdot t\_latency}{t\_receive} = \frac{6}{1} = 6 \tag{3.5}$$

GC	*t_send*	$2 \cdot t_{clk}$
GC	*t_receive*	$1 \cdot t_{clk}$
LC	*t_execute*	$5 \cdot t_{clk}$
Comm.	*t_latency*	$0.5 \cdot t_{clk}$

Table 3.4: Parameters used in the third case.

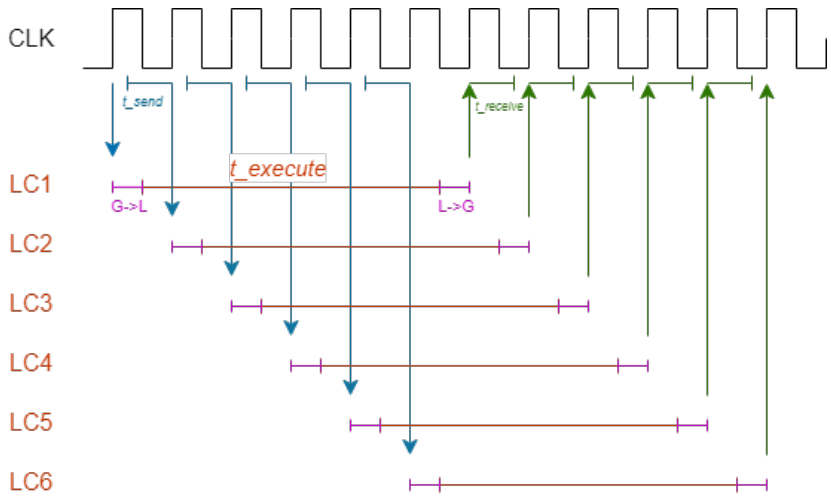


Figure 3.8

In this section, various workloads of the control plane were examined, and their corresponding parallelism was evaluated through the analysis of time diagrams. These findings shed light on the interplay between latencies and workload types, providing valuable insights into the system's performance. The next section focuses on the development of a complete model, which will encapsulate the collective understanding of execution profiling, time diagrams, and instruction execution, aiming to provide a mathematical formulation that accurately represents the system's behavior and characteristics.

3.3. Model definition

Once the execution has been completely profiled in Section 3.1 and its parallelism has been estimated in Section 3.2, the goal is now to create a comprehensive model that uniquely defines the parallelism of the control plane.

From the time diagram analysis, we observed that the parallelism of the Local Controllers is always bounded between the two extremes defined by the *send* and *receive* operations while it is incremented by the communication latency. The relationship can be therefore expressed as follows:

$$\frac{t\_execute_L + t\_latency}{max(t\_send_G, t\_receive_G)} \leq \#LCs \leq \frac{t\_execute_L + t\_latency}{min(t\_send_G, t\_receive_G)} \tag{3.6}$$

Equation 3.6 lays the initial foundation for the module's development, serving as a starting point for further improvements. The next step is to establish the precise definition of each quantity by incorporating the execution parameters outlined in Section 3.1. The following equivalences hold true:

- $t\_execute_L \equiv \beta \cdot Tc_L(i) + Tqex_L(i) + \varepsilon \cdot Tact_L(i) + \xi \cdot Twb_L$
- $t\_latency \equiv \gamma \cdot RTT_C(j) + \delta \cdot Tpd_C(j,k)$
- $t\_send_G \equiv \alpha \cdot Tpe_G(i)$
- $t\_receive_G \equiv \zeta \cdot Tpo_G(i)$

Consequently, including these definitions in Eq.3.6 we obtain the following equations:

$$\#LCs \leq \frac{\beta \cdot Tc_L(i) + Tqex_L(i) + \varepsilon \cdot Tact_L(i) + \xi \cdot Twb_L + \gamma \cdot RTT_C(j) + \delta Tpd_C(j,k)}{min(\alpha \cdot Tpe_G(i), \zeta Tpo_G(i))} \tag{3.7}$$

$$\#LCs \geq \frac{\beta \cdot Tc_L(i) + Tqex_L(i) + \varepsilon \cdot Tact_L(i) + \xi \cdot Twb_L + \gamma \cdot RTT_C(j) + \delta Tpd_C(j,k)}{max(\alpha \cdot Tpe_G(i), \zeta \cdot Tpo_G(i))} \tag{3.8}$$

For the rest of the analysis, the focus will be on Equation 3.7, but equivalent results can be derived with analog rationale for Equation 3.8. It is worth noting that the parameter values used in the precedent section were just example quantities, and the actual analysis is only valid when the latencies are representative of the real system.

Based on the dependency of the parameters on the instruction executed (*i*) and on the physical location of the nodes involved (*j,k*), it can be inferred that also the parallelism is dependant on the same variables. Consequently, a more correct relationship is defined in Equation 3.9.

$$\#LCs(i,j,k) \leq \frac{\beta \cdot Tc_L(i) + Tqex_L(i) + \varepsilon \cdot Tact_L(i) + \xi \cdot Twb_L + \gamma \cdot RTT_C(j) + \delta \cdot (j,k)}{min(\alpha \cdot Tpe_G(i), \zeta Tpo_G(i))} \tag{3.9}$$

Based on this intermediate outcome, the maximum achievable parallelism for controlling Local Controllers in parallel during each QISA instruction is already determined. To further generalize and construct the final general model, a few additional considerations need to be taken into account. These deliberations will contribute to refining the model and capturing a holistic representation of the system's behavior:

1. Remove the physical dependency from the parallelism value, by accounting for worst-case scenarios. Call $\#LCs(i)$ the result of Eq.3.9 for each instruction.
2. Define w_i as a weight representing the statistical frequency of the instruction *i* to appear in a given workload
3. Define M as the total number of QISA instructions creating the workloads

The complete final model calculating the control plane parallelism can be then defined as the weighted average of the parallelism of each instruction:

$$\#LCs = \frac{\sum_{i=1}^{M} \#LCs(i) \cdot w_i}{\sum_{i=1}^{M} w_i} \tag{3.10}$$

By defining the comprehensive and unified model shown in Equation 3.10, the control plane's achievable parallelism can now be accurately calculated, providing valuable insights into the system's performance and optimization potential.

3.4. Conclusion

In this chapter, we focused on modeling the control plane of the system to understand its behavior and performance. We began by analyzing the instruction execution process, presenting a detailed flowchart and profiling of the steps involved. This provided valuable insights into the sequence and timing of instructions. With the help of time diagrams, different workload scenarios and their impact on parallelism have been evaluated.

This served as a foundation for the definition of a model that provides a unified framework to calculate the achievable parallelism of the control plane, taking into account various factors and parameters. In this thesis work, we will eventually utilize this model to evaluate the design of the global controller.

4

qRV32 Instruction Set Extension

This chapter represents our first step in transitioning the focus of the work toward the operation of the global controller within a diamond-based color-center quantum computer. As we shift our focus, we embark on designing an instruction extension following the RISC-V standard and tailored to a Quantum Instruction Set Architecture (QISA) for NV-center quantum computing. This extension will allow us to incorporate the essential functionalities required for efficient control and coordination of the quantum computing system into an existing industry-level ISA. By encoding these additional instructions, we pave the way for the realization of the global controller.

Throughout this chapter, we will explore the intricacies of the qRV32 set extension and its design considerations and trade-offs. Firstly, RISC-V extensions (Section 4.1) and the Diamond QISA (Section 4.2) are introduced. The focus then shifts to the qRV32 Instruction Set Extension (Section 4.3), covering instructions encoding (Section 4.3.1) and communication with LCs (Section 4.3.2). The chapter concludes with a summary (Section 4.4).

4.1. RISC-V Extensions

The RISC-V ISA is based on a modular instruction set architecture design, which natively welcomes Instruction Set Extensions (ISE), particularly supporting the design of domain-specific central computing units. Implementing RISC-V extensions requires careful consideration of various factors, in particular, the compatibility with the existing RISC-V standard must be ensured to maintain interoperability.

As described in the RISC-V Instruction Set Manual [46], the ISA provides a flexible framework for incorporating both standard and custom extensions. The base ISA, either RV32I or RV64I, can be combined with selected standard extensions, such as IMAFD, Zicsr, and Zifencei, to form a ”general-purpose” ISA denoted as RV32G or RV64G. In addition to supporting general-purpose software development, RISC-V also aims to provide a foundation for specialized instruction-set extensions and customized accelerators. The instruction encoding spaces and optional variable-length instruction encoding facilitate leveraging the standard ISA toolchain when designing customized processors.

The major opcodes for the RISC-V general-purpose ISA, or *RVG*, are presented in Table **??**, where Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. *Custom-0* and *custom-1* opcodes are intended for use by custom instruction-set extensions, while *custom-2/rv128* and *custom-3/rv128* opcodes are reserved for future use in RV128. It is important to

inst[4:2] inst[6:5]	000	001	010	011	100	101	110
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*

Table 4.1: RISC-V base opcode map, inst[1:0]=11 [46]

avoid using reserved opcodes to ensure compatibility with future standard extensions.

Each instruction encoding space represents a specific number of instruction bits used to encode a base ISA or an ISA extension. RISC-V supports varying instruction lengths, and even within a single length, different encoding space sizes are available. For example, the base ISA is defined within a 30-bit encoding space (bits 31-2 of the 32-bit instruction), while the atomic extension ”A” fits within a 25-bit encoding space (bits 31-7). The term ”prefix” refers to the bits to the right of an instruction encoding space, and it is stored at earlier memory addresses in little-endian instruction fetch order. The base ISA prefix is the two-bit ”11” field (bits 1-0), while the atomic extension ”A” prefix is the seven-bit ”0101111” field (bits 6-0) representing the AMO major opcode.

To ensure compatibility and coexistence of various instruction encoding spaces, a standard-compatible global encoding allocates a unique non-conflicting prefix for each included space, targeting one or more available opcode spaces of Table **??**. This allows the base ISA, standard extensions, and non-standard extensions to be part of the global encoding. Non-standard extensions can be included as long as they do not conflict with standard extensions and can use standard prefixes if the associated standard extensions are not included. This approach enables a common toolchain to target the standard subset of any RISC-V standard-compatible global encoding.

4.2. Diamond QISA

The control hardware architecture [16] for the system is structured into layers, as illustrated in Figure 4.1. At the core of this architecture is the Global Controller, serving as the central component responsible for overseeing the entire system. It receives input in the form of a program written in the Quantum Instruction Set Architecture. Using these instructions, the global controller performs necessary computations and issues commands to the local controllers accordingly.

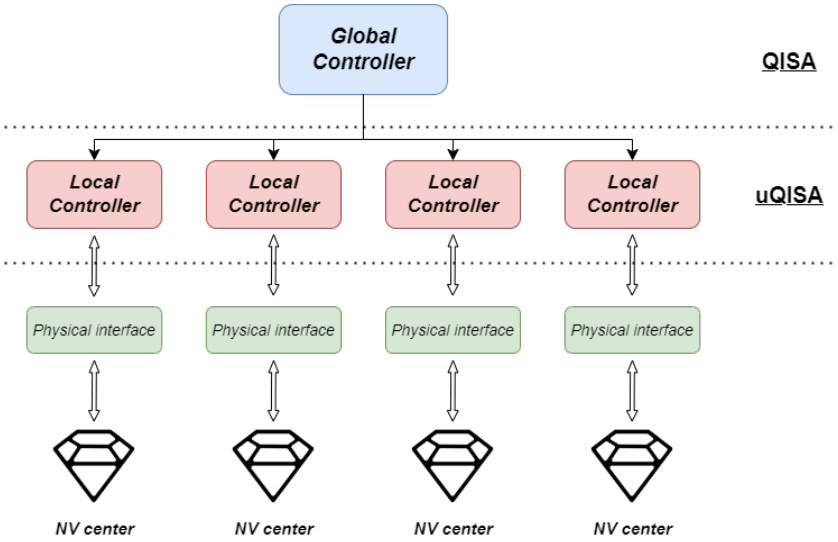


Figure 4.1: Schematic of the layered digital control adopted .

The literature provides multiple versions of Instruction Set Architectures specifically tailored for NV centers quantum computing. An initial version is presented in [47], followed by a more accurate and enhanced alternative described in [40] and [16].

For the purpose of this work, the QISA proposed in [16] is adopted as the reference. It offers a clear separation of functionalities between the Global and Local controllers, along with improved instruction specificity. While a partial listing of the QISA instructions can be found in Appendix A, further details, currently under embargo, are out of the scope of this project.

4.3. qRV32 Instruction Set Extension

In this section, we delve into the design of the Instruction Set Extension. Our objective is to create a RISC-V **standard-compatible global encoding** that can effectively handle the complete range of

functionalities offered by the QISA and its associated parameters. Furthermore, we address the crucial aspect of establishing a communication standard between the Global and Local Controllers.

4.3.1. Instructions encoding

The **qRV32 ISE** targets the Global Controller of a diamond-based quantum computer, and it must grant classical instruction for communication and data management on top of correctly mapping the structure of the QISA. Specifically, it must include **memory operations** (load, store), **immediate** operations, and **move** operations, because it is envisioned that the GC hardware will employ a separate register file, called *QREG*, to store the parameters of the QISA. Eventually, the architecture is envisioned to grant Direct Memory Access (DMA) to the LCs to write back their results, therefore avoiding the need for *send* and *receive* instructions.

A fundamental aspect of the process is the analysis of the instruction parameters. Due to the limited encoding space of 32bits instructions, not all the parameters can be expressed in immediate fields and may need to be preemptively stored in the QREG. Therefore a study of their nature and possible values is now presented. More information about this can be found in Table B.1 of Appendix B.

- The "*preserve*" and "*dir*" parameters used in *qgate-* instructions represent binary values, and can therefore be expressed using 1 bit
- The "*basis*" used during the *swap* instruction can be either x, y, z, therefore can be expressed using 2 bits
- While the entanglement process and the network structure are still not completely defined, a "*direction*" parameter is used to indicate the relative position of NV centers. A 3-bit binary number is used to represent 8 different directions
- Up to 32 registers are anticipated in the LCs, therefore 5 bits must be used to address them
- The "*phase*" and "*angle*" parameters used in *qgate-* instructions represent fractions of π. Therefore a fixed point <16,14> binary number is used to represent angles with accuracy of $\pi/16384$
- The frequencies "*sweepStart*", "*sweepStep*", "*sweepStop*" for the *detectCarbon* and *magbias* instructions can be expressed as 16-bit unsigned numbers
- Similarly, the times "*sweepStart*", "*sweepStep*", "*sweepStop*" for the *rabicheck* instruction are also 16-bit unsigned numbers

It looks clear that, among all the parameters, only "*preserve*", "*dir*", "*basis*", "*phase*" and "*angle*" can actually be expressed in immediate fields and still leave space for the actual encoding. Eventually, the parameters "*NV*" (number of NV centers per GC) and "*C13*" (number of ^{13}C qubits per NV center), as well as the depth of the *QREG* file, need to be maximized during the encoding process. The **target** for the number of NV centers per GC is set at 1000, while at least 10 ^{13}C qubits are planned per NV center.

It is worth noticing that the encoding process involves a trade-off between the number of addressable elements and the size of instruction identification fields. Increasing the number of bits used to identify specific elements, such as NV centers, reduces the available bits for identifying individual instructions, and vice-versa. As a result, the Instruction Set Extension may require additional major encoding spaces to accommodate the growing number of instructions.

The rest of the section presents three different encoding schemes for the qrv32 ISE. A complete listing of the encodings, comprehending all the instructions, can be found in Tables B.2, B.3, B.4 of Appendix B.

Encoding A

The first encoding presented in this work does not employ any optimization. It aims to encode all the instructions of the QISA in a single RISC-V major opcode, *custom-0*, and can be seen as a starting point for the design. The instructions can be divided into subgroups, accounting for similarities in functionality and parameters. Their encodings are presented in the following list.

1. *G-type* instructions comprehend the *qgate-* instructions and also *set* and *entangle*. The 3-bit parameter field can be used to express single-bit flags, like *dir* and *preserve*.
2. *N-ype* instructions comprehend the remaining qubit operations. The field *function-4* (*(f4)* provides a unique identification for the instructions included. The 5-bit parameter field is used for entanglement *direction* or swap *basis*.

31 - 30	29 - 27	26 - 23	22 - 19	18 - 16	15 - 10	9 - 7	6 0	1 1
f2	par.	QRS2	QRS1	C13	NVNode	$ID	0 0 0 1 0	1 1

31 - 29	28 — 23	22 —- 19	18 - 16	15 —— 10	9 - 7	6 0	1 1
f4	x	QREG	C13	NVNode	$ID	0 0 0 1 0	1 1

3. *C-type* instructions comprehend the calibration functions, such as *detectCarbon*, *magbias* and *rabicheck*. Those instructions require up to 4 QREGs to correctly store all the needed parameters, which are not immediates.

31 — 28	27 — 24	23 — 20	19 — 16	15 —- 10	9 - 7	6 0	1 1
QREG 4	QREG 3	QREG 2	QREG 1	NVNode	$ID	0 0 0 1 0	1 1

4. The instructions *move*, *qld* and *qst* present in their encodings space to specify either one 4-bit QREG and one gpREG field, or two 4-bit QREG fields. The immediate field *function-2 (f2)* is used to distinguish among these 4 instructions. The memory operations supported, use the available encoding space for the offset parameter (11-bit) passed as immediate.

31 ————- 21	20 - 19	18 —— 14	13 ——10	8 - 7	6 0	1 1
immediate [11:0]	f2	QREG2	QREG1	$ID	0 0 0 1 0	1 1

5. The *qldi* instruction is used to load a 16-bit wide immediate value into a QREG.

31 ———————— 16	15 - 14	13 — 10	8 - 7	6 0	1 1
immediate [14:0]	x	QREG1	$ID	0 0 0 1 0	1 1

While the encoding scheme enables the representation of quantum elements and instructions, it is important to acknowledge that the current implementation falls significantly short of the desired targets. Specifically, the architecture allows for a maximum of 8 ^{13}C qubits per NV center, 64 color centers per GC, 16 quantum registers, and supports 16-bit immediate loads. It is clear that these values fall well below the desired targets, and is therefore crucial to address the problem.

Encoding B
It is clear, from encoding A, that the 25-bit space is limiting the number of addressable elements. To overcome such limitations, an additional instruction *NV-choose* is introduced to set which node is going to be addressed. As for the previous encoding scheme, subgroups are now listed.

1. The *NV-choose* instruction provides a 13-bit wide field to specify the NV node address. This allows the system to increase the number of color centers per GC.

31 — 27	26 ———————— 14	13 - 10	9 - 7	6 0	1 1
1 1 1 1 1	NV node	x	$ID	0 0 0 1 0	1 1

2. In encoding B, the *Q-type* subgroup comprehends all the quantum operations, from gates to initialization. This is achieved by employing the encoding space earned by the removal of the NV center address to specify QREGs, parameters and a *function-5* field.

31 — 27	26 - 24	23 — 19	18 — 14	13 - 10	9 - 7	6 0	1 1
f5	param.	QRS2	QRS1	C13	$ID	0 0 0 1 0	1 1

31 - 30	29 — 25	24 — 20	19 — 15	14 — 10	9 - 7	6 0	
f2	QREG 4	QREG 3	QREG 2	QREG 1	$ID	0 0 0 1 0	1 1

3. The *C-type* (calibration) subgroup, comprehending *detectCarbon*, *magbias* and *rabicheck*, still employs 4 QREGs to store all the parameters. The newly available encoding space allows increasing the address fields up to 5-bit.
4. The instructions *move*, *qld* and *qst* and *qldi* present the same encoding discussed in the previous paragraph.

Encoding B completely mitigates the limitations created by encoding A. In fact, the architecture allows for 16 ^{13}C qubits per NV center, 8000 color centers per GC and 32 quantum registers while still offering a 16-bit immediate load. Unfortunately, the architectural separation between *NV-choose* and the quantum operation instructions can create a non-negligible overhead in the compilation and execution of algorithms.

Encoding C
This last encoding format tackles the limitations of encoding A with another solution: the QREG file structure is modified, and each register is envisioned to be 48-bit wide, which each 16-bit chuck as an independently loadable subregister. This reduces the portion of encoding space needed to declare multiple quantum registers, like in *C-type* instructions. Once again, instruction types are listed below:

1. *G-type* instructions comprehend all the quantum gates available. Within the parameter field, C13 (4-bit) and a single-bit flag can be specified.

31 - 29	28 - 24	23 — 19	18 ——————— 9	8 - 7	6 0	
f3	param.	QREGs	NV node	$ID	0 0 0 1 0	1 1

2. The remaining quantum operations belong to the *Q-type* instructions. The 6-bit parameter field can be used to specify QREG and LREG addresses (5-bit), C13, basi (2-bit), or direct (3-bit).

31	30 - 27	26 ———— 19	18 ———————— 9	8 - 7	6 0	
f1	f4	parameters	NV node	$ID	0 0 0 1 0	1 1

3. The *move* instruction, available in two flavors (move from and move to) based on the value of *f1*, does not use additional parameters

31- 30	29	28 — 24	23 — 19	18 ———————— 9	8 - 7	6 0	
1 0	f1	gpREG	qREG	x	$ID	0 0 0 1 0	1 1

4. As usual, *qld* and *qst* use the available encoding space for the offset parameter (11-bit) passed as immediate.
   Ex. *qst rs2 $rs1(imm)*

31- 30	29 —— 24	23 — 19	18 —— 14	13 —— 9	8 - 7	6 0	
f2	imm [10:5]	QREG2	QREG1	imm [4:0]	$ID	0 0 0 1 0	1 1

5. The *qldi* instruction is used to load an immediate value in a sub-register. Once again the immediate is 16-bit wide, and the *sel* parameter identifies which sub-register will be loaded.

31 ———— 24	23 —- 19	18 - 17	16 ———— 9	8 - 7	6 0	
immediate[15:8]	QREGd	sel	immediate[7:0]	$ID	0 0 0 1 0	1 1

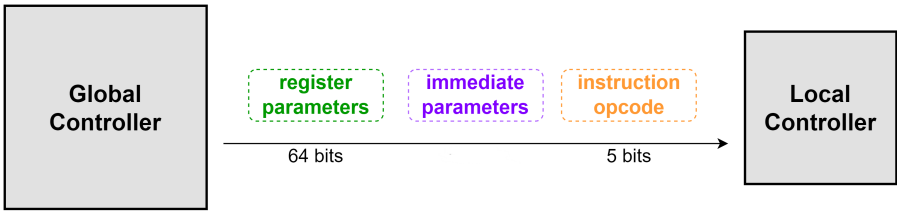


Figure 4.2: Global to Local communication packet

By adopting Encoding C, the limitations introduced by Encoding A are effectively mitigated. This enhanced architecture now supports up to 16 ^{13}C qubits per NV center, 1000 color centers per GC, 32 quantum registers, and retains the 16-bit immediate load capability, all without requiring any architectural modifications. However, it is important to note that this advancement comes at the cost of increased hardware complexity within the QREG structure.

Comparison
In this section, we explored three different encoding approaches for integrating the QISA into the RISC-V standard. Encoding A served as the initial design, fulfilling the required functionalities but falling short of meeting the design targets. Encoding B and Encoding C were proposed as alternative solutions to address the limitations. While Encoding B offered a straightforward approach, it introduced challenges in the quantum stack. In contrast, Encoding C appeared to overcome this limitation by increasing the complexity of the qreg. Ultimately, Encoding C emerged as the optimal and final solution, as the hardware overhead it introduced was manageable and easily implementable.

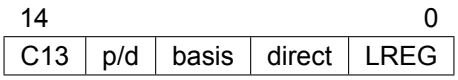
4.3.2. Communication with LCs

The preceding section provided an in-depth explanation of the instruction encoding design, which established a clear syntax for communication with the Global Controller. However, once the GC decodes and processes the instructions, it needs to transmit the appropriate control signals and instruction data to each Local Controller. Therefore, a communication standard between the GC and LCs must also be defined.

This communication packet must encompass the quantum operation to be executed, as well as all the immediate and register parameters required for its execution. Figure 4.2 visually illustrates the message and its individual components. The *register parameters* occupy a width of 64 bits and directly correspond to an entry in the Quantum Register File. The *instruction opcode* consists of a 5-bit code that uniquely identifies the QISA instructions, following the encoding table B.5 provided in Appendix B. Lastly, the *immediate parameters* field encompasses all the possible parameters expressed as immediate values at the instruction level.

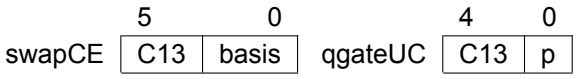
Notably, the *immediate parameters* field varies for each executed instruction, making its encoding more complex. The remaining part of this section will present two potential encodings and compare them, taking into account their physical effects and trade-offs.

Fixed-length packet
The immediate parameters can be encoded using a fixed-length packet, which offers a simpler approach. In this method, each instruction sends 15 bits within the communication message to transmit the value of each possible immediate parameter. If a parameter is not utilized, the corresponding bits are set to zero. Since the order of the parameters remains constant, the decoding logic in the local controller is straightforward. An illustration of the fixed-length packet representation is provided below.



Variable-length packet
Alternatively, a variable-length packet can be employed to transmit only the parameters required by the instruction. This can be achieved by maintaining a fixed order of possible parameters but sending out only those that are actually used. This approach reduces the message length since the packet

is always smaller than 6 bits. Two examples of such encoding, one for the *swapEC* instruction and another for the *qgateUC* instruction, are demonstrated below.



Comparison
The two encoding schemes, fixed-length and variable-length packets, have different tradeoffs and considerations in terms of decoding logic, power consumption, and overall efficiency.

- Decoding Logic on LC: In the fixed-length packet scheme, the decoding logic in the Local Controller is relatively simple since the order of the parameters is always the same. Each parameter has a fixed position in the packet, making it easier to extract and interpret the values. On the other hand, the variable-length packet requires additional logic to determine which parameters are present in the message and their respective positions. This increases the complexity of the decoding process in the LC.
- Transmission Power Consumption: The transmission power consumption is influenced by the width of the communication message and the distance it needs to travel. In the fixed-length packet scheme, all 15 bits are transmitted regardless of whether a parameter is used or not. This results in a wider message and potentially higher power consumption during transmission, especially when considering longer distances or operating at different temperatures. In contrast, the variable-length packet only transmits the necessary parameters, reducing the width of the message and potentially lowering power consumption.

The variable-length packet scheme offers advantages in terms of energy efficiency and logic power overhead. By transmitting only the required parameters, it minimizes the energy consumed during transmission. Additionally, the power consumption in the LC at $T = 4K$ remains similar to room temperature, and the decoding process only requires a few logic levels on top of the logic already used to translate the instructions to μcode. In contrast, the fixed-length packet scheme may result in higher power consumption due to its wider message width which requires a wider bus and therefore additional capacitance.
Considering the tradeoffs and benefits, the variable-length packet scheme emerges as a more favorable option and is therefore chosen for this work. The final structure of the communication message, with the variable-length packet encoding, consists of 75 bits in total, including the necessary instruction opcode, register parameters, and immediate parameters.

Overall, the variable-length packet scheme provides a more efficient approach for transmitting the required parameters within the communication message, balancing power consumption and decoding complexity.

4.4. Conclusion

In this chapter, we explored the integration of the diamond QISA proposed in [16] into the qRV32 architecture, focusing on the instruction encodings and communication packet syntax. We began by discussing the RISC-V extensions and the importance of incorporating the QISA functionality into the existing architecture.

Subsequently, we then delved into the details of the qRV32 Instruction Set Extension, which forms the core of our design. We examined the encoding of instructions, considering three different options: Encoding A, Encoding B, and Encoding C. Each encoding had its strengths and limitations, and we carefully analyzed their tradeoffs in terms of hardware complexity, addressing elements, and instruction identification fields.

Furthermore, we discussed the communication protocol between the Global Controller and the Local Controllers. We described the structure of the communication message, which includes the quantum operation to be performed, register parameters, and immediate parameters. We explored two different approaches for encoding the immediate parameters: fixed-length packets and variable-length packets. We compared these approaches based on decoding logic, power consumption, and overall efficiency.

Taking into account the considerations regarding instruction encodings and communication packet syntax, we have reached a conclusion. Firstly, Encoding C emerged as the best among the prosed solution. It overcomes the encoding space limitation by introducing an increased complexity to the QREG file. Despite the hardware overhead associated with this encoding, it remains the preferred choice, effectively addressing the limitations and providing a robust solution for the desired functionality. Secondly, the variable-length packet scheme, despite introducing some additional complexity in the decoding process, emerged as the more favorable option. It offers energy efficiency in transmission, minimizing power consumption, and avoids significant logic power overhead in the LCs. With the final structure of the communication message incorporating the variable-length packet encoding, we have achieved a well-balanced design.

In conclusion, the integration of the Diamond QISA into the qRV32 architecture, with careful consideration of instruction encodings and communication packet syntax, provides an effective and efficient framework for quantum computing. The qRV32 architecture, with its instruction set extension and optimized communication protocol, lays the foundation for quantum computing tasks and opens new possibilities in the field of quantum information processing.

5

Hardware Implementation

This chapter focuses on the hardware implementation phase following the design of the qRV32 Instruction Set Extension in the previous chapter. It begins with a survey of 32-bit RISC-V cores (Section 5.1), followed by a comparison (Section 5.1.1). The design of the Global Controller is discussed (Section 5.2), including ISE execution (Section 5.2.1), architecture (Section 5.2.2), and qRV32 functional units (Section 5.2.3). System on Chip design is then covered (Section 5.3), encompassing system memory (Section 5.3.1), UART (Section 5.3.2), and SoC (Section 5.3.3). The chapter concludes with the implementation results (Section 5.4) and a summary (Section 5.5).

5.1. 32-bit RISC-V Cores Survey

To address the choice for the Global Controller base processor core, the existing comparative literature [21]-[22] has been examined. Specifically, our goal is to identify and compare the best available open-source 32-bit RISC-V cores. Additionally, only cores that offer a native SoC architecture and are intended for FPGA implementation have been taken into account.

The subsequent portions of the section will show, examine, and evaluate four RISC-V-based cores to determine which one is the best fit for the project. The ability to execute the *qrv32* instruction will subsequently be added to the selected core in the next section.

PicoRV32

PicoRV32 [36] is a 32-bit CPU core designed to support the RV32I[M][C] or RV32E instruction set. The core's high *fmax* capability enables seamless integration into existing designs, eliminating the need for cross-clock domain communication. Furthermore, PicoRV32 demonstrates robust timing characteristics, even at lower frequencies, ensuring compatibility with diverse design requirements without compromising timing closure.

VexRiscv

VexRiscv is a feature-rich RISC-V CPU implementation utilizing a 5-stage in-order pipeline architecture [44]. It serves as a versatile foundation that allows for the seamless integration of optional plugins to augment its functionality. The core fully supports the *RV32I[M][A][F][D][C]* instruction set and offers the flexibility to customize the number of pipeline stages, ranging from 2 to 5. Notably, VexRiscv is specifically designed for FPGA implementations, making it independent of any vendor-specific IP blocks or primitives.

NeoRV32

The NEORV32 CPU [30] architecture offers a unique approach that combines elements of both traditional pipelined and conventional multi-cycle architectures, resulting in a distinct trade-off between performance and size. NEORV32 fully supports the *RV32I[B][C][M][U][X]* ISA and expands upon it with additional features. One notable feature of NEORV32 is its ability to incorporate custom RISC-V instructions through the inclusion of a Custom Functions Unit for CPU-internal custom instructions and a Custom Functions Subsystem for tightly-coupled co-processors within the core.

CV32E40P

CV32E40P [11] is a CPU core that belongs to the CV32E family developed by OpenHW Group. It is designed based on the RISC-V ISA, specifically targeting the RV32IMC instruction set variant. The core offers a balance between performance and efficiency, making it suitable for a wide range of embedded applications. It features a pipelined architecture with 4 stages for instruction execution, including fetch, decode, execute, and writeback stages. The CV32E40P core also supports non-standard Xpulp instructions. A detailed description of the core is available at [19].

5.1.1. Comparison

Having explained the characteristics of the cores, is now possible to perform a comparison. Common evaluation metrics, such as area and performance, are presented in relative comparison, and the strengths and weaknesses of each core are discussed. Metrics and evaluations are taken from the official documentation cited before.

Area and performance

Figures 5.1 and 5.2 visualize the comparison of the four cores in area, maximal execution frequency, and $performance/MHz$ using the CoreMark benchmark [10]. In particular, by comparing the CoreMark/MHz values, one can gain insights into how efficiently each core performs computations relative to its clock frequency, enabling a more meaningful assessment of its overall performance.

In terms of performance and area, the comparison reveals interesting findings. The picoRV core exhibits a significantly higher operating frequency ($714MHz$), nearly 14 times that of the CV32E40P core($55MHz$), while occupying less than half the area when implemented on Artix-X7 FPGAs. On the other hand, the CV32E40P core showcases exceptional performance per MHz, boasting an impressive 3.19 CoreMarks/MHz, while picoRV set the lowest result at 0.4 CoreMarks/MHz. Moving on to VexRiscV and NeoRV32, these cores demonstrate more average profiles in terms of area and frequency. However, VexRiscV grants more than twice the performance of NeoRV32 on the CoreMark benchmark at the price of less than twice the area utilization.

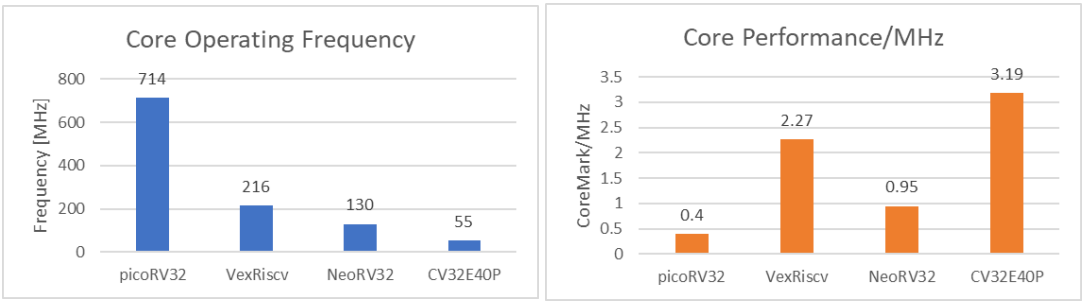


Figure 5.1: Performance comparison of the four cores analyzed.

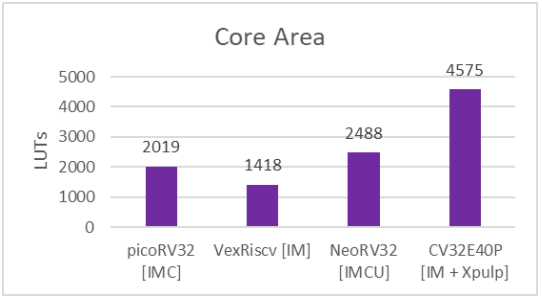


Figure 5.2: Area comparison of the four cores analyzed implemented on Artix X7 series FPGAs.

Features
Each core has its unique approach to **custom processing**. The picoRV32 core offers a native co-processor interface, allowing for the seamless integration of custom co-processors. In contrast, the VexRiscv provides flexibility in extending the instruction set via the plugin system, with complete support for custom instructions. The NeoRV32 core embodies a limited native custom instruction co-processor, enabling the addition of instructions with predefined encoding. Lastly, the CV32E40P core provides native support only to the Xpulp extension, for signal processing.

When it comes to **System On Chip** (SoC) architecture and available peripherals, picoRV32 offers the PicoSoC, which is a limited SoC solution. VexRiscv provides the *Pinsec* SoC, which includes essential peripherals such as UART, GPIO, and timers. NeoRV32 goes further by offering a complete microcontroller-like SoC, with additional features like SPI and I2C, making it suitable for a wide range of embedded applications. Among them, CV32E40P stands out with its PULPissimo SoC, which offers advanced features like DMA (Direct Memory Access), making it the most comprehensive option in terms of SoC architecture and available peripherals.

Discussion
Upon careful evaluation of the four cores, namely picoRV, vexRV, NeoRV32, and cv32e40p, each core exhibits distinct features and trade-offs that influence their suitability for the project.

The picoRV core exhibits solid performance and efficient area utilization, making it a desirable option in terms of resource efficiency. However, its limited SoC capabilities hinder its potential for more complex applications. Additionally, picoRV lacks support for custom instructions, which could limit its adaptability to specific task requirements.

The VexRiscv core, in contrast, stands out for its thorough support for instruction extensions, enabling customization of the instruction set. The SoC capabilities of the VexRiscv, however, are rather constrained, and there are only a few peripherals offered. Additionally, developers may experience a learning curve when using SpinalHDL, a more complicated hardware description language.

The NeoRV32 core offers a complete microcontroller-like SoC with a comprehensive range of peripherals. However, it lacks a key feature, DMA, which can significantly impact its efficiency in handling data transfers. Additionally, the existing custom instruction co-processor severely limits the design freedom in instruction encoding. Eventually, the core's overall performance-area trade-off is at best average compared to the other options.

In terms of the CV32E40P core, it emerges as a highly compelling choice. It not only provides a feature-rich SoC design but also offers native support for custom ISA extensions. This inherent native support for the *Xpulp* ISE makes its architecture naturally adaptable to accommodate other custom instructions. With appropriate modifications, the core can be configured to execute the instructions specific to the qRV32 architecture using part of the decoding logic designed for *Xpulp*. Furthermore, the CV32E40P core demonstrates good performance metrics and provides a more comprehensive SoC architecture, including valuable features like DMA, which enhances its versatility and efficiency in handling data-intensive tasks.

Considering these factors, the CV32E40P core presents the most favorable combination of a feature-rich SoC design, support for custom extensions, and strong performance metrics. Its architecture, which already implements the desired extension, provides a solid foundation for easy customization and integration. Therefore, based on these assessments, the CV32E40P core is the designed choice for the hardware implementation of the Global Controller.

5.2. Global Controller Design

The hardware implementation of the global controller is a critical component in the overall design of a system, as it plays a vital role in coordinating and controlling various modules and subsystems. In this section, we focus on the intricate details of the global controller design, aiming to develop an efficient and robust solution. Our primary objective is to present the architectural choices and considerations made during the design process, ensuring seamless integration with the target system. Additionally, we thoroughly examine the instruction pipeline, which forms the backbone of the controller's operation. Furthermore, we pay special attention to the specific functional units tailored to support the qrv32 Instruction Set Extension, ensuring the successful execution of its unique instructions and functionalities. Through a comprehensive exploration of these aspects, we aim to achieve a well-designed,

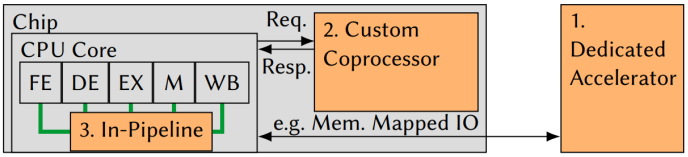


Figure 5.3: Different Implementation approaches: 1. Dedicated Accelerator, 2. Custom Coprocessor, 3. In-Pipeline. Deep integration can limit inter-core portability. Off-chip designs may affect latency, throughput, and hardware cost. [12]

high-performance global controller that effectively orchestrates a diamond-based quantum computer control layer.

5.2.1. ISE Execution

When it comes to implementing the execution of custom instructions in the Global Controller, several architectural approaches can be considered, each with its own set of advantages and drawbacks. These approaches include the use of a co-processor, in-pipeline execution, or a custom accelerator. Figure 5.3 represents the different approaches in a high-level diagram.

- Tightly embedding the custom execution within the pipeline stages offers the advantage of maximizing throughput and optimizing hardware resource utilization. However, this approach requires a higher level of design effort and may face challenges in terms of scalability.
- On the other hand, employing a custom co-processor provides better scalability and facilitates the use of standard communication protocols. However, it may incur additional area and timing costs, and there may be limitations in terms of custom control-flow.
- Alternatively, implementing the custom instruction execution in a custom accelerator can introduce latency penalties and limited throughput. Additionally, it often comes with higher area and power costs.

Based on the specific requirements of the control core in the control layer, the decision has been made to implement the execution of the ISE using an in-pipeline architecture. This choice is driven by the need to effectively coordinate multiple LCs and maximize throughput, while also minimizing latency. By embedding the custom instruction execution within the pipeline stages, we can achieve efficient instruction execution and control over the LCs, resulting in improved overall system performance. Furthermore, the in-pipeline architecture allows for better utilization of hardware resources and offers a balanced trade-off between throughput and latency.

5.2.2. Architecture

The previous section defines the way of implementing the custom instruction execution in an existing core design. Accordingly, the resulting modified four-stage core pipeline can be observed in Figure 5.4. The execution flow has been modified to include qrv32 functional units, identified in the picture by filled rectangles.

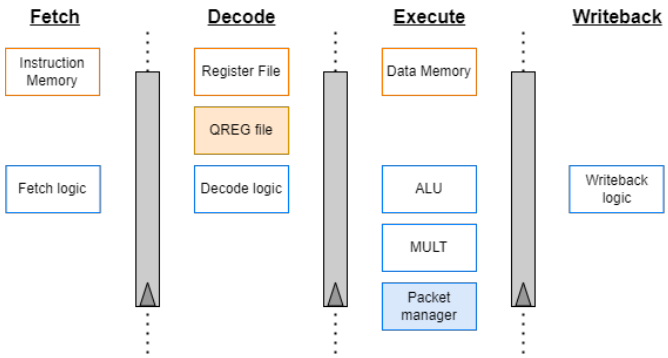


Figure 5.4: GC modified architecture pipeline. Filled rectangles identify the custom additional functional unit and register file.

The architecture of the global controller, including the CV32E40P core and the integrated qrv32 units, can be understood by analyzing the instruction flow through different stages. Figure 5.5 provides a clear visual representation of this architecture. As the fetched instruction enters the decode stage, it is processed by the decoder, which, in addition to identifying instruction operands, also distinguishes between custom and standard instructions. To accommodate these different types, separate register files are utilized for storing the corresponding data. After decoding, the instruction operands are passed to the execute stage, where they are processed either by the ALU and MULT units or assembled into a packet by the qrv32 packet manager. The resulting data is then written back to the data memory using the Load/Store Unit (LSU) or back to the register files through the writeback logic.

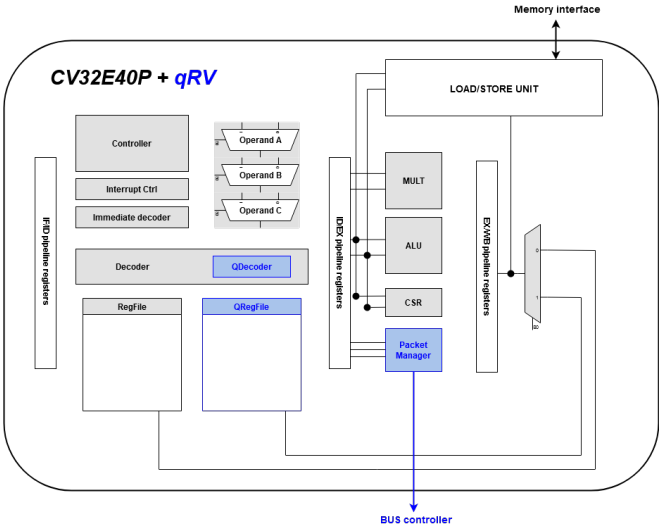


Figure 5.5: GC architecture. In grey, native components of the CV32E40P core; in blue, additional units required for the execution of qRV32 ISE

The decode and execute stages of the CV32E40P core form the central components for efficiently processing instructions, therefore they have undergone substantial modifications with the inclusion of qrv32 units and logic.

During the decode stage, the fetched instruction is decoded by a combination of instruction decoders and control logic. The instruction decoders are responsible for identifying the opcode and other essential fields of the instruction, while the control logic generates control signals required for the subsequent stages. An immediate decoder completes the identification of instruction operands started in the decoder. Eventually, each operand is fed to the multiplexed ports *OperandA*, *OperandB* and *OperandC*, which are responsible for passing meaningful data onto the execute stage.

The decoded instruction is then passed on to the execute stage, which consists of multiple functional units, including an Arithmetic Logic Unit and a Multiplier. The ALU handles various arithmetic and logical operations, such as addition, subtraction, bitwise AND, and OR. On the other hand, the MULT unit executes multiplication operations, providing essential support for arithmetic-intensive tasks. In addition to that, the qrv32 packet manager is in charge to assemble the communication message to LCs, incorporating the variable-length packet.

5.2.3. qRV32 Functional Units

In this subsection, we finally delve into the design of the custom hardware blocks, responsible for addressing qrv32 instructions. These specialized hardware blocks are specifically designed to address the unique requirements and functionalities of the qrv32 instruction set. To provide a comprehensive understanding of their purpose and operation, we will explore each custom hardware block in detail, outlining their individual contributions to the overall system architecture.

QDecoder

The decoder plays a crucial role in identifying the opcodes of instructions. It serves as the initial step in the decoding process, enabling the accurate interpretation of instructions and distinguishing among

various instruction parameters. In the case of CV32E40P, which supports both standard RISC-V IMC instructions and custom Xpulp instructions, modifications are necessary to accommodate qrv32 instructions. To ensure the availability of opcode space for qrv32 instructions, certain overlapping Xpulp instructions that utilize the same encodings need to be removed. This task is specific to the CV32E40P core and requires minimal additional effort. The freed opcodes encompass a range of instructions, including BIT-MANIPULATION instructions, custom MULT instructions, and HW-LOOP instructions.

In addition to opcode management, new decoding logic must be implemented to correctly handle qrv32 instructions. To address this requirement, a dedicated Quantum (Instruction) Decoder, referred to as the QDecoder, is incorporated within the decoder stage. The qRV32 custom instructions are identified by the *custom-0* opcode and are decoded based on the specifications provided in Table B.4. For detailed snippets of the design code related to the QDecoder implementation, refer to Listing C.1 in Appendix C.

QREG File

```
29    //Read port R1
30    input  logic [ADDR_WIDTH-1:0]   raddr_i,
31    output logic [DATA_WIDTH_R-1:0] rdata_o,
32
33    // Write port W1
34    input logic [ADDR_WIDTH-1:0]   waddr_i,     // write address
35    input logic [1:0]              wsel_i,      // write sub-reg select
36    input logic [DATA_WIDTH_W-1:0] wdata_i,     // write data
37    input logic                    we_i,        // write enable
38
39    //Read port R2 (32bit)
40    input  logic [ADDR_WIDTH-1:0]   raddr_mem_i,
41    output logic [DATA_WIDTH_M-1:0] rdata_mem_o,
42
43    //Read port R3 (32bit)
44    input  logic [ADDR_WIDTH-1:0]   raddr_mem2_i,
45    output logic [DATA_WIDTH_M-1:0] rdata_mem2_o,
46
47    //Write port W2 (32bit)
48    input logic [ADDR_WIDTH-1:0]    waddr_mem_i,
49    input logic [DATA_WIDTH_M-1:0]  wdata_mem_i,
50    input logic                     we_mem_i
```

Listing 5.1: QREG file ports - *qRV_register_file_ff.sv*

The design of the QREG file, an integral hardware component in the qRV32 architecture, takes into consideration the specific requirements of the instruction set encoding proposed in Section 4.3.1.

The QREG file module consists of 32 registers, each with a width of 64 bits. These registers are further divided into four 16-bit sub-registers, allowing for independent write operations. With one read port, which operates combinatorially, the QREG file enables concurrent access to the register contents. The read port is 64 bits wide, facilitating the retrieval of an entire register entry in a single operation, as described in Listing 5.1. For write operations, the QREG file features a sequential write port that is 16 bits wide. This allows for selective modification of the sub-registers within a register, granting fine-grained control over data updates. In addition, the QREG file includes two dedicated ports for memory instructions, providing a 32-bit wide read port and a 32-bit wide write port. These specialized ports streamline the execution of instructions involving data transfers between registers, such as load, write and move.

In summary, the QREG file design custom read and write ports address the specific requirements of qRV32 instructions. For further details on the QREG file module, including the complete SystemVerilog code implementation, please refer to Listing C.2 in Appendix C.

Packet Manager

The Packet Manager within the Global Controller is designed to assemble communication messages for the Local Controllers, adhering to the specific requirements of the qRV32 ISA extension on Global-to-Local communication.

It generates a quantum message which employs the variable-length format presented. In the case of the qRV32 ISA extension, the Packet Manager assembles packets with a total length of 75 bits.

This ensures consistent and compatible communication between the Global Controller and the Local Controllers, facilitating efficient data transfer and control signals. For more detailed information on the packet format and the operations performed by the Packet Manager, refer to Appendix C.

Minor Modifications

```
548 //qRV immediate extraction
549 assign imm_qmem_type = {{20{instr[29]}}, instr[29:24], instr[13:9]};          //bottom 11
        bits is the immediate value
550 assign imm_qldi_type = {16'b0, instr_rdata_i[31:24], instr_rdata_i[16:9]};    //bottom 16
        bits is the immediate value
```

Listing 5.2: qRV immediate extraction - *cv32e40p_id_stage.sv*

To accommodate additional control and data signals, as well as additional ports, several modifications were made to the original designs of the stages. This subsection focuses on highlighting the non-trivial changes implemented within the decoding stage to accommodate the custom immediate fields and other requirements.

One notable modification is related to the syntax differences between qRV32 memory instructions and standard instructions. These differences impact the width of immediate fields, necessitating the design of new logic to handle these variations. The specific implementation details of this logic can be found in Listing 5.2.

Furthermore, the forwarding logic underwent modifications to meet the specific requirements of the qRV32 instructions. The forwarding logic includes multiple levels of multiplexers that determine which data should be passed to subsequent stages based on the decoded instruction. In the context of memory operations, OperandA and OperandC are utilized to forward the read data from the QREG file. Additionally, OperandB is employed to handle the immediate parameters of the *qldi* instruction. The control logic is accordingly adjusted based on the opcode of the instruction. For detailed insights into the modifications made to the forwarding logic, please refer to Listings C.4 and C.5 in Appendix C.

Moreover, specific modifications were introduced to address the unique requirements of move instructions. Detailed information regarding these modifications can be found in the appendix.

5.3. System on Chip Design

The System On Chip design section focuses on the hardware components designed to facilitate the interaction and functionality of the Global Controller Core. While the presented design does not encompass a comprehensive and finalized SoC, it serves as a minimal configuration capable of enabling the core's functionality. Key components included in this design are the system memory and a simple UART interface, which serves as the interface for communication. By incorporating these essential elements, the SoC design provides a foundation for the proper operation and integration of the Global Controller Core.

5.3.1. System Memory

Instruction and data memory are fundamental parts of the system and are required to guarantee the functionality of the GC. To enable seamless operation, a ROM is incorporated into the SoC to provide a reliable mechanism for storing instructions and initiating the core's execution. Furthermore, a dedicated RAM component is included in the SoC design to support the scalability and efficient functioning of the control plane. This RAM serves as a versatile storage medium for critical data, ensuring its availability and facilitating smooth control operation execution.

By incorporating both the Boot ROM and RAM, the SoC design establishes a solid foundation for the Global Controller's functionality. This comprehensive approach ensures that the core receives the necessary instructions and data resources, enabling it to effectively perform its control tasks in a scalable and efficient manner.

ROM

The design of the Read Only Memory for the SoC incorporates a single-port memory block with a capacity of 32×256 bits, resulting in a $1KB$ ROM. Each word in the memory consists of 4 bytes. Since the memory is byte addressable - as from the RISC-V standard - the two least significant bits of the address are ignored during operations. The read access is combinatorial, providing instant access to

the stored instructions. At simulation time $t = 0$ or during synthesis, the memory is flashed with the instructions retrieved from the specified file indicated by the *instr_path* parameter. Write operations are not allowed in the ROM as it functions as read-only storage. Additionally, to accommodate the requirements of the core, additional ports are added to mimic the PCI protocol ports. These ports serve to establish the necessary communication interfaces for the core's functionality. The full SytemVerilog module is provided in Listing C.8 in Appendix C.

RAM

The RAM memory design in the SoC employs a single-port memory block with a capacity of 32×256 bits, resulting in a $1KB$ RAM block. Similar to the ROM, the RAM operates in a byte-addressable manner, consequently, the two least significant bits of the address are ignored during read operations. Moreover, the memory design takes into consideration the byte-addressability, utilizing a dedicated byte enable signal to determine the specific portion of the word to be stored.

Read access to the memory is combinatorial, ensuring instantaneous retrieval of the stored data. In contrast, write operations are sequential, allowing for the sequential updating of memory locations. Additionally, during simulation, a reset signal is employed for testing purposes, as the RAM implementation relies on FPGA block RAMs that do not require an explicit reset signal in synthesis. Eventually, to meet the requirements of the core's integration, supplementary ports are added also to the RAM module to mimic the PCI protocol ports. The complete module implementation is shown in Listing C.9 in Appendix C.

5.3.2. UART

In the context of the minimal SoC design, a UART interface is utilized for the communication of the 75-bit instruction message and the 10-bit NV center address. While the control plane implementation may require a custom bus interface between the local and global controller, this simplified SoC design relies on the UART interface for its communication requirements.

To accommodate the conversion of the total 85 bits required for transmission, specialized hardware is employed. Specifically, a sequencer module is responsible for handling the transmission process and operates with various inputs and outputs. It takes inputs such as the clock signal, the 85-bit input data to be transmitted, and the start control signals for initiating transmission. It provides outputs for buffer readiness, the start of transmission, and the serialized data transmitted in 8-bit chunks. The sequencer incorporates internal registers for tracking the position within a byte and managing the ongoing transmission. It ensures efficient and reliable data transmission within the SoC design. An always block triggered on the positive edge of the clock signal is utilized to handle the transmission process.

Considering the UART interface as the potential bottleneck in the design due to its low baud rate compared to the operating frequency, a FIFO (First-In-First-Out) structure is implemented. This FIFO has a size of 85x128 and employs a read enable signal to record only meaningful core data outputs, ensuring that no data is lost while the interface is busy.

Finally, the UART interface follows the standard communication protocol available in the literature [34], with specific attention given to the *baud_rate* and *clk_freq* parameters. These Verilog parameters must be properly configured to match the actual hardware parameters for a functional implementation on an FPGA platform. For detailed insights into the UART design, please refer to Listings from C.10 to C.14 in Appendix C.

5.3.3. SoC

The overall SoC design module serves as a central entity that defines the system's main input and output ports, clock frequency, and baud rate parameters. It plays a crucial role in connecting all the components of the system together. In this simplified SoC design, a bus is not implemented, and the peripherals (memory and UART) are directly connected to one or more ports of the core based on their respective functionality. A complete view of the elaborated design on Xilinx Vivado is shown in Figure 5.6.

5.4. Synthesis results

The Global Controller and SoC design was synthesized using Vivado FPGA tools, targeting the Kintex-7 FPGA family. The synthesis results showed a maximal operating frequency of $100MHz$, and the design

WSN	*TSN*	*WSH*	*Failed Routes*	*LUT*	*FF*	*BRAM*	*URAM*	*DSP*	*IO*
0.276	0.000	0.107	0	7157	4630	1.5	0	5	4

Table 5.1: Syntehsis results from Xilinx Vivado

utilized 7157 Look-Up Tables (LUTs). Additional synthesis results obtained on Vivado are summarized in Table 5.1.

It is crucial to note that the frequency reported in Section 5.1.1 cannot be directly compared to this result. The nominal frequency ($55MHz$) represents the standard operating frequency at which the core is designed to work under typical conditions. On the other hand, the maximal operating frequency obtained from synthesis represents the highest frequency at which the design can function without violating the timing constraints.

Furthermore, the synthesized design utilizes nearly double the number of Look-Up Tables (LUTs) compared to the base core. This increase in LUTs is mainly due to the incorporation of memory and UART peripherals, rather than the additional core functional units. The addition of these peripherals enhances the functionality and capabilities of the Global Controller and SoC design, enabling it to interact with the memory system and external devices efficiently.

5.5. Conclusion

In this chapter, we have explored the hardware implementation of the Global Controller, which serves as the control plane for our RISC-V-based system. We began by surveying various 32-bit RISC-V cores to understand their features and capabilities, providing a basis for our design choices.

The Global Controller design was then presented, focusing on its execution within the Instruction Set Extension. We examined the architecture of the Global Controller, including its functional units specifically tailored for the qRV32 ISE.

Furthermore, we delved into the design of the System on Chip to address the crucial aspects of instruction feeding and data memory management. The inclusion of dedicated system memory and UART interface ensures the smooth operation and scalability of the control plane.

Through this hardware implementation journey, we have demonstrated the integration of the Global Controller with the necessary peripherals, allowing for efficient communication and interaction within the system. The SoC design presented here provides a minimal yet functional setup for the Global Controller Core.

In conclusion, the hardware implementation chapter has provided valuable insights into the design and construction of the Global Controller and the associated components. The next chapter will focus on the software stack, where we will explore the development of an assembler targeting qRV32 in order to realize a fully functional RISC-V system.

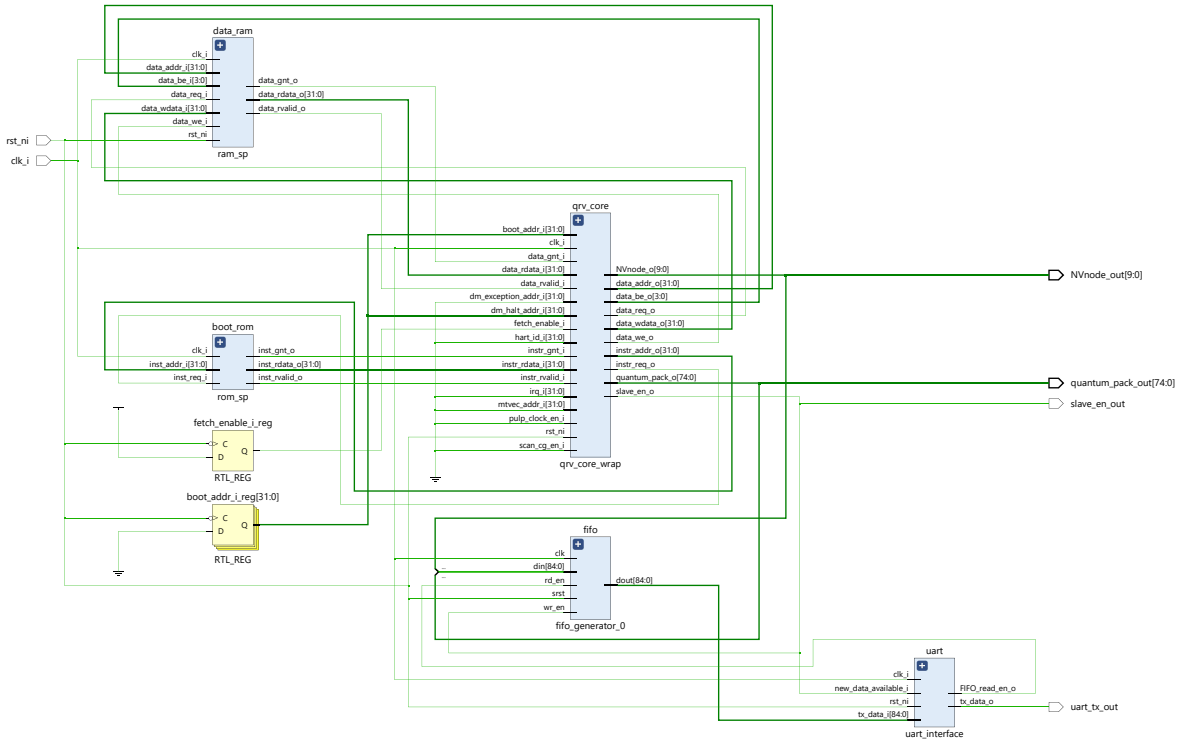


Figure 5.6: System On Chip elaborated design view on Vivado

6

qRV32 Assembler

This chapter focuses on the development of a custom assembler to target qRV32 instructions. The design of an assembler is a critical component in the creation of a functional system and allows thorough testing of the Global Controller hardware. The main goal of the assembler is to translate the quantum algorithms compiled at QISA level [16] to actual machine-level instructions. This involves parsing the assembly instructions, identifying the corresponding opcode and operands, and producing the binary representation.

By the end of this chapter, readers will have a comprehensive understanding of the assembler's purpose, design, and functionality. They will gain insights into the intricacies of translating assembly code into machine instructions, enabling them to write efficient and optimized programs for the system.

The chapter starts with the analysis of assembly-level instructions (Section 6.1). The development steps are then described in Section 6.2, including opcode identification (Section 6.2.1), substitution of pseudo-instructions (Section 6.2.2), register management (Section 6.2.3), and parameter translation (Section 6.2.4). The chapter discusses the assembler outputs (Section 6.3) and concludes with a summary (Section 6.4).

6.1. Assembly-level instructions analysis

```
qgatee q0 1.5707963267949 1.5707963267949
qgatecc q1 1 0.0 3.1415926535898
qgateuc q2 2 0.0 3.1415926535898 1
qgatedir q3 3 1.5707963267949 1.5707963267949 0
qgateze q4 1.5707963267949
qgatezc q5 5 1.5707963267949
initialize q4
measuree q5
crc q6
detectCarbon q7 100 5000 200 10000
magbias q8 1500 5000 200 10000
rabicheck q9 100 5000 200 10000
swapec q10 6
swapce q11 7 2
set_local q12 r15 0
nventangle q13 010
move_to r20 qr12
move_from qr30 r11
qld r24 0xF5(r20)
qst r5 0xF6(r20)
```

Listing 6.1: Example list of all QISA instructions, assembled

Listing 6.1 provides a comprehensive view of the complete set of instructions for diamond-based quantum computers, represented in QISA format. Familiarity with the syntax of these instructions is crucial for the accurate development of an assembler. Analyzing the structure of the instructions, it becomes evident that each instruction begins with a string that denotes the type of instruction. Subsequently,

the NV center specification follows, providing the unique address details about the targeted node. Finally, the instruction parameters are presented, adhering to the prescribed order as outlined in Table A.1. By closely examining this listing, one can gain a comprehensive understanding of the syntax and organization of the instructions, which serves as a foundation for developing a robust assembler for diamond-based quantum computers.

6.2. Development steps

The development of an assembler for the diamond-based quantum computer architecture involves several crucial steps. This section provides an in-depth exploration of these development steps, focusing on the key aspects required to transform QISA instructions into executable machine code and how to implement them in Python. The following subsections outline the fundamental stages of the assembler development process. The complete implementation code of the assembler and helper functions can be found in Appendix D.

6.2.1. Opcode Identification

The first step in the assembler development process is opcode identification. This involves analyzing the instruction set architecture and recognizing the specific opcodes associated with each instruction. By correctly identifying the opcodes, the assembler can map each instruction to its corresponding machine code representation. This is achieved by **dictionaries** linking the string keyword to the corresponding binary opcodes.

```
    "qgatee":      "000001011",
    "qgatecc":     "000001011",
    "qgateuc":     "000001011",
    "qgatedir":    "000001011",
    "qgateze":     "000001011",
    "qgatezc":     "000001011",
    ...
```

Listing 6.2: Portion of opcode identification dictionaries - *qRV32_assembler.py*

6.2.2. Pseudo-instructions Substitution

Pseudo-instructions, also known as macro instructions, provide a higher-level representation of complex operations. In this step, the assembler substitutes pseudo-instructions with the corresponding sequences of native instructions using the following **function**:

```
def pseudo_instruction_substitutor(opcode, asm_instruction):
```

This substitution ensures that the assembler can accurately translate these higher-level instructions into the appropriate machine code. Here is an example of the addition of load-immediate instructions insertions to correctly map the *qgatee* instruction:

```
    if opcode == "qgatee":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[1])])
        subreg += 1
        #2nd qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), "r"+str(parameter_register)])
```

Listing 6.3: Example of a pseudo-instruction substitution - *qRV32_assembler.py*

6.2.3. Register Management

Correct register management is crucial for code generation. In this step, the assembler handles the allocation and management of registers, as shown in the following snippet:

```
def find_next_available_register():
    # Find the first free register
    destination_register = -1  # Set to an invalid value as a fallback
```

```
    for i, status in enumerate(register_status):
        if status == 0:
            destination_register = i
            register_status[i] += 1  # Increment the register status value
            break

    # If no free registers are available, select the earliest used register
    if destination_register == -1:
        min_used_time = max(register_status) + 1  # Set to an invalid value as a fallback
        min_used_index = -1  # Set to an invalid value as a fallback
        for i, status in enumerate(register_status):
            if status < min_used_time:
                min_used_index = i
                min_used_time = status

        # If a register with the minimum value is found, set it as the destination register
        if min_used_index != -1:
            destination_register = min_used_index
            register_status[min_used_index] += 1  # Increment the register status value

    return destination_register
```

Listing 6.4: Function for register management - *qRV32_assembler.py*

The assembler assigns appropriate registers to store intermediate values, avoids register dependencies, and ensures complete utilization of available registers. Proper register management is crucial to ensure reliable mapping between assembly and machine-level code.

6.2.4. Parameters Translation

The translation of instruction parameters is another vital aspect of the assembler development process. Instructions often contain parameters that need to be translated into their appropriate machine code representation. This step involves converting these parameters, such as immediate values or memory addresses, into a suitable binary format that the target architecture can interpret correctly. In particular, immediate values need to be represented accordingly to Table B.1. This is achieved by the following **helper function**.

```
def float_to_fixed_point(num):
    # Multiply the floating point number by 2^14 to shift the decimal point 14 bits to the
        left.
    # This converts the floating point number to a fixed point number with 14 fractional bits
        .
    fixed = int(round(num * (2 ** 14)))

    # Convert the fixed point number to a binary string with 16 bits (including 14 fractional
         bits).
    binary = bin(fixed)[2:].zfill(16)

    return binary
```

Listing 6.5: Parameter translation function - *helper_functions.py*

6.3. Assembler outputs

The Listing below (6.6) shows an example of a diamond-based compliant quantum circuit compiled at QISA level. The circuit is the Hadamard Gate (*H-gate*) [5], which turns $|0\rangle$ into $(|0\rangle + |1\rangle)/2$ and $|1$ into $(|0\rangle + |1\rangle)/2$.

```
1 initialize q0
2 swapec q0 0
3 qgateuc q0 0 1.5707963267949 1.5707963267949 1
4 qgateuc q0 0 0.0 3.1415926535898 1
```

Listing 6.6: Hgate circuit, an example of quantum algorithm

Using the developed assembler, the quantum algorithm source code is compiled, resulting in the machine-level binary code provided in Listing 6.7. Notably, a *nop* instruction is added at the beginning of the algorithm and a *ebreak* instruction is added at the end to better isolate the execution during simulation.

```
00000000000000000000000000010011   //nop
10100111111110000000000010001011   //initialize q0
11000000011110000000000010001011   //swapec q0 0
01100100000000010001000110001011   //qldi_a r0 0 1.5707963267949
01100100000000110001000110001011   //qldi_a r0 1 1.5707963267949
01010000000000000000000000001011   //qgateuc q0 0 r0 1
00000000000010000000000110001011   //qldi_a r1 0 0.0
11001001000010100010000110001011   //qldi_a r1 1 3.1415926535898
01010000000010000000000000001011   //qgateuc q0 0 r1 1
00000000000100000000000001110011   //ebreak
```

Listing 6.7: Hgate circuit translated to machine-level qRV32 instructions

The compiled binary code accurately represents the corresponding QISA instructions in machine-level format. Each line of the binary code represents a translated instruction, with the opcode and associated parameters encoded in binary form. A comparison with the syntax defined in Table B.4 confirms its correctness.

The successful generation of the binary code from the QISA source code demonstrates the functionality and reliability of the assembler in translating quantum algorithms into executable machine code. By comparing the original QISA source code with the resulting binary code, it is evident that the assembler effectively performs the necessary translations, ensuring the accurate representation of quantum instructions in machine code format.

6.4. Conclusion

The development and evaluation of the assembler have provided valuable insights into the process of translating quantum algorithms from the QISA level to machine-level binary code. Through the identification of opcodes, substitution of pseudo-instructions, management of registers, and translation of parameters, the assembler successfully converts QISA instructions into executable machine code for the diamond-based quantum computer architecture.

The assembler has demonstrated its ability to perform the necessary translations, ensuring the faithful representation of quantum instructions in binary format. By examining the compiled binary code and comparing it with the original QISA source code, we can confidently conclude that the assembler is a reliable tool for converting quantum algorithms into executable machine code.

In conclusion, the assembler plays a crucial role in bridging the gap between the high-level representation of quantum algorithms and the low-level machine code required for their execution. Its successful implementation highlights its importance as a key component in the development and deployment of quantum stack, facilitating the realization of complex quantum computations on diamond-based quantum computers.

7

Results and Discussion

This chapter presents the results and analysis of the Global Controller architecture and of the control plane. Through software simulations and model evaluations, the functionality of the design is thoroughly examined.

The first result presented in Section 7.1 is the outcome of the core's architecture simulations. In particular, using Xilinx Vivado the behavior of Global Controller SoC is evaluated. Section 7.2 presents an evaluation of the controller and system performance, making use of the model discussed in Chapter 3. It provides an assessment of critical aspects such as timing and scalability.

The chapter concludes with discussions on the implications of the results, highlighting the strengths and limitations of the Global Controller (Section 7.3). These insights contribute to a deeper understanding of the system's performance and pave the way for future research in the field of quantum computing.

7.1. Software Simulations

The output generated by the assembler in the form of object files serves as a direct input for the Global Controller core, enabling the thorough testing of its behavior and functionality through logical simulations. In order to facilitate comprehensive simulations, two additional ports have been added to the SoC, allowing the direct recording of the NV center address and the communication packet generated for each instruction.

The generated output of the core when loaded with H-gate instructions (List. 6.6, 6.7) is presented below. This outcome serves as a valuable reference for future comparisons with alternative simulation approaches, such as the simulator developed in [40], as well as potential hardware realizations of the Global Controller.

```
boot_instr = D:/User/Documenti/TuDelft/Thesis/qrv_assembler/outputs/hgate_bin.txt
    packet #          1
    NVnode       =    0
    quantum_pack = 000000000000000c

    packet #          2
    NVnode       =    0
    quantum_pack = 600000000000000f

    packet #          3
    NVnode       =    0
    quantum_pack = 2000000000c910c9103

    packet #          4
    NVnode       =    0
    quantum_pack = 2000000001922000003
```

Listing 7.1: Post-implementation simulation results for the H-gate circuit

Furthermore, listing E.1 of Appendix E showcases the core's example outputs for each qRV32 instruction. These simulations provide an accurate representation of the core's behavior, closely resembling its real hardware counterpart. For additional simulation results, please refer to Appendix E.

7.2. Model evaluations

The model introduced in Chapter 3 offers a means to gauge the efficacy of the Global Controller by providing a metric. While the model's primary purpose is to assess the entire control plane, it can provide a partial evaluation of the system's parallelism based on the characteristics of the Global Controller.

While the relationship among the latency parameters (Table 3.1) has been defined in Equation 3.10, by only considering pre-processing time, write-back time, and quantum execution time, one can generate an early estimation of the system's parallelism. This implies disregarding the remaining parameters by setting to zero their respective coefficients. Accordingly, by employing Equation 7.1, a simplified version of Equation 3.8, the parallelism is calculated as described above.

$$\#LCs(i) \geq \frac{Tqex_L(i) + Twb_L}{Tpe_G(i)} \tag{7.1}$$

For the analysis, it is assumed that both Global and Local Controllers operate at the nominal operating frequency achievable by the base core, which has been reported as $55MHz$ in Section 5.1.1. Furthermore, it is assumed that the Local Controllers require only one clock cycle to write back results to their internal registers. The measured quantum execution times are reported in table A.2 of Appendix A,

The graph below illustrates the available data and corresponding results for a subset of the QISA, specifically the subset for which the quantum execution times have been measured and excluding calibration instructions (magbias, detectCarbon and rabicheck). By employing Equation 7.1, the parallelism is calculated as described in the paragraph above. The resulting value for each instruction is depicted on the vertical axis.

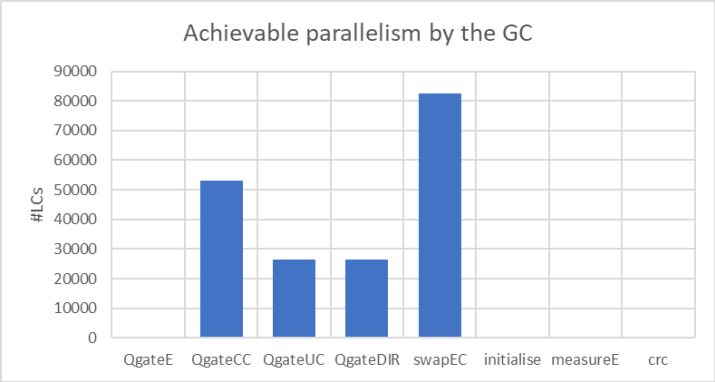


Figure 7.1: Achievable parallelism by the GC for QISA instructions, at $f_{clk} = 55MHz$

It is evident that the quantum execution time varies significantly among the listed instructions. Slower instructions such as *QgateCC* and *SwapEC* exhibit high parallelism, while faster ones like *QgateE* and *initialise* necessitate the GC to address them in the subsequent clock cycle. Without considering the statistical frequency of each instruction, the model suggests that an initial average of 23605 LCs can be simultaneously addressed by the GC, a result that exceeds the design target of 1000 LCs technically addressable by the encoding format (Sec.4.3). To obtain an optimized architecture in line with the requirements, two critical factors, namely operating frequency and communication delay, must be further analyzed.

Firstly, the presented simplified formula disregards communication delays, failing to account for the significant latency of the system. By introducing a fixed and explanatory $1\mu s$ Round-Trip Time delay time, it becomes evident that instructions with exceptionally swift execution times ($Tqex$) can enhance their parallelism. In this scenario, the instruction *initialise* reaches a parallelism of 56 LCs.

Secondly, the tuning of operating frequencies for both the Global and Local Controllers allows the adjustment of parallelism value. In particular, the parallelism value is directly proportional to the GC frequency, allowing for flexible scaling of parallelism for each instruction. For example, reducing the GC clock frequency to $5.5MHz$ allows the same architecture to control 2364 LCs concurrently. Additionally, tuning the LC frequency leads to a secondary effect that particularly impacts the parallelism of instructions with extremely fast execution times. For instance, lowering the LCs frequency to $100KHz$, for example to limit their power consumption, results in an increased average parallelism of 23971 LCs.

In conclusion, the model provides valuable insights into the efficacy of the Global Controller and the system's parallelism. Selecting appropriate values for all these parameters is imperative in the final system design to meet the parallelism requirement, either considering the entire QISA with the average as a target or ensuring parallelism ≥ 1000 for each instruction.

7.3. Conclusion

This chapter presented the evaluation of the proposed Global Controller architecture, which aims to provide a scalable and efficient solution for the coordination of multiple Local Controllers in a distributed system. The evaluation consisted of two parts: software simulations and model evaluations.

The software simulations were performed using Vivado, and they tested the functionality of the design under different scenarios and parameters. The results showed how the core behaves when loaded with different instructions and demonstrated the robustness of the controller. The model evaluations, on the other hand, offered a more abstract view of the system's performance, estimating its parallelism. The partial results suggest a potentially significant achievable parallelism for the GC, but that it can be tuned by considering factors such as communication delays and clock frequency in the system.

In conclusion, this chapter validated the feasibility and effectiveness of the proposed global controller architecture and provided some insights into its strengths and weaknesses. This analysis can be used to further improve the design and pave the way for future research in quantum computing. The next chapter will discuss future work and possible extensions of this research.

8

Conclusion

The last chapter of this thesis concludes the work. In Section 8.1, a summary of each chapter will be provided. Section 8.2 presents the main contribution of the thesis. Eventually, Section 8.3 lists suggestions for future work.

8.1. Summary

This section provides a summary of the proposed work.

- **Chapter** 2 provides essential background knowledge for understanding the thesis work. It covers Quantum Information Theory (Sec.2.1), including principles of quantum computing. NV Center Quantum Computers (Sec.2.2) are explored, focusing on qubits and control mechanisms like analog (Sec.2.2.2) and digital control (Sec.2.2.3). The chapter also introduces the RISC-V Instruction Set Architecture (Sec.2.3). This foundation enables subsequent chapters to delve into the implementation of control electronics in quantum computers.
- **Chapter** 3 develops a mathematical model for optimizing the allocation of local controllers to global controllers in the Digital Control Layer. It explores control plane modeling, including instruction execution (Sec.3.1) and time diagrams for ideal, processing, and communication scenarios (Sec.3.2.1 - 3.2.3). This model will be utilized in the subsequent thesis work to evaluate the design of the global controller and its performance.
- **Chapter** 4 integrates the diamond QISA into the qRV32 architecture. It introduces RISC-V extensions (Sec.4.1) and the Diamond QISA (Sec.4.2). The qRV32 ISE is explored, covering instruction encodings (Sec.4.3.1) and communication with Local Controllers (Sec.4.3.2). Three encoding options are analyzed, considering factors like hardware complexity. The communication protocol between the Global Controller and the Local Controllers is discussed, highlighting the structure of the communication message and encoding approaches. Encoding C and the variable-length packet scheme are selected for the design.
- **Chapter** 5 delves into the hardware implementation phase, which follows the design of the qRV32 ISE discussed in the previous chapter. It surveys various 32-bit RISC-V cores (Sec.5.1) and presents the design of the Global Controller (Sec.5.2), including its architecture (Sec.5.2.2) and tailored functional units (Sec.5.2.3). The chapter also covers the design of the Soc (Sec.5.3), addressing aspects such as system memory (Sec.5.3.1) and UART interface (Sec.5.3.2). Through this hardware implementation journey, the integration of the Global Controller with the necessary peripherals is achieved. The SoC design presented here provides a minimal yet functional setup for the GC Core.
- **Chapter** 6 describes the development of a custom assembler for qRV32 instructions, translating quantum algorithms from the QISA level to machine-level code (Sec.6.1). It analyzes assembly-level instructions and describes the development steps including opcode identification 6.2.1, the substitution of pseudo-instructions (Sec.6.2.2), register management (Sec.6.2.3), and parameter translation (Sec.6.2.4). The assembler's successful implementation highlights its importance in the quantum stack, enabling the execution of complex quantum computations on diamond-based quantum computers (Sec.6.3).

- **Chapter** 7 evaluates the proposed Global Controller design for coordinating multiple Local Controllers in a distributed system. It includes software simulations using Vivado to assess the behavior of the Global Controller SoC (Sec.7.1), as well as model evaluations (Sec.7.2) of system performance based on the control plane model (Ch.3). The results provide valuable insights into the strengths and weaknesses of the Global Controller, facilitating further design enhancements and future research in the field of quantum computing. The chapter concludes by discussing the implications of the evaluation results (Sec.7.3), paving the way for future work and extensions in this area.

8.2. Main contributions

In this section, the main contributions of this work are presented:

1. **Developed a mathematical experimental model to estimate the achievable parallelism of the control system**. The model takes into account all the elements of the layered control system. It is fundamental to estimate the capabilities of the system and allows evaluation of future design choices.
2. **Designed a RISC-V-compliant Instruction Set Extension for the control of a Diamond-based quantum computer**. To meet the first goal of this work, the structure and content of the QISA proposed in [16] have been transposed into a novel RISC-V ISE, called qRV32. The final encoding ensures compatibility with the requirements listed in Section 4.3.1. Three different encodings have been explored and evaluated.
3. **Defined a communication standard between GC and LC of the system**: Local Controllers are the final target of the QISA, as they control the NV centers. To correctly send controls and data to the LCs, a 75-bit message has been structured, comprehending instruction parameters and opcode. This met the second goal set for this thesis.
4. **Designed the first version of the GC hardware, based on the CV32E40P core**: A control core and relative SoC have been designed to address the execution of qRV32 instructions. The in-pipeline execution of custom instruction is achieved by the design of functional units, such as the QDecoder, achieving the third objective of this work. The minimal SoC comprises instruction and data memories, fundamental for the correct execution of quantum programs.
5. **Developed a functional assembler targeting the qRV32 ISE**: A functional assembler that translates assembly-level QISA instructions into machine-level qRV32 binary instructions has been developed in Python. The assembler plays a crucial role in bridging the gap between the high-level representation of quantum algorithms and the low-level machine code required for their execution.

8.3. Future work

Despite the proposed ISE and the control core do meet the design goal set, future work can be done.

System On Chip

The proposed work only describes the initial step in the design of the complete Global Controller architecture, and additional work is required for the completion of the System On Chip. In particular, part of the reason for choosing CV32E40P as the base core is its compatibility with the PULP project and its SoCs. The GC core can, in fact, be integrated as part of the PULPissimo Micro-controller Unit (MCU) to exploit the native DMA interface available.

Moreover, specific hardware must be designed to connect the GC and the LCs network. This can be achieved by custom BUS or router systems, which can directly connect to the ports provided on the GC core. In designing such structure, particular attention must be given to latency and power consumption, due to the high number of elements and the size of the communication packet.

Assembler

The assembler currently employs a naive implementation that achieves the basic requirements of assembling assembly code into machine code. However, several crucial areas remain to be addressed to enhance the assembler's overall performance and functionality.

Firstly, optimizations must be explored and implemented to improve the efficiency of the translation process, enabling the assembler to generate machine code more quickly and effectively. Additionally, support for advanced features such as macros, conditional assembly, and modular programming should be incorporated to enhance code reusability and maintainability. Furthermore, error handling and reporting mechanisms need to be strengthened to provide more informative and user-friendly error messages, aiding developers in debugging their assembly code effectively. Lastly, rigorous testing and validation procedures are necessary to ensure the correctness and reliability of the assembler, including the ability to handle edge cases and adhere to the specifications of the target architecture.

By addressing these areas, the assembler can evolve into a robust and high-performance tool, offering developers a comprehensive and efficient solution for assembling assembly code to qRV32.

Toolchain

The integration of the assembler and the compiler developed in [16] poses an essential step in creating a comprehensive software toolchain capable of generating object files from a high-level representation of quantum algorithms. To accomplish this integration, a series of steps must be followed.

Initially, the compiler needs to be seamlessly integrated into the toolchain, necessitating the creation of a command-line interface or API that enables programmatic invocation of the compiler while also defining the desired output format. As for the main assembler component, an open-source RISC-V assembler must be selected, and its corresponding command-line interface or API identified for smooth integration. Additionally, the qRV32 assembler, being a custom component, necessitates the development of a command-line interface. Furthermore, the toolchain must be configured to pass the assembly code generated by the standard assembler to the custom assembler for further processing. Lastly, the object file generation process entails implementing a mechanism to merge or combine the output object files generated by both the standard and custom assemblers into a single object file. Additionally, steps need to be taken to address any conflicts that may arise concerning symbols or memory addresses between the standard and custom assembly code.

By following these steps, a cohesive and powerful software toolchain can be created, enabling efficient object file generation from a high-level quantum algorithm representation.

References

[1] AIMultiple. *Quantum Entanglement: What is it & Why is it Important in 2023?* 2021. URL: https://research.aimultiple.com/quantum-computing-entanglement/.

[2] K. Asanović and et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] Z. Z. Bandic and et al. *High performance RISC-V embedded SweRV™ EH1 core: microarchitecture, performance and SSD controller implementations*. Tech. rep. Western Digital, 2019.

[4] H. Bernien. "Control, Measurement and Entanglement of Remote Quantum Spin Registers in Diamond". MA thesis. TU Delft, 2014.

[5] G. Casati and G. Benenti. "Quantum Computation and Chaos". In: *Encyclopedia of Condensed Matter Physics*. Ed. by Franco Bassani, Gerald L. Liedl, and Peter Wyder. Oxford: Elsevier, 2005, pp. 9–17. ISBN: 978-0-12-369401-0. DOI: https://doi.org/10.1016/B0-12-369401-9/01146-3. URL: https://www.sciencedirect.com/science/article/pii/B0123694019011463.

[6] M. Cerezo. *The More You Know: Bloch Sphere*. 2016. URL: https://entangledphysics.com/2016/03/23/the-more-you-know-bloch-sphere/.

[7] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. "Experimental Implementation of Fast Quantum Searching". In: *Phys. Rev. Lett.* 80 (1998), pp. 3408–3411.

[8] Rigetti Computing. *Aspen-m-2 quantum processor*. 2022. URL: https://qcs.rigetti.com/qpus.

[9] *Concurrence (quantum computing)*. URL: https://en.wikipedia.org/wiki/Concurrence_%28quantum_computing%29.

[10] *CoreMark CPU Benchmark*. URL: https://www.eembc.org/coremark/.

[11] *CVE3040P*. URL: https://github.com/pulp-platform/cv32e40p.

[12] Mihaela Damian et al. "Scaie-V". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference* (2022). DOI: 10.1145/3489517.3530432.

[13] M. H. Devoret, A. Wallraff, and J. M. Martinis. *Superconducting Qubits: A Short Review*. 2004. arXiv: cond-mat/0411174 [cond-mat.mes-hall].

[14] Ivan B. Djordjevic. "Chapter 7 - Quantum Information Theory Fundamentals". In: *Quantum Information Processing, Quantum Computing, and Quantum Error Correction (Second Edition)*. Ed. by Ivan B. Djordjevic. Second Edition. Academic Press, 2021, pp. 251–286. ISBN: 978-0-12-821982-9. DOI: https://doi.org/10.1016/B978-0-12-821982-9.00012-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128219829000125.

[15] F. et al. Dolde. "High-fidelity spin entanglement using optimal control". In: *Nature Communications* 5.3371 (2014). DOI: https://doi.org/10.1038/ncomms4371.

[16] Matti Dreef. *Compiler for Color Center-based Quantum Computers*. 2023. URL: http://resolver.tudelft.nl/uuid:5bdc7e3a-7d83-4343-8b61-23126ea7c764.

[17] Richard P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6-7 (1982), pp. 467–488.

[18] A. et al. Gottscholl. "Initialization and read-out of intrinsic spin defects in a van der Waals crystal at room temperature". In: *Nature Materials* 19 (2020), pp. 540–545. DOI: https://doi.org/10.1038/s41563-020-0619-6.

[19] F. K. Gürkaynak and L. Benini. *Working with RISC-V - Part 1 of 4 : Introduction to RISC-V ISA*. URL: https://pulp-platform.org/docs/hipeac/acaces2020/01_Intro_RISC-V.pdf.

[20] Y. et al. He. “A two-qubit gate between phosphorus donor electrons in silicon”. In: *Nature* 571 (2019), pp. 371–375.

[21] Carsten Heinz et al. “A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors”. In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019, pp. 1–8. DOI: 10.1109/ReConFig48160.2019.8994796.

[22] Roland Höller et al. “Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation”. In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. 2019, pp. 1–6. DOI: 10.1109/MECO.2019.8760205.

[23] IBM. *IBM unveils breakthrough 127-qubit quantum processor*. 2021. URL: https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor.

[24] H. et al. Kaufmann. “Fast ion swapping for quantum-information processing”. In: *Phys. Rev. A* 95.5 (2017).

[25] T. Knaflič. *The Weak Measurement in Quantum Mechanics*. 2012. URL: http://www-f1.ijs.si/~ramsak/seminarji/KnaflicSibka.pdf.

[26] E. Knill and et al. “Introduction to Quantum Information Processing”. In: *LA Science* (2002). DOI: https://doi.org/10.48550/arXiv.quant-ph/0207171.

[27] J. Martinis and S. Boixo. *Quantum Supremacy Using a Programmable Superconducting Processor*. 2019. URL: https://ai.googleblog.com/2019/10/quantum-supremacy-using-programmable.html.

[28] *Maskless and targeted creation of arrays of color centers in diamond with i-FIB*. URL: https://www.orsayphysics.com/204-i-fib.

[29] Wolfram MathWorld. *Bloch Vector*. URL: https://mathworld.wolfram.com/BlochVector.html.

[30] *Neorv32*. URL: https://github.com/stnolting/neorv32.

[31] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary edition. Cambridge University Press, 2010. Chap. 4.2 Single qubit operations.

[32] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary edition. Cambridge University Press, 2010. Chap. 4.3 Controlled operations.

[33] TU Delft OCW. *NV center qubits*. URL: https://ocw.tudelft.nl/course-lectures/3-2-1-nv-center-qubits/.

[34] E. Peňa and M. G. Legaspi. *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter*. URL: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html.

[35] S. Pezzagna and J. Meijer. “Quantum computer based on color centers in diamond”. In: *Appl. Phys. Rev.* 8 (2021). DOI: https://doi.org/10.1063/5.0007444.

[36] *picorv32*. URL: https://github.com/YosysHQ/picorv32.

[37] *RISC-V ”V” Vector Extension*. Tech. rep. RISC-V International, 2023. URL: https://github.com/riscv/riscv-v-spec.

[38] *RISC-V Cryptography Extensions Volume I*. Tech. rep. RISC-V International, 2022. URL: https://github.com/riscv/riscv-crypto.

[39] *RISC-V International*. URL: https://riscv.org/.

[40] Folkert de Ronde. *Quantum simulator for electronical control of diamond spin qubits*. 2022. URL: https://repository.tudelft.nl/islandora/object/uuid:6e56a4c9-a248-4fc0-8d63-425688c06b4d?collection=education.

[41] Peter W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM Journal on Computing* 26.6 (1996), pp. 1484–1509.

[42] Berkley University. *C191 - Lectures 8 and 9 - Measurement in Quantum Mechanics*. 2021. URL: https://inst.eecs.berkeley.edu/~cs191/fa14/lectures/lecture89.pdf.

[43] C. S. Unnikrishnan. “Quantum Non-Demolition Measurements: Concepts, Theory and Practice”. In: *Current Science* 109.11 (Dec. 2015), p. 2052. DOI: 10.18520/v109/i11/2052-2060. URL: https://doi.org/10.18520%2Fv109%2Fi11%2F2052-2060.

[44] *VexRiscv*. URL: https://github.com/SpinalHDL/VexRiscv.

[45] A. Waterman. “Design of the RISC-V Instruction Set Architecture”. PhD thesis. EECS Department, University of California, Berkeley, Jan. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

[46] A. Waterman, K. Asanovi, and SiFive Inc. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. Tech. rep. CS Division, EECS Department, University of California, Berkeley, 2019.

[47] Q. van Wingerden. “Quantum Computer Microarchitecture”. MA thesis. TU Delft, 2021.

A

QISA Instructions

This appendix contains the QISA proposed in [16] [40] and the relative measured quantum execution times.

Instruction	*Parameters*
set	NV, register, value
qgateE	NV, phase, angle
qgateZE	NV, angle
qgateZC	NV, C13, angle
qgateCC	NV, C13, phase, angle
qgateUC	NV, C13, phase, angle, preserve
qgateDIR	NV, C13, phase, angle, dir
initialize	NV
measureE	NV
swapEC	NV, C13
swapCE	NV, C13, basis
entangle	NV, direction
detectCarbon	NV, sweepStart, sweepStep, sweepStop, measMax
magbias	NV, sweepStart, sweepStep, sweepStop
rabicheck	NV, sweepStart, sweepStep, sweepStop, measMax
crc	NV

Table A.1: Proposed Quantum Instruction Set Architecture [16] [40]

Instruction	*Quantum Execution time* [s]
QgateE	2.70E-10
QgateCC	2.90E-03
QgateUC	1.45E-03
QgateDIR	1.45E-03
SwapEC	1.50E-03
initialise	6.00E-09
measureE	6.00E-09
magBias	5.00E-05
detectCarbon	7.40E-04
rabiCheck	6.27E-09
crc	5.00E-08

Table A.2: Measured quantum execution times. These preliminary values where provided by Folkert de Ronde (TU Delft, QCE) at the start of the thesis project, and are taken as a reference.

B

qRV32 figures

B.1. qRV32 Parameters Analysis

The following table contains the description of the qrV32 parameters units and their order of magnitude.

Instruction	Paramer	Encoding	Unit
qgate-	phase	$< 16, 14 >$ unsigned	π
	angle	$< 16, 14 >$ unsigned	π
magbias	sweepStart	16 bit unsigned	MHz
	sweepStep	16 bit unsigned	KHz
	sweepStop	16 bit unsigned	MHz
detectcarbon	sweepStart	16 bit unsigned	KHz
	sweepStep	16 bit unsigned	Hz
	sweepStop	16 bit unsigned	KHz
	measAmt	16 bit unsigned	-
rabicheck	sweepStart	16 bit unsigned	ns
	sweepStep	16 bit unsigned	ps
	sweepStop	16 bit unsigned	ns
	countMax	16 bit unsigned	-

Table B.1: qrV32 parameters units and order of magnitude

B.2. qRV32 Encoding Tables

This section presents the three different qrV32 encoding tables proposed in this report.

Table B.2: qRV32 Encoding A

31 - 30	29 - 27	26 - 23	22 - 19	18 - 16	15 - 10	9 - 7	6 0		
0 0	x	QRS2	QRS1	x	NVNode	0 0 0	0 0 0 1 0	1 1	qgateE
0 1	x	QRS2	QRS1	C13	NVNode	0 0 0	0 0 0 1 0	1 1	qgateCC
1 0	p	QRS2	QRS1	C13	NVNode	0 0 0	0 0 0 1 0	1 1	qgateUC
1 1	d	QRS2	QRS1	C13	NVNode	0 0 0	0 0 0 1 0	1 1	qgateDIR
0 0	x	QRS2	QRS1	x	NVNode	0 0 1	0 0 0 1 0	1 1	zgateE
1 1	x	QRS2	QRS1	C13	NVNode	0 0 1	0 0 0 1 0	1 1	zgateCC

31 - 28	27 — 23	22 —- 19	18 - 16	15 —— 10	9 - 7	6 0		
1 0 0 0	x	QREG	x	NVNode	0 0 1	0 0 0 1 0	1 1	set
1 0 0 1	direction	QREG	x	NVNode	0 0 1	0 0 0 1 0	1 1	entangle
1 0 1 0	x	x	x	NVNode	0 0 1	0 0 0 1 0	1 1	initialize
1 0 1 1	x	x	C13	NVNode	0 0 1	0 0 0 1 0	1 1	measureE
1 1 0 0	x	x	C13	NVNode	0 0 1	0 0 0 1 0	1 1	swapEC
1 1 0 1	basis	x	C13	NVNode	0 0 1	0 0 0 1 0	1 1	swapCE
1 1 1 0	x	x	C13	NVNode	0 0 1	0 0 0 1 0	1 1	crc

31 — 28	27 — 24	23 — 20	19 — 16	15 —- 10	9 - 7	6 0		
QREG 4	QREG 3	QREG 2	QREG 1	NVNode	0 1 0	0 0 0 1 0	1 1	detectC13
QREG 4	QREG 3	QREG 2	QREG 1	NVNode	0 1 1	0 0 0 1 0	1 1	magbias
QREG 4	QREG 3	QREG 2	QREG 1	NVNode	1 0 0	0 0 0 1 0	1 1	rabicheck

31 ————— 21	20 - 19	18 —— 14	13 ——10	8 - 7	6 0		
immediate [11:0]	0 0	QREG2	QREG1	1 0 1	0 0 0 1 0	1 1	qld
immediate [11:0]	0 1	QREG2	QREG1	1 0 1	0 0 0 1 0	1 1	qst
x	1 0	gpREG	QREG	1 0 1	0 0 0 1 0	1 1	move *from*
x	1 1	gpEG2	QREG	1 0 1	0 0 0 1 0	1 1	move *to*

31 ———————————— 16	15 - 14	13 — 10	8 - 7	6 0		
immediate [14:0]	x	QREG1	1 1 1	0 0 0 1 0	1 1	qldi

Table B.3: qRV32 Encoding B

31 — 27	26 ——————————— 14	13 - 10	9 - 7	6 0	
1 1 1 1 1	NV node	x	0 0 0	0 0 0 1 0	1 1 NV-choose

31 — 27	26 - 24	23 — 19	18 — 14	13 - 10	9 - 7	6 0	
0 0 0 0 0	x	QRS2	QRS1	x	0 0 0	0 0 0 1 0	1 1 qgateE
0 0 0 0 1	x	QRS2	QRS1	C13	0 0 0	0 0 0 1 0	1 1 qgateCC
0 0 0 1 0	p	QRS2	QRS1	C13	0 0 0	0 0 0 1 0	1 1 qgateUC
0 0 0 1 1	d	QRS2	QRS1	C13	0 0 0	0 0 0 1 0	1 1 qgateDIR
0 0 1 0 0	x	QRS2	QRS1	x	0 0 0	0 0 0 1 0	1 1 zgateE
0 0 1 0 1	x	QRS2	QRS1	C13	0 0 0	0 0 0 1 0	1 1 zgateCC
0 0 1 1 0	x	QRS2	QRS1	x	0 0 0	0 0 0 1 0	1 1 set
0 0 1 1 1	direct	x	x	x	0 0 0	0 0 0 1 0	1 1 entangle
0 1 0 0 0	x	x	x	x	0 0 0	0 0 0 1 0	1 1 initialize
0 1 0 0 1	x	x	x	C13	0 0 0	0 0 0 1 0	1 1 measureE
0 1 0 1 0	x	x	x	C13	0 0 0	0 0 0 1 0	1 1 swapEC
0 1 0 1 1	basis	x	x	C13	0 0 0	0 0 0 1 0	1 1 swapCE
0 1 1 0 0	x	x	x	x	0 0 0	0 0 0 1 0	1 1 crc

31 - 30	29 — 25	24 — 20	19 — 15	14 — 10	9 - 7	6 0	
0 0	QREG 4	QREG 3	QREG 2	QREG 1	1 0 0	0 0 0 1 0	1 1 detectC13
0 1	QREG 4	QREG 3	QREG 2	QREG 1	1 0 0	0 0 0 1 0	1 1 magbias
1 0	QREG 4	QREG 3	QREG 2	QREG 1	1 0 0	0 0 0 1 0	1 1 rabicheck

31 ————— 21	20 - 19	18 —— 14	13 ——10	8 - 7	6 0	
immediate [11:0]	0 0	QREG2	QREG1	1 0 1	0 0 0 1 0	1 1 qld
immediate [11:0]	0 1	QREG2	QREG1	1 0 1	0 0 0 1 0	1 1 qst
x	1 0	gpREG	QREG	1 0 1	0 0 0 1 0	1 1 move *from*
x	1 1	gpEG2	QREG	1 0 1	0 0 0 1 0	1 1 move *to*

31 ——————————— 16	15 - 14	13 — 10	8 - 7	6 0	
immediate [14:0]	x	QREG1	1 1 1	0 0 0 1 0	1 1 qldi

Table B.4: qRV32 Encoding C

31 - 29	28	27 — 24	23 —— 19	18 ———————— 9	8 - 7	6 ——— 2	1 0	
0 0 0	x	x	QREGs	NVNode	0 0	0 0 0 1 0	1 1	QgateE
0 0 1	x	C13	QREGs	NVNode	0 0	0 0 0 1 0	1 1	QgateCC
0 1 0	P	C13	QREGs	NVNode	0 0	0 0 0 1 0	1 1	QgateUC
0 1 1	D	C13	QREGs	NVNode	0 0	0 0 0 1 0	1 1	QgateDIR
1 0 0	x	x	QREGs	NVNode	0 0	0 0 0 1 0	1 1	ZgateE
1 0 1	x	C13	QREGs	NVNode	0 0	0 0 0 1 0	1 1	ZgateCC

31	30 — 27	26 – 24	23 —— 19	18 ———————— 9	8 - 7	6 ——— 2	1 0	
1	0 0 0 1	x	QREGs	NVNode	0 1	0 0 0 1 0	1 1	detectC13
1	0 0 1 0	x	QREGs	NVNode	0 1	0 0 0 1 0	1 1	magBias
1	0 0 1 1	x	QREGs	NVNode	0 1	0 0 0 1 0	1 1	RabiCheck
1	0 1 0 0	x		NVNode	0 1	0 0 0 1 0	1 1	initialise
1	0 1 0 1	x		NVNode	0 1	0 0 0 1 0	1 1	measureE
1	0 1 1 0	x		NVNode	0 1	0 0 0 1 0	1 1	crc
1	1 1 1 1	x		NVNode	0 1	0 0 0 1 0	1 1	escape

31	30 — 27	26 — 23	22 21	20 19	18 ———————— 9	8 - 7	6 ——— 2	1 0	
1	0 1 1 1	C13	basis	x	NVNode	0 1	0 0 0 1 0	1 1	swapCE
1	1 0 0 0	C13	x	x	NVNode	0 1	0 0 0 1 0	1 1	swapEC

31	30 29	28 27	26 – 24	23 —— 19	18 ———————— 9	8 7	6 ——— 2	1 0	
0	0 1	x	direct	x	NVNode	0 1	0 0 0 1 0	1 1	entangle
0	0 0	LREG		QREGs	NVNode	0 1	0 0 0 1 0	1 1	set

31 30	29	28 ——— 24	23 ——— 19	18 ———————— 9	8 7	6 ——— 2	1 0	
1 0	0	gpREGs	QREGd	x	1 0	0 0 0 1 0	1 1	move *to*
1 0	1	gpREGd	QREGs	x	1 0	0 0 0 1 0	1 1	move *from*

31 30	29 ——— 24	23 —— 19	18 — 14	13 ——— 9	8 7	6 ——— 2	1 0	
0 0	imm[10:5]	QREGd	QREGs	imm[4:0]	1 0	0 0 0 1 0	1 1	qld rd
0 1	imm[10:5]	$QREGs_2$	$QREGs_1$	imm[4:0]	1 0	0 0 0 1 0	1 1	qst rs2

31 ———————— 24	23 —— 19	18 17	16 ———————— 9	8 7	6 ——— 2	1 0	
immediate[15:8]	QREGd	sel	immediate[7:0]	1 1	0 0 0 1 0	1 1	qldi

B.3. qRV32 Global To Local Communication

This section explicates the encoding utilized in the global-to-local communication package.

Instruction	Binary Encoding
qRV_reset_op	00000
qRV_escape	11111
qRV_QgateE	00001
qRV_QgateCC	00010
qRV_QgateUC	00011
qRV_QgateDIR	00100
qRV_ZgateE	00101
qRV_ZgateCC	00110
qRV_set	00111
qRV_ent	01000
qRV_detectC	01001
qRV_magBias	01010
qRV_RabiCheck	01011
qRV_init	01100
qRV_measurE	01101
qRV_crc	01110
qRV_swapEC	01111
qRV_swapCE	10000
qRV_ld	10001
qRV_st	10010
qRV_mv_to_q	10011
qRV_mv_from_q	10100
qRV_ldi	10101

Table B.5: qRV32 opcode-field encoding

C

GC & SoC HDL design

C.1. Global Controller

This section contains snippets of SystemVerilog code that describes the Global Controller hardware.

QDecoder

```
3051
3052 ////////////////////////////////////////////////
3053 ////////////////////////////////////////////////
3054 //         qRV32 custom instructions          //
3055 ////////////////////////////////////////////////
3056 ////////////////////////////////////////////////
3057
3058 OPCODE_CUST0: begin
3059     //decoding of the first 2 bits of the instruction
3060     unique case (instr_rdata_i[8:7])
3061       2'b00: begin
3062         //routing fixed parts of the instruction
3063         NVnode = instr_rdata_i[18:9];       //node address
3064         qReg_rAddr = instr_rdata_i[23:19];    //quantum register address
3065         //decoding of the last 3 bits of the instruction
3066         unique case (instr_rdata_i[31:29])
3067           3'b000: begin //instruction 1: QgateE
3068             qOP = qRV_QgateE;   //internal operation code
3069           end
3070           3'b001: begin //instruction 2: QgateCC
3071             qOP = qRV_QgateCC;
3072             qParameters = instr_rdata_i[29:24];
3073           end
3074           3'b010: begin //instruction 3: QgateUC
3075             qOP = qRV_QgateUC;
3076             qParameters = instr_rdata_i[29:24];
3077           end
3078           3'b011: begin //instruction 4: QgateDIR
3079             qOP = qRV_QgateDIR;
3080             qParameters = instr_rdata_i[29:24];
3081           end
3082           3'b100: begin //instruction 5: ZgateE
3083             qOP = qRV_ZgateE;
3084           end
3085           3'b101: begin //instruction 6: ZgateCC
3086             qOP = qRV_ZgateCC;
3087             qParameters = instr_rdata_i[29:24];
3088           end
3089           default: begin
3090             illegal_insn_o = 1'b1;
3091           end
3092         endcase
3093       end
3094       2'b01: begin
```

```
3095        //routing fixed parts of the instruction
3096        NVnode = instr_rdata_i[18:9];
3097        //decoding of the last bit of the instruction
3098        unique case (instr_rdata_i[31])
3099
3100          1'b0: begin
3101            //decoding of bits 29 to 30 of the instruction
3102            unique case (instr_rdata_i[30:29])
3103              2'b00: begin //instruction 7: set
3104                qOP = qRV_set;
3105                qReg_rAddr = instr_rdata_i[23:19];
3106                qParameters = instr_rdata_i[29:24];
3107              end
3108              2'b01: begin //instruction 8: entangle
3109                qOP = qRV_ent;
3110                qParameters = instr_rdata_i[29:24];
3111              end
3112              default: begin
3113                illegal_insn_o = 1'b1;
3114              end
3115            endcase
3116          end
3117          1'b1: begin
3118            //routing fixed parts of the instruction
3119            NVnode = instr_rdata_i[18:9];
3120            //decoding of bits 27 to 30 of the instruction
3121            unique case (instr_rdata_i[30:27])
3122              4'b0001: begin //instruction 10: detectC13
3123                qOP = qRV_detectC;
3124                qReg_rAddr = instr_rdata_i[23:19];
3125              end
3126              4'b0010: begin //instruction 11: magneticBiasing
3127                qOP = qRV_magBias;
3128                qReg_rAddr = instr_rdata_i[23:19];
3129              end
3130              4'b0011: begin //instruction 12: RabiCheck
3131                qOP = qRV_RabiCheck;
3132                qReg_rAddr = instr_rdata_i[23:19];
3133              end
3134              4'b0100: begin //instruction 13: initialise
3135                qOP = qRV_init;
3136                //no additional parameters
3137              end
3138              4'b0101: begin //instruction 14: measureE
3139                qOP = qRV_measurE;
3140                //no additional parameters
3141              end
3142              4'b0110: begin //instruction 15: crc
3143                qOP = qRV_crc;
3144                //no additional parameters
3145              end
3146              4'b0111: begin //instruction 16: swapCE
3147                qOP = qRV_swapCE;
3148                qParameters = instr_rdata_i[26:21];
3149              end
3150              4'b1000: begin //instruction 17: swapEC
3151                qOP = qRV_swapEC;
3152                qParameters = instr_rdata_i[26:21];
3153              end
3154              4'b1111: begin //instruction 19: escape
3155                qOP = qRV_escape;
3156                //leave empty
3157              end
3158            endcase
3159          end
3160          default: begin
3161            illegal_insn_o = 1'b1;
3162          end
3163
3164        endcase //instr_rdata_i[31]
3165      end // minor opcode = 01
```

```
      2'b10: begin
        //decoding bits 31 and 30 of the instruction
        unique case ({instr_rdata_i[31:30]})
          2'b00: begin //instruction 20: qld
            qOP = qRV_ld;
            data_req = 1'b1;                //communication requested with memory
            data_sign_extension_o = 2'b00;  //no sign extension on red data
            data_type_o = 2'b00;            //load word
            data_load_event_o = 1'b1;       //load event

            qReg_mem_we = 1'b1;                       //w_en to qReg
            qReg_mem_wAddr = instr_rdata_i[23:19];    //write address to qReg mem port (destination register)
            qReg_mem_rAddr = instr_rdata_i[18:14];    //read address to qReg mem port (memory base address)

            //using existing hardware paths to the WB stage (LSU)
            alu_op_a_mux_sel_o  = OP_A_REGA_OR_FWD;
            //operand_a_fw_mux_sel     = SEL_QREGFILE;  //this comes from the controller
            imm_b_mux_sel_o     = IMMB_QREG;  //send immediate field to LSU
            alu_op_b_mux_sel_o  = OP_B_IMM;
            alu_operator_o = ALU_ADD;
          end
          2'b01: begin //instruction 21: qst
            qOP = qRV_st;
            data_req = 1'b1;     //communication requested with memory
            data_we_o = 1'b1;    //write to memory
            data_type_o = 2'b00; //store word

            qReg_mem_rAddr = instr_rdata_i[23:19];   //read wdata from qReg2
            qReg_mem2_rAddr = instr_rdata_i[18:14];  //read memory base address from qReg1

            alu_op_a_mux_sel_o  = OP_A_REGA_OR_FWD;     //send address to LSU (qReg_mem2_rAddr)
            //operand_a_fw_mux_sel     = SEL_QREGFILE;
            imm_b_mux_sel_o     = IMMB_QREG;            //send immediate field to LSU
            alu_op_b_mux_sel_o  = OP_B_IMM;
            alu_op_c_mux_sel_o = OP_C_REGC_OR_FWD;      //send write data to LSU (qReg_mem_rAddr)
            //operand_a_fw_mux_sel     = SEL_QREGFILE;
            alu_operator_o = ALU_ADD;
          end
          2'b10: begin //instruction 18: move
            unique case (instr_rdata_i[29])
              1'b0: begin   //move to qReg: 18a
                qOP = qRV_mv_to_q;
                //read from regfile B port and send data to EX stage for qReg
                  //done at id stage level
                //send address and we to EX stage for qReg
                  qReg_mem_we = 1'b1;
                  qReg_mem_wAddr = instr_rdata_i[23:19];
                //route the data to the output port of ID stage: opA = 0, opB = regfile port B rData
                  alu_op_a_mux_sel_o  = OP_A_IMM;
                  imm_a_mux_sel_o     = IMMA_ZERO;
                  alu_op_b_mux_sel_o  = OP_B_REGB_OR_FWD;
                  //operand_b_fw_mux_sel     = SEL_REGFILE;  //this comes from the controller
                  alu_operator_o      = ALU_ADD;
              end
              1'b1: begin   //move from qReg. 18b
                qOP = qRV_mv_from_q;
                //read from qReg mv port and send data to EX stage for regfile
                  qReg_mem_rAddr = instr_rdata_i[23:19];   //32bit data
                //send address and we to EX stage for regfile
                  regfile_alu_we = 1'b1;
                //route the data to the output port of ID stage: opA = 0, opB = qReg_mv_rData
                  alu_op_a_mux_sel_o  = OP_A_IMM;
                  imm_a_mux_sel_o     = IMMA_ZERO;
                  alu_op_b_mux_sel_o  = OP_B_REGB_OR_FWD;
                  //operand_b_fw_mux_sel     = SEL_QREGFILE;  //this comes from the controller
```

```
                    alu_operator_o      = ALU_ADD;
                end
                default: begin
                  illegal_insn_o = 1'b1;
                end
                endcase
            end

            default: begin
              illegal_insn_o = 1'b1;
            end
          endcase

        end
        2'b11: begin //instruction 22: qldi
          qOP = qRV_ldi;
          qReg_wAddr = instr_rdata_i[23:19];
          qReg_sel = instr_rdata_i[18:17];   //selecting the sub-qReg to write on
          qReg_we = 1'b1;
          alu_op_a_mux_sel_o  = OP_A_IMM;
          alu_op_b_mux_sel_o  = OP_B_IMM;
          imm_a_mux_sel_o     = IMMA_ZERO;
          imm_b_mux_sel_o     = IMMB_QLDI;
          alu_operator_o      = ALU_ADD;
        end
        default: begin
          illegal_insn_o = 1'b1;
        end
      endcase

end // case: OPCODE_CUST0
```

Listing C.1: QDecoder logic - *cv32e40p_id_stage.sv*

QREG File

```

////////////////////////////////////////////////////////////////////////////////
// Engineer:       Jacopo Costantini - j.costantini@student.tudelft.nl        //
//                                                                            //
//                                                                            //
// Design Name:    qRV register file                                          //
// Project Name:   qRV ISA extension                                          //
// Language:       SystemVerilog                                              //
//                                                                            //
// Description:    Register file with 32x 64-bit wide registers, made of 4    //
//                 sub-registers each, independently writeable.               //
//                 Is suggested to always perform 4x 16-bit ldi operations.   //
//                 One read port, 64-bit wide; one write port 16-bit wide.    //
//                 Two additional ports are provided for mv instructions      //
//                 (one read, one write, 32-bit wide).                        //
//                 This register file is based on flip-flops.                 //
//                 Read is combinational, write is sequential.                //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

module qRV_qregister_file #(
    parameter ADDR_WIDTH = 5,
    parameter DATA_WIDTH_R = 64,    // 64-bit wide read port
    parameter DATA_WIDTH_W = 16,    // 16-bit wide write port
    parameter DATA_WIDTH_M = 32    // 32-bit wide ports for move instructions
) (
    // Clock and Reset
    input logic clk,
    input logic rst_n,

    //Read port R1
    input  logic [ADDR_WIDTH-1:0]   raddr_i,
    output logic [DATA_WIDTH_R-1:0] rdata_o,

    // Write port W1
```

```
36      input logic [ADDR_WIDTH-1:0]   waddr_i,     // write address
37      input logic [1:0]              wsel_i,      // write sub-reg select
38      input logic [DATA_WIDTH_W-1:0] wdata_i,     // write data
39      input logic                    we_i,        // write enable
40
41      //Read port R2 (32bit)
42      input  logic [ADDR_WIDTH-1:0]   raddr_mem_i,
43      output logic [DATA_WIDTH_M-1:0] rdata_mem_o,
44
45      //Read port R3 (32bit)
46      input  logic [ADDR_WIDTH-1:0]   raddr_mem2_i,
47      output logic [DATA_WIDTH_M-1:0] rdata_mem2_o,
48
49      //Write port W2 (32bit)
50      input logic [ADDR_WIDTH-1:0]    waddr_mem_i,
51      input logic [DATA_WIDTH_M-1:0]  wdata_mem_i,
52      input logic                     we_mem_i
53
54
55  );
56
57    // number of integer registers
58    localparam NUM_WORDS = 2 ** (ADDR_WIDTH);     //2^5 = 32 registers
59
60    // integer register file
61    logic [NUM_WORDS-1:0][DATA_WIDTH_R-1:0] mem;  // 32x 64-bit wide registers - "mem"
62
63    // masked write addresses
64    logic [ADDR_WIDTH-1:0] waddr;                 // masked write address - "waddr"
65    logic [ADDR_WIDTH-1:0] waddr_mem;              // masked write address for 32bit ports - "
          waddr_mem"
66
67    // write enable signals for all registers
68    logic [NUM_WORDS-1:0] we_dec;                 // write enable signals for all registers - "
          we_dec"
69    logic [NUM_WORDS-1:0] we_mem_dec;              // write enable signals for all registers -
          "we_mem_dec"
70
71
72    //-------------------------------------------------------------------------
73    //-- READ : Read address decoder RAD
74    //-------------------------------------------------------------------------
75    assign rdata_o = mem[raddr_i];
76    assign rdata_mem_o = mem[raddr_mem_i][31:0];    // read out only lower 32 bits of the
          register
77    assign rdata_mem2_o = mem[raddr_mem2_i][31:0];  // read out only lower 32 bits of the
          register
78
79
80    //-------------------------------------------------------------------------
81    //-- WRITE : Write Address Decoder (WAD), combinatorial process
82    //-------------------------------------------------------------------------
83
84    // Mask top bit of write address to disable fp regfile
85    assign waddr = waddr_i;
86    assign waddr_mem = waddr_mem_i;
87
88    genvar reg_index;
89    generate
90      for (reg_index = 0; reg_index < NUM_WORDS; reg_index++) begin : gen_we_decoder
91        assign we_dec[reg_index]    = (waddr == reg_index)    ? we_i : 1'b0;
92        assign we_mem_dec[reg_index] = (waddr_mem == reg_index) ? we_mem_i : 1'b0;
93      end
94    endgenerate
95
96    genvar i;
97    generate
98      //-------------------------------------------------------------------------
99      //-- WRITE : Write operation
100     //-------------------------------------------------------------------------
101
```

```
    // all 32 registers
    for (i = 0; i < NUM_WORDS; i++) begin : gen_rf

      always_ff @(posedge clk, negedge rst_n) begin : register_write_behavioral
        if (rst_n == 1'b0) begin    // reset
          mem[i] <= 64'b0;

        end else begin    // normal operation
          if (we_dec[i] == 1'b1)
            unique case (wsel_i)
              2'b00:
                mem[i][15: 0] <= wdata_i;
              2'b01:
                mem[i][31:16] <= wdata_i;
              2'b10:
                mem[i][47:32] <= wdata_i;
              2'b11:
                mem[i][63:48] <= wdata_i;
            endcase

          // write to the register for mem instruction
          else if (we_mem_dec[i] == 1'b1)
            mem[i] <= {32'b0, wdata_mem_i};     // write to the lower 32 bits of the register

        end
      end

    end

  endgenerate

endmodule
```

Listing C.2: QREG file - *qRV_register_file_ff.sv*

Packet manager

```

////////////////////////////////////////////////////////////////////////////////
// Engineer:       Jacopo Costantini - j.costantini@student.tudelft.nl        //
//                                                                            //
//                                                                            //
// Design Name:    qRV imm_params manager                                     //
// Project Name:   qRV ISA extension                                          //
// Language:       SystemVerilog                                              //
//                                                                            //
// Description:    Quantum imm_params manager, assembler and router of data   //
//                 imm_paramss to the NV nodes. Uses the OBI adapter.         //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

module qRV_master
  import cv32e40p_pkg::*;
  import cv32e40p_apu_core_pkg::*;
  import qRV_pkg::*;
#(
    parameter nodes = 1024,
    parameter ADDR_WIDTH = $clog2(nodes),
    parameter DATA_WIDTH = 75,
    parameter words = 19
) (

    input logic        clk,
    input logic        rst_n,

    //inputs from the EX stage
    input logic [63:0] reg_params_i,
    input logic [ 5:0] imm_params_i,
    input qRV_op       opcode_i,
    input logic [ 9:0] node_addr_i,

```

```
35      //output to the bus interface
36      output logic        slave_en_o,
37      output logic [ 9:0] node_addr_o,
38      output logic [74:0] packet_o
39
40  );
41
42      ////////////////////////////////// PACKET ASSEMBLER ///////////////////////////////////
43      logic [75:0] packet;
44      logic [ 5:0] imm_params;
45      logic        slave_en;
46
47      always @ (opcode_i)
48      begin
49        if (opcode_i == qRV_ldi || opcode_i == qRV_st || opcode_i == qRV_ld || opcode_i ==
              qRV_mv_to_q
50              || opcode_i == qRV_mv_from_q || opcode_i == qRV_reset_op) begin
51          packet = '0;
52          slave_en = 1'b0;
53
54        end else begin
55          slave_en = 1'b1;
56            unique case (opcode_i)          //some (default) cases are just included for
                  completetness
57              qRV_QgateE: begin
58                  imm_params = 5'b00000;
59                  packet = {imm_params, reg_params_i, opcode_i};
60                  end
61              qRV_QgateCC: begin
62                  imm_params = {2'b00, imm_params_i[3:0]};                        //C13
63                  packet = {imm_params, reg_params_i, opcode_i};
64                  end
65              qRV_QgateUC: begin
66                  imm_params = {1'b0, imm_params_i[4], imm_params_i[3:0]};          //P,C13
67                  packet = {imm_params, reg_params_i, opcode_i};
68                  end
69              qRV_QgateDIR: begin
70                  imm_params = {1'b0, imm_params_i[4], imm_params_i[3:0]};          //D,C13
71                  packet = {imm_params, reg_params_i, opcode_i};
72                  end
73              qRV_ZgateE: begin
74                  imm_params = 5'b00000;
75                  packet = {imm_params, reg_params_i, opcode_i};
76                  end
77              qRV_ZgateCC: begin
78                  imm_params = {2'b00, imm_params_i[3:0]};                        //C13
79                  packet = {imm_params, reg_params_i, opcode_i};
80                  end
81
82              qRV_swapEC: begin
83                  imm_params = {imm_params_i[1:0], imm_params_i[5:2]};              //basis,
                        C13
84                  packet = {imm_params, reg_params_i, opcode_i};
85                  end
86              qRV_swapCE: begin
87                  imm_params = {2'b00, imm_params_i[5:2]};                         //C13
88                  packet = {imm_params, reg_params_i, opcode_i};
89                  end
90
91              qRV_set: begin
92                  imm_params = {1'b0, imm_params_i[4:0]};                         //local
                        register
93                  packet = {imm_params, reg_params_i, opcode_i};
94                  end
95              qRV_ent: begin
96                  imm_params = {3'b000, imm_params_i[2:0]};                       //ent
                        direction
97                  packet = {imm_params, reg_params_i, opcode_i};
98                  end
99
100             default: begin    //detectC13, magBias, RabiCheck, init, measureE, crc, escape...
```

```
                imm_params = 5'b00000;
                packet = {imm_params, reg_params_i, opcode_i};
                end
            endcase //end case
        end //end if
    end; //packet assemblers

    ////////////////////////////////// PACKET ROUTER ///////////////////////////////////
    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            node_addr_o <= 10'b0;
            packet_o    <= 75'b0;
            slave_en_o  <= 1'b0;
        end else begin
            node_addr_o <= node_addr_i;
            packet_o    <= packet;
            slave_en_o  <= slave_en;
        end
    end


endmodule
```

Listing C.3: qRV32 Packet Manager - *qRV_master.sv*

Forwarding logic

```

  ////////////////////////////////////////////////////////
  //   ___                                  _      _    //
  //  / _ \ _ __   ___ _ __ __ _ _ __   __| |    / \    //
  // | | | | '_ \ / _ \ '__/ _` | '_ \ / _` |   / _ \   //
  // | |_| | |_) |  __/ | | (_| | | | | (_| |  / ___ \  //
  //  \___/| .__/ \___|_|  \__,_|_| |_|\__,_| /_/   \_\ //
  //       |_|                                          //
  ////////////////////////////////////////////////////////

  // ALU_Op_a Mux
  always_comb begin : alu_operand_a_mux   // alu_operand_a_mux is the input to the ALU for operand A
    case (alu_op_a_mux_sel)               // alu_op_a_mux_sel is the control signal for the mux
      OP_A_REGA_OR_FWD: alu_operand_a = operand_a_fw_id;    //alu_operand_a = operand a forwarded
      OP_A_REGB_OR_FWD: alu_operand_a = operand_b_fw_id;    //alu_operand_a = operand b forwarded
      OP_A_REGC_OR_FWD: alu_operand_a = operand_c_fw_id;    //alu_operand_a = operand c forwarded
      OP_A_CURRPC:      alu_operand_a = pc_id_i;            //alu_operand_a = program counter
      OP_A_IMM:         alu_operand_a = imm_a;              //alu_operand_a = immediate
      default:          alu_operand_a = operand_a_fw_id;
    endcase
    ;  // case (alu_op_a_mux_sel)
  end

  always_comb begin : immediate_a_mux   // immediate_a_mux is the input to the ALU for operand A (for the immediate)
    unique case (imm_a_mux_sel)         // imm_a_mux_sel is the control signal for the mux
      IMMA_Z:    imm_a = imm_z_type;    //imm_a = immediate z
      IMMA_ZERO: imm_a = '0;            //imm_a = 0 for load immediate instructions
    endcase
  end

  // Operand a forwarding mux
  always_comb begin : operand_a_fw_mux    // operand_a_fw_mux is the input to the ALU for operand A (for the forwarding)
    case (operand_a_fw_mux_sel)           // operand_a_fw_mux_sel is the control signal for the mux
      SEL_FW_EX:   operand_a_fw_id = regfile_alu_wdata_fw_i;      //forward alu result
      SEL_FW_WB:   operand_a_fw_id = regfile_wdata_wb_i;          //forward writeback result
```

```
705       SEL_REGFILE: operand_a_fw_id = regfile_data_ra_id;          //forward register file
              read data port A
706       SEL_QREGFILE: operand_a_fw_id = qReg_mem2_rData_id;        //forward qregister file
              read data port MEM
707       default:     operand_a_fw_id = regfile_data_ra_id;
708     endcase
709     ;  // case (operand_a_fw_mux_sel)
710   end
711
712   ////////////////////////////////////////////////////
713   //   ___                                  _   ____   //
714   //  / _ \ _ __   ___ _ __ __ _ _ __   __| | | __ )  //
715   // | | | | '_ \ / _ \ '__/ _` | '_ \ / _` | |  _ \  //
716   // | |_| | |_) |  __/ | | (_| | | | | (_| | | |_) | //
717   //  \___/| .__/ \___|_|  \__,_|_| |_|\__,_| |____/  //
718   //       |_|                                        //
719   ////////////////////////////////////////////////////
720
721   // Immediate Mux for operand B
722   always_comb begin : immediate_b_mux
723     unique case (imm_b_mux_sel)
724       IMMB_I:      imm_b = imm_i_type;
725       IMMB_S:      imm_b = imm_s_type;
726       IMMB_U:      imm_b = imm_u_type;
727       IMMB_PCINCR: imm_b = is_compressed_i ? 32'h2 : 32'h4;
728       IMMB_S2:     imm_b = imm_s2_type;
729       IMMB_BI:     imm_b = imm_bi_type;
730       IMMB_S3:     imm_b = imm_s3_type;
731       IMMB_VS:     imm_b = imm_vs_type;
732       IMMB_VU:     imm_b = imm_vu_type;
733       IMMB_SHUF:   imm_b = imm_shuffle_type;
734       IMMB_CLIP:   imm_b = {1'b0, imm_clip_type[31:1]};
735       IMMB_QREG:   imm_b = imm_qmem_type;
736       IMMB_QLDI:   imm_b = imm_qldi_type;
737       default:     imm_b = imm_i_type;
738     endcase
739   end
740
741   // ALU_Op_b Mux
742   always_comb begin : alu_operand_b_mux
743     case (alu_op_b_mux_sel)
744       OP_B_REGA_OR_FWD: operand_b = operand_a_fw_id;
745       OP_B_REGB_OR_FWD: operand_b = operand_b_fw_id;
746       OP_B_REGC_OR_FWD: operand_b = operand_c_fw_id;
747       OP_B_IMM:         operand_b = imm_b;
748       OP_B_BMASK:       operand_b = $unsigned(operand_b_fw_id[4:0]);
749       default:          operand_b = operand_b_fw_id;
750     endcase  // case (alu_op_b_mux_sel)
751   end
752
753
754   // scalar replication for operand B and shuffle type
755   always_comb begin   // operand_b_vec is the replicated operand B
756     if (alu_vec_mode == VEC_MODE8) begin      // VEC_MODE8 is 8-bit vector mode
757       operand_b_vec    = {4{operand_b[7:0]}};
758       imm_shuffle_type = imm_shuffleb_type;
759     end else begin                            // VEC_MODE16 is 16-bit vector mode
760       operand_b_vec    = {2{operand_b[15:0]}};
761       imm_shuffle_type = imm_shuffleh_type;
762     end
763   end
764
765   // choose normal or scalar replicated version of operand b
766   assign alu_operand_b = (scalar_replication == 1'b1) ? operand_b_vec : operand_b;
767
768   // Operand b forwarding mux
769   always_comb begin : operand_b_fw_mux
770     case (operand_b_fw_mux_sel)
771       SEL_FW_EX:    operand_b_fw_id = regfile_alu_wdata_fw_i;
772       SEL_FW_WB:    operand_b_fw_id = regfile_wdata_wb_i;    // regfile_wdata_wb_i is the
              write data for the writeback stage
```

```
773       SEL_REGFILE:  operand_b_fw_id = regfile_data_rb_id;
774       SEL_QREGFILE: operand_b_fw_id = qReg_mem_rData_id;
775       default:      operand_b_fw_id = regfile_data_rb_id;
776     endcase
777     ;  // case (operand_b_fw_mux_sel)
778   end
779
780
781   ////////////////////////////////////////////////////
782   //   ___                                   _    ____  //
783   //  / _ \ _ __   ___ _ __ __ _ _ __   __| |  / ___| //
784   // | | | | '_ \ / _ \ '__/ _` | '_ \ / _` | | |     //
785   // | |_| | |_) |  __/ | | (_| | | | | (_| | | |___  //
786   //  \___/| .__/ \___|_|  \__,_|_| |_|\__,_|  \____| //
787   //       |_|                                       //
788   ////////////////////////////////////////////////////
789
790   // ALU OP C Mux
791   always_comb begin : alu_operand_c_mux
792     case (alu_op_c_mux_sel)
793       OP_C_REGC_OR_FWD: operand_c = operand_c_fw_id;
794       OP_C_REGB_OR_FWD: operand_c = operand_b_fw_id;
795       OP_C_JT:          operand_c = jump_target;
796       default:          operand_c = operand_c_fw_id;
797     endcase  // case (alu_op_c_mux_sel)
798   end
799
800
801   // scalar replication for operand C and shuffle type
802   always_comb begin
803     if (alu_vec_mode == VEC_MODE8) begin
804       operand_c_vec = {4{operand_c[7:0]}};
805     end else begin
806       operand_c_vec = {2{operand_c[15:0]}};
807     end
808   end
809
810   // choose normal or scalar replicated version of operand b
811   assign alu_operand_c = (scalar_replication_c == 1'b1) ? operand_c_vec : operand_c;
812
813
814   // Operand c forwarding mux
815   always_comb begin : operand_c_fw_mux
816     case (operand_c_fw_mux_sel)
817       SEL_FW_EX:   operand_c_fw_id = regfile_alu_wdata_fw_i;
818       SEL_FW_WB:   operand_c_fw_id = regfile_wdata_wb_i;
819       SEL_REGFILE: operand_c_fw_id = regfile_data_rc_id;
820       SEL_QREGFILE: operand_c_fw_id = qReg_mem_rData_id;
821       default:     operand_c_fw_id = regfile_data_rc_id;
822     endcase
823     ;  // case (operand_c_fw_mux_sel)
824   end
```

Listing C.4: Forwarding multiplexers - *cv32e40p_id_stage.sv*

```
1363 // Forwarding control unit
1364   always_comb
1365   begin
1366     // default assignements
1367     operand_a_fw_mux_sel_o = SEL_REGFILE;
1368     operand_b_fw_mux_sel_o = SEL_REGFILE;
1369     operand_c_fw_mux_sel_o = SEL_REGFILE;
1370
1371     unique case (qRV_op_i)
1372       qRV_mv_from_q: begin
1373         operand_b_fw_mux_sel_o = SEL_QREGFILE;
1374       end
1375       qRV_ld: begin
1376         operand_a_fw_mux_sel_o = SEL_QREGFILE;
1377       end
1378       qRV_st: begin
```

```
1379          operand_a_fw_mux_sel_o = SEL_QREGFILE;
1380          operand_c_fw_mux_sel_o = SEL_QREGFILE;
1381        end
1382        default: begin
1383          // default assignements
1384          operand_a_fw_mux_sel_o = SEL_REGFILE;
1385          operand_b_fw_mux_sel_o = SEL_REGFILE;
1386          operand_c_fw_mux_sel_o = SEL_REGFILE;
1387        end
1388      endcase
1389
1390      // Forwarding WB -> ID
1391      if (regfile_we_wb_i == 1'b1)
1392      begin
1393        if (reg_d_wb_is_reg_a_i == 1'b1)
1394          operand_a_fw_mux_sel_o = SEL_FW_WB;
1395        if (reg_d_wb_is_reg_b_i == 1'b1)
1396          operand_b_fw_mux_sel_o = SEL_FW_WB;
1397        if (reg_d_wb_is_reg_c_i == 1'b1)
1398          operand_c_fw_mux_sel_o = SEL_FW_WB;
1399      end
1400
1401      // Forwarding EX -> ID
1402      if (regfile_alu_we_fw_i == 1'b1)
1403      begin
1404       if (reg_d_alu_is_reg_a_i == 1'b1)
1405         operand_a_fw_mux_sel_o = SEL_FW_EX;
1406       if (reg_d_alu_is_reg_b_i == 1'b1)
1407         operand_b_fw_mux_sel_o = SEL_FW_EX;
1408       if (reg_d_alu_is_reg_c_i == 1'b1)
1409         operand_c_fw_mux_sel_o = SEL_FW_EX;
1410      end
1411
1412      // for misaligned memory accesses
1413      if (data_misaligned_i)
1414      begin
1415        operand_a_fw_mux_sel_o  = SEL_FW_EX;
1416        operand_b_fw_mux_sel_o  = SEL_REGFILE;
1417      end else if (mult_multicycle_i) begin
1418        operand_c_fw_mux_sel_o  = SEL_FW_EX;
1419      end
1420    end
```

Listing C.5: Forwarding multiplexers select control - *cv32e40p_controller.sv*

Move instruction

```
596 //-----------------------------------------------------------------------------
597 // source register selection regfile_fp_x=1 <=> CV32E40P_REG_x is a FP-register
598 //-----------------------------------------------------------------------------
599 assign regfile_addr_ra_id = {regfile_fp_a, instr[REG_S1_MSB:REG_S1_LSB]};
600
601 //assign regfile_addr_rb_id = {regfile_fp_b, instr[REG_S2_MSB:REG_S2_LSB]};
602 assign regfile_addr_rb_id = (qOP_id == (qRV_mv_to_q)) ?
603     {1'b0, instr[28:24]} : {regfile_fp_b, instr[REG_S2_MSB:REG_S2_LSB]};
```

Listing C.6: Source register selection - *cv32e40p_id_stage.sv*

```
596 //-----------------------------------------------------------------------------
597 // destination registers regfile_fp_d=1 <=> REG_D is a FP-register
598 //-----------------------------------------------------------------------------
599 //addiotinal qRV mux:
600 assign regfile_waddr_id = (qOP_id == (qRV_mv_from_q)) ?
601     {1'b0, instr[28:24]} : {regfile_fp_d, instr[REG_D_MSB:REG_D_LSB]};   //create a mux that
          selects a different address if qOP = qRV_mv_from_q
```

Listing C.7: Destination register selection - *cv32e40p_id_stage.sv*

C.2. System on Chip

This section contains snippets of SystemVerilog code that describes the System on Chip hardware.

C.2.1. Sytem Memory

Read Only Memory

```

////////////////////////////////////////////////////////////////////////////////
// Engineer:       Jacopo Costantini - j.costantini@student.tudelft.nl        //
//                                                                            //
//                                                                            //
// Design Name:    rom block                                                  //
// Project Name:   qRV ISA extension                                          //
// Language:       SystemVerilog                                              //
//                                                                            //
// Description:    Behavioral implementation of SoC memory, with additional   //
//                 ports to mimic the PCI protocol.                           //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

module rom_sp #(
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 32,
    parameter memory_depth = 256,
    parameter instr_path = "D:/User/Documenti/TuDelft/Thesis/qrv_assembler/outputs/test_bin.txt"
) (
    // Clock and Reset
    input logic clk_i,

    // Instruction interface
    input  logic                  inst_req_i,
    input  logic [ADDR_WIDTH-1:0] inst_addr_i,

    output logic [DATA_WIDTH-1:0] inst_rdata_o,
    output logic                  inst_rvalid_o,
    output logic                  inst_gnt_o
);

    //this ROM is 1KB (256 words of 4 bytes each)
    //since the memory is byte addressable and the data width is 32 bits (4 bytes), the two least significant bits of the address are ignored

    //the full addressable space is 2^32 bytes (4GB) or 2^30 words

    localparam NUM_WORDS = memory_depth;
    (* rom_style = "block" *)
    reg [DATA_WIDTH-1:0] memory [NUM_WORDS-1:0];

    //--------------------------------------------------------------------------
    //-- READ : Read address decoder RAD
    //--------------------------------------------------------------------------
    always_comb begin
        inst_rdata_o = '0;
        inst_rvalid_o = 1'b0;
        inst_gnt_o = 1'b0;

        if (inst_req_i) begin
            inst_rdata_o = memory[inst_addr_i[ADDR_WIDTH-1:2]];
            inst_rvalid_o = 1'b1;
            inst_gnt_o = 1'b1;
        end
    end

    //--------------------------------------------------------------------------
    //-- WRITE : Flash memory with instructions
    //--------------------------------------------------------------------------

    // initial begin
    //     $readmemb(instr_path, memory, 0);   //read the file and store it in the memory
    //     // set the words that havent been initialised by the file to 32'b0
    //     for (int i = 0; i < NUM_WORDS; i++) begin
    //         if ($isunknown(memory[i])) begin
    //             memory[i] <= 32'b0;
```

```
    //          end
    //      end
    // end

    initial begin

        // Initialize memory with zeros
        for (int i = 0; i < NUM_WORDS; i++) begin
            memory[i] = 32'b0;
        end

        // Read file and update memory with file contents
        $readmemb(instr_path, memory, 0);
  end

endmodule
```

Listing C.8: ROM Module - *rom_sp.sv*

Random Access Memory

```

////////////////////////////////////////////////////////////////////////////////
// Engineer:       Jacopo Costantini - j.costantini@student.tudelft.nl        //
//                                                                            //
//                                                                            //
// Design Name:    ram block                                                  //
// Project Name:   qRV ISA extension                                          //
// Language:       SystemVerilog                                              //
//                                                                            //
// Description:    Behavioral implementation of sp RAM block with additional  //
//                 ports to mimic the PCI protocol.                           //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

module ram_sp #(
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 32,
    parameter memory_depth = 256
) (
    // Clock and Reset
    input logic clk_i,
    input logic rst_ni,

    // Data interface
    input  logic                  data_req_i,
    input  logic [ADDR_WIDTH-1:0] data_addr_i,
    input  logic                  data_we_i,
    input  logic [3:0]            data_be_i,
    input  logic [DATA_WIDTH-1:0] data_wdata_i,

    output logic [DATA_WIDTH-1:0] data_rdata_o,
    output logic                  data_rvalid_o,
    output logic                  data_gnt_o
);

    //this memory is default to only use 1KB (256 words of 4 bytes each)
    //since the memory is byte addressable and the data width is 32 bits (4 bytes), the two
        least significant bits of the address are ignored

    //the full addressable space is 2^32 bytes (4GB) or 2^30 words

    localparam NUM_WORDS = memory_depth;
    (* ram_style = "block" *)
    reg [DATA_WIDTH-1:0] memory [NUM_WORDS-1:0];

    //--------------------------------------------------------------------------
    //-- READ : Read address decoder RAD
    //--------------------------------------------------------------------------
    always_comb begin
        data_rdata_o = '0;
```

```
        data_rvalid_o = 1'b0;
        data_gnt_o = 1'b0;

        if (data_req_i) begin
            data_rdata_o = memory[data_addr_i[ADDR_WIDTH-1:2]];
            data_rvalid_o = 1'b1;
            data_gnt_o = 1'b1;
        end
    end

    //------------------------------------------------------------------------
    //-- WRITE : Write operation
    //------------------------------------------------------------------------
    always_ff @(posedge clk_i) begin
        //reset: only for simulation
        /*if (rst_ni == 1'b0) begin
            for (int i = 0; i < NUM_WORDS; i++) begin
                memory[i] = 32'b0;
            end
        end else*/
        if (data_we_i) begin  //write operation
            //memory[ADDR_WIDTH-1:2] <= data_wdata_i;
            if (data_be_i[0] == 1'b1) begin
                memory[data_addr_i[ADDR_WIDTH-1:2]][7:0] <= data_wdata_i[7:0];
            end
            if (data_be_i[1] == 1'b1) begin
                memory[data_addr_i[ADDR_WIDTH-1:2]][15:8] <= data_wdata_i[15:8];
            end
            if (data_be_i[2] == 1'b1) begin
                memory[data_addr_i[ADDR_WIDTH-1:2]][23:16] <= data_wdata_i[23:16];
            end
            if (data_be_i[3] == 1'b1) begin
                memory[data_addr_i[ADDR_WIDTH-1:2]][31:24] <= data_wdata_i[31:24];
            end
        end // if
    end

endmodule
```

Listing C.9: RAM Module - *ram_sp.sv*

C.2.2. UART

UART Interface

```
module uart_interface#(
    parameter clk_freq = 100000000,
    parameter baud_rate = 115200
    )(
    input logic clk_i,
    input logic rst_ni,
    input logic [84:0] tx_data_i,
    input logic new_data_available_i,

    //output logic ready_o,
    output logic FIFO_read_en_o,
    output logic tx_data_o
);

//signals
logic [7:0] tx_byte_data;

logic uart_tx_done;
logic uart_tx_start;
logic uart_tx_active;
//logic tx_data_stb;
//logic tx_ack_out;

logic sequencer_load;
logic sequencer_start;
logic sequencer_tready;
logic sequencer_ready;
```

```
logic sequencer_tactive;
logic sequencer_done;

logic ready_for_transmission;
logic data_processed;
logic [ 8:0] data_counter;

//assigments
//ready for transmission when there is data available in FIFO and the sequencer and the UART are ready
assign ready_for_transmission = ((data_counter > 0) && sequencer_ready) ? 1'b1 : 1'b0;
assign uart_tx_start = sequencer_tactive;

//state machine
enum logic [5:0] {IDLE, START, LOAD_DATA, UART_TX, WAIT1, WAIT2} state, next_state;

//submodules
 uart #(
     .clk_freq(clk_freq),
     .baud_rate(baud_rate)
 ) uart (
     .clk(clk_i),
     .rst_n(rst_ni),
     .tx_data_in(tx_byte_data),
     .start(uart_tx_start),
     .tx(tx_data_o),
     .tx_active(uart_tx_active),
     .done_tx(uart_tx_done)
 ); //uart

//Sthefan Lung UART implementation:
//UART #(
//    .CLOCK_FREQUENCY(clk_freq),
//    .BAUD_RATE(baud_rate)
//) uart (
//    .CLOCK(clk_i),
//    .RESET_n(rst_ni),
//    .DATA_STREAM_IN(tx_byte_data),
//    .DATA_STREAM_IN_ACK(tx_ack_out),
//    .DATA_STREAM_IN_STB(tx_data_stb),
//    .DATA_STREAM_OUT(),
//    .DATA_STREAM_OUT_STB(),
//    .DATA_STREAM_OUT_ACK(),
//    .TX(tx_out),
//    .TX_ready(uart_tx_done),
//    .RX()
//); //uart

uart_seq_transmitter sequencer(
    .clk(clk_i),
    .tbuf(tx_data_i),
    .load_i(sequencer_load),
    .start_i(sequencer_start),
    .tready_i(sequencer_tready),
    .ready_o(sequencer_ready),
    .tcompleted_o(sequencer_done),
    .tactive_o(sequencer_tactive),
    .tbus(tx_byte_data)
);

//fsm
always_ff @(posedge clk_i) begin
    if (~rst_ni) begin
        state <= IDLE;
    end else begin
        state <= next_state;
    end
end
always_comb begin : next_state_logic
    //next_state = IDLE;    //default state
    case(state)
```

```
        IDLE:
              if (ready_for_transmission) begin
                  next_state = START;
              end else next_state = IDLE;
        START: next_state = LOAD_DATA;    //just wait for 1 clock cycle
        LOAD_DATA: next_state = WAIT1;
        WAIT1: if (sequencer_ready) begin
                  next_state = UART_TX;
              end else next_state = WAIT1;
        UART_TX:
            next_state = WAIT2;
        WAIT2:
            if (uart_tx_done) begin
                if (~sequencer_done) begin
                    next_state = UART_TX;
                end else next_state = IDLE;
            end else next_state = WAIT2;
        default: next_state = IDLE;
    endcase
end
always_comb begin : control_logic
    //default values
    FIFO_read_en_o = 1'b0;
    sequencer_load = 1'b0;
    sequencer_start = 1'b0;
    data_processed = 1'b0;

    case(state)
        IDLE: begin
            sequencer_tready = 1'b0;
            end
        START: begin
            sequencer_tready = 1'b1;
            FIFO_read_en_o = 1'b1;
            end
        LOAD_DATA: begin
            sequencer_load = 1'b1;
            data_processed = 1'b1;
            end
        WAIT2: begin
            FIFO_read_en_o = 1'b0;
            sequencer_load = 1'b0;
            end
        UART_TX:
            sequencer_start = 1'b1;
        WAIT2:
            sequencer_start = 1'b0;
    endcase
end

//FIFO read enable process
always_ff @(posedge clk_i) begin : data_cnt
    if (~rst_ni) begin
        data_counter <= 0;
    end else if (new_data_available_i) begin
        data_counter <= data_counter + 1;
    end else if (data_processed) begin
        data_counter <= data_counter - 1;
    end
end

endmodule
```

Listing C.10: UART interface module - *uart_interface.sv*

UART Sequencer

```
////////////////////////////////////////////////////////////////////////////////////////

//The module uart_seq_transmitter takes in the following inputs:

```

```
//    - clk: Clock signal for synchronization.
//    - tbuf: The 85-bit input data to be transmitted.
//    - start_i: Input signal to initiate transmission.
//    - tready_i: Input signal indicating receiver readiness.

//The module provides the following outputs:

//    - ready_o: Output signal indicating buffer readiness.
//    - tstart_o: Output signal indicating the start of transmission.
//    - tbus: Output signal representing the serialized data transmitted in 8-bit chunks.

//Internal registers:

//    - counter: A 7-bit register used as a counter to keep track of the current position
//        within a data byte.
//    - data_remaining: An 84-bit register representing the number of bits remaining to
//        be transmitted.
//    - running: A 1-bit register indicating whether the transmission is ongoing.

//The module uses an always block triggered on the positive edge of the clk signal to
//handle the transmission process.

//When ready_o is high, indicating the receiver is ready to accept data, the module checks
//the following conditions:
//    * If start_i is high and running is low, it initiates a new transmission. It sets
//        data_remaining to 84 (indicating 84 bits remaining), sets tstart_o to high, and
//        sets running to high.
//    * If running is high, it transmits data byte by byte. It uses the counter to keep
//        track of the current position within a byte and assigns the corresponding 8 bits
//        from tbuf to tbus. It decrements data_remaining by 8 and decrements counter by 1
//        until all 84 bits are transmitted.
//    * If running is low, indicating the transmission is completed, it sets tstart_o to low.

//The assign statement assigns the complement of running to the ready_o signal,
//indicating buffer readiness.

//The module is inspired from the following source:
//    https://github.com/Digilent/Genesys-2-Keyboard/blob/master/src/hdl/uart_buf_con.v

//~Jacopo Costantini, May 2023

//////////////////////////////////////////////////////////////////////////////////


module uart_seq_transmitter(
    input             clk,
    input      [84:0] tbuf,
    input             load_i,
    input             start_i,
    input             tready_i,
    output            ready_o,
    output reg        tcompleted_o,
    output reg        tactive_o,
    output reg [7:0]  tbus
);
    reg [ 7:0] data_remaining = 7'd0;
    reg        running = 1'b0;
    reg [84:0] internal_tbuf = 85'd0;
    reg        old_start = 1'b0;

    always @(posedge clk) begin
        old_start <= start_i;
        if (tready_i) begin
            if (start_i && ~running && data_remaining >= 8) begin
                // Start a new transmission
                data_remaining <= data_remaining;
                tactive_o <= 1'b0;
                running <= 1'b1;
                tcompleted_o <= 1'b0;
            end else if (old_start && running && data_remaining >= 8) begin
                // Transmit data byte by byte
```

```
                case (data_remaining)
                    7'd88: tbus <= internal_tbuf[84:77];
                    7'd80: tbus <= internal_tbuf[76:69];
                    7'd72: tbus <= internal_tbuf[68:61];
                    7'd64: tbus <= internal_tbuf[60:53];
                    7'd56: tbus <= internal_tbuf[52:45];
                    7'd48: tbus <= internal_tbuf[44:37];
                    7'd40: tbus <= internal_tbuf[36:29];
                    7'd32: tbus <= internal_tbuf[28:21];
                    7'd24: tbus <= internal_tbuf[20:13];
                    7'd16: tbus <= internal_tbuf[12:5];
                    7'd8:  tbus <= {internal_tbuf[4:0], 3'b0};
                    default: tbus <= 8'd0;
                endcase
                tactive_o <= 1'b1;
                tcompleted_o <= 1'b0;
                data_remaining <= data_remaining - 8;
            end else if (~start_i && running && data_remaining >= 8) begin
                //Trasmission is paused
                data_remaining <= data_remaining;
                tbus <= 8'd0;
                tactive_o <= 1'b0;
                tcompleted_o <= 1'b0;
            end else if (data_remaining < 8) begin
                //Transmission completed
                tcompleted_o <= 1'b1;
                tactive_o <= 1'b0;
                tbus <= 8'd0;
                running <= 1'b0;
                // Load new data
                if (load_i) begin
                    internal_tbuf <= tbuf;
                    data_remaining <= 7'd88;
                end
            end
        end else begin
            // Receiver not ready, wait for tready_i to become high
            tactive_o <= 1'b0;
            tcompleted_o <= 1'b0;
        end
    end

    assign ready_o = ~running;

endmodule
```

Listing C.11: UART Sequencer module - *uart_seq_transmitter.v*

UART

```
/*
 *  Copyright (C) 2018  Siddharth J <www.siddharth.pro>
 *
 *  Permission to use, copy, modify, and/or distribute this software for any
 *  purpose with or without fee is hereby granted, provided that the above
 *  copyright notice and this permission notice appear in all copies.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 *  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 *  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 *  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 *  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 *  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 */

module uart(clk,rst_n,rx,tx_data_in,start,rx_data_out,tx,tx_active,done_tx);

parameter clk_freq = 15_000_000; //MHz
parameter baud_rate = 115200; //bits per second
```

```
22
23    input clk,rst_n;
24    input rx;
25    input [7:0] tx_data_in;
26    input start;
27    output tx;
28    output [7:0] rx_data_out;
29    output tx_active;
30    output done_tx;
31
32
33  uart_rx
34        #(.clk_freq(clk_freq),
35       .baud_rate(baud_rate)
36      )
37        receiver
38               (
39                .clk(clk),
40            .rst(rst),
41            .rx(rx),
42            .rx_data_out(rx_data_out)
43               );
44
45
46  uart_tx
47         #(.clk_freq(clk_freq),
48       .baud_rate(baud_rate)
49          )
50        transmitter
51                 (
52                  .clk(clk),
53          .rst(rst),
54          .start(start),
55          .tx_data_in(tx_data_in),
56          .tx(tx),
57          .tx_active(tx_active),
58          .done_tx(done_tx)
59                 );
60
61  endmodule
```

Listing C.12: UART module - *uart.sv*

```
1  /*
2   *  Copyright (C) 2018  Siddharth J <www.siddharth.pro>
3   *
4   *  Permission to use, copy, modify, and/or distribute this software for any
5   *  purpose with or without fee is hereby granted, provided that the above
6   *  copyright notice and this permission notice appear in all copies.
7   *
8   *  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9   *  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10  *  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11  *  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12  *  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13  *  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14  *  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15  *
16  */
17
18  module uart_tx(clk,rst,start,tx_data_in,tx,tx_active,done_tx);
19
20  parameter clk_freq = 100_000_000; //MHz
21  parameter baud_rate = 115200; //bits per second
22  //parameter baud_rate = 1_000_000; //fake baud rate for simulation
23  input clk,rst;
24  input start;
25  input [7:0] tx_data_in;
26  output tx;
27  output tx_active;
28  output logic done_tx;
```

```
29
30 localparam clock_divide = (clk_freq/baud_rate); //clock divide value: counts to this vaule
       every bit time
31
32 enum bit [2:0]{ tx_IDLE = 3'b000, tx_START = 3'b001, tx_DATA = 3'b010, tx_STOP = 3'b011,
       tx_DONE = 3'b100 } tx_STATE, tx_NEXT;
33
34 logic [11:0] clk_div_reg,clk_div_next;
35 logic [7:0] tx_data_reg, tx_data_next;
36 logic tx_out_reg,tx_out_next;
37 logic [2:0] index_bit_reg, index_bit_next;  //index of bit to be transmitted
38
39 assign tx_active = (tx_STATE == tx_DATA);
40 assign tx = tx_out_reg;
41
42 always_ff @(posedge clk) begin
43     if(~rst) begin  //reset
44         tx_STATE <= tx_IDLE;
45         clk_div_reg <= 0;
46         tx_out_reg <= 0;
47         tx_data_reg <= 0;
48         index_bit_reg <= 0;
49     end
50     else begin   //next state update
51         tx_STATE <= tx_NEXT;
52         clk_div_reg <= clk_div_next;
53         tx_out_reg <= tx_out_next;
54         tx_data_reg <= tx_data_next;
55         index_bit_reg <= index_bit_next;
56     end
57 end
58
59 always @(*) begin   //next state logic
60     tx_NEXT = tx_STATE;
61     clk_div_next = clk_div_reg;
62     tx_out_next = tx_out_reg;
63     tx_data_next = tx_data_reg;
64     index_bit_next = index_bit_reg;
65     done_tx = 0;
66
67 case(tx_STATE)
68
69     tx_IDLE: begin
70         tx_out_next = 1;
71         clk_div_next = 0;
72         index_bit_next = 0;
73         if(start == 1) begin
74             tx_data_next = tx_data_in;
75             tx_NEXT = tx_START;
76         end
77         else begin
78             tx_NEXT = tx_IDLE;
79         end
80     end
81
82     tx_START: begin
83         tx_out_next = 0;
84         if(clk_div_reg < clock_divide-1) begin
85             clk_div_next = clk_div_reg + 1'b1;
86             tx_NEXT = tx_START;
87         end
88         else begin
89             clk_div_next = 0;
90             tx_NEXT = tx_DATA;
91         end
92     end
93
94     tx_DATA: begin
95         tx_out_next = tx_data_reg[index_bit_reg];
96         if(clk_div_reg < clock_divide-1) begin
97             clk_div_next = clk_div_reg + 1'b1;
```

```
            tx_NEXT = tx_DATA;
        end
        else begin
            clk_div_next = 0;
            if(index_bit_reg < 7) begin
                index_bit_next = index_bit_reg + 1'b1;
                tx_NEXT = tx_DATA;
            end
            else begin
                index_bit_next = 0;
                tx_NEXT = tx_STOP;
            end
        end
    end

    tx_STOP: begin
    tx_out_next = 1;
    if(clk_div_reg < clock_divide-1) begin
        clk_div_next = clk_div_reg + 1'b1;
        tx_NEXT = tx_STOP;
    end
    else begin
        clk_div_next = 0;
        tx_NEXT = tx_DONE;
    end
    end

    tx_DONE: begin
        done_tx = 1;
        tx_NEXT = tx_IDLE;
    end

    default: tx_NEXT = tx_IDLE;
endcase
end

endmodule
```

Listing C.13: UART TX module - *uart_tx.sv*

```
/*
 *  Copyright (C) 2018  Siddharth J <www.siddharth.pro>
 *
 *  Permission to use, copy, modify, and/or distribute this software for any
 *  purpose with or without fee is hereby granted, provided that the above
 *  copyright notice and this permission notice appear in all copies.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 *  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 *  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 *  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 *  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 *  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 */

module uart_rx(clk,rst,rx,rx_data_out);

parameter clk_freq = 50000000; //MHz
parameter baud_rate = 19200; //bits per second
input clk;
input rst;
input rx;
output [7:0] rx_data_out;

localparam clock_divide = (clk_freq/baud_rate);

enum bit [2:0] { rx_IDLE = 3'b000,
                 rx_START = 3'b001,
         rx_DATA = 3'b010,
```

```
32          rx_STOP = 3'b011,
33          rx_DONE = 3'b100 } rx_STATE, rx_NEXT;
34
35 logic [11:0] clk_div_reg,clk_div_next;
36 logic [7:0] rx_data_reg,rx_data_next;
37 logic [2:0] index_bit_reg,index_bit_next;
38
39
40 always_ff @(posedge clk) begin
41 if(~rst) begin
42 rx_STATE <= rx_IDLE;
43 clk_div_reg <= 0;
44 rx_data_reg <= 0;
45 index_bit_reg <= 0;
46 end
47 else begin
48 rx_STATE <= rx_NEXT;
49 clk_div_reg <= clk_div_next;
50 rx_data_reg <= rx_data_next;
51 index_bit_reg <= index_bit_next;
52 end
53 end
54
55 always @(*) begin
56 rx_NEXT = rx_STATE;
57 clk_div_next = clk_div_reg;
58 rx_data_next = rx_data_reg;
59 index_bit_next = index_bit_reg;
60
61 case(rx_STATE)
62
63 rx_IDLE: begin
64 clk_div_next = 0;
65 index_bit_next = 0;
66 if(rx == 0) begin
67 rx_NEXT = rx_START;
68 end
69 else begin
70 rx_NEXT = rx_IDLE;
71 end
72 end
73
74 rx_START: begin
75 if(clk_div_reg == (clock_divide-1)/2) begin
76 if(rx == 0) begin
77 clk_div_next = 0;
78 rx_NEXT = rx_DATA;
79 end
80 else begin
81 rx_NEXT = rx_IDLE;
82 end
83 end
84 else begin
85 clk_div_next = clk_div_reg + 1'b1;
86 rx_NEXT = rx_START;
87 end
88 end
89
90 rx_DATA: begin
91 if(clk_div_reg < clock_divide-1) begin
92 clk_div_next = clk_div_reg + 1'b1;
93 rx_NEXT = rx_DATA;
94 end
95 else begin
96 clk_div_next = 0;
97 rx_data_next[index_bit_reg] = rx;
98 if(index_bit_reg < 7) begin
99 index_bit_next = index_bit_reg + 1'b1;
100 rx_NEXT = rx_DATA;
101 end
102 else begin
```

```
103 index_bit_next = 0;
104 rx_NEXT = rx_STOP;
105 end
106 end
107 end
108
109 rx_STOP: begin
110 if(clk_div_reg < clock_divide - 1) begin
111 clk_div_next = clk_div_reg + 1'b1;
112 rx_NEXT = rx_STOP;
113 end
114 else begin
115 clk_div_next = 0;
116 rx_NEXT = rx_DONE;
117 end
118 end
119
120 rx_DONE: begin
121 rx_NEXT = rx_IDLE;
122 end
123
124 default: rx_NEXT = rx_IDLE;
125 endcase
126 end
127
128 assign rx_data_out = rx_data_reg;
129
130 endmodule
```

Listing C.14: UART RX module - *uart_rx.sv*

D

Assembler code

This appendix provides the full code for the qRV32 custom assembler.

Assembler

```
from helper_functions import *

source_path = "D:/User/Documenti/TuDelft/Thesis/qrv_assembler/sources/hgate.nva"
output_path = "D:/User/Documenti/TuDelft/Thesis/qrv_assembler/outputs/hgate_bin.txt"

#initialize register status
global register_status
global source_pseudoinstruction
register_status = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0]
angle = True

#dictionaries
opcode_lsb = {
    "qgatee":      "000001011",
    "qgatecc":     "000001011",
    "qgateuc":     "000001011",
    "qgatedir":    "000001011",
    "qgateze":     "000001011",
    "qgatezc":     "000001011",

    "detectCarbon": "010001011",
    "magbias":      "010001011",
    "rabicheck":    "010001011",
    "initialize":   "010001011",
    "measuree":      "010001011",
    "crc":          "010001011",

    "swapec":      "010001011",
    "swapce":      "010001011",

    "nventangle":   "010001011",
    "set_local":    "010001011",

    "move_to":      "100001011",
    "move_from":    "100001011",

    "qld":          "100001011",
    "qst":          "100001011",

    "qldi":        "110001011",
    "qldi_a":      "110001011",

    "nop": "0010011",
    }
opcode_msb = {
```

```
    "qgatee":      "000",
    "qgatecc":     "001",
    "qgateuc":     "010",
    "qgatedir":    "011",
    "qgateze":     "100",
    "qgatezc":     "101",

    "detectCarbon": "10001",
    "magbias":      "10010",
    "rabicheck":    "10011",
    "initialize":   "10100",
    "measuree":      "10101",
    "crc":          "10110",

    "swapce":      "10111",
    "swapec":      "11000",

    "nventangle":   "001",
    "set_local":    "000",

    "move_to":      "100",
    "move_from":    "101",

    "qld":          "00",
    "qst":          "01",

    "qldi":        None,
    "qldi_a":      None,
    }

#helper functions
def opcode_lsb_gen(opcode):
    for key, value in opcode_lsb.items():
        if key == opcode:
            return value

def opcode_msb_gen(opcode):
    for key, value in opcode_msb.items():
        if key == opcode:
            return value

def find_next_available_register():
    # Find the first free register
    destination_register = -1  # Set to an invalid value as a fallback
    for i, status in enumerate(register_status):
        if status == 0:
            destination_register = i
            register_status[i] += 1  # Increment the register status value
            break

    # If no free registers are available, select the earliest used register
    if destination_register == -1:
        min_used_time = max(register_status) + 1  # Set to an invalid value as a fallback
        min_used_index = -1  # Set to an invalid value as a fallback
        for i, status in enumerate(register_status):
            if status < min_used_time:
                min_used_index = i
                min_used_time = status

        # If a register with the minimum value is found, set it as the destination register
        if min_used_index != -1:
            destination_register = min_used_index
            register_status[min_used_index] += 1  # Increment the register status value

    return destination_register


def pseudo_instruction_substitutor(opcode, asm_instruction):
    arguments = asm_instruction[1:]    #get the arguments to pass to new instructions
    asm_instruction_list = []
    parameter_register = 0
```

```
    subreg = 0

    #GATE INSTRUCTIONS: decompose in 2 qldi and 1 Qgate
    if opcode == "qgatee":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[1])])
        subreg += 1
        #2nd qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), "r"+str(parameter_register)])
    elif opcode == "qgatecc":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        subreg += 1
        #2nd qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[3])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), str(arguments[1]), "r"+str(
            parameter_register)])
    elif opcode == "qgateuc" or opcode == "qgatedir":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        subreg += 1
        #2nd qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[3])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), str(arguments[1]), "r"+str(
            parameter_register), str(arguments[4])])
    elif opcode == "qgatezc":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), str(arguments[1]), "r"+str(
            parameter_register)])
    elif opcode == "qgateze":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi_a", "r"+str(parameter_register), str(subreg), str(
            arguments[1])])
        #Qgate instruction
        asm_instruction_list.append([opcode, str(arguments[0]), "r"+str(parameter_register)])


    #CALIBRATION INSTRUCTIONS: decompose in 4 qldi and 1 original
    elif opcode == "detectCarbon" or opcode == "magbias" or opcode == "rabicheck":
        #find the next available register to load the parameters
        parameter_register = find_next_available_register()
        #1st qldi
        asm_instruction_list.append(["qldi", "r"+str(parameter_register), str(subreg), str(
            arguments[1])])
        subreg += 1
        #2nd qldi
        asm_instruction_list.append(["qldi", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
```

```
        subreg += 1
        #3rd qldi
        asm_instruction_list.append(["qldi", "r"+str(parameter_register), str(subreg), str(
            arguments[3])])
        subreg += 1
        #4th qldi
        asm_instruction_list.append(["qldi", "r"+str(parameter_register), str(subreg), str(
            arguments[4])])
        #original instruction
        asm_instruction_list.append([opcode, str(arguments[0]), "r"+str(parameter_register)])

    elif opcode == "set_local":
        #find the next available register to load the parameter
        parameter_register = find_next_available_register()
        #qldi
        asm_instruction_list.append(["qldi", "r"+str(parameter_register), str(subreg), str(
            arguments[2])])
        #original instruction
        asm_instruction_list.append([opcode, str(arguments[0]), str(arguments[1]) ,"r"+str(
            parameter_register)])

    else:   #no pseudo instruction found, return the original instruction
        asm_instruction_list.append(asm_instruction)

    return asm_instruction_list

def asm_to_bin(asm_instruction):
    bin_instruction = []

    opcode = asm_instruction[0]
    arguments = asm_instruction[1:]

    #msb opcode translation
    if (opcode != "qldi" and opcode != "qldi_a" and opcode != "nop"):
        bin_instruction.append(opcode_msb_gen(opcode))   #add the MSB opcode to the binary
            instruction

    #argument translation
    if (opcode == "qgatee" or opcode == "qgateze"):
        bin_instruction.append("11111")                                         #don't care bits
        bin_instruction.append(bin(int(arguments[1][1:]))[2:].zfill(5))      #add the
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)
    elif (opcode == "qgatecc" or opcode == "qgatezc"):
        bin_instruction.append("1")                                             #don't care bit
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(4))          #add the c13
            address (4 bit binary number)
        bin_instruction.append(bin(int(arguments[2][1:]))[2:].zfill(5))      #add the
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)
    elif opcode == "qgateuc" or opcode == "qgatedir":
        bin_instruction.append(int(arguments[3]))                               #add the
            preserve/dir bit
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(4))          #add the c13
            address (4 bit binary number)
        bin_instruction.append(bin(int(arguments[2][1:]))[2:].zfill(5))      #add the
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "initialize" or opcode == "measuree" or opcode == "crc":
        bin_instruction.append("11111111")                                      #don't care bits
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "detectCarbon" or opcode == "magbias" or opcode == "rabicheck":
        bin_instruction.append("111")                                           #don't care bits
        bin_instruction.append(bin(int(arguments[1][1:]))[2:].zfill(5))      #add the
            register address (5 bit binary number)
```

```
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "swapec":
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(4))          #add the c13
            address (4 bit binary number)
        bin_instruction.append("1111")                                       #don't care bits
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)
    elif opcode == "swapce":
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(4))          #add the c13
            address (4 bit binary number)
        bin_instruction.append(bin(int(arguments[2]))[2:].zfill(2))          #add the basis
            (2 bit binary number)
        bin_instruction.append("11")                                         #don't care bits
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "set_local":
        bin_instruction.append(bin(int(arguments[1][1:]))[2:].zfill(5))      #add the local
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[2][1:]))[2:].zfill(5))      #add the global
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "nventangle":
        bin_instruction.append("11")                                         #don't care bits
        bin_instruction.append(arguments[1])                                 #add the
            entanglement direction (3 bit binary number)
        bin_instruction.append("11111")                                      #don't care bits
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(10))     #add the NV
            center address (10 bit binary number)

    elif opcode == "move_to":
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(5))     #source register
            (general purpose register)
        bin_instruction.append(bin(int(arguments[1][2:]))[2:].zfill(5))     #destination
            register (quantum register)
        bin_instruction.append("1111111111")                                 #don't care bits
    elif opcode == "move_from":
        bin_instruction.append(bin(int(arguments[1][1:]))[2:].zfill(5))     #destination
            register (general purpose register)
        bin_instruction.append(bin(int(arguments[0][2:]))[2:].zfill(5))     #source register
            (quantum register)
        bin_instruction.append("1111111111")                                #don't care bits

    elif opcode == "qld" or opcode == "qst":
        immediate = bin(int(arguments[1][2:4], 16))[2:].zfill(11)             #get the
            immediate value and translate if from hex (11 bit binary number)
        imm_lsb = immediate[6:11]
        imm_msb = immediate[0:6]
        #imm_msb = immediate[5:11]
        #imm_lsb = immediate[0:5]

        address_source_register = bin(int(arguments[1][6:8]))[2:].zfill(5)    #get the source
             register address (5 bit binary number)

        bin_instruction.append(imm_msb)                                      #add the
            immediate mSB (5 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(5))      #add the
            register address (5 bit binary number)
        bin_instruction.append(address_source_register)                      #add the source
            register address (5 bit binary number)
        bin_instruction.append(imm_lsb)                                      #add the
            immediate lSB (5 bit binary number)

    elif opcode == "qldi_a":
        immediate = float_to_fixed_point(float(arguments[2]))                #if the
            instruction is a qgate, the immediate is a angle rotation
        imm_lsb = immediate[8:16]
```

```
        imm_msb = immediate[0:8]

        bin_instruction.append(imm_msb)                                         #add the
            immediate MSB (7 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(5))         #add the
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(2))             #add the
            subregister address (2 bit binary number)
        bin_instruction.append(imm_lsb)                                         #add the
            immediate LSB (8 bit binary number)

    elif opcode == "qldi":
        immediate = bin(int(arguments[2]))[2:].zfill(16)                        #else, is a
            integer number
        imm_lsb = immediate[8:16]
        imm_msb = immediate[0:8]

        bin_instruction.append(imm_msb)                                         #add the
            immediate MSB (7 bit binary number)
        bin_instruction.append(bin(int(arguments[0][1:]))[2:].zfill(5))         #add the
            register address (5 bit binary number)
        bin_instruction.append(bin(int(arguments[1]))[2:].zfill(2))             #add the
            subregister address (2 bit binary number)
        bin_instruction.append(imm_lsb)                                         #add the
            immediate LSB (8 bit binary number)

    elif opcode == "nop":
        bin_instruction.append('00000000000000000000000000')           #add the nop
            instruction (32 bit binary number)

    #lsb opcode translation
    bin_instruction.append(opcode_lsb_gen(opcode))   #add the MSB opcode to the binary
        instruction

    bin_instruction_string = ''.join(map(str, bin_instruction))
    return bin_instruction_string
    #return bin_instruction

def instructions_to_bin(asm_instruction_list):
    bin_instruction_list = []
    for asm_instruction in asm_instruction_list:
        bin_instruction_list.append(asm_to_bin(asm_instruction))
    return bin_instruction_list

#main function
asm_instruction_list = []       #list of instructions to be translated into binary
bin_instruction_list = []       #list of binary instructions
with open(source_path, "r") as source:
    for line in source:
        line = line.strip( )                #remove the newline character

        asm_instruction = line.split( )     #split the line into a list of strings
        opcode = asm_instruction[0]

        asm_instruction_list.append(pseudo_instruction_substitutor(opcode, asm_instruction))

    asm_instruction_list = [item for sublist in asm_instruction_list for item in sublist]
              #flatten the list
    #print(asm_instruction_list)
    bin_instruction_list = instructions_to_bin(asm_instruction_list)
    #print(bin_instruction_list)

asm_instruction_list_string = []
for sublist in asm_instruction_list:
    # Use the join method to concatenate the elements in the sublist into a single string
    sublist = ' '.join(sublist)
    # Append the string to the list of strings
    asm_instruction_list_string.append(sublist)

with open(output_path, 'w') as f:
    #insert dummy instruction
```

```
    f.write('00000000000000000000000000010011   //nop\n')
    # Loop through the lists and write each string to the file
    for b, asm in zip(bin_instruction_list, asm_instruction_list_string):
        f.write(b + '    ' + '//' + asm + '\n')  # Add content after the string, separated by
            a space
    f.write('00000000000100000000000001110011   //ebreak\n') #insert ebreak instruction
```

Listing D.1: qRV32_assembler.py

Helper functions

```
def float_to_fixed_point(num):
    # Multiply the floating point number by 2^14 to shift the decimal point 14 bits to the
        left.
    # This converts the floating point number to a fixed point number with 14 fractional bits
        .
    fixed = int(round(num * (2 ** 14)))

    # Convert the fixed point number to a binary string with 16 bits (including 14 fractional
         bits).
    binary = bin(fixed)[2:].zfill(16)

    return binary
```

Listing D.2: helper_functions.py

E

Simulation Results

The following appendix presents the recorded output of the Global Controller, at different levels of simulation. Firstly, it includes the execution of the whole Instruction Set Extension, and secondly, the realization of the H-gate circuit.

qRV32 ISE example instructions

```
00000000000000000000000000010011   //nop
01100100000000010000110110001011   //qldi_a r0 0 1.5707
01100100000000110001000110001011   //qldi_a r0 1 1.5707963267949
00011111000000000000000000001011   //qgatee q0 r0
00000000000010000000000110001011   //qldi_a r1 0 0.0
11001001000010100001111110001011   //qldi_a r1 1 3.1415535898
00110001000010000000001000001011   //qgatecc q1 1 r1
00000000000100000000000110001011   //qldi_a r2 0 0.0
11001001000100100010000110001011   //qldi_a r2 1 3.1415926535898
01010010000100000000010000001011   //qgateuc q2 2 r2 1
01100100000110010001000110001011   //qldi_a r3 0 1.570796367949
01100100000110110001000110001011   //qldi_a r3 1 1.57077949
01100011000110000000011000001011   //qgatedir q3 3 r3 0
01101101001000001001111110001011   //qldi_a r4 0 1.707963267949
10011111001000000000100000001011   //qgateze q4 r4
00100100001010010001000110001011   //qldi_a r5 0 0.5707963267949
10110101001010000000101000001011   //qgatezc q5 5 r5
10100111111110000001101010001011   //initialize q13
10101111111110000001111010001011   //measuree q15
10110111111110000010001010001011   //crc q17
00000000001100001100100110001011   //qldi r6 0 100
00010011001100110001000110001011   //qldi r6 1 5000
00000000001101011001000110001011   //qldi r6 2 200
00100111001101100010000110001011   //qldi r6 3 10000
10001111001100000111001010001011   //detectCarbon q57 r6
00000101001110011011100110001011   //qldi r7 0 1500
00010011001110110001000110001011   //qldi r7 1 5000
00000000001111011001000110001011   //qldi r7 2 200
00100111001111100010000110001011   //qldi r7 3 10000
10010111001110000111010010001011   //magbias q58 r7
00000000010000001100100110001011   //qldi r8 0 100
00010011010000110001000110001011   //qldi r8 1 5000
00000000010001011001000110001011   //qldi r8 2 200
00100111010001100010000110001011   //qldi r8 3 10000
10011111010000000111011010001011   //rabicheck q59 r8
11000011011111001011000010001011   //swapec q600 6
10111011110111001100011010001011   //swapce q611 7 2
11110001010010000100000110001011   //qldi r9 0 123456
00001111010011001100100010001011   //set_local q612 r15 r9
00111010111111001100101010001011   //nventangle q613 010
00000000000100000000000001110011   //ebreak
```

Simulation outputs

```
boot_instr = D:/User/Documenti/TuDelft/Thesis/qrv_assembler/outputs/test_bin.txt
packet #          1
NVnode       =    0
quantum_pack = 0000000000c910c90c1

packet #          2
NVnode       =    1
quantum_pack = 0200000001921e00002

packet #          3
NVnode       =    2
quantum_pack = 2400000001922000003

packet #          4
NVnode       =    3
quantum_pack = 0600000000c910c9104

packet #          5
NVnode       =    4
quantum_pack = 00000000000000da9e5

packet #          6
NVnode       =    5
quantum_pack = 0a00000000000049106

packet #          7
NVnode       =   13
quantum_pack = 0000000000c910c90cc

packet #          8
NVnode       =   15
quantum_pack = 0000000000c910c90cd

packet #          9
NVnode       =   17
quantum_pack = 0000000000c910c90ce

packet #         10
NVnode       =   57
quantum_pack = 004e200190271000c89

packet #         11
NVnode       =   58
quantum_pack = 004e20019027100bb8a

packet #         12
NVnode       =   59
quantum_pack = 004e200190271000c8b

packet #         13
NVnode       =  600
quantum_pack = 6c00000000c910c90cf

packet #         14
NVnode       =  611
quantum_pack = 0e00000000c910c90d0

packet #         15
NVnode       =  612
quantum_pack = 1e000000000001e2407

packet #         16
NVnode       =  613
quantum_pack = 0400000000c910c90c8
```

Listing E.1: Behavioral simulation results