Qtenon: Towards Low-Latency Architecture Integration for Accelerating Hybrid Quantum-Classical Computing

Chenning Tao
Zhejiang University
Hangzhou, China
tcn@zju.edu.cn

Liqiang Lu*
Zhejiang University
Hangzhou, China
liqianglu@zju.edu.cn

Size Zheng
ByteDance Seed
Beijing, China
zheng.size@bytedance.com

Li-Wen Chang
ByteDance Seed
Beijing, China
liwen.chang@bytedance.com

Minghua Shen
Sun Yat-sen University
Guangzhou, China
shenmh6@mail.sysu.edu.cn

Hanyu Zhang
Zhejiang University
Hangzhou, China
hyzz@zju.edu.cn

Fangxin Liu
Shanghai Jiao Tong University
Shanghai, China
liufangxin@sjtu.edu.cn

Kaiwen Zhou
Zhejiang University
Hangzhou, China
kaiwenzhou@zju.edu.cn

Jianwei Yin*
Zhejiang University
Hangzhou, China
zjuyjw@zju.edu.cn

Abstract

Hybrid quantum-classical algorithms have shown great promise in leveraging the computational potential of quantum systems. However, the efficiency of these algorithms is severely constrained by the limitations of current quantum hardware architectures. These architectures, which typically feature a decoupled design, lack both hardware support for low-latency communication and software support for fine-grained optimization.

In this paper, we propose Qtenon, a tightly coupled system for efficient hybrid quantum-classical algorithm acceleration. Qtenon is composed of both hardware part and software part. To enable efficient communication and computation, the hardware part provides a unified memory hierarchy, an efficient quantum controller, as well as a multi-stage processing pipeline. The unified memory hierarchy functions as a communication buffer between host and quantum accelerators, with dedicated data paths and interfaces provided by the quantum controller. The multi-stage pipeline leverages hardware pipelines to fully exploit parallelism. To program hybrid quantum-classical algorithms on the hardware, our software part provides a set of instructions for data communication and computation. The instructions also enable fine-grained synchronization and efficient scheduling for quantum-host interaction. We design Qtenon as a RISC-V extended chip and implement it using Chisel. In evaluation, we achieve up to 14.9× end-to-end speedup compared to state-of-the-art work for hybrid quantum-classical algorithms.

CCS Concepts

• **Hardware → Quantum technologies**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

*Corresponding Author

Keywords

Quantum computing, Instruction set architecture (ISA), Quantum-classical hybrid computing, RISC-V

ACM Reference Format:
Chenning Tao, Liqiang Lu, Size Zheng, Li-Wen Chang, Minghua Shen, Hanyu Zhang, Fangxin Liu, Kaiwen Zhou, and Jianwei Yin. 2025. Qtenon: Towards Low-Latency Architecture Integration for Accelerating Hybrid Quantum-Classical Computing. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731087

1 Introduction

Quantum computing has demonstrated great potential for accelerating complex algorithms that are inefficient on classical computers, such as combinatorial optimization [18] and quantum chemistry simulations [25]. To harness this potential, numerous hybrid quantum-classical algorithms [19, 29] have been proposed. However, current quantum computing systems struggle to execute these algorithms efficiently. Figure 1 (a) illustrates the performance of current quantum hardware architecture by evaluating three hybrid quantum-classical algorithms. The results indicate that the quantum execution contributes only a minor fraction of the overall runtime. For example, the quantum computation of 64-qubit Variational Quantum Eigensolver (VQE) [32] accounts for just 7.9% of the total runtime. Our profiled result in Figure 1(b) shows that the runtime is dominated by classical computation on the host and the communication between quantum and host. Over 90% of the runtime comes from communication, computation, and repeated compilations, underscoring a substantial space for performance enhancements.

The inefficiency of current quantum computing systems [13–15] arises from their decoupled architecture design and suboptimal software scheduling. In these systems, the architecture comprises a host processor and a quantum accelerator, with their interaction

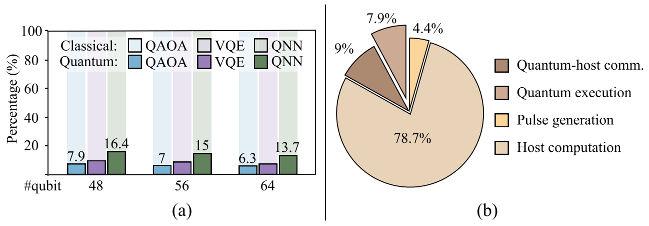


Figure 1: (a) The percentage of quantum and classical execution time for three hybrid quantum-classical algorithms: QAOA, VQE, and QNN. (b) Detailed time breakdown of executing 64-qubit VQE based on our profiling analysis.

typically managed by a dedicated FPGA controller. The host handles classical tasks such as parameter computation and cost function evaluation, while the quantum accelerator performs quantum computations. However, this straightforward decoupled design introduces significant limitations: **(1) Communication bottlenecks**: the communication between the host and quantum accelerator relies on low-speed network-based links, resulting in unacceptable transmission latency. **(2) Inefficient quantum-host interaction**: the decoupled architecture hinders the implementation of efficient instruction sets for seamless quantum-host interaction. It prevents the development of effective data exchange protocols or interfaces. These hardware limitations result in low software performance at both compile-time and runtime.

A tightly coupled system that integrates the host and quantum accelerator into a single, efficient unit could effectively address the aforementioned issues. However, building such an integrated system presents several challenges: **(1) Unified memory space design challenge**: integration requires a unified memory space to facilitate fast communication between the host and quantum accelerator. This demands an efficient memory organization strategy to ensure low-latency data exchange and high performance. **(2) Quantum controller design challenge**: The quantum accelerator imposes unique bandwidth demands. To meet these requirements, a high-performance quantum controller is essential to support enough bandwidth and enable efficient host-accelerator interaction through robust hardware interfaces. **(3) Instruction set architecture (ISA) design challenge**: Existing instruction sets are inadequate for tightly coupled architectures. Developing such systems requires a new ISA tailored for hybrid quantum-classical algorithms, supporting both efficient computation and data communication while maintaining well-defined memory consistency. These challenges highlight the complexity of designing a cohesive and tightly integrated system that seamlessly combines the host and quantum accelerator.

In this paper, we present Qtenon, a tightly coupled system designed to efficiently execute hybrid quantum-classical workloads, as shown in Figure 3. Qtenon addresses the challenges outlined for both the hardware side and the software side. For the hardware design, Qtenon first incorporates a unified memory space that seamlessly integrates the host and quantum accelerator, enabling efficient data sharing and management. To ensure scalability, the memory hierarchy is organized into a 2D space, where the unified memory is divided into segments, with each dedicated to the qubits used in hybrid quantum-classical algorithms. Then, built upon the unified memory space, Qtenon features an advanced quantum controller designed to optimize communication between the host and quantum accelerator. It provides four dedicated data paths with efficient hardware interfaces, allowing for transparent memory access and enabling low-latency access to the host memory hierarchy from the quantum side. Finally, leveraging the efficient host-accelerator communication, Qtenon implements a multi-stage pipeline for control pulse computation. This pipeline maximizes computational parallelism and minimizes redundant operations, ensuring highly efficient execution of quantum-classical tasks.

For the software design, we propose a customized instruction set architecture (ISA) tailored to the hardware architecture, incorporating both computation and data communication instructions. The ISA ensures memory consistency within the unified memory space, enabling fine-grained synchronization. Fine-grained control provided by the ISA allows for incremental compilation at runtime, reducing redundant compilation overheads and significantly improving overall system efficiency. Building on the ISA, we also develop a scheduling algorithm for quantum-host interaction, ensuring seamless coordination and efficient execution. The hardware and software parts together enable efficient execution of hybrid quantum-classical algorithms. In summary, our contributions are as follows:

- We propose Qtenon, an integrated hardware-software system optimized for hybrid quantum-classical workloads. We implement Qtenon in Chisel as an RISC-V extended ASIC chip.
- For hardware innovations, we introduce a unified memory space managed by an efficient quantum controller and implement a multi-stage pipeline for efficient pulse generation.
- For software innovations, we design a custom ISA that includes computation and data communication instructions. The ISA enables both memory consistency support and efficient quantum-host scheduling.

Experimental results demonstrate that Qtenon efficiently handles various Variational Quantum Algorithms (VQAs), achieving a speedup of up to 441.5× for classical processing. Furthermore, it delivers up to 14.9× end-to-end speedup compared to state-of-the-art quantum hardware architectures.

2 Background

2.1 Hybrid Quantum-Classical Algorithm

Hybrid quantum-classical algorithms are a promising approach for leveraging the strengths of both quantum and classical computing, particularly on Noisy Intermediate-Scale Quantum (NISQ) devices [16]. One of the most widely studied hybrid algorithms types is the Variational Quantum Algorithms (VQAs) [6, 8, 9, 19, 27, 34]. VQAs address optimization problems by employing parameterized quantum circuits to explore solution spaces and using classical optimization methods to minimize a cost function. Prominent examples of VQAs include the Quantum Approximate Optimization Algorithm (QAOA) [4, 22, 23], the Variational Quantum Eigensolver (VQE) [32], and Quantum Neural Networks (QNNs) [41].

QAOA is tailored for combinatorial optimization problems, which involve finding the best configuration of variables under specific

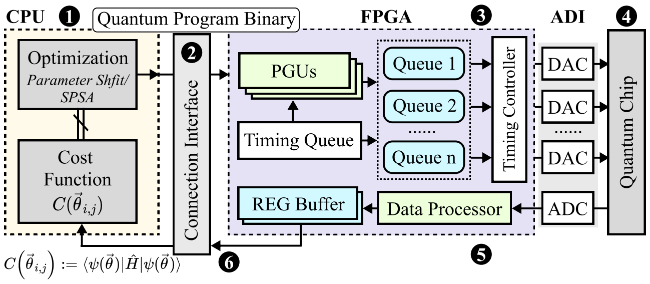


Figure 2: Example of existing hardware architecture when running a hybrid quantum-classical algorithm.

constraints—a challenge central to fields like operations research and logistics. VQE applies the variational principle to determine the ground-state energy of a Hamiltonian, addressing critical problems in quantum chemistry and condensed matter physics. Meanwhile, QNNs bring the potential of quantum-enhanced machine learning, leveraging quantum systems' high-dimensional feature space to solve complex computational tasks. In addition, hybrid approaches have been proposed for accelerating classical high-complexity problems, such as propositional satisfiability problem (SAT) problems [29].

2.2 Existing Quantum Hardware Design

Existing quantum hardware systems [21, 37] use decoupled system design and often consist of three main components: a host, which manages high-level control tasks such as parameter tuning and circuit compilation; one FPGA controller, which manages the quantum chip by converting high-level quantum programs into low-level pulse instructions and manages data transmission to and from the Analog-Digital Interface (ADI); and the quantum chip itself.

Figure 2 shows the process of one iteration of the hybrid quantum-classical algorithm (e.g., QAOA). The host transpiles the quantum circuit based on the quantum hardware. The transpiled quantum circuits are then compiled into binary programs based on the quantum ISA. The compiled instructions are then transmitted through ❷ Gigabyte Ethernet to FPGA. FPGA controller then uses ❸ Pulse Generation Units (PGUs) to process the compiled instructions and calculate corresponding pulses to control the evolution of superconducting quantum bits. These pulses are to be combined with the IQ mixer, converted into the analog signal by the DAC, and transmitted to the ❹ quantum chip. The results of the quantum chip can be retrieved through ADC, which are then processed by ❺ data processors to produce state determination, after which they are sent back to the host for cost estimation and parameter update.

2.3 Existing Quantum Software Design

The existing ISAs for hybrid quantum-classical workloads can be divided into two parts: quantum-dedicated [13, 15] and unified ISA [5]. Quantum-dedicated ISA only supports instructions for quantum chips. eQASM [13] uses ISA that can be translated from OpenQASM [7] and supports 7-qubit programming. HiSEP-Q [15] extends eQASM's ISA with a more efficient qubit encoding method and extends the qubit number to 128. On the contrary, unified ISA extends RISC-V ISA so that the instructions can be used by both host side and accelerator side. QUASAR and its vector extension, qV [5] are typical examples. Their support is up to 512 qubits. They also support a single instruction multiple data (SIMD) programming model. Unified ISA makes it easier to develop efficient programs for tightly coupled systems, but the synchronization between host and quantum accelerator is not efficient in previous work. They use `FENCE` instruction for synchronization without fine-grained memory consistency support. The `FENCE` instruction enforces a strict ordering of program execution. Although `FENCE` has good support on various systems, the coarse-grained strict synchronization overhead is unacceptable.

3 Motivational Example

Table 1: The comparison of different quantum system architectures.

System	Decoupled System		Tightly Coupled System
	eQASM [13]	HiSEP-Q [15]	**Qtenon (ours)**
Unified Memory	✘	✘	✔
Memory Consistency	✘	✘	✔
Data Interface	USB	Ethernet	**Tilelink [30] & RoCC [1]**
Q-H Comm. Support*	✘	✘	✔
Comm. Latency	$\sim 1ms$	$\sim 10ms$	**$10ns\sim 100ns$**
Instruction Counts**	✘	$\sim 3\times 10^4$	**~285**
Recompile Overhead	$1ms\sim 100ms$	$1ms\sim 100ms$	**$10ns\sim 100ns$**
Execution	Sequential	Sequential	**Interleaved**

* Q-H Comm. Support: Quantum-host communication support.
** We estimate the 64-qubit QAOA algorithm with five layers, running for ten iterations with a gradient descent optimizer. Since other ISAs only support the quantum part, this count includes only the quantum instructions.

To motivate our work, we use Table 1 as a comparison between two different system designs. The decoupled system design separates the host side and the quantum side and uses a dedicated FPGA controller to connect the two parts. eQASM [13] and HiSEP-Q [15] are typical examples of decoupled system design. They use a decoupled architecture and thus separate the quantum ISA from the host ISA for software. As for controller design, they leverage FPGA storage for quantum control and communication between the host and the quantum accelerator. During execution, the data is transmitted through the FPGA interfaces (usually USB or Ethernet), with a latency from $1ms$ to $10ms$, On the contrary, our tightly coupled system design employs a unified memory space hierarchy for the whole system with memory consistency support for both parts. The host and accelerator are linked by low-latency surfaces (e.g., Tilelink and RoCC) with dedicated instruction and hardware data path. The data transfer overhead ranges from $10ns$ to $100ns$, which is significantly better than the decoupled architecture.

In addition, decoupled system design usually uses dedicated ISA for quantum chips without consideration for the interaction between host and quantum accelerator. The dedicated ISA often encodes qubit index statically into the program, resulting in long instruction sequences after compilation (more than 10^4 instructions). Also, the lack of communication support forces the code to be recompiled from scratch each iteration, and the recompilation overhead ranges from $1ms$ to $100ms$. Worse still, the compiled program must be executed in a single pass without any overlapping

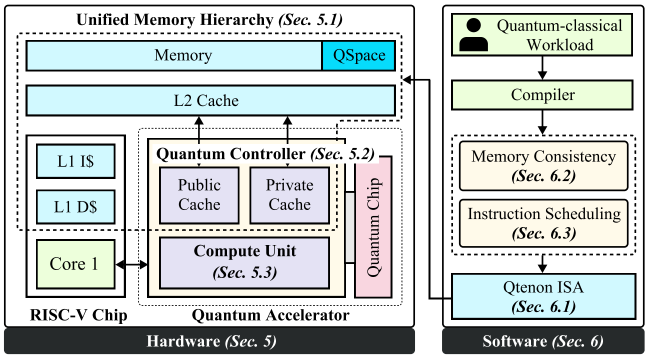


Figure 3: Overview of Qtenon system.

possibilities with host processing. As for our tightly coupled design, we support both computation and communication instructions in our ISA, enabling fine-grained synchronization control between host and quantum accelerator. Moreover, based on the scalable unified memory management from hardware design, our ISA is able to encode a variety lengths of qubits with much less instructions (e.g., 285 is enough for 64 qubits). The communication ability also makes it possible to perform incremental compilation, reducing recompilation overhead to less than 100ns in practice.

Overall, the benefits of both the hardware side and software side motivate our design and implementation of Qtenon, which is a tightly coupled system designed to efficiently execute hybrid quantum-classical workloads.

4 Qtenon Overview

The overall design of the Qtenon system is illustrated in Figure 3. It consists of two key components: hardware and software. For seamless integration and high-performance communication between the quantum and host systems, we propose a unified memory hierarchy (Section 5.1). Quantum program execution is facilitated by a quantum controller featuring specialized interfaces for cache communication and dedicated data paths (Section 5.2). Additionally, a multi-stage hardware pipeline is implemented to perform quantum control pulse calculations (Section 5.3). The hardware design is elaborated in Section 5. To enable efficient communication in this tightly integrated system, we introduce Qtenon ISA (Section 6.1). The user's hybrid quantum-classical workload can be programmed using our ISA with computation and communication instructions. The program's execution flow is optimized by an instruction scheduling algorithm (Section 6.3) to overlap quantum execution with classical processing with the support of fine-grained memory consistency (Section 6.2). The software design is detailed in Section 6.

5 Hardware Design of Qtenon

In this Section, we explain the hardware design of Qtenon. To tightly integrate the host and quantum accelerator components, we introduce a unified memory hierarchy and enable fast communication between the host side and the quantum chip side via a quantum controller; we also introduce a multi-stage hardware pipeline for fast computation of quantum chip control pulses.

Table 2: Quantum controller cache design for 64 qubits.

Segment	Description	Size
.program	• **Quantum program instructions.** • #Entries: 64 set×1024 entry • Entry size: type (4b)+reg_flag (1b)+ data (27b)+status (3b)+qaddr (30b)	**520 KB**
.pulse	• **Control pulses for quantum chip.** • #Entries: 64 set×1024 entry • Entry size: 10×64 bit	**5 MB**
.measure	• **Processed readout data.** • #Entries: 5120 entry • Entry size: 64 bit	**40 KB**
.slt	• **Skip Lookup Table** • #Entries: 64 set×2 way×128 entry • Entry size: tag (20b)+qaddr (30b)+ valid (1b)+count (5b)	**112 KB**
.regfile	• **Frequently updated parameters.** • #Entries: 1024 entry • Entry size: 32 bit	**4 KB**
Total		**5.66 MB**

5.1 Unified Memory Hierarchy

The tightly coupled architecture of Qtenon necessitates a unified memory hierarchy that seamlessly integrates both the host and quantum accelerator. To achieve this, we adopt a standard multi-level cache design and introduce a new memory space, referred to as the *quantum controller cache*, as illustrated in Figure 4. The quantum controller cache is implemented as an SRAM buffer and positioned at the same hierarchical level as the host's L1 cache, providing high bandwidth for data communication.

To achieve scalable management of the quantum controller cache, we organize it as a 2D space. For the first dimension, we divide the memory space into five segments; for the second dimension, we divide each segment into a list of qubit chunks. This 2D space makes the system flexible and easy to expand to larger quantum qubit space. We show the five segments in Figure 4 and explain the usage of each segment in Table 2. The *.slt* and *.pulse* segments are kept private to ensure system integrity. The Skip Lookup Table (*.slt*) segment, analogous to cache tag management in classical architectures, operates under exclusive hardware control through dedicated on-chip logic. This design choice stems from two critical factors: 1) the absence of QAddress mapping prevents direct CPU access, and 2) the SLT requires dynamic real-time updates during pulse generation to maintain temporal consistency between the *.program* instructions and *.pulse* outputs. The *.pulse* segment contains PGU-generated control pulses that could theoretically support user access. However, public exposure would mandate three-way synchronization across the interdependent *.program*, *.pulse*, and *.slt* segments, creating significant hardware coordination overhead and software complexity. Our architecture preserves both performance efficiency and data security by keeping these segments private through hardware isolation. The *.program*, *.regfile*, and *.measure* segments are public and accessible to users. The *.program* segment stores the quantum program instructions including data type, reg_flag, and data. The *.measure* segment stores processed readout data from the quantum chip. The *.reg* segment stores the register

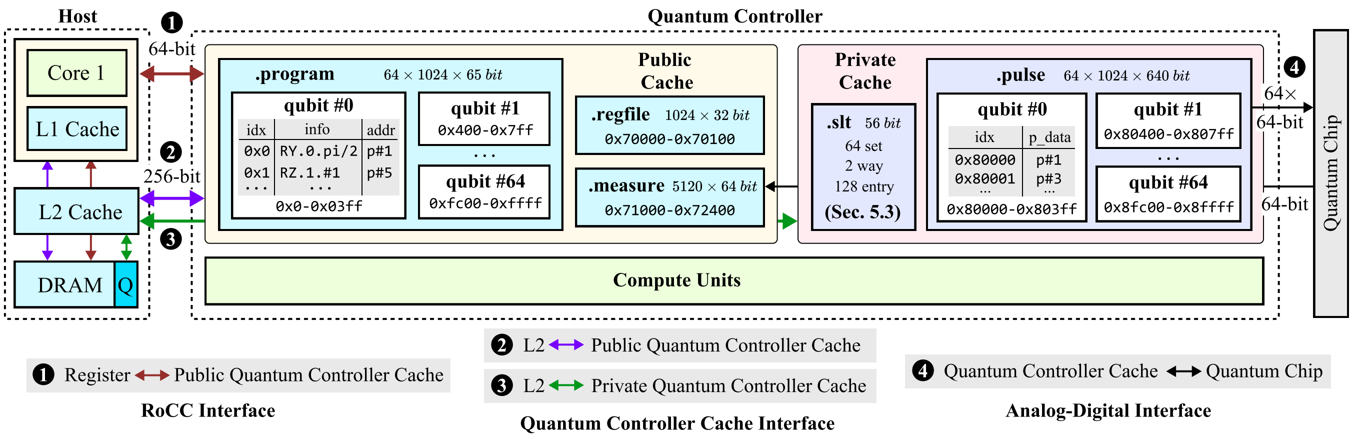


Figure 4: Unified memory space and quantum controller of Qtenon with 64 qubits.

parameters, which are to be linked to the *.program* segment after compilation.

Each segment is further organized as a list of qubit chunks. The chunks provide dedicated memory addresses for each qubit as shown in Figure 4, and this address is called QAddress. For *.regfile* and *.measure* segments, their memory space is shared by all qubits. Organizing the segments into chunks makes it possible to reduce quantum program code size and quantum data transfer overhead. By assigning a dedicated memory address range to each qubit, we eliminate the need to include the qubit index in each entry of the quantum program definition, as the index is inherently encoded within the address space. This also makes it more efficient to transfer large quantum programs.

Figure 4 and Table 2 show an example for a 64 qubit design: The whole *.program* segment space is implemented with 64 sets, corresponding to 64 qubits. Each qubit is assigned to a program memory of 1024 entries, the address range for the first qubit is defined as `0x0-0x3ff`, for the second qubit as `0x400-0x7ff`, and so forth. The assigned address will be used as an index to access the corresponding entry. Each entry employs a type and data space to define the gate parameter. If this gate's parameter is frequently updated, the reg_flag bit is set to 1, and the data field stores the *.regfile* index. QAddress links the *.pulse* data using quantum address. The *.regfile* is set with 1024 entries, each 32 bits long, starting at `0x70000`. The *.measure* segment employs 5120 entries, with each entry 64 bits long, and the start address is `0x71000`. The *.pulse* and *.slt* segments also have the same number of sets as the number of qubits. The *.pulse* segment employs 1024 entries, with each entry 640 bits long to meet the bandwidth requirement for the output data. The total memory space for the quantum controller cache is 5.66 MB, with the majority of the space allocated to *.pulse* segment.

5.2 Quantum Controller

Quantum controller is used to control the quantum chip, which is composed of quantum controller cache and compute units for pulses. The quantum controller has three outer connection interfaces for it to interact with the host and the quantum chip.

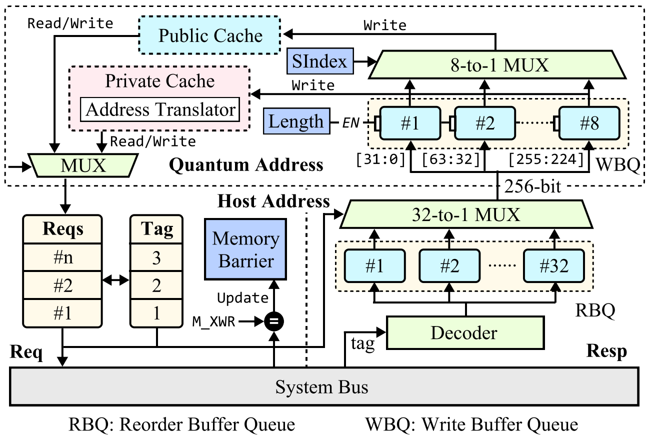


Figure 5: Quantum controller cache interfaces.

For the connection between the quantum controller and host, Qtenon supports three data paths: ❶ host core register and public quantum controller cache, ❷ host L2 cache and public quantum controller cache, and ❸ host L2 cache and private quantum controller cache. For the connection between the quantum controller and the quantum chip, Qtenon uses the datapath ❹. The data path ❶ uses the RoCC interface for transmitting data. This data path features one-cycle latency and transmits data in 64-bit segments, making it ideal for transmitting small data, such as updated parameters or information from the quantum. The data path ❷ and ❸ use the quantum controller cache interface shown in Figure 5 for transmission. The data path ❷ has a higher latency than ❶ but enables larger data transmissions, such as transferring complete quantum program instructions. Data path ❶ and ❷ feature transmission in public quantum controller cache space. The data path ❸ establishes a communication pathway between the host L2 cache and the private memory space within the quantum controller, which remains inaccessible to the user. A dedicated DRAM region, QSpace, reserved exclusively for quantum data, is directly linked to the controller's private memory space. This memory address range is shielded from the CPU and will not be accessed by the host core.

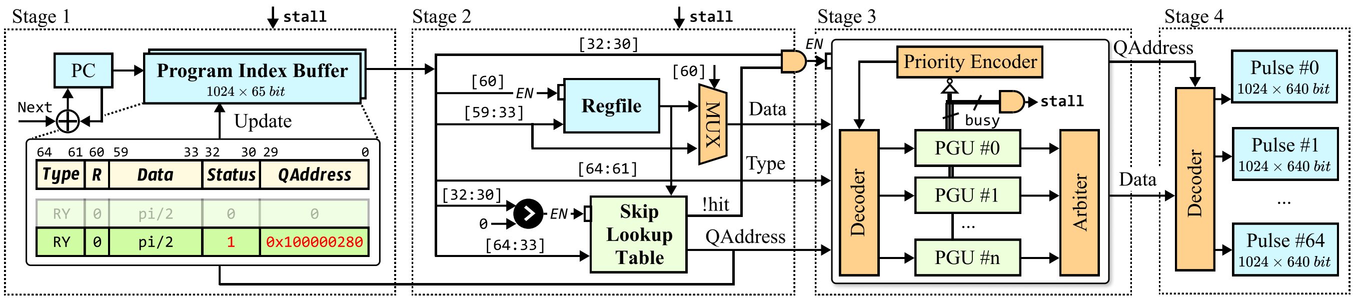


Figure 6: Four-stage computation workflow.

This provides a connection for the quantum controller to leverage the host memory hierarchy.

Enabling datapath ❷❸ requires establishing a connection between the host L2 cache and quantum cache. This presents three challenges: maintaining memory consistency between quantum and host, managing out-of-order responses from the system bus, and handling varying data widths across components. A memory barrier mechanism is implemented to maintain high-efficiency cache consistency between the quantum and host. The Reorder Buffer Queue (RBQ) is introduced to manage out-of-order responses. The Write Buffer Queue (WBQ) design is employed for efficient processing between different data widths. The detailed architecture is illustrated in Figure 5. The address space is divided into the quantum address domain and the host address domain.

The RBQ address the out-of-order response issue. The system bus employs a TileLink interface, with requests buffered in an output queue. Each request is associated with a unique 5-bit tag. Requests are issued whenever the bus is free, and a unique tag is available. Since responses arrive out of order, an RBQ (containing 32 entries corresponding to the number of unique tag numbers) is utilized to realign responses based on their tags. Responses are enqueued into their corresponding queues, and during output, the tag queue determines the specific data to be dequeued and output in the correct order. The memory barrier will be updated if the request is a write instruction for the cache, indicating those addresses are synchronized.

To address the varying data widths in the public quantum controller cache space (e.g., 32-bit width for programs) and the system bus, a WBQ with 8 separate 32-bit queues is employed. Each queue corresponds to a 32-bit segment, and during an enqueue operation, the requested data length is used to determine which queues can accommodate the data. This ensures that requests of varying lengths are efficiently mapped to the available queues. The indexing mechanism (SIndex) then determines which data will be written in the public space.

The quantum controller cache .pulse directly connects the quantum chip through the Analog-Digital Interface ❹. This aligns with the high bandwidth requirement of the ADI interface for controlling the quantum chip. Qtenon assumes each qubit requires two 16-bit, 2GHz Digital-to-Analog Converters (DACs). This setup imposes a bandwidth requirement of 64 bits/ns (16 bits × 2 DACs × 2 GHz), equivalent to 8 GB/s per qubit. The generated pulse is organized in the output order with each entry of 640 bit width. Assume our SRAM runs at 200MHz. To meet the output requirement of the DACs, each data entry is put into ten parallel 64 bit buffers before executing. It is then fed into a SerDes unit, which bridges the SRAM and DAC by serializing the data at the target 2 GHz DAC frequency. This organization ensures the chip can handle the data throughput required by the system.

5.3 Multi-stage Hardware Pipeline

To maximize the parallelism of the preprocessing process and fully utilize the compute power, we propose a four-stage hardware pipeline, illustrated in Figure 6, for executing the pulse computing workload. Such a multi-stage pipeline is only feasible with the support of our unified memory space and quantum controller.

The processing begins with Stage 1, which reads the circuit definition from the Program Index Buffer based on the address and sends the data to Stage 2. In Stage 2, the processing data is decoded. If the *R* field is set to 1, gate data is fetched from the Regfile; otherwise, the data is taken directly from the Program Index Buffer. The *Status* field indicates whether the *QAddress* in the Program Index Buffer is valid. If the status is 0, the *QAddress* is invalid, further processing is needed. The *Type* and *Data* are forwarded into the SLT for processing, which is used to either retrieve the cached *QAddress* if this parameter has been computed before or allocate a new *QAddress*. The returned *QAddress* is updated in the Program Index Buffer. If this parameter has not been computed before, the allocated QAddress is also passed to Stage 3.

In Stage 3, the pulse generation process is executed, leveraging compute parallelism through a Priority Encoder to select a free PGU and using a decoder to assign data accordingly. If all PGUs are occupied, a stall signal is sent to stall Stage 1 and 2. However, Stage 4 is unaffected by the stall and is connected to Stage 3 using a ready-valid signal. All PGUs are linked to an Arbiter to handle data contention, which resolves conflicts by selecting the valid signal from the PGU. The Arbiter then forwards the output signal to the Decoder, which writes the results to the corresponding QAddress in Pulse Cache. This design optimizes parallelism by ensuring pulse generation continues seamlessly, even in resource contention.

To skip the redundant computation of control pulses, we design an SLT workflow to link the parameters with the corresponding pulse address in the quantum controller if the pulse has been computed before. The detailed workflow for SLT is shown in Figure 7. Each qubit has its own SLT. Each SLT is configured in two sets, each containing 128 entries. ❶ The input data is truncated into a 3-bit

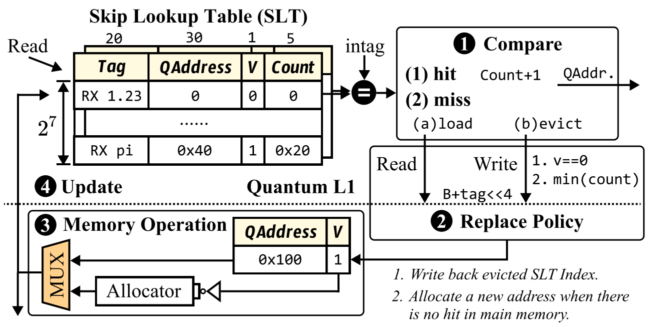


Figure 7: The workflow for SLT.

type field and a 4-bit data field (representing two digits before and after the decimal point). These fields are concatenated to form an index used to query the SLT. Upon querying, the SLT compares the input tag with the stored tag in each entry. If a match is found and the valid bit is set to 1, it returns the corresponding QAddress to the Program Index Buffer and disables further pulse generation. If all entries in the SLT result in a miss, the SLT initiates the replacement policy, which follows the Least Count (LC) approach.

❷ The LC replacement policy prioritizes invalid entries for replacement. If any entry has its valid bit set to 0, it is replaced without requiring a write-back to classical memory. However, if all entries are valid, the policy evicts the entry with the least count. When an entry is evicted, the SLT writes it back to the QSpace in classical memory, involving an address translation process based on the tag and the base address of QSpace. Additionally, the SLT requests the tag stored in QSpace through the same translation mechanism. ❸ QSpace, designed for quantum state storage, allocates $2^{20} \times 4 = 4$ MB per qubit, derived from the 20-bit tag width and 4-byte entry size. Suppose the entry in QSpace corresponding to the requested tag is invalid. In that case, the system signals the allocator, which generates a new QAddress and passes it to the next pipeline stage for pulse generation. Conversely, if the QSpace entry is valid, the existing QAddress is returned. ❹ Finally, the SLT updates its corresponding entry to reflect the current state, ensuring consistency across the memory hierarchy.

6 Software Design of Qtenon

In this Section, we explain the software part of Qtenon. The software part comprises three components: ISA design and compilation; memory consistency; and software scheduling.

Table 3: Qtenon's extended ISA.

Type	Instruction	Explanation
Data Comm.	q_update	Host Register ⟶ Quantum Controller Cache.
	q_set	Host Memory ⟶ Quantum Controller Cache.
	q_acquire	Quantum Controller Cache ⟶ Host Memory.
Computation	q_gen	Generate pulse.
	q_run	Run the quantum program for the specified number of shots.

6.1 ISA Extension and Compilation

The key insight of our ISA is to treat the quantum program as computable data rather than as a sequential static list of instructions,

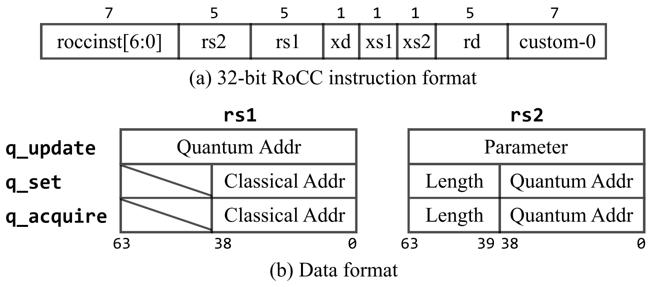


Figure 8: *Data communication* instructions.

since the quantum accelerator will inherently process and order instructions based on their timing. This reduces the code transmission size by encoding indices as quantum addresses. In detail, we propose two types of ISA: *computation* and *data communication*, which are shown in Table 3. Both types of instructions follow the RoCC extension format. The RoCC instruction format is explained in Figure 8 (a). The *roccinst* field specifies the instruction type. The *rs1* and *rs2* fields are used to refer to source register files. The *rd* field is used to refer to the destination register. The *xd*, *xs1*, *xs2* fields are used to indicate whether the corresponding register is used in the instruction.

For computation, we add two instructions: `q_gen` and `q_run`. `q_gen` instruction triggers the pulse computation, which generates the pulses to control the quantum chip, while `q_run` instruction runs the quantum program multiple times (which is determined by the value stored in `rs1` register) and retrieves the measurement results in the corresponding memory segment. For communication, we add three instructions: `q_set`, `q_update`, and `q_acquire`. The `q_update` instruction transfers data from the host core register to the quantum controller cache. This instruction uses data path ❶ in Figure 4. The *rs1* field specifies the destination quantum address, while the *rs2* field contains the data to be transferred. The `q_set` and `q_acquire` instructions handle the data communication between host memory and quantum controller cache. These two instructions use the data path ❷ in Figure 4. For these two instructions, *rs1* field holds the starting address in the classical memory, while the lower 39 bits of register *#rs2* represent the starting address in quantum memory space. The upper 25 bits of register *#rs2* indicate the data length to be transferred. The `q_set` instruction is used to load the quantum program to the program segment, whereas the `q_acquire` operation is used to retrieve data from .measure segment.

Dynamic Incremental Compilation: For the hybrid quantum-classical algorithms, the quantum programs across consecutive iterations exhibit quantum locality—only part of the parameters need updates, while all other program codes remain identical. Previous work [5, 13, 15] compiles the algorithms and generates code from scratch each iteration (just-in-time compilation), incurring huge compilation overhead at runtime. Furthermore, the newly generated code should be transferred from the host to the quantum controller as a whole, resulting in low runtime performance. On the contrary, our ISA uses communication instructions to enable incremental compilation at runtime. In detail, we treat every gate in a quantum program as a parameter, which can be computed and updated independently. We include a *reg_flag* bit within the program definition to indicate whether the registers can be updated

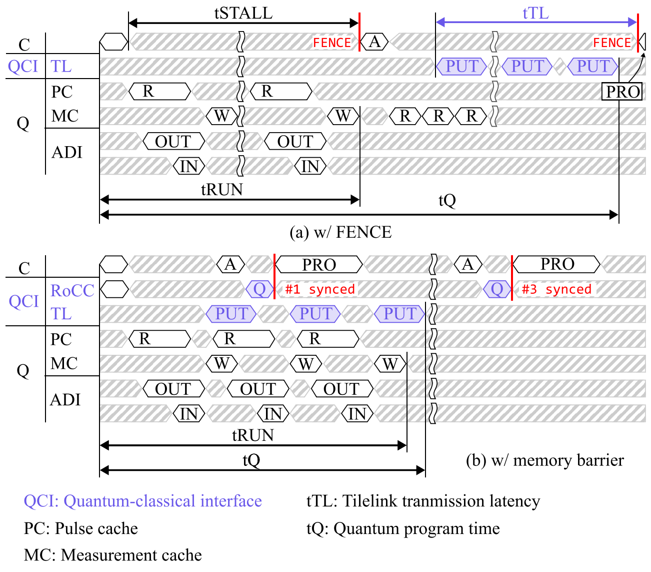


Figure 9: Compare the timing of different synchronization methods. (a)FENCE operation stalls the classical pipeline until all quantum operations are complete. (b) Fine-grained synchronization allows the overlapping of quantum execution and classical post-processing. Memory consistency is ensured by querying through the RoCC interface.

directly in each iteration. Once this bit is set, subsequent updates to the quantum program code only happen to the selected registers using the q_update instruction, eliminating the need to recompile the entire program.

6.2 Memory Consistency

Our tightly coupled system requires memory consistency between the quantum controller cache and host memory. Two potential data races could occur in this setup: 1. Data race between q_set and q_gen: Initializing the pulse generation process on an address before the program setting on this address is completed. 2. Data race between q_run, q_acquire, and post-processing: Acquiring a quantum address before execution completes or the host accessing memory before the quantum controller finishes writing. For the first case, a barrier can be added to the quantum controller cache, effectively resolving the issue without requiring any modifications to the software. For the second case, the design differs for different architectures. Previous work [36] uses FENCE instruction to synchronize the data. But directly using FENCE instruction for quantum controller cache and host memory introduces significant performance overhead. We use an example in Figure 9 (a) for explanation. In this example, two FENCE instructions are required. The first FENCE instruction is used to resolve the data race between q_run and q_acquire, resulting in a stall time of tSTALL on the host side. The second FENCE instruction solves the data race between q_acquire and host post-processing. The host can only start the post-processing process after all Tilelink transmissions are completed.

Algorithm 1: Batched Transmission Policy

```
    Input: Number of qubits $N$, bus width $B$, total shots $S$
1   Compute transmission interval $K \leftarrow \lfloor B/N \rfloor$;
2   Initialize $batch \leftarrow \emptyset$;
3   Initialize $batch\_count \leftarrow 0$;
4   Initialize $addr \leftarrow host\_addr$;
5   for each shots $r = 1, 2, \ldots, S$ do
6       Run quantum circuit for $N$ qubits;
7       $result \leftarrow$ measurement results;
8       Append $result$ to $batch$;
9       $batch\_count \leftarrow batch\_count + 1$;
10      if $batch\_count = K$ then
11          Tilelink PUT $\leftarrow batch, addr$ ;
12          $addr = addr + \lceil N/8 \rceil \times K$;
13          Clear $batch$ and reset $batch\_count \leftarrow 0$;
14  if $batch \neq \emptyset$ then
15      Transmit remaining $batch$;
16      Clear $batch$;
```

To address the latency introduced by the FENCE instruction, we use fine-grained synchronization in the quantum controller cache. The benefit is that it allows efficient instruction scheduling and overlapping for quantum, quantum-host transmission, and host processing, which can effectively mask some transmission and post-processing latency. The example in Figure 9 (b) illustrates the benefits of this fine-grained synchronization. The q_run and q_acquire instructions are allowed to overlap. The TileLink PUT instruction is initiated immediately after each quantum run and executed in parallel with the MC's write operation. The CPU can access and post-process the part of the synchronized data before the whole execution is completed.

To implement fine-grained synchronization, we introduced a cache consistency protocol for the quantum controller cache and host memory by implementing a soft memory barrier on the host memory address designated for synchronization with the quantum controller cache. When the CPU attempts to access a host address synchronized by the quantum controller cache, it first queries the memory barrier within the quantum controller via the RoCC interface. This query is non-blocking, allowing ongoing instructions on the quantum controller to proceed uninterrupted and incurring only a single-cycle latency. The memory barrier (see Section 5.2) monitors the status of the PUT requests issued by the controller. If the corresponding address's write request has been sent through the system bus, the controller returns a valid signal.

6.3 Efficient Quantum-Host Scheduling

Our proposed synchronization method, incorporating a memory barrier, enables fine-grained instruction scheduling between quantum and host, allowing instructions such as q_run, q_acquire, and host post-processing to overlap effectively. This capability is crucial for achieving efficient quantum-classical integration.

A simple transmission scheduling approach transmits the measurement results immediately after each quantum circuit run. While

Table 4: Hardware configurations for Qtenon.

Part	Configuration	
Core	Rocket[1] @ 1GHz	Boom-L [40] @ 1GHz
L1	16KB 4-way I-Cache, 16KB 4-way D-Cache	
QCC	5.66 MB, configured according to Table 2	
QC	64 qubits, 8 PGUs	
L2	512KB 8-bank 4-way	
Memory	16GB DDR3 4-bank	

* QC: Quantum Controller
** QCC: Quantum Controller Cache

straightforward, this method has the drawback of increasing the number of bus accesses. For example, our 64-qubit setup triggers a 64-bit transmission after every measurement, resulting in four times the demand on the system bus due to under-utilization of bus bandwidth (256 bits/cycle).

To improve bandwidth utilization, we propose batched transmission shown in Algorithm 1. This scheduling approach leverages speed and efficiency by batching the measurement results to utilize the bandwidth fully. Given the bus width B and the number of qubits N, the transmission interval is determined as $K = \lfloor B/N \rfloor$. This means the transmission is scheduled to happen every K shots (e.g., four shots per transmission in our setup). The result is appended to the batch for each shot and increments the batch_count (lines 6-9). As the batch_count reaches K, the controller outputs a Tilelink `PUT` request while increasing the *addr* given the data width (lines 10-12). After all the shots are finished, the remaining data in the batch is transmitted to the host (lines 14-16).

7 Experiment

7.1 Experiment Setup

Qtenon Design: Qtenon is configured as outlined in Table 4. A custom quantum controller is implemented as a RoCC extension, with design code developed in Chisel [2]. The quantum processing element includes PGUs, treated as a black box with an enforced latency of 1000 cycles, approximating realistic operational times [14, 31]. We configure eight such PGUs in our quantum control system for our experiments. The configuration for the quantum controller cache follows Table 2. On the host side, we experiment with two core configurations, Rocket [1] and Boom-Large [40], both set at a frequency of 1 GHz. These cores share the same memory hierarchy, which includes a 16 KB, 4-way set-associative L1 cache for both instructions and data. The L2 cache comprises an eight-banked configuration with 512 KiB capacity and 8-way set-associativity across eight banks, while the main memory consists of a 16 GB DDR3 module divided across four memory banks.

Experimental Methodology: Qtenon is designed as an ASIC chip and is simulated by Firesim [20]. This FPGA-accelerated hardware simulation platform provides cycle-accurate modeling of our design's hardware and software model, including I/O and DRAM [3]. The simulation is run on the Xilinx Alveo U200 Accelerator Card. The RTL hardware implementation frequency is set at 50MHz for Rocket-based Qtenon and 30MHz for Boom-based Qtenon. Figure 10 shows the floorplan of our Qtenon-64 with Rocket core. For the quantum chip input and output, we use simulator data obtained

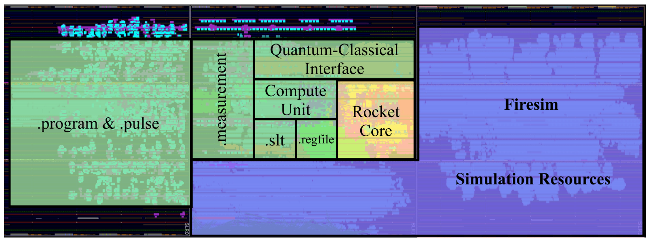


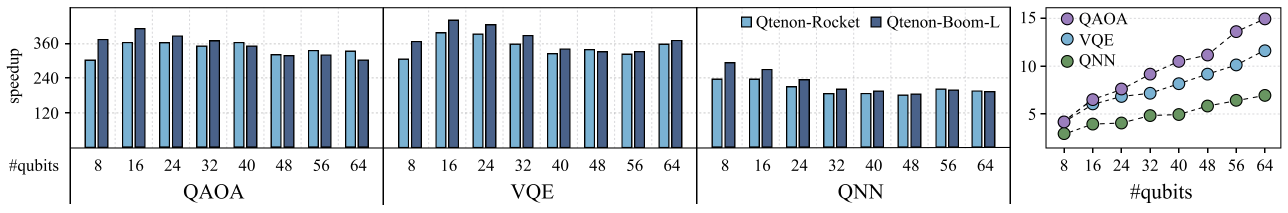
Figure 10: Floorplan of Qtenon-64 with Rocket Chip architecture, simulated on Alveo U200 using FireSim.

from Qiskit. For the software part, we modified the RISC-V GNU Toolchain to support Qtenon's extended ISA.

We use cycle count to measure the performance of Qtenon and is obtained using the `RDCYCLE` [12] control status register [11]. For the quantum execution time, we standardized based on the following assumptions: the gate times for common operations include 20ns for single-qubit gates and 40ns for two-qubit gates. Measurement operations generally require a pulse of 100$ns \sim 2\mu s$, followed by an equivalent duration to process the measurement result [24]. In our experiments, this measurement time is set to 600ns [39]. The time of the whole quantum-classical algorithm can be divided into two parts: quantum execution time and classical execution time. The classical execution time in our experiment can be broken down into the following parts: quantum-host communication time, pulse generation time, and host computation time.

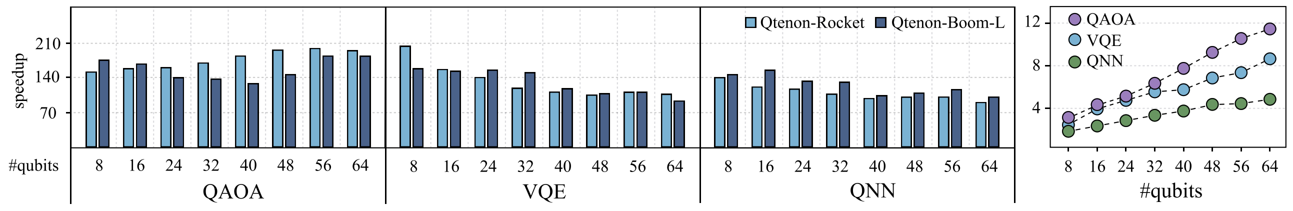
Baseline Configuration: We compare Qtenon against a decoupled hardware system. The host is configured with an Intel i9-14900K CPU and 64 GB of DDR5 RAM. The quantum circuit is generated using Qiskit [17] and compiled into OpenQASM [7]. The host system and the quantum chip controller (an FPGA) are connected using a 100-gigabyte Internet connection with UDP protocol. We omit the overhead of using possible switches and other network devices. The FPGA execution time is considered under optimal conditions and focused solely on pulse generation, which is set to a fixed latency of 1000ns per pulse [14, 31]. The Analog-Digital Interface (ADI) latency is assumed to be a fixed 100ns for each direction [26].

Benchmark: he benchmarks in our evaluation include three variational quantum algorithms (VQAs). 1) QAOA is set to solve the MAX-CUT problem on n_q number of nodes using the standard alternating ansatz with five layers [10]. n_q is the number of qubits. 2) VQE is applied to molecular ground state simulations, where the number of qubits corresponds to the number of molecular spin-orbitals. 3) QNN is implemented through hardware-efficient ansatz with alternating $Ry(\theta)$ and CZ gates in 2 layers. The number of sampling shots for each quantum circuit is set to 500, and the number of iterations is 10. We employ two parameter optimization algorithms for the quantum variational algorithms: the Gradient Descent (GD) method, which uses the parameter shift rule to compute the gradient, and the Simultaneous Perturbation Stochastic Approximation (SPSA). These two methods correspond to different computational scenarios. In the GD method, one parameter is updated at a time, leading to simpler classical post-processing and pulse generation for each quantum-classical iteration, but requiring more communication rounds. In contrast, the SPSA method updates all parameters



(a) Classical execution time speedup (b) End-to-end speedup

Figure 11: The overall performance of Qtenon compared with baseline when running VQA optimized by the GD optimizer.



(a) Classical execution time speedup (b) End-to-end speedup

Figure 12: The overall performance of Qtenon compared with baseline when running VQA optimized by the SPSA optimizer.

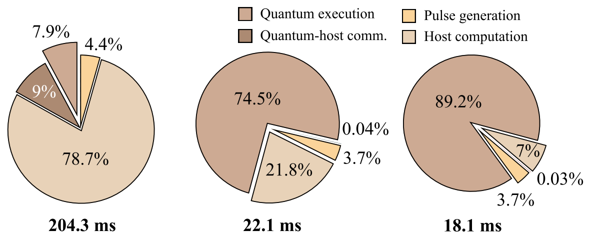
simultaneously, which increases the computational effort for classical post-processing and pulse generation per iteration but reduces the overall number of communication rounds.

7.2 Performance Comparison

We evaluate the performance of Qtenon with two RISC-V core configurations against the decoupled baseline system setup across different qubit configurations, ranging from 8 to 64 qubits. In both parameter optimization algorithms, as the number of qubits increases, the end-to-end execution speed continues to improve compared to the baseline system, as shown in Figure 11 (b) and Figure 12 (b). Specifically, for 64-qubit QAOA, VQE, and QNN algorithms, Qtenon achieves end-to-end speedups of 14.7×, 11.7×, and 6.9× under GD optimization, and 14.9×, 11.5×, and 6.9× under SPSA optimization, respectively.

In terms of classical execution time, the GD optimization algorithm requires more communication rounds, which increase with the number of qubits. Qtenon reduces communication overhead through a specialized quantum controller and shortens parameter optimization time via ISA extensions and incremental compilation, achieving average speedups of 354.0× for QAOA, 375.8× for VQE, and 221.7× for QNN. In contrast, the SPSA optimization method maintains a consistent number of communication and parameter optimization rounds, regardless of the number of qubits, and is significantly lower than the GD method. As a result, the speedup is slightly lower than the GD method, but it still achieves average speedups of 167.1× for QAOA, 131.8× for VQE, and 124.6× for QNN. By co-designing software and hardware to tightly couple the classical and quantum components, Qtenon demonstrates strong adaptability and efficiency across different variational quantum algorithms.

An example of the end-to-end time breakdown is shown in Figure 13. The effective quantum execution time only contributes to 7.9% percent of the total running time in Figure 13 (a). With Qtenon



(a) Baseline (b) Qtenon w/o software (c) Qtenon

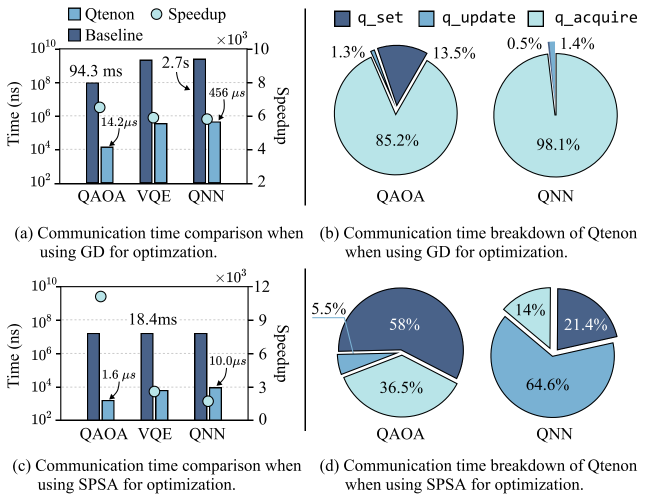
Figure 13: The end-to-end breakdown of 64-qubit VQE optimized by SPSA optimizer.

hardware proposed in Section 5, the total execution time decreases from 204.3*ms* to 22.1*ms*. In particular, the quantum-host communication time is reduced to almost negligible compared to other parts. The most time-consuming part is the host computation, which still contributes to 21.8% percent of the total running time. This time could be further reduced by applying the memory consistency model and instruction scheduling method presented in Section 6. In Figure 13 (c), the quantum time is able to occupy nearly 90% of the total execution time, significantly reducing the classical execution overhead.

7.3 Latency Profiling

We profile the latency of the three classical parts involved in executing 64-qubit variational quantum algorithms: quantum-host communication, pulse generation, and host computation.

Quantum-host Communication: We profile the quantum-host communication time of Qtenon using the Boom core. For the GD optimization method, the number of parameters is positive correlated with the algorithm's communication time. VQE and QNN require more parameters, leading to more frequent communication and thus significantly longer communication times in the baseline system compared to QAOA, as shown in Figure 14 (a). For example,



(a) Communication time comparison when using GD for optimzation.
(b) Communication time breakdown of Qtenon when using GD for optimization.
(c) Communication time comparison when using SPSA for optimization.
(d) Communication time breakdown of Qtenon when using SPSA for optimization.

Figure 14: Analysis of quantum-host communication time.

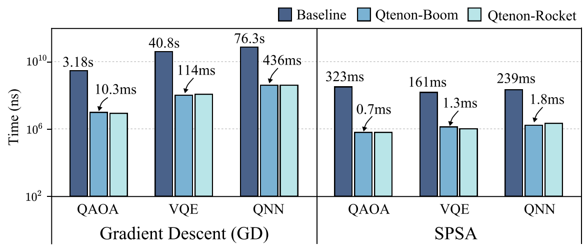
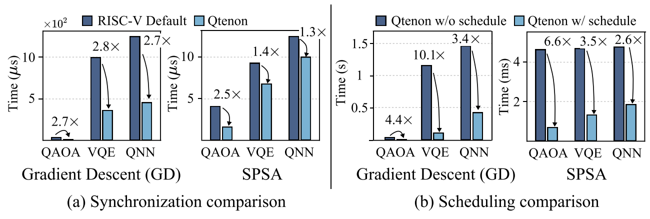


Figure 15: Host execution time comparison.



(a) Synchronization comparison
(b) Scheduling comparison

Figure 16: The software optimization result of Qtenon.

Table 5: Pulse generation speedup and computation requirement reduction.

	QAOA		VQE		QNN	
	Speedup	Reduction	Speedup	Reduction	Speedup	Reduction
GD	204.2×	96.8%	339.0×	98.3%	647.9×	98.9%
SPSA	23.3×	61.3%	13.5×	55.7%	27.8×	72.1%

under ideal assumptions, QNN requires up to 2.7*s* of quantum-host communication time on the baseline system, while QAOA requires 94.3*ms*. Qtenon significantly reduces these times to 456*μs* and 14.2*μs*, achieving speedups of 5921.1× and 6647.2×, respectively. For the SPSA method, the communication time in the baseline system only depends on the number of algorithm iterations, so the communication time is the same for all algorithms, as shown in Figure 14 (c). By using incremental compilation, Qtenon only optimizes the changed parameters. As a result, QAOA requires fewer parameters and, therefore, needs less quantum-host communication time compared to VQE and QNN on the Qtenon system.

To gain further insights into Qtenon's communication characteristics, we break down the communication time into three main categories that correspond to three data communication instructions we defined earlier: `q_set`, `q_update`, and `q_acquire`. In the case of GD optimization, the majority of Qtenon's time is taken by `q_acquire` instruction, given the massive count of transmission time required. Specifically, the `q_acquire` instruction accounts for 85.2% of the communication time for QAOA and 98.1% for QNN, as shown in Figure 14 (b). For SPSA optimization, most of the execution time is spent on the `q_set` and `q_update` instructions, as SPSA optimization requires less data movement. The time breakdown between QAOA and QNN reflects the higher frequency of parameter updates in QNN, which also results in longer communication delays—10*μs* for QNN compared to 1.6*μs* for QAOA.

Pulse Generation: Table 5 compares Qtenon with the baseline system in terms of the pulse generation time. For the GD method, only one parameter is updated at a time, while the rest of the quantum program remains unchanged. Using dynamic incremental compilation, Qtenon can leverage 'quantum locality' to update only the relevant parameters, thereby reducing computational requirements. Specifically, the computation requirement is reduced by 96.8%, 98.3%, and 98.9% for QAOA, VQE, and QNN, respectively. This approach also delivers impressive speedups, with the generation time accelerating by 204.2×, 339.0×, and 647.9× for these algorithms.

On the other hand, the SPSA method optimizes all parameters simultaneously, which has less potential for reductions in computation. Nevertheless, Qtenon still achieves reductions in computational demand of 61.3%, 55.7%, and 72.1% for QAOA, VQE, and QNN, respectively. Due to runtime incremental compilation, the need to recompile the entire program is eliminated, resulting in speedups of 23.3×, 13.5×, and 27.8× compared to the baseline hardware system, respectively.

Host Computation: The host computation time is profiled under two core configurations, as shown in Figure 15. The host computation time for the two cores is almost identical. Specifically, Qtenon with Boom core achieves 308.7×, 357.9×, 175.0× of speedup for QAOA, VQE, and QNN, respectively, using the GD method, and 461.4×, 123.8×, and 132.8× for QAOA, VQE, and QNN, respectively, using the SPSA method. The performance gain mainly comes from Qtenon's ability to support dynamic incremental compilation and quantum-host scheduling method supported by our fine-grained synchronization technique.

7.4 Software Optimization

In this part, we compare the effectiveness of two software designs: memory consistency policy (Section 6.2) and instruction scheduling (Section 6.3) when running 64-qubit VQA.

Memory Consistency: Figure 16 (a) compares the quantum-host transmission time using two synchronization methods: the RISC-V default approach, which relies on the `FENCE` operation for memory consistency, and Qtenon, which employs fine-grained synchronization. When using the GD optimization method, most of the

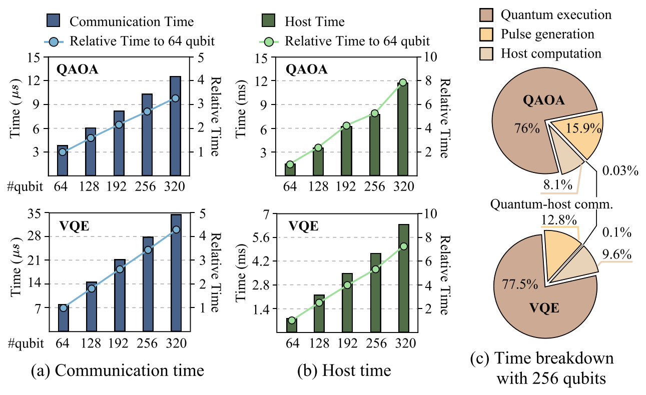


Figure 17: The scalability test of Qtenon.

communication time is spent retrieving data from the quantum controller cache. Our memory consistency model significantly reduces this transmission overhead through fine-grained synchronization, achieving a more significant communication time reduction in VQE and QNN compared to the SPSA optimization method. For QAOA, since the transmission costs of both optimization methods are closer, the resulting speedups are also more comparable, with improvements of 2.7× and 2.5×, respectively.

Instruction Scheduling: Figure 16 (b) compares the host computation time of our instruction scheduling method. Our instruction scheduling method achieves considerable performance gains for VQAs under two optimization methods. For the GD method, our scheduling method achieves 4.4×, 10.1×, and 3.4× speedup for QAOA, VQE, and QNN, respectively. For the SPSA method, our scheduling method achieves 6.6×, 3.5×, and 2.6× speedup, respectively. This improvement comes from batching the measurement results, which maximizes bus bandwidth usage and reduces data transfer time.

7.5 Scalability

On the software side, our ISA design and optimization policies support qubit expansion as long as sufficient QAddress space is available. The address space of the QAddress is 2^{39}. However, two hardware factors may limit stability. First, the cache size required for the quantum controller increases linearly with the number of qubits. For instance, controlling 256 qubits requires a cache size of 22.63MB. Second, the number of available pins on the chip imposes a limitation. Each qubit requires two DACs, so the qubit count can only be increased if the chip provides enough pins for ADC and DAC connections, as well as sufficient SRAM for the cache design.

Figure 17 illustrates the scalability of Qtenon when executing QAOA and VQE algorithms across an increasing number of qubits, assuming sufficient cache and output connections. Figure 17(a) depicts the quantum-host communication time, which scales nearly linearly with qubit number. At 320 qubits, VQE with SPSA optimization requires only $34.4\mu s$, while QAOA with SPSA requires $12.5\mu s$. The higher communication time for VQE stems from its greater number of parameter updates compared to QAOA. Figure 17(b) presents the classical computation time, which also grows almost linearly. Here, QAOA and VQE require 11.8 ms and 6.4 ms, respectively, for 320 qubits. Figure 17(c) details the time breakdown for 256 qubits, revealing that quantum execution dominates the runtime. Pulse generation and host computation account for a growing but smaller fraction, while quantum-host communication remains minimal. Notably, pulse generation and host computation times could be further reduced by integrating additional PGUs in the quantum controller and leveraging more RISC-V processor cores. These results demonstrate that Qtenon's design retains efficiency and scalability even as hardware systems expand.

8 Related Work

On the hardware side, FPGA-based systems are widely used to control and measure superconducting quantum processing units (QPUs) [14, 26, 37, 38]. These systems typically consist of three components: room-temperature electronics hardware, FPGA control logic, and corresponding software, often implemented on top of a quantum ISA. The room-temperature electronics hardware manages signal conversion between the digital and analog domains through three key modules: the Analog-to-Digital Converter (ADC)/Digital-to-Analog Converter (DAC) for generating and detecting intermediate frequency (IF) signals, the RF mixing module for converting signals between IF and target frequencies, and the Local Oscillator (LO) generation module for producing low-noise LO signals. The FPGA acts as a gateway, translating high-level quantum circuit definitions into precise control pulses that are sent to the electronics hardware for qubit manipulation.

Various software approaches have been proposed to control current quantum hardware for noisy intermediate-scale quantum (NISQ) applications [5, 13, 15, 37]. For example, eQASM [13], HiSEP-Q [15], and QubiC [38] introduce specialized ISAs tailored to their respective hardware architectures. Additionally, efforts such as QUASAR [5] aim to leverage the computational capabilities of RISC-V by extending it with quantum-specific ISAs. For fault-tolerant quantum computation (FTQC) [28, 33, 35] where quantum programs operate on encoded logical qubits, each composed of thousands of physical qubits. Some dedicated ISAs for those applications have also been proposed [39].

9 Conclusion

Accelerating hybrid quantum-classical algorithms is of vital importance for quantum systems. Previous decoupled systems lack both hardware support for low-latency communication and software support for fine-grained optimization. In this paper, we propose Qtenon, a tightly coupled system for efficient hybrid quantum-classical algorithm acceleration. For hardware design, we propose a unified memory hierarchy, an efficient quantum controller, and a multi-stage processing pipeline. For software design, we propose ISA for data communication and computation, which enables fine-grained synchronization and efficient scheduling. In evaluation, we achieve up to 14.9× end-to-end speedup compared to state-of-the-art work.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.62472374) and the Zhejiang Provincial Natural Science Foundation of China under Grant (No.LR25F020002).

References

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. doi:10.1145/2228360.2228584

[3] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 330–339. doi:10.1145/3289602.3293894

[4] Kostas Blekos, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui-Hao Li, Komal Pandya, and Alessandro Summer. 2024. A review on Quantum Approximate Optimization Algorithm and its variants. *Physics Reports* 1068 (June 2024), 1–66. doi:10.1016/j.physrep.2024.03.002

[5] Anastasiia Butko, George Michelogiannakis, Samuel Williams, Costin Iancu, David Donofrio, John Shalf, Jonathan Carter, and Irfan Siddiqi. 2020. Understanding Quantum Control Processor Capabilities and Limitations through Circuit Characterization. In *2020 International Conference on Rebooting Computing (ICRC)*. IEEE Computer Society, Los Alamitos, CA, USA, 66–75. doi:10.1109/ICRC2020.2020.00011

[6] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. 2021. Variational quantum algorithms. *Nature Reviews Physics* 3, 9 (Aug. 2021), 625–644. doi:10.1038/s42254-021-00348-9

[7] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (Sept. 2022), 50 pages. doi:10.1145/3505636

[8] Siddharth Dangwal, Gokul Subramanian Ravi, Poulami Das, Kaitlin N. Smith, Jonathan Mark Baker, and Frederic T. Chong. 2024. VarSaw: Application-tailored Measurement Error Mitigation for Variational Quantum Algorithms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 362–377. doi:10.1145/3623278.3624764

[9] Suguru Endo, Jinzhao Sun, Ying Li, Simon C Benjamin, and Xiao Yuan. 2020. Variational quantum simulation of general processes. *Physical Review Letters* 125, 1 (2020), 010501.

[10] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph] https://arxiv.org/abs/1411.4028

[11] Five EmbedDev. 2023. Control and Status Register (CSR) Instructions. https://five-embeddev.com/riscv-isa-manual/latest/csr.html Accessed: 2024-11-11.

[12] Five EmbedDev. 2023. Counters. https://five-embeddev.com/riscv-isa-manual/latest/counters.html Accessed: 2024-11-11.

[13] X. Fu, L. Riesebos, M. A. Rol, Jeroen van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2019. eQASM: An Executable Quantum Instruction Set Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 224–237. doi:10.1109/HPCA.2019.00040

[14] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2017. An experimental microarchitecture for a superconducting quantum processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 813–825. doi:10.1145/3123939.3123952

[15] Xiaorang Guo, Kun Qin, and Martin Schulz. 2023. HiSEP-Q: A Highly Scalable and Efficient Quantum Control Processor for Superconducting Qubits. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 86–93. doi:10.1109/ICCD58817.2023.00023

[16] He-Liang Huang, Xiao-Yue Xu, Chu Guo, Guojing Tian, Shi-Jie Wei, Xiaoming Sun, Wan-Su Bao, and Gui-Lu Long. 2023. Near-term quantum computing techniques: Variational quantum algorithms, error mitigation, circuit compilation, benchmarking and classical simulation. *Science China Physics, Mechanics & Astronomy* 66, 5 (2023), 250302.

[17] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. arXiv:2405.08810 [quant-ph] https://arxiv.org/abs/2405.08810

[18] Mingrui Jiang, Keyi Shan, Chengping He, and Can Li. 2023. Efficient combinatorial optimization by quantum-inspired parallel annealing in analogue memristor crossbar. *Nature Communications* 14, 1 (2023), 5927.

[19] Yuwei Jin, Zirui Li, Fei Hua, Tianyi Hao, Huiyang Zhou, Yipeng Huang, and Eddy Z Zhang. 2024. Tetris: A Compilation Framework for VQA Applications in Quantum Computing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 277–292.

[20] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. doi:10.1109/ISCA.2018.00014

[21] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. 2019. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews* 6, 2 (June 2019). doi:10.1063/1.5089550

[22] Vladimir Kremenetski, Anuj Apte, Tad Hogg, Stuart Hadfield, and Norm M. Tubman. 2023. Quantum Alternating Operator Ansatz (QAOA) beyond low depth with gradually changing unitaries. arXiv:2305.04455 [quant-ph] https://arxiv.org/abs/2305.04455

[23] Zhiding Liang, Zhixin Song, Jinglei Cheng, Zichang He, Ji Liu, Hanrui Wang, Ruiyang Qin, Yiru Wang, Song Han, Xuehai Qian, et al. 2023. Hybrid gate-pulse model for variational quantum algorithms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[24] Satvik Maurya and Swamit Tannu. 2022. COMPAQT: Compressed Waveform Memory Architecture for Scalable Qubit Control. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. doi:10.1109/micro56248.2022.00076

[25] Honghui Shang, Yi Fan, Li Shen, Chu Guo, Jie Liu, Xiaohui Duan, Fang Li, and Zhenyu Li. 2023. Towards practical and massively parallel quantum computing emulation for quantum chemistry. *npj Quantum Information* 9, 1 (2023), 33.

[26] Leandro Stefanazzi, Kenneth Treptow, Neal Wilcer, Chris Stoughton, Collin Bradford, Sho Uemura, Silvia Zorzetti, Salvatore Montella, Gustavo Cancelo, Sara Sussman, Andrew Houck, Shefali Saxena, Horacio Arnaldi, Ankur Agrawal, Helin Zhang, Chunyang Ding, and David I. Schuster. 2022. The QICK (Quantum Instrumentation Control Kit): Readout and control for qubits and detectors. *Review of Scientific Instruments* 93, 4 (April 2022). doi:10.1063/5.0076249

[27] Samuel Stein, Nathan Wiebe, Yufei Ding, Peng Bo, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. 2022. EQC: ensembled quantum computing for variational quantum algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 59–71. doi:10.1145/3470496.3527434

[28] Daniel Bochen Tan, Murphy Yuezhen Niu, and Craig Gidney. 2024. A SAT Scalpel for Lattice Surgery: Representation and Synthesis of Subroutines for Surface-Code Fault-Tolerant Quantum Computing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 325–339. doi:10.1109/isca59077.2024.00032

[29] Siwei Tan, Mingqian Yu, Andre Python, Yongheng Shang, Tingting Li, Liqiang Lu, and Jianwei Yin. 2023. HyQSAT: A Hybrid Approach for 3-SAT Problems by Integrating Quantum Annealer with CDCL. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 731–744. doi:10.1109/HPCA56546.2023.10071022

[30] Wesley W Terpstra. 2017. TileLink: A Free And Open Source, High Performance Scalable Cache Coherent Fabric Designed for RISC-V. In *Proc. 7th RISC-V Workshop*.

[31] Wuwei Tian, Xinghui Jia, Siwei Tan, Zixuan Song, Liqiang Lu, and Jianwei Yin. 2023. QPulseLib: Accelerating the Pulse Generation of Quantum Circuit with Reusable Patterns. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 01–09. doi:10.1109/ICCAD57390.2023.10323711

[32] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. 2022. The Variational Quantum Eigensolver: A review of methods and best practices. *Physics Reports* 986 (Nov. 2022), 1–128. doi:10.1016/j.physrep.2022.08.003

[33] Suhas Vittal, Poulami Das, and Moinuddin Qureshi. 2023. Astrea: Accurate Quantum Error-Decoding via Practical Minimum-Weight Perfect-Matching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 16 pages. doi:10.1145/3579371.3589037

[34] Meng Wang, Poulami Das, and Prashant J. Nair. 2024. Qoncord: A Multi-Device Job Scheduling Framework for Variational Quantum Algorithms. arXiv:2409.12432 [quant-ph] https://arxiv.org/abs/2409.12432

[35] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yufei Ding, and Yuan Xie. 2022. A synthesis framework for stitching surface code with superconducting quantum devices. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 337–350. doi:10.1145/3470496.3527381

[36] Pengcheng Xu and Yun Liang. 2020. Automatic Code Generation for Rocket Chip RoCC Accelerators. In *Proceedings of the Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[37] Yilun Xu, Gang Huang, Jan Balewski, Ravi Naik, Alexis Morvan, Bradley Mitchell, Kasra Nowrouzi, David I Santiago, and Irfan Siddiqi. 2021. QubiC: An open-source FPGA-based control and measurement system for superconducting quantum information processors. *IEEE Transactions on Quantum Engineering* 2 (2021), 1–11.

[38] Yilun Xu, Gang Huang, Neelay Fruitwala, Abhi Rajagopala, Ravi K. Naik, Kasra Nowrouzi, David I. Santiago, and Irfan Siddiqi. 2023. QubiC 2.0: An Extensible Open-Source Qubit Control System Capable of Mid-Circuit Measurement and Feed-Forward. arXiv:2309.10333 [quant-ph] https://arxiv.org/abs/2309.10333

[39] Fang Zhang, Xing Zhu, Rui Chao, Cupjin Huang, Linghang Kong, Guoyang Chen, Dawei Ding, Haishan Feng, Yihuai Gao, Xiaotong Ni, Liwei Qiu, Zhe Wei, Yueming Yang, Yang Zhao, Yaoyun Shi, Weifeng Zhang, Peng Zhou, and Jianxin Chen. 2023. A Classical Architecture for Digital Quantum Computers. *ACM Transactions on Quantum Computing* 5, 1, Article 3 (Dec. 2023), 24 pages. doi:10.1145/3626199

[40] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).

[41] Renxin Zhao and Shi Wang. 2021. A review of Quantum Neural Networks: Methods, Models, Dilemma. arXiv:2109.01840 [cs.ET] https://arxiv.org/abs/2109.01840