A Classical Architecture for Digital Quantum Computers

FANG ZHANG, Quantum Laboratory, DAMO Academy, USA
XING ZHU, Quantum Laboratory, DAMO Academy, P.R. China
RUI CHAO and CUPJIN HUANG, Quantum Laboratory, DAMO Academy, USA
LINGHANG KONG, Quantum Laboratory, DAMO Academy, P.R. China
GUOYANG CHEN, Alibaba Cloud Intelligence, Alibaba Group, USA
DAWEI DING, Quantum Laboratory, DAMO Academy, USA
HAISHAN FENG, Alibaba Cloud Intelligence, Alibaba Group, P.R. China
YIHUAI GAO, XIAOTONG NI, and LIWEI QIU, Quantum Laboratory, DAMO Academy, P.R. China
ZHE WEI, YUEMING YANG, and YANG ZHAO, Alibaba Cloud Intelligence, Alibaba Group, P.R. China
YAOYUN SHI, Quantum Laboratory, DAMO Academy, USA
WEIFENG ZHANG and PENG ZHOU, Alibaba Cloud Intelligence, Alibaba Group, USA
JIANXIN CHEN, Quantum Laboratory, DAMO Academy, USA

Scaling bottlenecks the making of digital quantum computers, posing challenges from both the quantum and the classical components. We present a classical architecture to cope with a comprehensive list of the latter challenges *all at once* and implement it fully in an end-to-end system by integrating a multi-core RISC-V CPU with our in-house control electronics.

Our architecture enables scalable, high-precision control of large quantum processors and accommodates evolving requirements of quantum hardware. A central feature is a microarchitecture executing quantum operations in parallel on arbitrary predefined qubit groups. Another key feature is a reconfigurable quantum instruction set that supports easy qubit re-grouping and instructions extensions.

As a demonstration, we implement the surface code quantum computing workflow. Our design, for the first time, reduces instruction issuing and transmission costs to constants, which do not scale with the number of qubits, without adding any overheads in decoding or dispatching.

Our system uses a dedicated general-purpose CPU for both qubit control and classical computation, including syndrome decoding. Implementing recent theoretical proposals as decoding firmware that parallelizes general inner decoders, we can achieve unprecedented decoding capabilities of up to distances 47 and 67

F. Zhang and X. Zhu contributed equally to this research.

This work was supported by Alibaba Group through Alibaba Research Intern Program and conducted when Yihuai Gao and Liwei Qiu were research interns at Alibaba Group.

Authors' addresses: F. Zhang, R. Chao, C. Huang, D. Ding, Y. Shi, and J. Chen, Quantum Laboratory, DAMO Academy, 205 108th Ave. NE, Suite 400, Bellevue WA, 98004, USA; e-mails: {y.shi, liangjian.cjx}@alibaba-inc.com; X. Zhu, L.Kong, Y. Gao, X. Ni, and L. Qiu, Quantum Laboratory, DAMO Academy, 8 Lixian Road, Yuhang District, Hangzhou, P. R. China; G. Chen, W. Zhang and P. Zhou, Alibaba Cloud Intelligence, Alibaba Group, 525 Almanor Ave, Suite 400, Sunnyvale, CA 94085, USA; H. Feng, Z. Wei, Y. Yang, and Y. Zhao, Alibaba Cloud Intelligence, Alibaba Group, 969 Wenyi West Road, Yuhang District, Hangzhou, P.R.China.

2643-6817/2023/12-ART3
https://doi.org/10.1145/3626199

with the currently available systems-on-chips for physical error rate $p = 0.001$ and $p = 0.0001$, respectively, all in just 1 μs.

CCS Concepts: • **Computer systems organization → Quantum computing**; • **Hardware → Quantum error correction and fault tolerance;**

Additional Key Words and Phrases: fault-tolerant quantum computing, quantum computer architecture, parallel decoding

ACM Reference format:
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 Trans. Quantum Comput.* 5, 1, Article 3 (December 2023), 24 pages.
https://doi.org/10.1145/3626199

1 MOTIVATIONS AND SUMMARY OF RESULTS

As quantum computers become more sophisticated [2, 3, 16, 41], their demands on the classical control multiply accordingly. In this section, we analyze those challenges, then summarize our solutions. We confine this work to the superconducting-circuit platform, the focus of our team. We first review the setup as the starting point for our discussion.

Superconducting system setup. Figure 1 illustrates a standard setup for a superconducting quantum computing system. Quantum information is stored physically on superconducting qubits on a quantum chip. To enable superconductivity and to suppress thermal noise, the quantum chip is cooled cryogenically inside a dilution refrigerator. To enable state evolution and measurement, the superconducting circuits are coupled to drive lines connecting to room-temperature control electronics, which in turn comprise **arbitrary waveform generators (AWGs)**, digitizers, IQ mixers, and so on. The control electronics are further driven by a general-purpose processing unit such as a PC.

Quantum computing workflows. Applications are the end goals of quantum computers, thus the origins of their design requirements. Most applications belong to one of the two main paradigms: **noisy intermediate-scale quantum (NISQ)** applications and **fault-tolerant quantum computations (FTQC)**. NISQ applications operate on noisy, unprotected physical qubits, limited in scale and in precision. FTQCs operate on encoded logical qubits, each consisting of (likely) thousands of physical qubits. The logical qubits have drastically reduced sensitivity to physical-level noises, allowing computations of an arbitrary length and scale, thus consequently the ultimate quantum advantages.

In NISQ, the PC sends the quantum circuit to the control electronics. The latter parse the circuit into microwave waveforms, play them synchronously on the drive lines to the qubits, process the measurement responses from the quantum chip, and finally, return the measurement results to the PC. The PC can then perform a classical post-processing, before possibly starting the next round of quantum circuit execution.

FTQC differs from NISQ in several key aspects. First, it requires constant extraction and decoding of the classical error syndromes, which are constantly churned out by the faulty quantum circuits. The decoding, in turn, requires real-time and intense classical computation. Second, while NISQ executes a static circuit, FTQC requires dynamic quantum circuit generation according to the decoding results.

Both NISQ and FTQC demand seamless coordination and collaboration between classical and quantum computational resources, which, in turn, require a co-design of classical and quantum

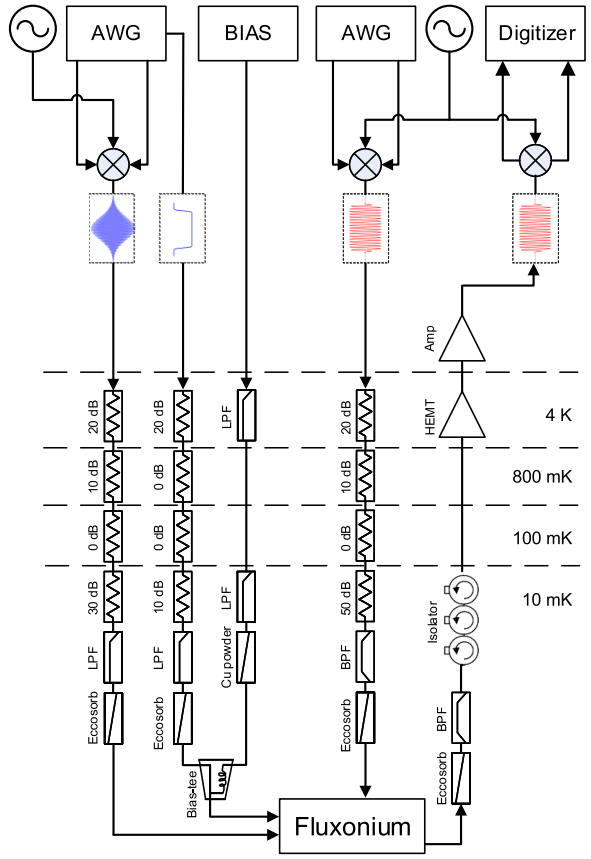


Fig. 1. An experimental setup for qubit driving and measurement. The dilution refrigerator is depicted as the cyan box, with different temperature zones separated by dashed lines. The PC driving the control electronics is omitted.

architecture. We focus on the design and implementation of classical architectures. We analyze the challenges from two perspectives: scaling-up and actual implementation of a complete system.

Challenges in scaling up the classical architecture. Maintaining a high precision in the control of quantum hardware is the primary requirement here, as it would directly affect fidelities of the quantum operations involved. Failing it would result in performance loss that eventually needs to be compensated by the quantum hardware, compounding the difficulty for the latter. Specifically, for superconducting qubits, microwave pulses played on different AWG channels and the sampling window of digitizer channels need to be synchronized at the picosecond level to ensure high-fidelity physical operations [42].

A second set of challenges are caused by the large number of instructions—the efficiency of their issuance, transmission, and execution as the number of qubits grows. These problems have been recognized by several authors [6, 10, 18], and we refer to them together as "instruction stresses." In FTQC, dynamic quantum instructions need to be issued and transmitted fast enough to keep in pace with the rapid quantum execution, posing a hard constraint on the classical architecture. This may not be required for NISQ but is still desirable, as it would decrease the total running time.

Syndrome decoding is yet another major bottleneck to FTQC classical architecture [36]. For surface code schemes on present-day superconducting qubits, one round of syndrome extraction

takes roughly 1μs [1] and generates $O(d^2)$ bits of syndrome information in parallel, for d being the code distance. Against this increasing syndrome size, the decoding algorithm needs to keep up with the constant syndrome extraction time and to avoid exponential syndrome backlog.

Multiple decoding schemes were proposed to tackle this problem [11, 12, 17, 21, 37] but can only handle code distances no more than 11, even with specialized hardware. Recently, a new parallel decoding scheme was proposed independently in References [33, 34]. An implementation of the scheme achieved a code distance of 11 for physical error rate $p = 0.4\%$ [31].

A fourth set of challenges originates from a desirable feature that we call "permissiveness," which means the ability to accommodate evolving requirements by other components of a quantum computer. Our field experiences indicate that implementing a complete classical architecture is time-consuming and labor-intensive. However, in this early stage of quantum computing, changes are rapid in applications, hardware characteristics, and error-correction schemes. Thus, a stable yet permissive classical architecture would be cost-effective in the classical-quantum co-design process.

Challenges for implementing a complete system. Many researchers have proposed innovative solutions addressing one or a few of the above problems. Ultimately, a single system needs to be built for a real quantum computer. Building such a complete system has the additional challenge of balancing competing objectives with currently available and compatible technologies. To our knowledge, there has not been a system implementation addressing all the aforementioned challenges in scalability.

Our contributions. We present and implement a classical architecture to address all the scalability challenges mentioned above in one single system.

(1) Our system provides *high-fidelity qubit control* by interconnecting one-chassis PXIe systems through a star-like hierarchy with high-density connectors. This design synchronizes, with high accuracy, pulses from different control electronics, enabling precise qubit control even as the system size increases, thereby maintaining high-fidelity. This conclusion is supported by the extensive testing of the channel-to-channel and phase jitter of the AWG outputs.
(2) To address *instruction stresses*, we develop an efficient "quantum instruction pipeline" that combines **Single-Instruction-Multiple-Data (SIMD)** with a broadcasting mechanism. This pipeline enables parallel application of the same type of gate on arbitrarily-sized qubit groups. The operation types and the qubit groups of application-specific instructions can be easily configured either prior to the execution of the quantum program or during runtime. Additionally, the costs across instruction issuing, transmission, dispatching, and execution do not scale with the size of each qubit group.
(3) Our system's *permissiveness* is achieved through a combination of features, including a reconfigurable quantum instruction set, **Memory-Mapped IO (MMIO)** in the microarchitecture, and a portable general-purpose CPU. The instruction set and the underlying MMIO-based microarchitecture facilitate the incorporation of new quantum instructions.
(4) We achieve unprecedented performances on *decoding throughput* for surface codes, a mainstream approach that our architecture and the implemented system are nevertheless not restricted to. More specifically, we implement the surface code and a parallel decoding firmware based on the recent theoretical proposals [31, 34] in a dedicated CPU and benchmark its performance on a development board. By leveraging our in-house Union-Find and PyMatching 2 [20] as inner decoders, we can decode up to distances 13 and 31 on SiFive P650 [32] or T-head C910 [9], or 41 and 67 with ET-SoC-1 [15], all in just 1 microsecond for physical error rate $p = 0.0001$.

The proposed classical architecture is implemented fully in an end-to-end quantum computer system by integrating a multi-core, vectorized RISC-V CPU with our in-house control electronics.

Our system also features an MLIR-based compiler to support the proposed reconfigurable quantum instruction set and enables optimization possibilities on various abstraction layers. We highlight the following features among the many of our implemented system.

(5) *Low communication latency* A key metric for FTQC is the "decoding latency," i.e., the time between the completions of syndrome generation and decoding. Such latency consists of the decoding algorithm latency and the communication latency between the control system and the quantum device. In our design, we use on-board communication to reduce the latter. This design also enables other capabilities where latency plays a critical role, such as on-the-fly calibration [19, 26, 29] and just-in-time compilation [39, 40].

(6) *Load balancing through multi-core CPU* The bottleneck in classical computation is not always syndrome decoding and can vary during the computational process. To accommodate different scenarios, we use a dedicated multi-core CPU in our system, allowing dynamic allocation of cores to syndrome decoding, qubit control, or other computation-heavy tasks. This design allows us to achieve optimal performance while avoiding the unnecessary complexities and cost of using specialized hardware for syndrome decoding.

Comparison with previous work.

To the best of our knowledge, our architecture proposal and the resulting actual implementation represent the first attempt to address, in a single system, the above comprehensive list of scaling challenges for the classical architecture.

Instruction stresses have been known for long, with various mitigating approaches proposed [6, 10, 18]. Those include using SIMD and **Very-Long-Instruction-Word (VLIW)** to reduce the instruction issuance rate [18] and multiprocessors to increase quantum operation and circuit-level parallelism [43]. However, those methods provide only constant-factor improvements and are insufficient to cope with the increasing overhead that scales with the code distance in surface code quantum computing.

The QuEST proposal [35] addresses the instruction bandwidth problem by employing dedicated programmable micro-code engines. While it shows promise for enabling real-time instruction issuing, it crucially relies on an assumption from the underlying primeline microarchitecture [22]: that all qubits driven at a given time must share the same frequency.

This may hold for some quantum computing platforms, such as cold atoms or trapped ions, but not for superconducting qubits, where frequency differences are likely inevitable and sometimes a design preference. Furthermore, the absence of scaling analysis makes it unclear how the frequency requirement would affect the performance in an actual implementation.

Syndrome decoding has been another well-known concern in the quantum computing community for over a decade [36], with proposals ranging from efficient algorithms to specific microarchitectures [11, 13, 14, 20]. Before our work, it remained an open problem if a general-purpose CPU with on-board communication to the control electronics would be sufficient to provide the required decoding throughput. We answer this question affirmatively for the first time by combining the recent parallel decoding schemes [31, 34] with an efficient in-house implementation for the Union-Find decoder and a recent implementation of the **Minimum Weight Perfect Matching (MWPM)** algorithm [20].

2 ARCHITECTURE DESIGN AND SYSTEM IMPLEMENTATION

2.1 Architecture Design

See Figure 2 for a block diagram of our system design. The MCU contains a dedicated CPU. Besides controlling the qubits via the electronics driver, the CPU can also execute classical tasks offloaded from the host PC, using dedicated cores labeled the **classical computing unit (CCU)**. Such tasks

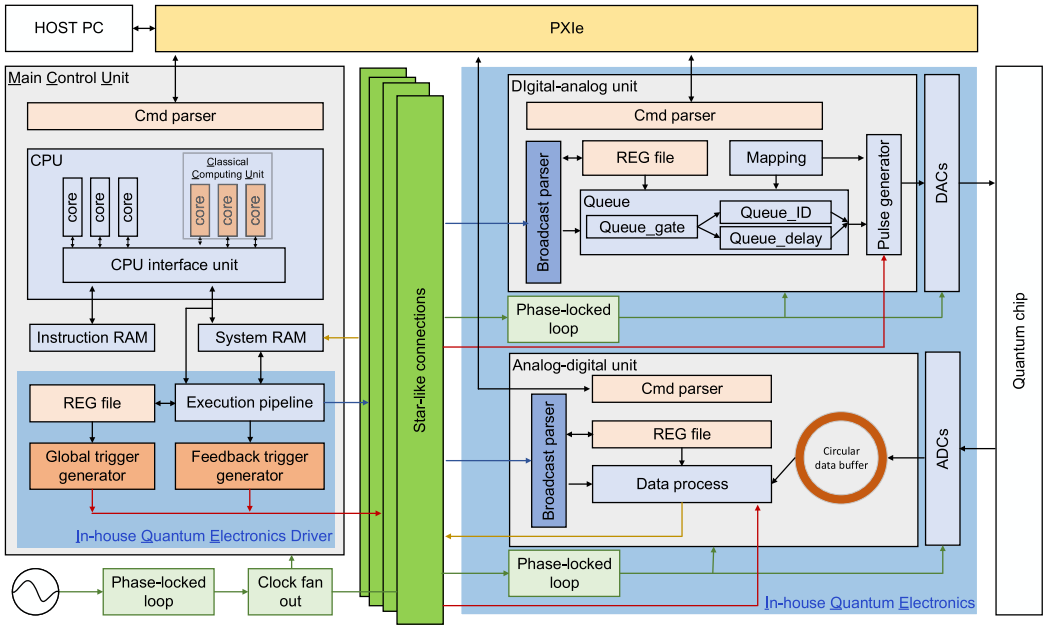


Fig. 2. Block diagram of the proposed classical architecture. The architecture consists of a host PC, a main control unit (MCU), control electronics (In-house Quantum Electronics). The quantum chip is connected with the control electronics via drive lines, while the host PC, the MCU, and the electronics are jointly connected via PXIe. Additionally, the MCU connects with all electronics via a star-like connection. A dedicated unit in the MCU is responsible for driving the control electronics. The MCU is equipped with a portable CPU, and a portion of it, called the classical computing unit, is allocated for runtime computation-heavy tasks such as syndrome decoding. In our implementation, the digital-analog and analog-digital units are made in-house and are called in-house quantum electronics (IQE), and their corresponding driver unit in the MCU is called the IQE driver. Please note that all other modules can be configured via the command parser; however, we have omitted the corresponding arrows in the diagram for the sake of simplicity.

naturally arise from logical quantum program execution, dynamic calibration, just-in-time compilation, and so on. The offloading greatly shortens the communication latency with the QPU.

A quantum program generally comprises both quantum and classical components that collaborate to solve a problem. In Section 4, we will introduce our front-end language and the corresponding compilation support. However, the design and workflow of our system are not restricted to specific quantum programming languages. When a quantum program is executed on a host PC, the quantum subroutines and, depending on the implementation, potentially some classical subroutines will be sent to the MCU. The MCU then processes these quantum or quantum-classical hybrid tasks by issuing both classical and quantum instructions. Classical instructions are carried out on the dedicated CPU for classical control and computations, while quantum instructions are executed through requests to our **in-house quantum electronics (IQE)** driver. The IQE driver dispatches corresponding "IQE instructions," or "commands," to IQE, which, in turn, drives the quantum processor. At the CPU level, the "quantum instructions" are implemented as pseudo-instructions that expand to MMIO load/store instructions. These MMIO instructions interact with a special memory region, and the electronics driver decodes them and dispatches "electronics-level instructions," which will be explained shortly, through broadcasting for communication with the control electronics.

The electronics-level instructions specify the pulse sequences and their corresponding timing information to the control electronics. The latter parse the instructions and feed the pulse sequence information into a local queue. The pulse sequence is not played until a special "trigger" signal arrives at the control electronics, which then plays the pulse sequence through its ports and empties the queue, waiting for the next round of pulse instructions.

We use various "instruction" terminologies. For clarity, Figure 4 exhibits a taxonomy, with more details in the main text.

In addition to the aforementioned general setup, a key feature of our architecture is a *quantum instruction pipeline* that naturally supports a large number of parallel repetition of a same gate and allows for easy reconfiguration. This is enabled jointly by several components, which we elaborate below.

Reconfigurable quantum instruction set. Exploiting MMIO's flexibility, our modular quantum instruction set comprises "pulse-level instructions" for qubit control and calibration and "gate-level instructions" for quantum circuit execution. By having both levels available, it allows for flexibility in implementing quantum algorithms and calibrating quantum devices, similar to other systems [6, 18, 43]. We distinctly exploit what we call the *brickwork structure* found in typical quantum circuits: There is a small number of single-layer sub-circuit of the form $\bigotimes_i G^{S_i}$, for some partition $\{S_i\}_i$ of either the whole set or a large subset of the qubits into equal-sized subsets, and an identical gate G acting on each subset. We allocate different MMIO addresses for the partition identifiers and specify the gate type via the message written to the address. Decoding and dispatching of the instruction are left to the underlying microarchitecture. This allows a lightweight specification of application-specific instructions on user-defined qubit partitions, which, in turn, significantly alleviates the cost of instruction issuing and data transmission.

Instruction dispatching via broadcasting. Some designs may prioritize certain aspects of instruction processing at the expense of others. For instance, adding complex instructions to reduce the instruction issuing rate can lead to more complex decoding and dispatching. However, our microarchitecture support does not come with any hidden costs. This means that we have successfully reduced costs at every stage of the instruction processing pipeline. When dispatching a single gate instruction to multiple control electronics, one-to-one communication would scale the cost linearly with the number of control electronics, impeding scalability. We avoid this problem by exploiting the few-distinct-partition property of the brickwork structure through the built-in broadcasting mechanism of the star-like connection.

Each signal transmitted from the electronics driver broadcasts automatically through the star-like connection, thus each IQE instruction is sent to a collection of control electronics simultaneously, regardless of if a control electronic is meant to be involved in the instruction. To utilize this, each electronic device holds a "partition mask" specifying which partitions it is in. Each IQE instruction broadcast from the electronics driver comes with a partition identifier. Upon receiving an instruction, each electronic device checks whether the partition identifier of the broadcast instruction matches one of the partition identifiers in its partition mask. If so, then it proceeds to process the instruction and ignores it otherwise.

The MCU, thus, can issue at once a same instruction to all devices with a common partition identifier, realizing microarchitecture-level SIMD. The partition masks are stored in local **registration entry (REG)** files on each electronic device. They are easily reconfigurable, either using PXIe between runs or in real-time via the same star-like connection.

Instruction decoding. To decode a quantum instruction received from the CPU, the electronics driver extracts the electronics-level instruction type determined by the value written through MMIO and appends it with the partition identifier determined by the MMIO address. Both mappings are stored in a local REG file that can be reconfigured if necessary. The assembled

electronics-level instruction is then dispatched through the broadcasting system mentioned before.

This microarchitecture does not incur extra overhead on either decoding or dispatching when the partition size increases (as in the case of more qubits).

Apart from the above quantum instruction pipeline problems, we highlight some design choices that address scaling.

Pulse synchronization via triggers. All of the ADCs and the DACs are driven in the same clock domain through a phase-locked loop and a star-like connection, with one rubidium oscillator used as the system root clock. Our design further synchronizes pulses on different control electronics via a dedicated trigger mechanism. The pulses are not played through DACs immediately upon processing of the electronics-level instructions, but rather are stored in a local queue. When a trigger instruction is issued from the MCU, the trigger signal arrives at each control electronic device at the same time, guaranteeing pulse-level synchronization. For further information on IQE, please refer to Reference [42].

Portable, tightly integrated but loosely coupled dedicated CPU. Unlike previous schemes [6, 10, 18, 43] that handle the communication of the control electronics and the MCU by new and dedicated CPU instructions, ours aims to avoid substantial CPU modifications, thus works solely with the unmodified classical instruction set instead. The MCU and the electronics driver are coupled only through MMIO instructions. This loose coupling provides portability and extensibility, as the same communication scheme can in principle be used with all CPUs supporting the same underlying ISA, or even other classical ISAs, with little to no modification. However, the dedicated CPU is tightly integrated to the control electronics through onboard communication, which significantly reduces the communication cost.

2.2 System Implementation

Our design can in principle be implemented over various classical and quantum hardware. For our particular implementation, we assume room-temperature, as opposed to cryogenic, electronics, as they are more widely deployed today. While there is no fundamental reason to prefer RISC-V, ARM, or other instruction set architectures, we choose RISC-V for its potential in future system evolution. For instance, we anticipate that integrating the required quantum instruction pipelines into the RISC-V IP core would be less limited due to its open license business model.

Hardware setup. We implement a prototype by integrating a RISC-V IP core with our room-temperature electronics, which include a **timing control module (TCM)**, four-channel AWGs, four-channel data acquisition modules, a local oscillator, amplifiers, mixers, and a high-precision voltage source. As mentioned above, the in-house AWGs and the digitizers are collectively referred to as IQE.

We implement a real-time digital signal processing system on built-in FPGAs of the IQE, featuring precise timing control, arbitrary waveform generation, and parallel IQ demodulation for qubit state discrimination. The FPGA in TCM serves as the master FPGA running the MCU and the IQE driver. The master FPGA communicates with the AWGs and the digitizers through high-speed digital backplane transmissions.

In the aforementioned configuration depicted in Figure 2, we have assumed an unrestricted number of connections in the star-like topology. Now, we will explain how we can scale up from chassis-based systems that have a limited number of connections. A standard chassis with 18 slots meets the requirements of 10 qubits' control and readout. In such a one-chassis PXIe system, the master FPGA with a soft RISC-V IP core is used to provide triggers and instructions to other AWGs and digitizers. To control more qubits, the master FPGA in each one-chassis PXIe system can be

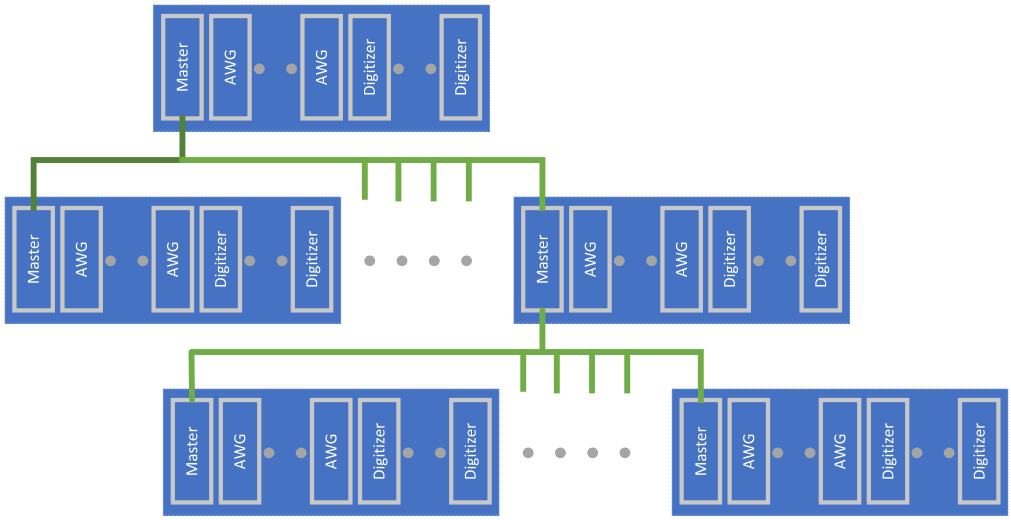


Fig. 3. Expansion scheme via star-like connection.

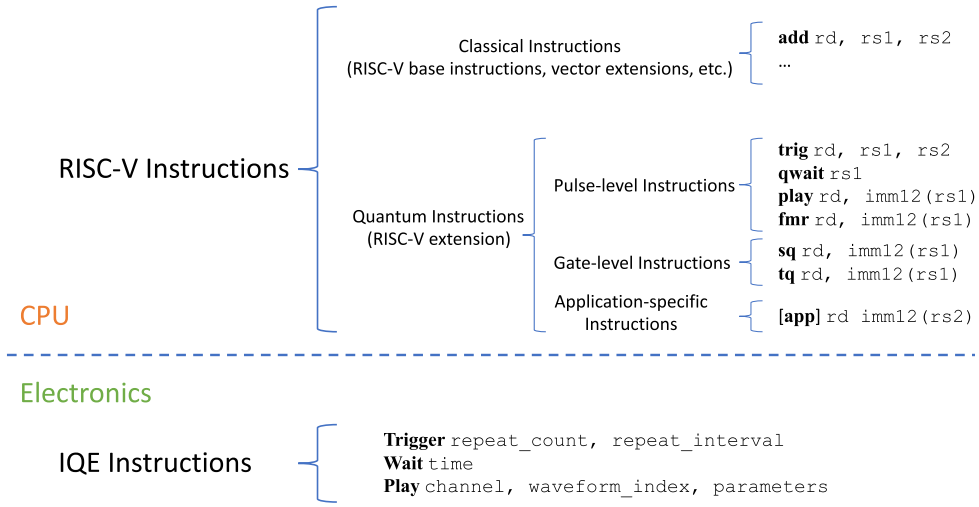


Fig. 4. Taxonomy of instructions.

interconnected through high-density connectors via a star-like expansion, as illustrated in Figure 3. Only one master FPGA needs to implement the soft RISC-V IP core as the MCU of the whole system. The MCU broadcasts the instructions to the master FPGAs of all those one-chassis PXIe subsystems by a daisy chain interface or star-like interface, and then each master FPGA broadcasts to AWGs and digitizers in the same chassis.

We will now provide a comprehensive overview of various types of instructions present in our architecture, including the IQE instructions as well as the RISC-V quantum instructions, as illustrated in Figure 4. Together they facilitate seamless control over the quantum processor.

Table 1. Summary of Instructions to the IQE Driver

Type	Operands
Trigger	**Repeat count**, *repeat interval*
Wait	**Time**
Play	Channel, **waveform index**, *parameters*

In the "Operands" column, underlined text indicates the "address operand" (an operand that is determined by the memory address rather than a value written); bold text indicates the "main operand" (i.e., writing this operand issues the instruction to the IQE driver); all other operands are in italic text, indicating that writing to them does not issue the instruction and that their values are preserved in the IQE driver memory.

IQE instructions. We specify the electronic-level instructions, or commands, broadcast by the IQE driver via the star-like connection, hereafter referred to as the "IQE instructions" for convenience (note that those "instructions" are not directly related to any CPU-level instructions). Currently, there are three types of IQE instructions, as summarized in Table 1:

— *"Trigger"*: As mentioned above, the "Trigger" instruction tells the IQE driver to actually start executing all quantum operations in the queue. To facilitate repeated measurements frequently occurring in qubit calibration, we ensure that the IQE driver has built-in functionality to *repeat* all quantum operations in the queue, with a specified number of repetitions and time intervals.
— *"Wait"*: The "Wait" instruction controls the relative timing between quantum operations in the same trigger, giving the user program full control on scheduling.
— *"Play"*: The "Play" instruction is the most basic quantum instruction. It plays a predefined waveform or a predefined combination of waveforms on one or more channels, which corresponds to quantum operations such as qubit reset, 1- or 2-qubit gates, or qubit measurement under the computational bases. Quantum measurement instructions differ from other operations in that they yield a result; these two cases are differentiated by the corresponding waveform indices, where indices $i \geq 128$ correspond to measurements, while indices < 128 are for no return values. The digitizer decodes the measurement instructions and sets the sampling window according to the parameters specified in the instruction. After IQ channel demodulation and data processing, the digitizer transmits the result to the CPU's system RAM.

Quantum instruction set. We here present an instantiation of a modular set of pseudo-instructions at the CPU level, consisting of pulse-level instructions for device calibration and gate-level instructions for algorithm implementations. These pseudo-instructions are not implemented directly, but are subsequently expanded to RISC-V MMIO operations via built-in load/store instructions. We also provide an example MMIO layout compatible with existing RISC-V architectures.

Table 2 illustrates the current design of the quantum instruction set and its corresponding expansion into MMIO load/store instructions. The instruction set features a hierarchical design, consisting of pulse-level, gate-level, and application-specific instructions. Each higher-level instruction can be decomposed into lower-level instructions with the same functionality, but using higher-level instructions reduced the decoding and dispatching overhead.

— "Pulse-level Instructions" `play`, `qwait`, and `trig`: specify pulses, their relative timing, and the issuance of the trigger signal, respectively. More precisely, `play` specifies the actual control pulse sequence, `qwait` specifies the scheduling of the corresponding pulses, and

Table 2. Summary of RISC-V Pseudo-instructions Designed for Communicating with the AQE Driver

Pseudo-instruction	Base instruction(s)	Meaning of parameters[1]	

Pulse-level	`trig rd, rs1, rs2`	`sw rd, ADDR_TRIGGER+8`
		`sw rs1, ADDR_TRIGGER+4`
		`sw rs2, ADDR_TRIGGER`
			`rd` — bit mask of channels
			`rs1` — repeat count
			`rs2` — repeat interval

	`qwait rs1`	`sw rs1, ADDR_WAIT`	`rs` — time
	`play rd, imm12(rs1)`	`sb rd, ADDR_PLAY + imm12(rs1)`	`rd` — waveform index `imm12(rs1)` — memory offset for the channel
	`fmr rd, imm12(rs1)`	`lw rd, ADDR_FMR + imm12(rs1)`	`rd` — destination register `imm12(rs1)` — result storage address
Gate-level	`sq rd, imm12(rs1)`	`sb rd, ADDR_GATE1Q + imm12(rs1)`	`rd` — gate index `imm12(rs1)` — memory offset for the qubit
	`tq rd, imm12(rs1)`	`sb rd, ADDR_GATE2Q + imm12(rs1)`	`rd` — gate index `imm12(rs1)` — memory offset for the qubit pair
Application-specific	`app rd, imm12(rs1)`	`sb rd, ADDR_APP + imm12(rs1)`	`rd` — operation index `imm12(rs1)` — memory offset for the operation grouping

`ADDR_*` are memory addresses that are determined at design time and thus are constant in the assembler.

`trig` triggers the actual execution of previously issued instructions. Additionally, `fmr` loads the qubit measurement results from previous runs from the predetermined addresses.

— "Gate-level Instructions" `sq` and `tq`: correspond to single-qubit and two-qubit operations, respectively. They share a similar expansion as the pulse-level "play" instructions, with different address offsets. As a single-qubit operation (e.g., a gate, a measurement, the qubit reset) usually consists of pulse plays on different physical channels with different timing constraints, a gate-level instruction is usually decoded into multiple IQE instructions by the IQE driver. This enables a relatively decoupled design of the MCU and the quantum architecture, as the exact interpretation of gate operations is only defined at the IQE driver level.

— "Application-specific Instructions" `app`: shares the same instruction format as the pulse-level `play`, but its decoding into IQE instructions is entirely left to the user. The user can design the decoding of `app` instructions as different combinations of IQE instructions, as long as it does not incur too much overhead on the IQE driver side. As the use cases of quantum processors in the near and far future remain largely uncertain, such customizable instruction design provides freedom of exploration with different potential use cases, including NISQ and fault-tolerant quantum computation.

We show in Table 3 an example MMIO layout supporting 0×4000, or 16,384, physical qubits, compatible with existing RISC-V architecture designs, including Si-Five, T-head, and so on. The allocated address space supports individual qubit control over quantum memory experiments on a surface code patch with a code distance of 90, or lattice surgery on two qubits with a code distance of up to 64. Note that neither distances is a hard constraint, as the MMIO address space is easily extendible to support a larger-scale computation.

3 EVALUATION METHODOLOGY

A full demonstration of scalability requires a large-scale quantum processor yet to be built. We thus focus on implementing essential features over surface code quantum computing using our architecture to argue that known scalability challenges can be resolved through our design.

More specifically, we reach our conclusion by focusing on components involved in large-scale computation or communication and analyzing how the incurred costs scale with the code distance and the quality of the quantum device. Our architecture design is not specific to surface-code-based

[1]The load/store instructions `lw, sw, sb` in this column are used to transfer a "word" or a "byte" between memory and registers.

Table 3. An Example of MMIO Address Layout

Name	Address	Type
ADDR_TRIGGER	0x40001000	int32
ADDR_WAIT	0x40002000	int32
ADDR_FMR	0x40003000	int32[0x1400]
ADDR_SQ	0x40010000	uint8[0x4000]
ADDR_TQ	0x40014000	uint8[0x8000]
ADDR_PLAY	0x4001c000	uint8[0x8000]
ADDR_APP	0x40024000	uint8[0x4000]

quantum computing, thus can be readily generalized to other quantum error correcting codes or fault-tolerant schemes.

3.1 Surface Code Quantum Computation

Surface code encodes the logical information of one qubit into a patch of $d \times d$ physical qubits, such that any error happening on at most $\lfloor (d-1)/2 \rfloor$ physical qubits can be detected through intermediate measurements and be corrected accordingly. A popular approach to realize logical Clifford operations for the surface code is "lattice surgery" [23]. Specifically, patches of logical qubits are arranged on a large grid, with additional physical qubits positioned in the "routing space" [7] between them. Then, lattice surgery allows measuring logical Pauli jointly over multiple patches, using interactions only between pairs of nearest-neighbor physical qubits.

There are alternatives to lattice surgery for realizing logical operations on surface codes (see Reference [5] and references therein). In this work, by **"surface code quantum computation" (SCQC)**, we refer to the approach through lattice surgery.

Figure 5 shows a schematic workflow of SCQC from the perspective of classical control. Upon receiving a pre-compiled quantum program, the MCU issues quantum instructions to an **"instruction decoding and dispatching unit" (IDDU)** through MMIO when needed. The IDDU then decodes the quantum instructions into pulse-level instructions readily executable on each of the electronics and dispatches them accordingly. The control electronics interact with the quantum hardware and return the raw measurement results to a dedicated memory region.

The incoming syndrome information is fed to and decoded by a classical firmware, called the **"syndrome decoding unit" (SDU)**, that runs on the dedicated CPU. Once decoded, the logical measurement results are fed to the MCU for adaptive real-time generation of the future instructions required by fault-tolerant quantum computing. In our implementation, the IDDU, the electronics, and the SDU correspond, respectively, to the IQE driver, the IQE, and part of the CCU.

Two essential subroutines of the SCQC are the "quantum memory experiment" and the "Bell-state experiment." Their quantum circuits are illustrated in Figure 6. The quantum memory experiment benchmarks the capability of the classical architecture for preserving quantum information, and the Bell-state experiment benchmarks that for essential steps in lattice surgery. As SCQC comprises mostly these two components (in addition to the preparation of a physical magic state and a single-patch logical measurement), we use them to validate our architecture.

Besides real-time execution of quantum circuits with large-scale parallel gates, these SCQC subroutines also require fast processing of classical information in "syndrome decoding." "Syndromes" are mid-circuit measurement results indicating errors occurring during the FTQC process. To identify the actual errors and correct them, a dedicated syndrome decoder is needed to deduce the most likely error given the syndrome information. Ideally, the syndrome decoder

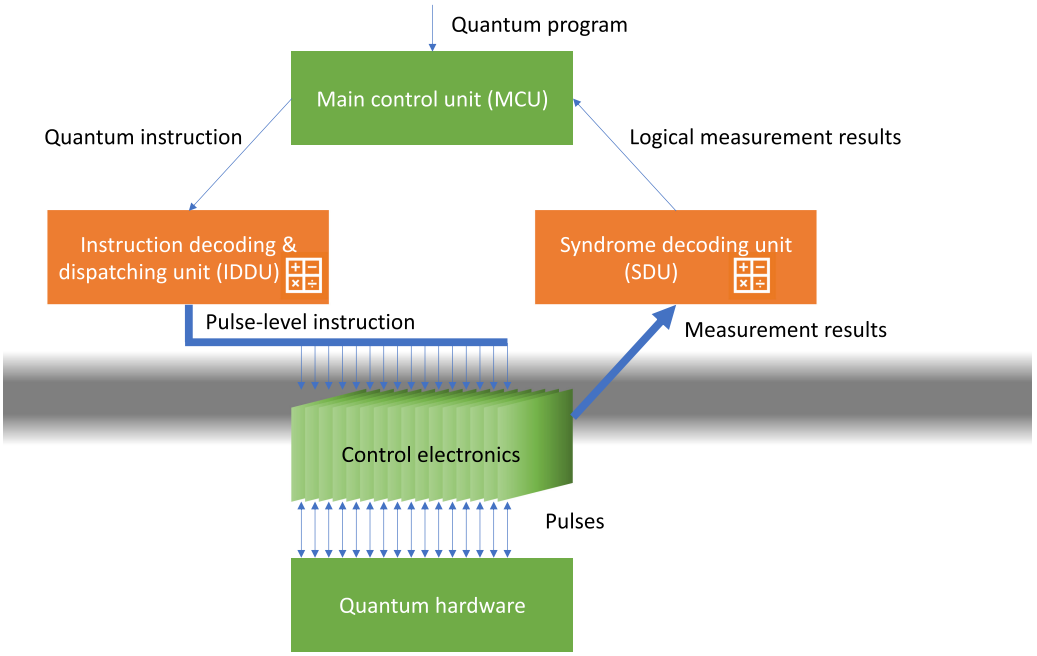


Fig. 5. Schematic workflow supporting surface code quantum computation (SCQC). The shaded area illustrates the blurred boundary of "classical" and "quantum" architecture, the former being our main focus.

needs to have a low error rate of inference and be fast enough not to cause exponential syndrome backlog [36]. Developing and implementing such a low-error, low-latency, and high-throughput syndrome decoder is vital to experimental realization of fault-tolerant quantum computation.

3.2 Validation of Scalability

We first establish the feasibility of our design by implementing an end-to-end prototype quantum computing system and validating that it functions properly with test qubit calibration programs. In addition, we examine the time variation ("jitter") of pulse control with increasing size of the star-like connection, confirming that our design admits scalable pulse synchronization. A low jitter ensures high-precision synchronization of pulses played on different AWG ports, ensuring high-fidelity controls.

To verify that our design resolves the instruction stress, we execute the aforementioned SCQC subroutines. We profile the running time of the classical controller against the running time of the quantum processor. The classical running time is estimated based on the instruction counts of an in-house CPU profiling tool over the QEMU RISC-V simulator. The running time of the quantum processor is estimated based on previously reported running time of each operation on a comparable superconducting platform [30]. We also quantitatively analyze the cost of instruction decoding and dispatching. Although neither is a scaling-up matter under our architecture design, we quantitatively show that the bandwidth of the differential pairs [24] can easily afford parallel gate instruction dispatching even under our proof-of-concept ISA implementation.

Real-time classical decoding was previously a hard problem, and even dedicated hardware struggled to achieve real-time decoding for code distance d larger than 11 [4, 13, 38]. However, recent advances [20, 31, 34] have made real-time decoding much more realistic even on general-purpose CPU. In particular, the sliding-window decoding schemes, introduced independently in References

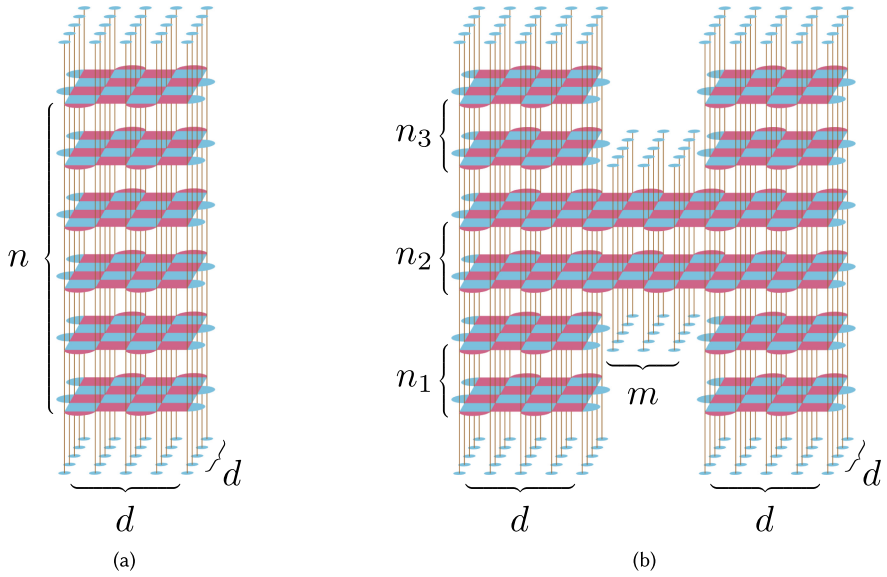


Fig. 6. Illustration of the quantum memory and the Bell-state experiments. A quantum memory experiment on a $d \times d$ lattice initiates n rounds of syndrome extractions. A Bell measurement experiment on two patches of $d \times d$ lattices with routing space length m initiates n_1 rounds of syndrome extractions on each patch, then initiates n_2 rounds of syndrome extractions on the joint patch by merging the two patches with the routing space, and finally initiates n_3 rounds of syndrome extractions on the two split patches. All data qubits are measured under the Z-basis before and after their corresponding syndrome extraction cycles.

[34] and [31], parallelize in scale: They split the decoding task evenly into an arbitrary number of parallel threads, with only a small constant overhead factor independent of the number of threads. We implement such a parallelized SDU on a RISC-V development board and benchmark its throughput on increasing code distances. We also give a rough estimation of the bandwidth required for syndrome transmission from the IQE digitizers to the SDU, finding it unlikely to become a bottleneck for our architecture.

4 SYSTEM EVALUATION

4.1 Real System Demonstration

We implement a prototype system by integrating a RISC-V IP core with our room-temperature electronics and a demo program to validate the end-to-end quantum computing system consisting of the prototype system, a quantum chip, and compilation toolchain. The demo program characterizes a qubit's relaxation time, i.e., the so-called T1 experiment. We compile an OpenQASM 3.0 front-end code to a RISC-V executable using our in-house compilation toolchain and test its correctness both on a pulse-level quantum simulator and on our in-house superconducting quantum processor. The result of the physical experiment is shown in Figure 7, demonstrating a successful run of the calibration routine.

4.2 Scalability of maintaining high-fidelity quantum operation

We now evaluate the feasibility of high-fidelity quantum operations when the chassis-based system is scaled up using a star-like connection.

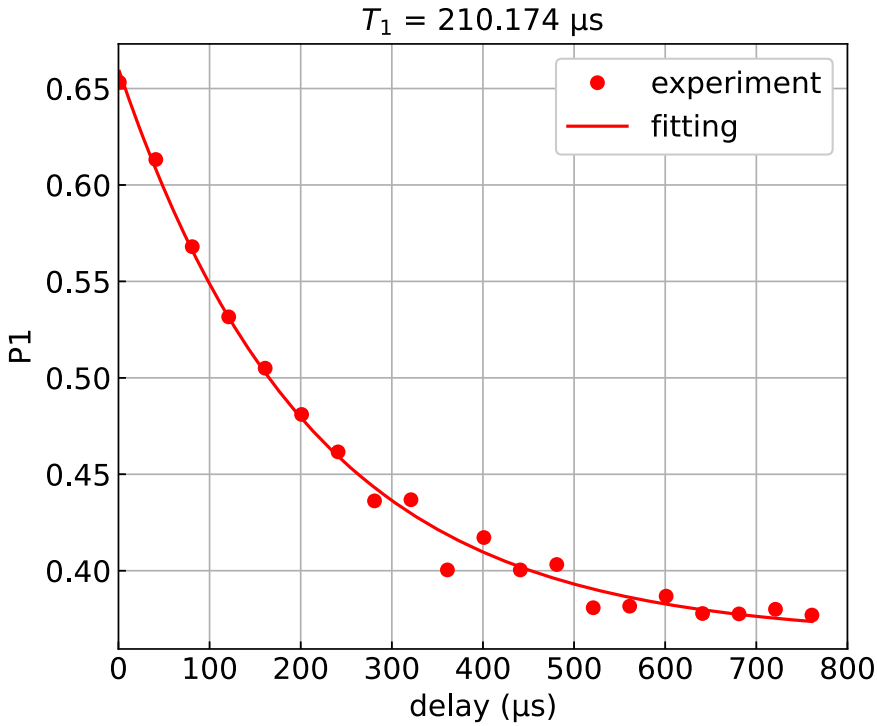


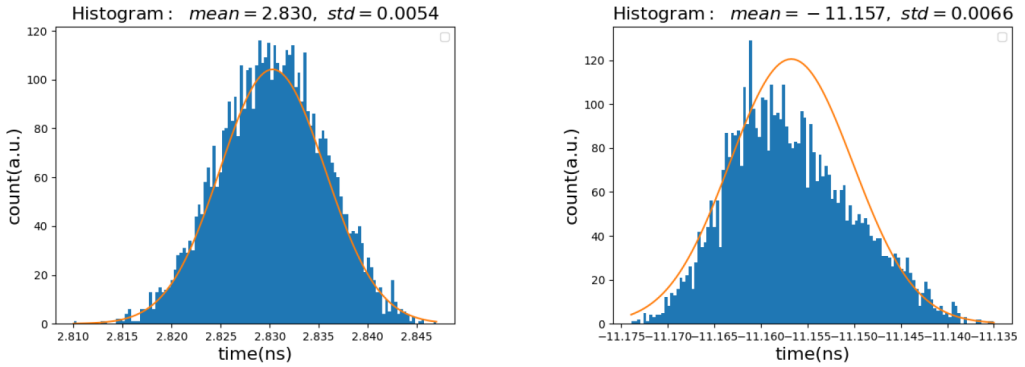
Fig. 7. T1 measurement on the prototype system.

Skew and jitter, which are crucial for system synchronization, can directly affect the accuracy of quantum operations. Skew, caused by variations in electrical connection length, can usually be compensated for, as it remains constant. Jitter, however, is a greater concern, as its effects cannot be calibrated.

To verify the fidelity of quantum operations in a larger system, we set up a 5-layer IQE platform with one MCU, one AWG, and one digitizer in each layer. The main trigger and the root system clock were generated by the MCU in the first layer and transmitted to the MCU in the second layer, and so on, for the subsequent layers. In each experiment, we pick the first layer and one other layer to test the jitter. The two AWG output channels from the chosen layers were then connected to a digitizer. We then use fixed-point phase analysis to calculate the jitter between these two signals as a proxy to evaluate pulse synchronization in larger systems. The critical aspect to consider is whether the jitter varies with the layer distance.

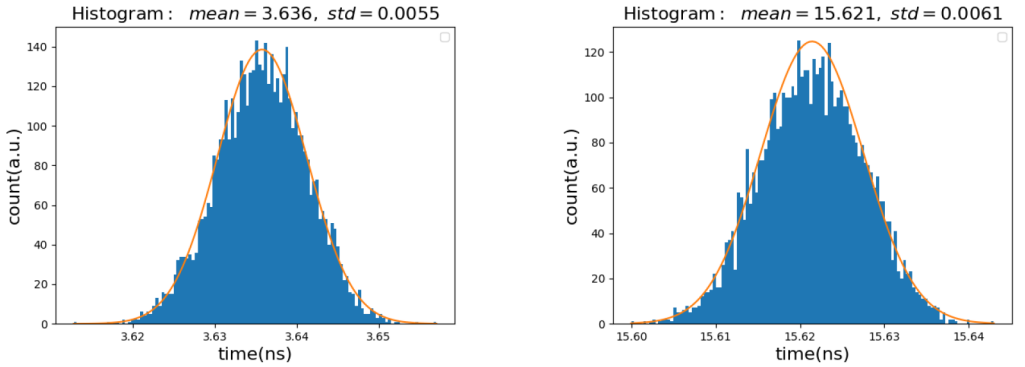
As depicted in Figure 8, the histograms display the jitter performance at different layer distances. Our measurements of layer-to-layer jitter show that the standard deviation is approximately 6 ps, and jitter does not increase with layer distance, indicating effective pulse synchronization within the system.

Based on the 5-layer results, we conclude that synchronization imprecision of microwave pulses from control electronics across different layers due to phase jitters is minuscule and will not become a major bottleneck for quantum computation. With a standard chassis that has 18 slots, the star-like expansion scheme is capable of supporting up to $10^4 + 10^3 + 10^2 + 10 + 1 = 11,111$ chassis and 111, 110 qubits based on the reasonable assumption that a single chassis can drive 10 additional chassis.



(a) Between layers 1 and 2.

(b) Between layers 1 and 3.



(c) Between layers 1 and 4.

(d) Between layers 1 and 5.

Fig. 8. Histogram of the channel-to-channel jitter of two AWGs across chassis in different layers.

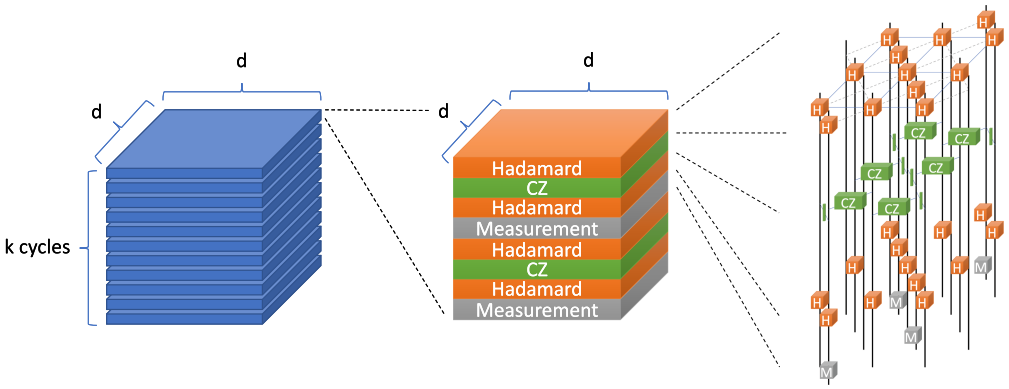
4.3 Scalability of the Instruction Pipeline

With the MMIO-based custom instruction design, we can test custom CPU-level instructions with different levels of abstraction against the quantum hardware execution time. In particular, we consider the following hierarchy of custom instruction abstraction, illustrated in Figure 9.

In addition to pulse- and gate-level instructions, the abstraction includes the following instructions:

— *Parallel-gate instructions* encode a layer of identical gates acting on a disjoint collection of qubits in a single instruction. Circuits involved in surface code quantum computing have the clear signature of the brickwork structure. Consequently, each round of syndrome extraction only costs a constant number of parallel-gate instructions. In this case, the number of parallel-gate instructions required per unit time no longer scales with the code distance d.

— *Logical-level instructions* further compresses all operations within a "logical cycle" [25] into a single instruction. A "logical cycle" refers to a repeated structure with d copies of identical syndrome extraction sub-routines, each consisting of constant layers of parallel operations with fixed patterns, with optional single-layer parallel operations before or after the repeats. Such a repeated structure is necessary for fault-tolerance against measurement errors and serves as building blocks for the SCQC. In this case, the total number of instructions throughout a quantum application stays a constant, regardless of the code distance, leaving




```
QMemory(k);
```

```
ParallelHadamard(Group0);
ParallelCZ(Group1);
…
ParallelHadamard(Group2);
ParallelCZ(Group3);
…
```

```
Hadamard(Qubit0);
Hadamard(Qubit1);
…
CZ(QubitPair0);
CZ(QubitPair1);
…
```

Fig. 9. A hierarchical instruction set design for SCQC. The applicability of this hierarchical design is made possible by the MMIO-based infrastructure and the brickwork structure of the syndrome extraction circuits. The figure only shows a particular syndrome extraction scheme following References [27, 30]; other syndrome extraction schemes also obey a brickwork structure, albeit differ slightly.

more room for improvement when dealing with other scaling factors, such as the number of logical qubits.

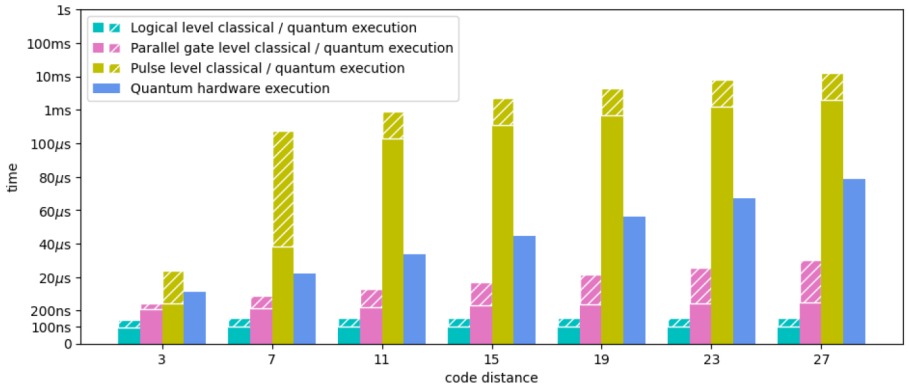
We estimate the execution time of custom instructions at each abstraction level, scaling in code distance, in Figure 10. It can be seen that the logical-level instructions stay constant with respect to code distance, and the parallel-gate level instructions scale linearly albeit with a smaller coefficient compared to the quantum running time. The pulse-level instructions scale with $\Theta(d^3)$ and quickly grow into the millisecond region, thus making them infeasible for surface code with reasonable sizes beyond a proof-of-principle demonstration.

For both the memory experiment and the Bell-state experiment, the majority of the quantum execution time is spent on syndrome extraction. Each syndrome cycle takes about 1 µs and takes 15 parallel-gate level instructions. This requires a throughput of 0.96 Gbps on the differential pair. This is well below the theoretical limit of the bandwidth of differential pairs [24]. As this estimation does not scale with respect to the code distance, the transmission of IQE instructions to the control electronics would not become a bottleneck for SCQC.

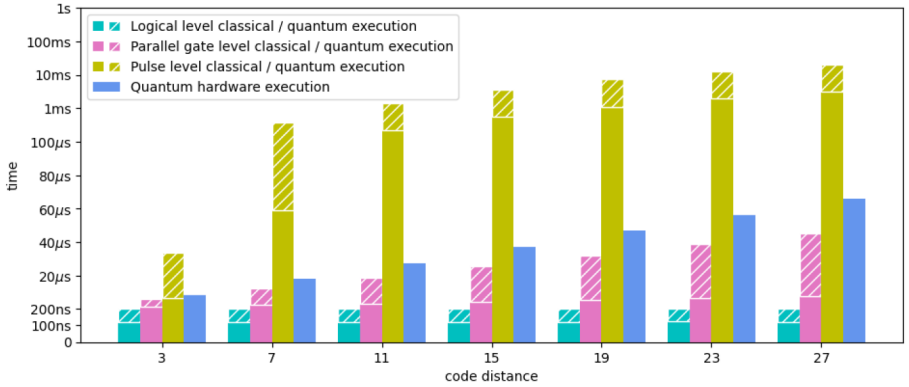
4.4 Scalability of the Syndrome Decoder

We implement an SDU with a parallel decoding firmware based on the "Sandwich Decoder" algorithm [31, 34]. This firmware splits the decoding task evenly into an arbitrary number of parallel threads, with a small constant overhead factor independent of the thread number, as long as the number of surface code rounds is sufficiently large.

We benchmark its throughput on the aforementioned SCQC subroutines. As a sanity check, we test the SDU implementation on a RISC-V development board and observe an agreement in results with our QEMU simulator. Benchmarking results are also used to extrapolate the expected throughput if implementing the SDU on other RISC-V IPs.



(a) Quantum memory experiment. The number of syndrome extraction rounds is set to $n = \frac{7}{2}(d+1)$.



(b) Bell-state experiment. The routing space length is set to $m = 3d$, and syndrome extraction rounds are set to $n_1 = n_2 = n_3 = d$.

Fig. 10. Comparison of the estimated classical execution time on the MCU against the quantum hardware execution time. The latter is estimated based on 20 ns for each single-qubit gate, 40 ns for each two-qubit gate, and 600 ns for each measurement and reset. The classical run time consists of two parts: (1) For classical instructions, the runtime estimated with the master frequency of the MCU CPU is 1 GHz and cycle counts from our in-house CPU profiling tools; (2) quantum instructions are executed via expansion into RISC-V base instructions for MMIO, and it takes up to 17 cycles for each MMIO communication between the MCU and the IQE driver through the system bus. The runtime is scaled piece-wise to reflect different running time scaling. The quantum execution time is marked separately with white hatches; note that the proportion of the quantum execution time versus the total classical execution time is distorted owing to the scaling distortion.

In deploying the Sandwich Decoder over the integrated multi-core CPU, the underlying inner decoders are an in-house Union-Find implementation and a recent PyMatching v2 [20]. We make several implementation-level improvements on the efficiency for our Union-Find decoder.

To benchmark the performance of our SDU, we apply the Sandwich Decoder to the Bell state experiment; this goes a step further than the memory experiment as in Reference [34]. For simplicity, we assume that the routing space between the two logical qubits is small compared to the code distance d and use a single large window to cover the two-qubit measurement part in the overall decoder graph (see Figure 11). All other windows are the same windows used in memory experiments. In our simulation experiments, we input the description of the large window as well

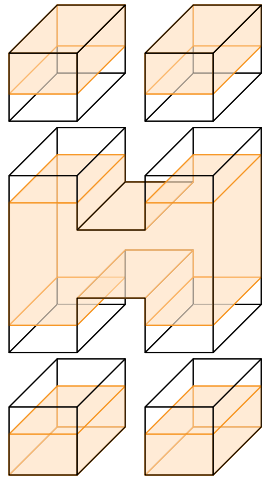


Fig. 11. Illustration of a division of the overall three-dimensional decoder graph of the Bell-state experiment (Figure 6) into windows. Under the assumption of a small routing space, even though the center window is large, its size is still $O(d) \times O(d) \times O(d)$. Note that the center window covers more layers than other windows; an alternative scheme (not depicted) is to divide the center window further so every window covers the same number of layers, which gives rise to more complicated windows.

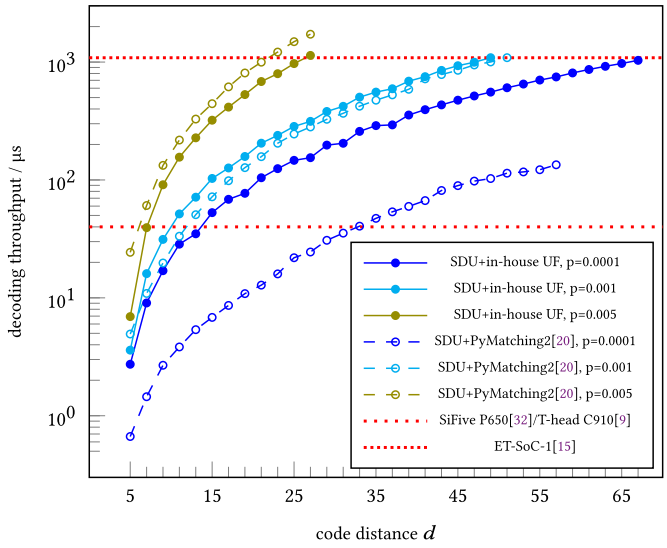
as the weight of each edge in that window to the CCU, which randomly generates error syndromes before invoking the decoding module.

We use the benchmarking results on the development board, shown in Figure 12, to estimate the decoding time when implementing our SDU on various RISC-V SoCs. With Union-Find and PyMatching 2 as inner decoders, the implemented SDU can decode up to distances 13 and 31 on SiFive P650 [32], T-head C910, or comparable alternatives [9] (16 cores at 2.5 GHz), or 67 and 57 with ET-SoC-1 [15] (1,088 cores at 1 GHz), all within just 1 microsecond for physical error rate $p = 0.0001$. Our evaluation of PyMatching 2 shows that its performance is constrained by the limited 1 GB memory available on the tested development board. We expect that PyMatching 2 could achieve even better results on a higher-end development board with a larger memory.

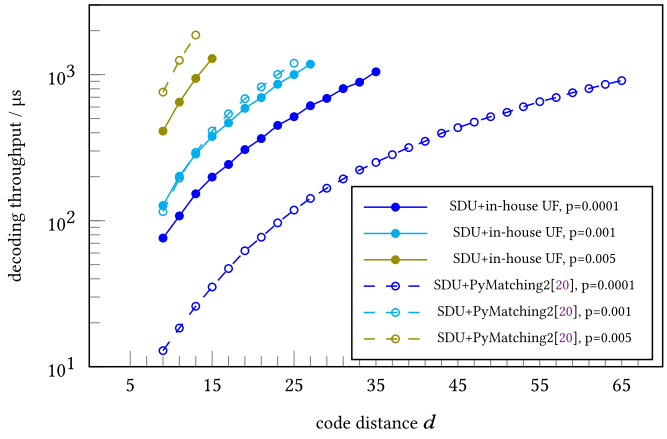
Besides the decoding throughput, another constraint for the decoder architecture is that the large amount of syndrome information generated throughout the SCQC process must not saturate the communication bandwidth. It is known that raw syndrome measurement results can quickly become a bandwidth bottleneck [13], thus must be compressed. One approach is to record the "detection events," i.e., changes in a sequence of syndrome bits, rather than all the syndrome bits. For a quantum memory experiment with a code distance d and a syndrome extraction cycle n, each ancilla qubit needs to generate $p_{\text{detect}} \cdot n \log_2 n$ bits, on average, assuming that each detection event happens with a probability p_{detect}. Roughly, the bandwidth requirement would become 100 Mbps for $p_{\text{detect}} = 0.02$ and $n = d = 33$. This compression can be done on each digitizer separately before transmission to the IQE driver. More advanced compression algorithms may achieve a better compression rate but may require conjoined processing from different digitizers. Such an algorithm can be placed in the IQE driver should there be a bottleneck in the MMIO bandwidth.

5 DISCUSSION AND OUTLOOK

We present a scalable design for the classical architecture of quantum computing. Our design aims towards easy scaling with no significant overhead. We evaluate its scalability on two basic



(a) Decoding throughput for the quantum memory experiment.



(b) Decoding throughput for the Bell-state experiment.

Fig. 12. The average syndrome decoding throughput for the quantum memory experiment and the two-qubit joint measurement in the Bell state experiment. Decoding throughput is defined as the average processing time per single layer of syndrome on a single core with 1 GHz master frequency. We experimentally benchmark the total running time of both experiments under different code distances and multiple runs and deduce the per-layer running time. For the ease of comparison, we set the step size as $t + 1 = (d + 1)/2$ and the window size as $3(t + 1)$ for both experiments. For the quantum memory experiments, the red dashed lines indicate the capability of the specific RISC-V SoCs, converted to the same scale as the experimental data. The data points below a certain dashed line indicate the feasibility of running the corresponding task on the corresponding SoC within 1 µs.

subroutines over a prominent fault-tolerant scheme and validate its practical feasibility with a prototype implementation.

A natural next step is to implement the real system with quantum processors of a much larger scale than that in our study. The current design is estimated to scale up easily over thousands of

qubits. Such an estimation is based on the size of the allocated MMIO addresses, the picosecond accuracy in the synchronization of the trigger signal across different electronics, and the physical size of the electronics stacks. Although most of the limiting factors can be lifted through a more careful design, it remains uncertain if unforeseen problems may arise with a larger-scale quantum processor. A possible further scaling-up through modularization is to let each MCU control one or a few logical components, such as a single logical qubit or a patch of the routing space, and let an upper-level control unit issue logical instructions to these logical components while maintaining synchronization.

In this study, we assume room-temperature devices for their wide adoption at the time of writing. However, our design, in principle, is not limited to such and may in particular work well for cryogenic electronics, such as cryo-CMOS [8] or single-flux-quantum [28], as long as the component functionalities can be implemented. Such demonstrations would be an interesting future direction.

Another important direction is to demonstrate through more sophisticated tasks than our two "toy-model" subroutines. Such experiments may lead to the discovery of currently unknown limiting factors for classical architecture in SCQC.

Classical architecture is just half of the story, as numerous challenges remain to be addressed in quantum architecture. Beyond the quantum processor's scale, hurdles such as **input/output (I/O)** management, interconnection, packaging, and heat and power dissipation must be overcome. Previous research in quantum architecture has often more focused on the feasibility of qubit control than the potential demands of intensive classical computation. Conversely, studies on classical architecture have primarily examined the viability of specific classical computation tasks such as syndrome decoding, either in-fridge or out-of-fridge, under the bold assumption that high-fidelity qubit control can be realistically achieved. A holistic evaluation of the FTQC workflow, encompassing both classical and quantum architectures, will aid in identifying potential bottlenecks and determining the most effective steps to move forward.

ACKNOWLEDGMENTS

We would like to thank all the members of DAMO Quantum Laboratory who contributed to the development of the quantum hardware used to demonstrate the end-to-end workflow in this study.

REFERENCES

[1] Rajeev Acharya, Igor Aleiner, Richard Allen, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Juan Atalaya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Joao Basso, Andreas Bengtsson, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Yu Chen, Zijun Chen, Ben Chiaro, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben Curtin, Dripto M. Debroy, Alexander Del Toro Barba, Sean Demura, Andrew Dunsworth, Daniel Eppens, Catherine Erickson, Lara Faoro, Edward Farhi,Reza Fatemi, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, William Giang, Craig Gidney, Dar Gilboa, Marissa Giustina, Alejandro Grajales Dau, Jonathan A. Gross, Steve Habegger, Michael C. Hamilton, Matthew P. Harrigan, Sean D. Harrington, Oscar Higgott, Jeremy Hilton, Markus Hoffmann, Sabrina Hong, Trent Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Pavol Juhas, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Tanuj Khattar, Mostafa Khezri, Mária Kieferová, Seon Kim, Alexei Kitaev, Paul V. Klimov, Andrey R. Klots, Alexander N. Korotkov, Fedor Kostritsa, John Mark Kreikebaum, David Landhuis, Pavel Laptev, Kim-Ming Lau, Lily Laws, Joonho Lee, Kenny Lee, Brian J. Lester, Alexander Lill, Wayne Liu, Aditya Locharla, Erik Lucero, Fionn D. Malone, Jeffrey Marshall, Orion Martin, Jarrod R. McClean, Trevor McCourt, Matt McEwen, Anthony Megrant, Bernardo Meurer Costa, Xiao Mi, Kevin C. Miao, Masoud Mohseni, Shirin Montazeri, Alexis Morvan, Emily Mount, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Murphy Yuezhen Niu, Thomas E. O'Brien, Alex Opremcak, John Platt, Andre Petukhov, Rebecca Potter, Leonid P. Pryadko, Chris Quintana, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus,

Christopher Schuster, Michael J. Shearn, Aaron Shorter, Vladimir Shvarts, Jindra Skruzny, Vadim Smelyanskiy, W. Clarke Smith, George Sterling, Doug Strain, Marco Szalay, Alfredo Torres, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Theodore White, Cheng Xing, Z. Jamie Yao, Ping Yeh, Juhwan Yoo, Grayson Young, Adam Zalcman, Yaxing Zhang, Ningfeng Zhu, and Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681. DOI: https://doi.org/10.1038/s41586-022-05434-1

[2] Google Quantum AI. 2021. Exponential suppression of bit or phase errors with cyclic error correction. *Nature* 595, 7867 (2021), 383–387.

[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (Oct. 2019), 505–510. DOI: https://doi.org/10.1038/s41586-019-1666-5

[4] Francesco Battistel, Christopher Chamberland, Kauser Johar, Ramon W. J. Overwater, Fabio Sebastiano, Luka Skoric, Yosuke Ueno, and Muhammad Usman. 2023. Real-time decoding for fault-tolerant quantum computing: Progress, challenges and outlook. arXiv:2303.00054

[5] Hector Bombin, Chris Dawson, Ryan V. Mishmash, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. 2023. Logical blocks for fault-tolerant topological quantum computation. *PRX Quant.* 4, 2 (2023), 020303.

[6] 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 *International Conference on Rebooting Computing (ICRC'20)*. IEEE, 66–75.

[7] Christopher Chamberland and Earl T. Campbell. 2022. Universal quantum computing with twist-free and temporally encoded lattice surgery. *PRX Quant.* 3, 1 (Feb. 2022), 010331.
DOI: https://doi.org/10.1103/PRXQuantum.3.010331

[8] E. Charbon, F. Sebastiano, A. Vladimirescu, H. Homulle, S. Visser, L. Song, and R. M. Incandela. 2016. Cryo-CMOS for quantum computing. In *IEEE International Electron Devices Meeting (IEDM'16)*. IEEE, 13–5.

[9] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension. In *ACM/IEEE 47th International Symposium on Computer Architecture (ISCA'20)*. IEEE Press, 52–64.DOI: https://doi.org/10.1109/ISCA45697.2020.00016

[10] James R. Cruise, Neil I. Gillespie, and Brendan Reid. 2020. Practical Quantum Computing: The Value of Local Computation. DOI: https://doi.org/10.48550/ARXIV.2009.08513

[11] Poulami Das, Aditya Locharla, and Cody Jones. 2022. LILLIPUT: A lightweight low-latency lookup-tab decoder for near-term quantum error correction. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. Association for Computing Machinery, New York, NY, 541–553.DOI: https://doi.org/10.1145/3503222.3507707

[12] Poulami Das, Christopher A. Pattison, Srilatha Manne, Douglas Carmean, Krysta Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2020. A scalable decoder micro-architecture for fault-tolerant quantum computing. *arXiv preprint* (2020). arXiv:2001.06598

[13] Poulami Das, Christopher A. Pattison, Srilatha Manne, Douglas M. Carmean, Krysta M. Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2022. AFS: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. IEEE, 259–273.

[14] Nicolas Delfosse and Naomi H. Nickerson. 2021. Almost-linear time decoding algorithm for topological codes. *Quantum* 5 (2021), 595.

[15] D. R. Ditzel and the Esperanto team. 2022. Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip. In *IEEE Micro*, 42, 3 (2022), 31–38. DOI: https://doi.org/10.1109/MM.2022.3140674

[16] Laird Egan, Dripto M. Debroy, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Michael Newman, Muyuan Li, Kenneth R. Brown, Marko Cetina, and Christopher Monroe. 2021. Fault-tolerant control of an error-corrected qubit. *Nature* 598, 7880 (2021), 281–286.

[17] Austin G. Fowler. 2015. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average O(1) parallel time. *Quant. Inf. Comput.* 15, 1-2 (2015), 145–158. DOI: https://doi.org/10.26421/QIC15.1-2-9

[18] Xiang Fu, Leon Riesebos, M. A. Rol, Jeroen Van Straten, J. Van Someren, Nader Khammassi, Imran 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 *IEEE International Symposium on High-Performance Computer Architecture (HPCA'19)*. IEEE, 224–237. DOI: https://doi.org/10.1109/HPCA.2019.00040 arXiv:1808.02449

[19] Christopher Granade, Christopher Ferrie, and David G. Cory. 2015. Accelerated randomized benchmarking. *New J. Phys.* 17, 1 (2015), 013042.

[20] Oscar Higgott and Craig Gidney. 2022. PyMatching v2. Retrieved from https://github.com/oscarhiggott/PyMatching

[21] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshan Ding, Massoud Pedram, and Frederic T. Chong. 2020. NISQ+: Boosting quantum computing power by approximating quantum error correction. In *47th ACM/IEEE International Symposium on Computer Architecture (ISCA'20)*. IEEE Press, 556–569. https://doi.org/10.1109/ISCA45697.2020.00053

[22] J. M. Hornibrook, J. I. Colless, I. D. Conway Lamb, S. J. Pauka, H. Lu, A. C. Gossard, J. D. Watson, G. C. Gardner, S. Fallahi, M. J. Manfra, and D. J. Reilly. 2015. Cryogenic control architecture for large-scale quantum computing. *Phys. Rev. Appl.* 3, 2 (2015), 024010.

[23] Clare Horsman, Austin G. Fowler, Simon Devitt, and Rodney Van Meter. 2012. Surface code quantum computing by lattice surgery. *New J. Phys.* 14, 12 (2012), 123011. DOI: https://doi.org/10.1088/1367-2630/14/12/123011

[24] Texas Instruments. 2002. Interface circuit for TIA/EIA-644 (LVDS). *SLLA038B, Application notes, Texas Instruments*. Retrieved from https://www.ti.com/lit/an/slla038b/slla038b.pdf

[25] Daniel Litinski. 2019. A game of surface codes: Large-scale quantum computing with lattice surgery. *Quantum* 3 (2019), 128. DOI: https://doi.org/10.22331/q-2019-03-05-128

[26] Rui Machado, Jorge Cabral, and Filipe Serra Alves. 2019. Recent developments and challenges in FPGA-based time-to-digital converters. *IEEE Trans. Instrument. Measur.* 68, 11 (2019), 4205–4221.

[27] J. F. Marques, B. M. Varbanov, M. S. Moreira, Hany Ali, Nandini Muthusubramanian, Christos Zachariadis, Francesco Battistel, Marc Beekman, Nadia Haider, Wouter Vlothuizen, A. Bruno, B. M. Terhal, and L. DiCarlo. 2022. Logical-qubit operations in an error-detecting surface code. *Nat. Phys.* 18, 1 (2022), 80–86.

[28] Oleg A. Mukhanov. 2011. Energy-efficient single flux quantum technology. *IEEE Trans. Appl. Superconduct.* 21, 3 (2011), 760–769.

[29] Rohit Navarathna, Tyler Jones, Tina Moghaddam, Anatoly Kulikov, Rohit Beriwal, Markus Jerger, Prasanna Pakkiam, and Arkady Fedorov. 2021. Neural networks for on-the-fly single-shot state classification. *Appl. Phys. Lett.* 119, 11 (2021), 114003.

[30] Thomas E. O'Brien, B. Tarasinski, and Leo DiCarlo. 2017. Density-matrix simulation of small surface codes under current and projected experimental noise. *npj Quant. Inf.* 3, 1 (2017), 39.

[31] Riverlane. 2022. Deltaflow. Decode Technical White Paper. Retrieved from https://www.riverlane.com/media/nz2dvqmi/deltaflow_decode_technical_white_paper_september_2022.pdf

[32] SiFive Development Team. 2022. SiFive Performance P650. Retrieved from https://www.sifive.com/cores/performance-p650

[33] Luka Skoric, Dan E. Browne, Kenton M. Barnes, Neil I. Gillespie, and Earl T. Campbell. 2022. Parallel window decoding enables scalable fault tolerant quantum computation. *arXiv preprint arXiv:2209.08552* (2022).

[34] Xinyu Tan, Fang Zhang, Rui Chao, Yaoyun Shi, and Jianxin Chen. 2022. Scalable surface code decoders with parallelization in time. *arXiv preprint arXiv:2209.09219* (2022).

[35] Swamit S. Tannu, Zachary A. Myers, Prashant J. Nair, Douglas M. Carmean, and Moinuddin K. Qureshi. 2017. Taming the instruction bandwidth of quantum computers via hardware-managed error correction. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. Association for Computing Machinery, New York, NY, 679–691.DOI: https://doi.org/10.1145/3123939.3123940

[36] Barbara M. Terhal. 2015. Quantum error correction for quantum memories. *Rev. Mod. Phys.* 87, 2 (2015), 307.

[37] Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki, and Yutaka Tabuchi. 2021. QECOOL: On-line quantum error correction with a superconducting decoder for surface code. In *58th ACM/IEEE Design Automation Conference*. 451–456. DOI: https://doi.org/10.1109/DAC18074.2021.9586326 arXiv:2103.14209

[38] Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki, and Yutaka Tabuchi. 2022. NEO-QEC: Neural network enhanced online superconducting decoder for surface codes. (2022). arXiv:2208.05758

[39] Joel J. Wallman and Joseph Emerson. 2016. Noise tailoring for scalable quantum computation via randomized compiling. *Phys. Rev. A* 94, 5 (2016), 052325.

[40] Ellis Wilson, Sudhakar Singh, and Frank Mueller. 2020. Just-in-time quantum circuit transpilation reduces noise. In *IEEE International Conference on Quantum Computing and Engineering (QCE'20)*. IEEE, 345–355.

[41] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Qingling Zhu, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. 2021. Strong quantum computational advantage using a superconducting quantum processor. *Phys. Rev. Lett.* 127, 18 (Oct. 2021), 180501. DOI: https://doi.org/10.1103/PhysRevLett.127.180501

[42] Yuchen Yang, Zhongtao Shen, Xing Zhu, Ziqi Wang, Gengyan Zhang, Jingwei Zhou, Xun Jiang, Chunqing Deng, and Shubin Liu. 2022. FPGA-based electronic system for the control and readout of superconducting quantum processors. *Rev. Scient. Instrum.* 93, 7 (2022), 074701.

[43] Mengyu Zhang, Lei Xie, Zhenxing Zhang, Qiaonian Yu, Guanglei Xi, Hualiang Zhang, Fuming Liu, Yarui Zheng, Yicong Zheng, and Shengyu Zhang. 2021. Exploiting different levels of parallelism in the quantum control microarchitecture for superconducting qubits. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Association for Computing Machinery, New York, NY, 898–911. DOI: https://doi.org/10.1145/3466752.3480116

Received 26 June 2023; accepted 28 August 2023