2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

# eQASM: An Executable Quantum Instruction Set Architecture

X. Fu,[1,2*] L. Riesebos,[1,2] M. A. Rol,[1,3] J. van Straten,[4] J. van Someren,[1,2] N. Khammassi,[1,2]
I. Ashraf,[1,2] R. F. L. Vermeulen,[1,3] V. Newsum,[5,1] K. K. L. Loh,[5,1] J. C. de Sterke,[6,1]
W. J. Vlothuizen,[5,1] R. N. Schouten,[1,3] C. G. Almudever,[1,2] L. DiCarlo,[1,3] K. Bertels[1,2†]
[1] *QuTech, Delft University of Technology, P.O. Box 5046, 2600 GA Delft, the Netherlands*
[2] *Quantum Computer Architecture Lab, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, the Netherlands*
[3] *Kavli Institute of Nanoscience, Delft University of Technology, P.O. Box 5046, 2600 GA Delft, the Netherlands*
[4] *Computer Engineering Lab, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, the Netherlands*
[5] *Netherlands Organisation for Applied Scientific Research (TNO), P.O. Box 155, 2600 AD Delft, the Netherlands*
[6] *Topic Embedded Systems B.V., P.O. Box 440, 5680 AK Best, the Netherlands*
∗ x.fu-1@tudelft.nl † k.l.m.bertels@tudelft.nl

***Abstract*—A widely-used quantum programming paradigm comprises of both the data flow and control flow. Existing quantum hardware cannot well support the control flow, significantly limiting the range of quantum software executable on the hardware. By analyzing the constraints in the control microarchitecture, we found that existing quantum assembly languages are either too high-level or too restricted to support comprehensive flow control on the hardware. Also, as observed with the quantum microinstruction set QuMIS [1], the quantum instruction set architecture (QISA) design may suffer from limited scalability and flexibility because of microarchitectural constraints. It is an open challenge to design a scalable and flexible QISA which provides a comprehensive abstraction of the quantum hardware.**

**In this paper, we propose an executable QISA, called eQASM, that can be translated from quantum assembly language (QASM), supports comprehensive quantum program flow control, and is executed on a quantum control microarchitecture. With efficient timing specification, single-operation-multiple-qubit execution, and a very-long-instruction-word architecture, eQASM presents better scalability than QuMIS. The definition of eQASM focuses on the assembly level to be expressive. Quantum operations are configured at compile time instead of being defined at QISA design time. We instantiate eQASM into a 32-bit instruction set targeting a seven-qubit superconducting quantum processor. We validate our design by performing several experiments on a two-qubit quantum processor.**

***Keywords*-Quantum instruction set architecture (QISA); eQASM; Quantum control microarchitecture; QuMA; superconducting quantum processor**

## I. Introduction

Quantum computing is promising with the potential to accelerate solving some problems which are inefficiently solved by classical computers, such as quantum chemistry simulation [2, 3]. A near-term goal is to develop a fully programmable quantum computer based on the circuit model with Noisy Intermediate-Scale Quantum (NISQ) technology [4] without quantum error correction [5]. To this end, a viable way for both quantum software and hardware is to support the widely-used "quantum data, classical control" programming paradigm [6]. In this paradigm, the data flow is the state evolution subject to a sequence of classical or quantum operations, and the control flow is the order of operations that are executed, which could be directed by qubit measurement results. It is embodied in a wide range of quantum applications, including active qubit reset [7], teleportation [8], non-deterministic quantum gate decomposition [9], iterative quantum phase estimation [10], etc.

Though high-level quantum software can support this paradigm well, current quantum hardware cannot because of limited executability of existing low-level quantum assembly languages. Assembly languages are introduced as human-readable representation of the interface to hardware (or machine code). But existing quantum assembly languages either incorporate too high-level constructs to be directly implemented by a microarchitecture (including QASM-HL [11], Quil [12], f-QASM [13], etc.), or are too restricted to provide a comprehensive abstraction of the quantum hardware which can support the required flow control (including OpenQASM [14], QuMIS [1], etc.). This fact significantly limits the range of quantum software which can be executed by the hardware.

Required is a scalable and flexible interface which can be executed by the hardware. Considering the constraints in microarchitecture implementation, designing such an interface is challenged by the difficulty of (1) providing a comprehensive abstraction of the quantum hardware [15] and (2) making the quantum instruction set architecture (QISA) scalable and flexible.

### A. Comprehensive Abstraction Challenge

Existing quantum software, including quantum programming languages [16], compilers [11, 17, 18], and quantum assembly languages [11–14], can well describe both the data flow and the control flow. The basic constructs of control flow include procedure, selection, loop, and recur-

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00040

sion [6, 16]. Among them, selection and loop may use feedback based on qubit measurement results to select which instructions to execute in the following. However, existing programmable quantum hardware mostly focuses on supporting the data flow. They cannot well support the control flow because they lack programmable feedback with sufficient flexibility [19–21] (though feedback has been demonstrated with customized hardware in multiple experiments [7, 22–24]).

The difficulty of supporting programmable feedback in the hardware roots in the strict requirements on the electrical signals (e.g., precise parameters and timing) used to control qubits. To satisfy these requirements, a three-step procedure is usually used: (i) defining waveforms in digital format that are long enough to include all operations of the quantum application, (ii) uploading the digital waveforms to arbitrary waveform generators (AWG), and (iii) converting these digital waveforms into analog ones at runtime by the AWGs. Because of the computational complexity and communication latency, step (i) and (ii) are performed at static time. The fact that waveforms are determined at static time makes runtime feedback almost impossible.

Using digital waveforms only as the interface to quantum hardware is cumbersome and presents poor scalability and flexibility. To address this issue, our previous work [1] proposed the quantum control microarchitecture QuMA, which implements the instruction-based waveform generation as an alternative. In this method, a set of short pulses in place of long waveforms are uploaded to AWGs, with each pulse representing a quantum operation. By executing instructions in the quantum microinstruction set QuMIS, desired pulses are selected and triggered at runtime, which in principle provides the foundation for runtime feedback. However, the timing of executing instructions is decoupled from that of pulse generation by a set of FIFOs [1]. As a consequence, a classical instruction that uses a qubit measurement result may start execution before the expected result is ready, or read another result when there are multiple instructions measuring the same qubit, which can lead to a wrong execution result. Hence, it is a challenge to design a mechanism that can correctly implement runtime feedback to support comprehensive quantum program flow control.

### B. Scalability & Flexibility Challenges

A scalability issue in QISA design was first observed with QuMIS, which was implemented by the control microarchitecture QuMA. Because quantum operations of a quantum application are applied on qubits with particular timing, quantum instructions should be fetched from the instruction memory and processed in time to ensure the described operations can be applied with the correct timing. Since every instruction can only encode a limited number of quantum operations, it would require a minimum number ($R_{\mathrm{req}}$) of instructions to be issued and processed per cycle for this application. However, the microarchitecture can only issue a limited number ($R_{\mathrm{allowed}}$) of instructions per cycle given a limited instruction issue rate. As the number of qubits grows, $R_{\mathrm{req}}$ in general increases. When $R_{\mathrm{req}} > R_{\mathrm{allowed}}$, the microarchitecture cannot execute the quantum program correctly. We call this problem the *quantum operation issue rate problem* [1, 25, 26]. The low instruction information density of QuMIS contributes to a large $R_{\mathrm{req}}$, exaggerating this problem. A QuMIS program has a relatively low instruction information density because (1) an explicit waiting instruction is required to separate any two consecutive timing points; (2) each target qubit of a quantum operation occupies a field in the instruction, making the instruction width a limitation for the number of target qubits in a single instruction; (3) two parallel and different operations cannot be combined into a single instruction. In our previous experiments, we found that the boundary condition $R_{\mathrm{req}} \leq R_{\mathrm{allowed}}$ cannot be satisfied for some applications with only two qubits. Hence, how to design a QISA with a high instruction information density forms a scalability challenge.

Though being able to provide flexibility in directing the program flow control at runtime, some QISA design suffers from no quantum semantics. For example, instructions of QuMIS and the Raytheon BBN APS2 instruction set [27] are low level and tightly bound to the electronic hardware implementation to ensure the executability. Compared to existing quantum assembly languages, these instructions are microinstructions without explicit quantum semantics. Thus, they do not qualify as a QISA, and it is another challenge to design an executable QISA with flexible quantum semantics.

### C. Contributions

In this paper, we propose an executable QISA based on QASM, named *executable QASM* (eQASM). eQASM can be generated by the compiler backend from a higher-level representation, like cQASM [28]. eQASM contains both quantum instructions and auxiliary classical instructions to support quantum program flow control. eQASM supports a set of discrete quantum operations to enable a microarchitecture implementation. The contributions of the paper are the following:

- **Comprehensive quantum control flow:** eQASM proposes two kinds of feedback with required microarchitectural mechanisms to implement them: *fast conditional execution* for simple but fast feedback, and *comprehensive feedback control* (CFC) for arbitrary user-definable feedback. Based on this, eQASM can support comprehensive program flow control required by the "quantum data, classical control" paradigm, and significantly broadens the range of quantum applications executable on hardware;
- **Operational implementation:** eQASM is a QISA framework with the definition focusing on the assembly level and the basic rules of mapping assembly to binary. It

requires customized instantiation for the binary format targeting a particular platform, which allows the pursuit of flexibility and practicability in microarchitecture implementation;

- **Increased instruction information density:** eQASM adopts *Single-Operation-Multiple-Qubit* (SOMQ) execution, a Very-Long-Instruction-Word (VLIW) architecture and a more efficient method for explicit timing specification, which can considerably alleviate the quantum operation issue rate problem when compared to QuMIS;
- **Configurable QISA at compile time:** As opposed to the classical instruction set architecture (ISA) whose operations are defined at ISA design time, eQASM enables the programmer to configure allowed quantum operations at compile time, leaving ample space for compiler-based optimization.

We instantiate eQASM into a 32-bit instruction set targeting a seven-qubit superconducting quantum processor and implement it using a control microarchitecture derived from QuMA as proposed in [1]. We validated eQASM by performing several experiments over a two-qubit superconducting quantum processor using the implemented microarchitecture.

This paper is organized as follows. Section II introduces the heterogeneous quantum programming model adopted by eQASM and an overview of eQASM. The quantum instructions of eQASM with related mechanisms are explained in Section III. Section IV describes the instantiation of eQASM targeting a seven-qubit quantum processor as well as its microarchitecture and implementation. Section V shows the experiments, and Section VI concludes.

## II. eQASM Overview

To our understanding, it is viable to integrate quantum computing in a similar way as a GPU or an FPGA in a heterogeneous architecture. The quantum part can be seen as an accelerator for particular classically-hard tasks. This section introduces the programming and compilation model, the design guidelines, the architectural state, and an overview of instructions of eQASM.

### A. Programming and Compilation Model

To take advantage of both classical computing and quantum computing, eQASM is defined based on OpenCL [29], which is an open industry standard for classical heterogeneous parallel computing. The programming and compilation model of eQASM is shown in Fig. 1.

A quantum-classical hybrid program contains a host program and one or more quantum kernels with the quantum kernel(s) accelerating particular parts of the computation. A hybrid compilation infrastructure compiles the host program into classical code using a conventional compiler such as GCC, which is later executed by the classical host CPU.

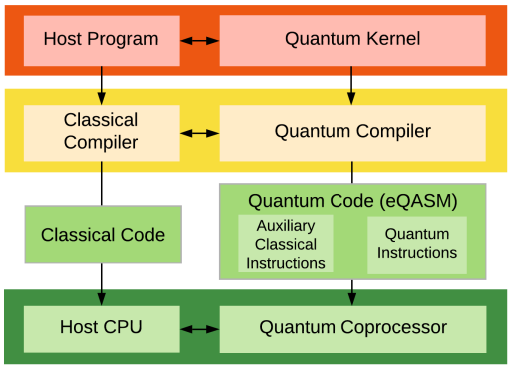


**Fig. 1.** Heterogeneous quantum programming and compilation model.

The quantum compiler, such as OpenQL [1], compiles the quantum kernels in two steps. First, quantum kernels are compiled into QASM, or a similar format mathematically equivalent to the circuit model [28]. This format is hardware independent, which enables high-level optimizations and can be ported across different platforms. Most of the hardware constraints are taken into account in the second step, where the compiler performs qubit mapping and scheduling, and low-level optimization. The output is the analog pulse configuration for physical operations [30], the microcode defining QISA-level operations using the pulse-defined physical operations, and the quantum code consisting of eQASM instructions. The quantum code contains quantum instructions as well as auxiliary classical instructions to support comprehensive quantum program flow control including runtime feedback. After the host CPU has loaded the quantum code, microcode, and pulses into the quantum processor, the quantum code can be directly executed. In the rest of this paper, we focus on the quantum processor, i.e., the microarchitecture in charge of controlling qubits. The interaction between the classical processor and the quantum processor is a research topic outside the scope of this paper.

### B. Design Guidelines

The core requirement of eQASM is being executable on real hardware but not bound to a particular electronic control setup. The design of eQASM focuses on providing an comprehensive abstraction at the architecture level which can support the "quantum data, classical control" programming paradigm as well as some quantum experiments such as measuring the relaxation time of qubits ($T_1$ experiment). Also, because some experiments and radical compiler-based optimization techniques such as quantum optimal control [31, 32] may use uncalibrated or uncommon quantum operations, eQASM should support the usage of user-defined quantum operations. In stark contrast to classical computation where time is irrelevant to correctness, timing plays a key role in the control over qubits, i.e., in the execution of algorithms and experiments. To ensure

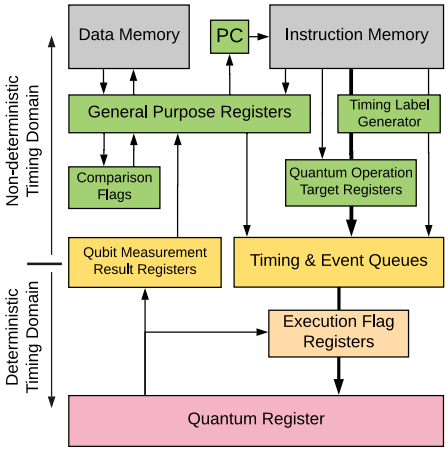


**Fig. 2.** Architectural state of eQASM. Arrows indicates the possible information flow. The thick arrows represent quantum operations, which read information from the modules passed through.

repeatability of quantum algorithms and experiments and reduce the risk of bug fixes or updates in software or hardware where timing is critical, timing of operations can be exposed at the architectural level as suggested by [33]. The design of eQASM is guided by five main principles:

1) eQASM should include classical instructions to support comprehensive quantum program flow control including runtime feedback;
2) eQASM should contain well-defined methods to specify the timing of quantum operations;
3) Low-level hardware information should be abstracted away from the eQASM assembly as much as possible to avoid eQASM being stuck to a particular hardware implementation;
4) The quantum operation issue rate is a potential bottleneck of the quantum microarchitecture, and should be addressed, e.g., by densely encoding the instructions such as done with SIMD and VLIW for classical architectures;
5) eQASM should be flexible to allow different quantum operations via configuration.

### C. Architectural State

Figure 2 illustrates the architectural state of the quantum processor.

*1) Classical Components:* The data memory can buffer intermediate computation results and serve as the communication channel between the host CPU and the quantum processor. The eQASM instructions are stored in the instruction memory, and the Program Counter (PC) contains the address of the next eQASM instruction to fetch. The general purpose register (GPR) file is a set of 32-bit registers, labeled as `Ri`, where `i` is the register address. The comparison flags store the comparison result of two general purpose registers which are used by comparison and branch related instructions (see Table I).

*2) Quantum Operation Target Registers:* Each quantum operation target register can be used as an operand of a quantum operation. Since most quantum technologies support physical operations applied on up to two qubits, there are two types of quantum operation target registers: single-qubit target registers for single-qubit operations [including measurement (MEASZ)], and two-qubit target registers for two-qubit operations.

Each single- (two-)qubit target register can store the physical addresses of a set of qubits (allowed qubit pairs). An *allowed qubit pair* is a pair of qubits on which we can directly apply a physical two-qubit gate. A single- (two-)qubit target register is labelled as `Si` (`Ti`), with `i` being the register address. eQASM does not define the format of target registers (see Section III-C for a discussion).

*3) Timing and Event Queues:* To support explicit timing specification of quantum operations, eQASM adopts a queue-based timing control scheme [1]. The timing and event queues are used to buffer timing points and operations generated from the execution of quantum instructions (see Section III-A). Together with the qubit measurement result registers, it separates the processor into two timing domains, the deterministic one and the non-deterministic one.

*4) Qubit Measurement Result Registers:* Each qubit measurement result register is 1-bit wide, and stores the result of the last finished measurement instruction on the corresponding qubit when it is valid (see Section III-F). It is labeled as `Qi`, where `i` is the physical address of the qubit.

*5) Execution Flag Registers:* Sometimes, the execution of a quantum operation depends on a simple combination of previous measurement results of this qubit [7, 22]. To this end, each qubit is associated with an execution flag register, which contains multiple flags derived automatically by the microarchitecture from the last measurement results of this qubit. The execution flag register file is used for fast conditional execution (see Section III-E).

*6) Quantum Register:* The quantum register is the collection of all physical qubits inside the quantum processor. Each qubit is assigned a unique index, known as the *physical address*. Since data in qubits can be superposed, eQASM does **not** allow direct access to the quantum data at the instruction level. Instead, users can measure qubits using measurement instructions and use the results later.

### D. Instruction Overview

To avoid the format of eQASM being stuck to a specific quantum technology implementation with particular properties, the definition of eQASM focuses on the assembly level and introduces basic rules of mapping the assembly code to binary instructions. The binary format is defined during the instantiation of eQASM targeting a concrete control electronic setup and quantum chip. This fact enables the eQASM assembly to be expressive while leaving considerable freedom to the (micro)architecture designer to pursue

**Table I.** Overview of eQASM instructions. The operator `::` concatenates the two bit strings.

| Type | Syntax | Description |
|---|---|---|
| Control | `CMP Rs, Rt` | **CoMP**are GPR `Rs` and `Rt` and store the result into the comparison flags. |
| | `BR <Comp. Flag>, Offset` | (**BR**anch) Jump to `PC + Offset` if the specified comparison flag is '1'. |
| Data Transfer | `FBR <Comp. Flag>, Rd` | (**F**etch **B**ranch **R**egister) Fetch the specified comparison flag into GPR Rd. |
| | `LDI Rd, Imm` | (**LoaD I**mmediate) Rd = sign_ext(Imm[19..0], 32). |
| | `LDUI Rd, Imm, Rs` | (**LoaD U**nsigned **I**mmediate) Rd = Imm[14..0]::Rs[16..0]. |
| | `LD Rd, Rt(Imm)` | (**LoaD** from memory) Load data from memory address `Rt + Imm` into GPR `Rd`. |
| | `ST Rs, Rt(Imm)` | (**STore** to memory) Store the value of GPR `Rs` in memory address `Rt + Imm`. |
| | `FMR Rd, Qi` | (**F**etch **M**easurement **R**esult) Fetch the result of the last measurement instruction on qubit `i` into GPR `Rd`. |
| Logical | `AND/OR/XOR Rd, Rs, Rt`<br>`NOT Rd, Rt` | Logical and, or, exclusive or, not. |
| Arithmetic | `ADD/SUB Rd, Rs, Rt` | Addition and subtraction. |
| Waiting | `QWAIT Imm`<br>`QWAITR Rs` | (**Q**uantum **WAIT I**mmediate/**R**egister) Specify a timing point by waiting for the number of cycles indicated by the immediate value `Imm` or the value of GPR `Rs`. |
| Target Specify | `SMIS Sd, <Qubit List>`<br>`SMIT Td, <Qubit Pair List>` | (**S**et **M**ask **I**mmediate for **S**ingle-/**T**wo-qubit operations) Update the single- (two-)qubit operation target register `Sd` (`Td`). |
| Q. Bundle | `[PI,] Q_Op [\| Q_Op]*` | Applying operations on qubits after waiting for a small number of cycles indicated by `PI`. |

microarchitectural practicability and performance.

An eQASM program can consist of interleaved quantum instructions and auxiliary classical instructions. An overview of the eQASM instructions is shown in Table I. Since the host CPU can provide classical computation power, auxiliary classical instructions are simple instructions to support the execution of quantum instructions. Complex instructions (e.g., floating-point instructions) are not included. The top part of Table I contains the auxiliary classical instructions. There are four types: *control*, *data transfer*, *logical*, and *arithmetic* instructions. These are all scalar instructions. The `FMR` instruction supports CFC and is explained in Section III-F.

The bottom part of Table I contains the quantum instructions. There are three types of instructions:

- Waiting instructions used to specify timing points,
- Quantum operation target register setting instructions, and
- Quantum bundle instructions, which consist of the specification of a small waiting time and multiple quantum operations.

These quantum instructions have several features based on the following four observations:

- Many quantum experiments, such as the $T_1$ experiment, require changing the timing of operations explicitly. Also, the timing of operations can significantly impact the fidelity of the final result as quantum errors accumulate during computation (Section V). eQASM can explicitly specify the timing of quantum operations to support quantum experiments and compiler-based timing optimization. The timing model is explained in Section III-A.
- To allow using different sets of quantum operations, quantum operations are specified by programmers at compile time via configuration (Section III-B). Only single- and two-qubit operations are allowed, and more-qubit operations should be decomposed by the compiler.
- To alleviate the quantum operation issue rate problem, we can reduce the required instruction issue rate $R_{\text{req}}$ by increasing the instruction information density at the architecture level, and/or increase the available instruction issue rate $R_{\text{allowed}}$ at the microarchitecture level. At the architecture level, eQASM reduces $R_{\text{req}}$ by adopting SOMQ execution, which supports applying a single quantum operation on multiple qubits (Section III-C), and a VLIW architecture, which can combine multiple different quantum operations into a quantum bundle (Section III-D). eQASM adopts the VLIW design because the microarchitecture implementation can be much simpler compared to a superscalar design. The tradeoff is that the compiler needs to perform more scheduling over the instructions as suggested by [34]. The microarchitecture can also introduce multiple-issue mechanisms as classical superscalar processors to increase $R_{\text{allowed}}$, which is out of the scope of this paper focusing on the architecture design.
- Two kinds of feedback are supported. Fast conditional execution performs a Go/No-go decision for every single-qubit operation based on the execution flag of the target qubit (Section III-E). To be more flexible, CFC allows programmers to define arbitrary feedback by redirecting the program flow based on the measurement results (Section III-F).

## III. Architecture

In this section, we construct the assembly syntax of quantum operations by introducing the aforementioned mechanisms.

### A. Timing Model

*1) Queue-based Timing Control:* eQASM adopts the queue-based timing control scheme proposed in [1] since it can support explicit timing specification. We briefly introduce this scheme and refer readers to the original paper for a detailed discussion.

In the queue-based timing control scheme, the execution of quantum instructions can be divided into a *reserve* phase in the non-deterministic timing domain and a *trigger* phase in the deterministic timing domain. A timeline is constructed by the reserve phase and consumed by the trigger phase: the result of executing quantum instructions in the reserve phase is consecutively creating new timing points on the timeline and associating events to them; the deterministic timing domain maintains a timer, and triggers all quantum operations associated with the timing point on the timeline that it reaches. Auxiliary classical instructions and mask setting instructions are not directly associated with timing points. The trigger phase is handled by the microarchitecture; we introduce the reserve phase in the following.

*2) Timeline Construction:* Quantum instructions fetched from the instruction memory form a quantum instruction stream. Instructions in the stream are executed in order; this constructs a timeline by generating consecutive timing points and assigning operations to them.

If the fetched instruction is a waiting instruction, `QWAIT Imm` or `QWAITR Rs`, a new timing point in the timeline is generated. The position of the new timing point is determined by the specification of the interval since the last generated timing point. The interval length comes from the immediate value `Imm` or GPR `Rs`. The first timing point of the timeline can be set by a dedicated instruction, or by an external trigger to the microarchitecture. Both waiting instructions use the unit cycle for the interval length.

If the fetched instruction is a quantum bundle instruction, the quantum operation(s) specified in the bundle instruction is associated with the last generated timing point. If multiple quantum operations are associated to the same timing point, these quantum operations will all start execution at that same timing point.

Based on our observation over some testbenches (see Section IV-D), short intervals between timing points are a common case. To improve the quantum operation issue rate, eQASM allows merging a `QWAIT PI` instruction followed by a quantum operation `<Quantum Operation>` into a single instruction

`[PI,] <Quantum Operation>.`

Square brackets `[···]` indicate that the content inside is optional. `PI` is short for `pre_interval`, which specifies a short interval between last generated timing point and the one when the operations in this instruction are to be triggered. It defaults to 1 if not specified. Value 0 is acceptable to both the `PI` and the waiting instructions, which means that the following timing point is identical to the last timing point.

*3) Example:* Assuming the durations of quantum operations `Q_OP0`, `Q_OP1`, `Q_OP2`, and `Q_OP3` all equal one-cycle time, the following code triggers these four operations back-to-back.

```
1 LDI  r0, 1      # r0 <- 1
2 Q_OP0
3 Q_OP1           # Default PI = 1
4 QWAITR r0       # Register-valued waiting
5 0, Q_OP2
6 QWAIT 0         # Equivalent to NOP
7 1, Q_OP3        # Explicity PI = 1
```

### B. Operation Definition & Decoding

eQASM does not define a fixed set of quantum operations at QISA design time, such as $\{H, T, \text{CNOT}, \cdots\}$. Instead, the available quantum operations can be configured by the programmer at compile time. Flexible quantum operation configuration is achieved through the configuration of the assembler, the microcode unit and the pulse generator (AWG) of the microarchitecture: on the one hand, the assembler is configured to translate a quantum operation, e.g., the $X$ gate, to the expected opcode, e.g., 0x01; on the other hand, the microcode unit translates the quantum opcodes into the expected microinstruction(s) using a microcode-based instruction decoding scheme. Each microinstruction represents one or more micro-operations, which are finally converted into pulses by the pulse generator with precise timing applying operations on qubits. The assembler, the microcode unit, and the pulse generator should be configured consistently at compile time. The pulses used in the configuration can be specified using a waveform specification format, such as OpenPulse [30] or an array as used in the open source quantum measurement environment PycQED [35].

### C. Address Mechanism

A quantum operation applied on multiple qubits is a common case. For example, quantum computation usually starts by preparing the superposition state from initialized qubits, which requires applying Hadamard gates on multiple qubits. eQASM uses SOMQ execution, which can apply a single quantum operation on multiple qubits at the same time. SOMQ is similar to classical single-instruction-multiple-data (SIMD) execution, with the operation target replaced by qubits. An instantiated eQASM can also be treated as an implementation of the previously proposed Multi-SIMD($k, d$) architecture [34] but removing the assumption of SIMD regions that in each region only a single quantum operation can be applied.

SOMQ is based on an indirect qubit addressing mechanism. The `SMIS` or `SMIT` instruction first defines a set of quantum operation target(s) in a quantum operation target register. Then a quantum operation can use the target register as the operand:

`<Operation Name>  <Target Register>.`

*1) Address of Allowed Qubit Pairs:* Since a two-qubit operation, such as a `CNOT` gate, can operate on its qubits differently, two qubits with different orders, i.e., `(Qubit A, Qubit B)` and `(Qubit B, Qubit A)`, are treated as different allowed qubit pairs. The term *quantum chip topology*

indicates the available qubits and allowed qubit pairs of a quantum chip (see Fig. 6 for an example). The quantum chip topology can be represented as a graph where each available qubit can be denoted as a vertex, and an allowed qubit pair as a directed edge. In the directed edge `(Qubit A, Qubit B)`, `Qubit A` is called the source qubit and `Qubit B` the target qubit of the pair.

*2) Translation from Assembly to Binary:* Since the efficiency of binary encoding the qubit list (qubit pair list) may depend on the target quantum chip topology, the designer can choose different binary encoding schemes for different target quantum processors during eQASM instantiation. In general, it is more efficient to put the address pairs in the instruction for a highly-connected quantum processor, while a mask format could be more efficient when the qubit connectivity is limited. For example, since at most two two-qubit gates can be applied and each qubit can be addressed with 3 bits in a fully connected 5-qubit trapped ion processor [36], only $2 \times 2 \times 3$ bits $= 12$ bits are required to specify the target of a two-qubit gate. This is more efficient than a mask of 20 bits with each bit in the mask indicating one of all 20 different allowed qubit pairs selected or not. In contrast, a mask of 6 bits is more efficient for the IBM QX2 [37], which also contains five qubits but has only six allowed qubit pairs.

*3) Example:* The following code sets the single-qubit target register `S7` to contain two qubits (0 and 1), and then applies an $Y$ gate on both qubits simultaneously.

```
1 SMIS S7, {0, 1}
2 Y    S7
```

The following code sets the two-qubit target register `T3` to contain two pairs of qubits (1, 3) and (2, 4), and then applies a `CNOT` gate on them.

```
1 SMIT T3, {(1, 3), (2, 4)}
2 CNOT T3
```

### D. Very Long Instruction Word

*1) Quantum Bundle Format:* Apart from SOMQ, different operations are also allowed to be applied on different qubits in parallel. eQASM can combine parallel quantum operations into a *quantum bundle* in a VLIW format. We define parallel quantum operations as operations starting at the same timing point, regardless of the duration of each operation. The format of a quantum bundle is:

```
[PI,] <Quantum Operation> [| <Quantum Operation>]*
```

The vertical bar `|` is used to separate different quantum operations in the same bundle. The asterisk $\star$ means the item in square brackets can repeat for $n \geq 0$ times.

*2) Translation from Assembly to Binary:* In the assembly code, an arbitrary number of quantum operations can be combined into a single quantum bundle. However, a single instruction can accommodate only a few quantum operations because of the limited instruction width. The *VLIW width* of eQASM characterizes the number of quantum operations that can be put in a single instruction word, which is defined during eQASM instantiation. Matching this, a single quantum bundle can be broken into multiple quantum bundle instructions with `PI` being 0. If the number of operations is not a multiple of the VLIW width, quantum no-operations (QNOP) fill up the last instruction. For example, the bundle `X S5 | H S7 | CNOT T3` can be decomposed by the assembler to two consecutive quantum bundle instructions `X S5 | H S7` and `0, CNOT T3 | QNOP` given a VLIW width of 2.

*3) Example:* In the code as shown in Fig. 3, the instruction `QWAIT 10000` initializes both qubits by idling them for 200 $\mu$s (assuming a cycle time of 20 ns). Line 6 applies a $Y$ gate on both qubits using SOMQ. Line 7 is a VLIW instruction, which applies an $X_{90}$ and $X$ gate on each qubit. In this paper, $X_{90}$ ($Y_{90}$) denotes the gate rotating the quantum state along the $x$- ($y$-)axis by a $\pi/2$ angle. $X_{\mathrm{m90}}$ ($Y_{\mathrm{m90}}$) denotes similar gates but with the rotation angle of $-\pi/2$. Line 8 measures both qubits using SOMQ. According to the `PI` value, the $Y$ gate happens immediately after the initialization, followed by the $X_{90}$ and $X$ gates 20 ns later and the measurement 40 ns later. The 1 $\mu$s waiting time (line 9) ensures no operations happening during the measurement.

```
1  SMIS S0, {0}
2  SMIS S2, {2}
3  SMIS S7, {0, 2}
4  ...
5  QWAIT    10000
6  0, Y     S7
7  1, X90   S0  | X  S2
8  1, MEASZ S7
9  QWAIT    50
10 ...
```

**Fig. 3.** Part of the code for a two-qubit *AllXY* experiment, which is used in validating eQASM in Section V.

### E. Fast Conditional Execution

Fast conditional execution allows executing or canceling a single-qubit operation when the micro-operation is triggered. The decision is made based on the value of a selected flag in the execution flag register corresponding to the target qubit. The value of the execution flag is derived by the microarchitecture using predefined combinatorial logic from the last measurement results of the same qubit. Once there returns a measurement result for a qubit, the corresponding execution flags are updated automatically. If the execution flag is '`1`', then the operation executes; otherwise, it is canceled. A selection signal is required for each micro-operation to select which execution flag to use, which can be generated by the microcode unit, or specified by an instruction field [38]. Except for the default execution flag that should always be '`1`', which and how many execution flags there are, should be defined during eQASM instantiation (see Section IV-B

for an example).

*Example:* In one instantiation of eQASM, the quantum operation `C_X` uses the execution flag which is '1' if and only if (**iff**) the last measurement result of the qubit is $|1\rangle$. Figure 4 shows the code for the active qubit reset experiment, where qubit 2 is put in an equal superposition using an $X_{90}$ gate after initializing it in the $|0\rangle$ state by idling it for 200 $\mu$s. After a measurement, a conditional `C_X` gate is applied to reset the qubit. Qubit 2 is measured again to read out the final state for verification.

```
SMIS   S2, {2}
QWAIT  10000
X90    S2
MEASZ  S2
QWAIT  50
C_X    S2 # the microcode unit is configured to
       # select the expected exe. flag for C_X
MEASZ  S2
```

**Fig. 4.** eQASM program for active qubit reset. The experimental result is shown in Section V.

### F. Comprehensive Feedback Control

CFC allows adjusting the program flow based on measurement results of any qubits to enable arbitrary user-defined feedback. This flexibility comes at the cost of longer feedback latency. We propose a three-step mechanism to implement CFC:

1) A measurement instruction is applied on the condition qubit `i`. At the moment that this measurement instruction is issued, `Qi` is invalidated. At the moment the measurement result is available, it is written in `Qi`. `Qi` turns back to valid if there are no more pending measurement instructions on qubit `i`.
2) The `FMR Rd, Qi` instruction fetches the value of the quantum measurement result register `Qi` into GPR `Rd`. If `Qi` is invalid, `FMR` should wait until `Qi` gets valid again. Thereafter, the value of `Qi` can be fetched into `Rd`. `Qi` remains valid until qubit `i` is measured again.
3) GPR `Rd` is then used in a `BR` instruction to select the program flow to follow. Note, multiple `FMR` and `BR` instructions can be combined to support more complex feedback logic.

*Example:* The eQASM program shown in Fig. 5 first measures qubit 1. If the measurement result is 1, a $Y$ gate is applied on qubit 0, otherwise, an $X$ gate is applied.

## IV. Instantiation & Implementation

This section introduces an instantiation, microarchitecture, and implementation of eQASM.

### A. Target Superconducting Quantum Chip

The quantum chip topology of the target seven-qubit superconducting quantum chip is shown in Fig. 6. It is part of a two-dimensional square lattice as proposed in [39]. It can implement a distance-2 surface code [40], which can

```
  SMIS  S0, {0}
  SMIS  S1, {1}
  LDI   R0, 1
  MEASZ S1
  QWAIT 30
  FMR   R1, Q1       # fetch msmt result
  CMP   R1, R0       # compare
  BR    EQ, eq_path # jump if R0 == R1
ne_path:
  X     S0           # happen if msmt result is 0
  BR    ALWAYS, next # this flag is always '1'
eq_path:
  Y     S0           # happen if msmt result is 1
next:
  ...
```

**Fig. 5.** eQASM program using CFC.

detect one physical error. In this figure, a vertex represents a qubit, and a directed edge represents an allowed qubit pair. Numbers besides the vertex (edge) are the addresses of qubits (allowed qubit pairs). For example, allowed qubit pair 0 has qubit **2** as the source qubit and qubit **0** as the target qubit. The feedlines are used to measure the nearby coupled qubits. Qubit 0, 2, 3, 5, and 6 (1 and 4) are coupled to feedline 0 (1). Each feedline has an input port and an output port. Besides, each qubit is connected to a microwave port and a flux port, which are not shown in Fig. 6.

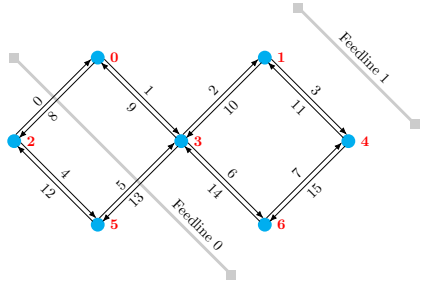


**Fig. 6.** Quantum chip topology of the target seven-qubit superconducting quantum chip.

Operations supported by this quantum processor include measurements, single-qubit $x$- or $y$-axis rotations, and a two-qubit controlled-phase (CZ) gate. A typical gate time is 20 ns for single-qubit gates and $\sim$ 40 ns for two-qubit gates. The duration of a measurement is typically 300 ns - 1 $\mu$s. A cycle time of 20 ns is used in this instantiation.

### B. Instantiation Design Space Exploration

To determine a suitable eQASM instantiation configuration for the target quantum processor [a single- (two-)qubit gate time of 1 (2) cycle(s), and a measurement time of 15 cycles], we perform analysis over three benchmarks using a quantum control architecture simulator derived from the previously proposed QPDO [41]. Because substantial time is spent on calibrating qubits before running applications with NISQ technology, the first benchmark we select is the widely-used calibration experiment randomized benchmarking (RB) [42], which might be limited by the high memory consumption when the required waveform for control is plainly stored in memory. Each qubit is subject to 4096

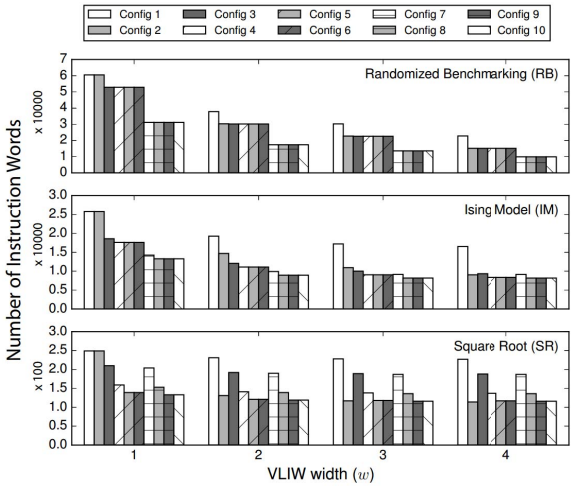


**Fig. 7.** Number of instructions for various architecture configurations for RB, IM, and SR.

single-qubit Clifford gates which have been decomposed into $x$ and $y$ rotations. Because every gate happens immediately following the previous one, randomized benchmarking cannot reveal timing patterns of quantum operations in real quantum algorithms, where the parallelism is limited by two-qubit gates. Addressing this, we also select two benchmarks from ScaffCC [11] as the representatives of small-scale quantum algorithms that might be executed with NISQ technology: a parallel algorithm (Ising model using 7 qubits, IM) which has $< 1\%$ two-qubit gates, and a relatively sequential algorithm (Grover's algorithm to calculate the square root using 8 qubits, which is the minimum number of qubits required, SR), which has $\sim 39\%$ two-qubit gates. The evaluation metric is the total number of instructions.

We investigate the impact of the VLIW width ($w$), three timing-specification methods, and SOMQ on the number of instructions. The three timing-specification methods include: the QuMIS fashion (specifying every timing point using separate `QWAIT` instructions, $ts_1$); including `QWAIT` in the quantum bundle instruction at the place of a quantum operation ($ts_2$); and using `PI` with various bit widths ($w_{\mathrm{PI}}$) to specify a small waiting time and using separate `QWAIT` instructions to specify longer waiting times ($ts_3$). The ten configurations and the simulation results are shown in Tab. II and Fig. 7, respectively. In the following analysis, Config 1 with $w = 1$ is chosen as the baseline.

**Table II.** Ten configurations used to investigate the binary format for this instantiation.

| | **Time Spec.** | $w_{\mathrm{PI}}$ | **SOMQ** |
|---|---|---|---|
| Config 1 | $ts_1$ | - | No |
| Config 2 | $ts_2$ (min $w = 2$) | - | No |
| Config 3/4/5/6 | $ts_3$ | 1/2/3/4 | No |
| Config 7/8/9/10 | $ts_3$ | 1/2/3/4 | Yes |

By increasing $w$ from 1 to 4 with Config 1, the number of instructions can be reduced up to 62% (RB). Benchmarks with substantial parallelism (RB and IM) benefit more from a big $w$. The instruction reduction in SR ($\sim 8\%$) indicates that large $w$ slightly improves quantum applications with limited parallelism.

Config 2 requires a minimum $w$ of 2 to distinguish $ts_2$ from $ts_1$. Compared with Config 1, by including the `QWAIT` operation as part of a quantum bundle instruction, Config 2 can reduce the number of instructions by 20 - 33% (RB), 24 - 45% (IM), 43 - 50% (SR) by varying $w$ from 2 to 4. SR benefits most because of two reasons. First, due to its sequential nature, it has relatively more `QWAIT` instructions. Second, limited parallelism in this algorithm leaves potential VLIW slots unused, which can be filled by `QWAIT` instructions.

Config 3 can reduce the number of instructions by 13 - 33% for RB and 28 - 44% for IM with $w$ varying from 1 to 4 compared with Config 1. Since the intervals between operations in RB and IM are mostly close to 1, further increasing $w_{\mathrm{PI}}$ up to $4$ bits introduces marginal benefit. Config 3 reduces the number of instructions of SR by $\sim 17\%$ regardless of $w$. Further increasing $w_{\mathrm{PI}}$ to 3 or $4$ bits can reduce the number of instructions of SR by up to 48%. Like SR, quantum algorithms are scheduled to be executed in a time as short as possible. The result of Config 3-6 suggests that most of the waiting time is short and can be encoded in a 3-bit `PI` field.

For Config 7/8/9/10, our analysis assumes that the target registers can always provide the required qubit (pair) list, and therefore shows the theoretical maximum benefit that can be obtained by SOMQ. Compared to Config 3/4/5/6, SOMQ can introduce a maximum reduction of 42% (Config 8, $w = 2$) in the number of instructions for RB, while it can only reduce at most 4% instructions for SR (Config 8, $w = 1$). Regardless of $w_{\mathrm{PI}}$, SOMQ can help reduce the number of instructions of IM by $\sim$ 24, 19, 9, and $2\%$ for different $w$. This fact suggests that SOMQ is more effective for highly parallel applications, especially when $w$ is small. An application that would benefit significantly from SOMQ is quantum error correction, which requires performing well-patterned error syndrome measurements repeatedly presenting high parallelism. As not shown in the figure, we also analyzed the number of effective quantum operations in each quantum bundle for Config 9, which is 1.795, 2.296, and 3.144 for RB, 1.485, 1.622, and 1.623 for IM, and 1.118, 1.147, and 1.147 for SR with $w$ varying from 2 to 4, respectively. It indicates that with the existence of SOMQ, $w > 2$ is not highly required for many quantum applications (RB is a special case with extreme parallelism).

As a result of the analysis, our eQASM instantiation adopts Config 9 ($ts_3$, $w_{\mathrm{PI}} = 3$, SOMQ) with $w = 2$. A width of 32 bits is used by all instructions for the memory alignment. Two instruction formats are used: the single format with the highest bit being '`0`' and the bundle format with the highest bit being '`1`'. Single format instructions use the other 31 bits to encode a single instruction, including all

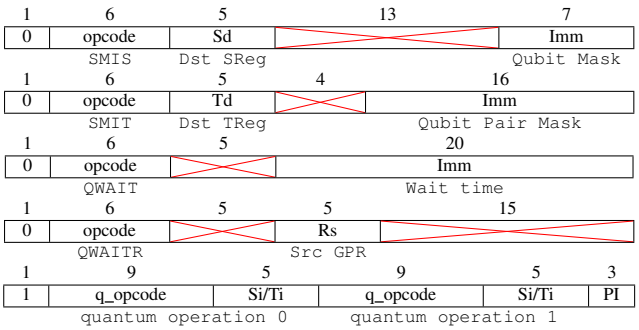


**Fig. 8.** Format of the SMIS and SMIT (top two), QWAIT and QWAITR (middle two), and quantum bundle (bottom) instruction.

auxiliary classical instructions, and SMIS, SMIT, QWAIT(R) instructions. For brevity, we only present the format of quantum instructions as shown in Fig. 8.

There are 32 single- (two-)qubit target registers, and the target register address width is 5 bits. The target registers use a mask format. The mask is 7- (16-)bit wide in the single- (two-)qubit target register. Each bit in the mask of the value '1' indicates that the corresponding qubit (allowed qubit pair) is selected. In the QWAIT(R) instruction, only the least significant 20 bits of the Imm field or GPR Rs are used to specify the waiting time. In the quantum bundle instruction, each quantum operation occupies 14 bits and the q_opcode is 9 bits.

### C. Microarchitecture

QuMIS is implemented by the control microarchitecture QuMA with codeword-based event control, queue-based event timing control and multi-level instruction decoding [1]. Adopting these three mechanisms, we redesign a quantum control microarchitecture, QuMA_v2, implementing the instantiated eQASM as shown in Fig. 9. It supports all features of eQASM. The classical pipeline maintains the PC and implements the GPR file and the comparison flags. The execution flag register is maintained by the fast conditional execution module. The classical pipeline fetches and processes instructions one by one from the instruction memory. All auxiliary classical instructions are processed by the classical pipeline while quantum instructions are forwarded to the quantum pipeline for further processing.

The timestamp manager processes the QWAIT(R) instructions and the PI field to generate timing points. The quantum pipeline contains a VLIW front end with two VLIW lanes, each lane processing one quantum operation. The SMIS (SMIT) instructions update the corresponding target registers in each VLIW lane. Inside each VLIW lane, the q_opcode is translated by the microcode unit into one micro-operation (labeled as $\mu\_op_{\rm s}$) for a single-qubit operation or two micro-operations (labeled as $\mu\_op_{\rm src}$ and $\mu\_op_{\rm tgt}$) for a two-qubit operation. $\mu\_op_{\rm src}$ ($\mu\_op_{\rm tgt}$) will be applied on the source (target) qubit of the target qubit pair. The configuration of

**Table III.** Definition of the micro-operation selection signal.

| Value | Operation to Select | Value | Operation to Select |
|---|---|---|---|
| '00' | None | '10' | $\mu\_op_{\rm tgt}$ |
| '01' | $\mu\_op_{\rm src}$ | '11' | $\mu\_op_{\rm s}$ |

the microcode unit is stored in the Q control store, which is implemented using a lookup table. The target register Si (Ti) is read for a single- (two-)qubit operation.

The quantum microinstruction buffer resolves the mask-based qubit address and associates the quantum operations to the last generated timing point. It resolves the qubit address in two steps.

First, the mask stored in Si (Ti) is translated into seven two-bit micro-operation selection signals $\mathrm{OpSel}_i$, where $i = 0, 1, \cdots, 6$, with each signal for one qubit. Table III lists the meaning of every case of the micro-operation selection signal. For single-qubit operations, $\mathrm{OpSel}_i$ is set to '11' ('00') if the $i$-th bit in the mask is '1' ('0'). For a two-qubit operation, $\mathrm{OpSel}_i$ is set to '00' if qubit $i$ is not contained in any selected allowed qubit pair. Otherwise, $\mathrm{OpSel}_i$ is '01' ('10') if the target qubit pair contains qubit $i$ as the source (target) qubit. Take qubit 0 as an example. It is connected to edges 0, 1, 8, and 9. When edge 0 or 9 (1 or 8) is selected in the mask, qubit 0 is the target (source) qubit and should be applied with $\mu\_op_{\rm tgt}$ ($\mu\_op_{\rm src}$). In other words, $\mathrm{OpSel}_0$ should be '10' ('01'), and can be generated using a simple OR ($\vee$) logic:

$$\mathrm{OpSel}_0 = (\mathtt{Ti}[0] \vee \mathtt{Ti}[9]) :: (\mathtt{Ti}[1] \vee \mathtt{Ti}[8]).$$

The assembler should check the validity of two-qubit target register values. For example, it is invalid if two edges connecting to the same qubit are selected in the same T register.

Second, based on $\mathrm{OpSel}_i$, either none or one micro-operation is output for qubit $i$. This step is fully parallel.

The operation combination module also works in a two-step fashion. First, since each VLIW lane outputs none or one micro-operation for each qubit, the operation combination module merges both micro-operations from both VLIW lanes. If both VLIW lanes output one micro-operation on the same qubit, an error is raised, and the quantum processor stops. Second, as explained in Section III-D, a long quantum bundle requires multiple quantum bundle instructions to describe it. The operation combination module buffers all micro-operations associated with the same timing point. Only when it detects that all quantum operations in the same quantum bundle have been collected, the operation combination module sends the buffered micro-operations to the device event distributor. This detection can be done, e.g., by recognizing a new timing point generated by the timestamp manager which is different to the one associated to the buffered micro-operations. Also, if two different quantum bundle instructions specify a quantum operation on the same qubit, an error is raised, and the quantum processor stops.

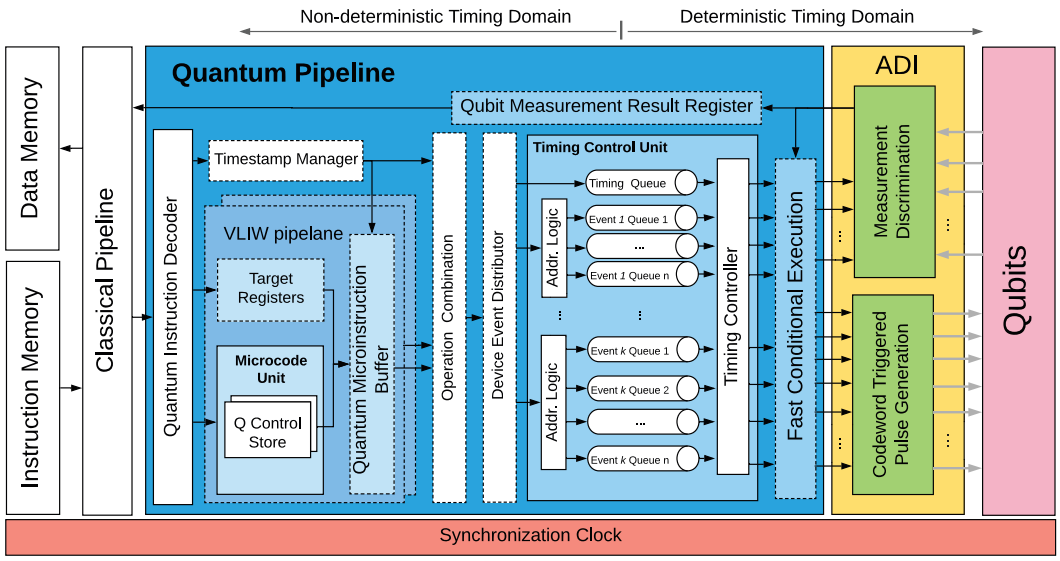


**Fig. 9.** Quantum microarchitecture (QuMA_v2) implementing the instantiated eQASM for the seven-qubit quantum processor. Dashed blocks highlight the newly introduced modules and mechanisms compared with QuMA proposed by [1].

As shown in Section IV-D, operating a qubit may require the collaboration of multiple electronic devices in the analog-digital-interface, and a single device may also control multiple qubits. Hence, the micro-operations should be reorganized into *device operations* to trigger the corresponding devices. The device event distributor reorganizes multiple micro-operations associated with the same timing label into different device operations. After that, each device operation with the associated timing label is buffered at an event queue of the timing control unit awaiting execution. The timing controller then triggers every device operation at its expected timing point.

After the device operations have been triggered by the timing controller, fast conditional execution is performed based on the selected execution flags of the target qubits. The execution flag selection signal comes from the microcode unit configured by the programmer. Only device operations for qubits of which the selected execution flag is '1' are released to the analog-digital interface (ADI). In this eQASM instantiation, four types of combinatorial logic are used to define the execution flags:

1) '1' (the default for unconditional execution);
2) '1' **iff** the last finished measurement result is $|1\rangle$;
3) '1' **iff** the last finished measurement result is $|0\rangle$;
4) '1' **iff** the last two finished measurements get the same result.

Note, the last finished measurement result refers to the result of the last finished measurement instruction on this qubit when these flags are used. It is irrelevant to the validity of the quantum measurement result register. Once there returns a measurement result for a qubit from the analog-digital interface, the fast conditional execution unit immediately update the execution flags corresponding to that qubit.

To support CFC, a counter `Ci` is attached to each qubit measurement result register `Qi`, with an initial value of 0. Once a measurement instruction acting on qubit $i$ is issued from the classical pipeline to the quantum pipeline, `Ci` increments by 1. If the measurement discrimination unit writes back a measurement result for qubit $i$, `Ci` decrements by 1. `Qi` is valid only when `Ci` is 0. If `Ci` is not 0 when the instruction `FMR Rd, Qi` is issued, the pipeline is stalled until `Ci` is 0. In this way, it is ensured that the instruction `FMR Rd, Qi` always fetches the result of the last measurement instruction acting on qubit $i$.

### D. Implementation

The hardware structure implementing the microarchitecture (Fig. 10) consists of a Central Controller responsible for orchestrating three modules containing slave devices for microwave control, flux control, and measurement.

The Central Controller is a digital device built with an Intel Altera Cyclone V SOC 5CSTFD6D5F31I7N Field Programmable Gate Array (FPGA) chip. The Central Controller implements the digital part of the microarchitecture (left to the ADI in Fig. 9). The timing controller and fast conditional execution module work at 50 MHz to get a cycle time of 20 ns. The other parts work at 100 MHz.

Single-qubit $x$ and $y$ rotations are performed by applying microwave pulses to the qubits. The pulses are generated by Zurich Instruments High Density Arbitrary Waveform Generators (HDAWG) and modulated using a Rohde & Schwarz (R&S) SGS100A microwave source. A custom-built vector switch matrix (VSM) is responsible for duplicating and routing the pulses to the respective qubits as well as tailoring the waveforms to the individual qubits [43] using a qubit-frequency reuse scheme that allows for efficient scaling of the microwave control module [39].

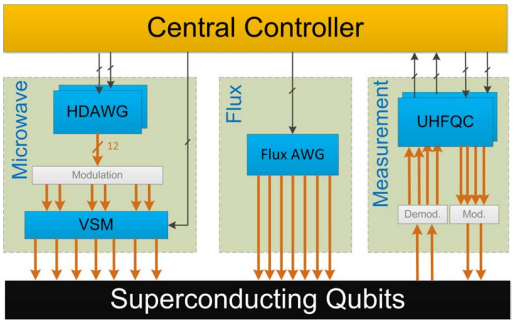


**Fig. 10.** Hardware structure implementing the instantiated eQASM for the seven-qubit superconducting quantum processor. Thin (thick) lines represent digital (analog) signals.

Flux pulses that implement two-qubit CZ gates and single-qubit $z$ rotations are performed by applying pulses generated by an HDAWG on the dedicated flux lines for each qubit.

The measurement discrimination unit is implemented using two Zurich Instruments Ultra-High-Frequency Quantum Controllers (UHFQC) connected to the two feedlines shown in Fig. 6. The UHFQC has two analog outputs that can be used to generate the measurement pulses and two analog inputs to sample the transmitted signals from which the UHFQC can infer the measurement result. The measurement pulses going to (coming from) the qubits are modulated (demodulated) using a single R&S SGS100A. All analog ports operate at $1.8$ GSa/s allowing for simultaneous measurement of up to 9 qubits per feedline using frequency multiplexing techniques.

The Central Controller connects to the UHFQCs and HDAWGs via a 32-bit digital interface working at $50$ MHz. Since measurement results are sent from the UHFQC to the Central Controller, $16$ bits of the connection are sent from the Central Controller to the UHFQC and the other $16$ bits the other way around. All operations on UHFQCs and HDAWGs are codeword triggered. The routing of microwave pulses by the VSM is controlled through seven digital signals with a sampling rate of 400 MSa/s.

## V. Experiment

Since the target seven-qubit quantum chip is still under test at the time of writing, we replaced the quantum chip of this microarchitecture with a two-qubit superconducting quantum processor to validate the eQASM design. The two qubits are interconnected and coupled to a single feedline. A configuration file is used to specify the quantum chip topology with the two qubits renamed as qubit 0 and 2. It is used by the quantum compiler and the assembler. eQASM programs used to perform the experiments as described below are all compiled from OpenQL descriptions with corresponding quantum operation configuration.

We first used eQASM to perform some single-qubit calibration experiments which utilize uncalibrated operations. For example, the Rabi oscillation [44] applies an $x$-rotation pulse on the qubit after initialization and then measures it. A sequence of fixed-length $x$-rotation pulses with variable amplitudes are used. Each pulse in the sequence is uploaded to the codeword triggered pulse generation unit of the microarchitecture and configured to be an operation in eQASM. As a result, this experiment calibrated the amplitude of the $X$ gate pulse. Together with other experiments, the fidelity of single-qubit quantum operations used later reached 99.90% as measured in the following RB experiment. It is worth mentioning that we observed considerable speedup in performing these experiments with the eQASM control paradigm in practice.

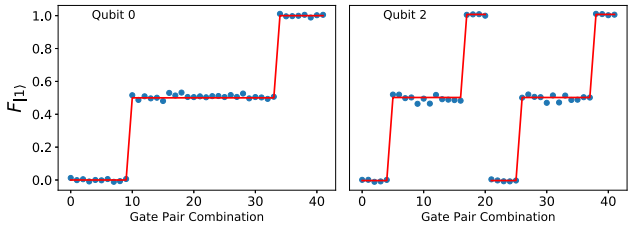


**Fig. 11.** Two-qubit *AllXY* result, corrected for readout errors.

eQASM is then configured to include single-qubit gates $\{I, X, Y, X_{90}, Y_{90}, X_{\mathrm{m90}}, Y_{\mathrm{m90}}\}$ and a two-qubit CZ gate for the following experiments. The *AllXY* experiment is typically used to calibrate single-qubit gates. In *AllXY*, pairs of single-qubit gates are chosen from the set $\{I, X, Y, X_{90}, Y_{90}\}$ and applied in such a way that the expected measurement outcomes produce a characteristic staircase pattern that is highly sensitive to gate errors (red line in Fig. 3). In the two-qubit *AllXY* experiment, the control pulses are applied on each qubit simultaneously. The sequence is modified to distinguish the qubits on which it is applied: each gate pair in the sequence is repeated on the first qubit while the entire sequence is repeated on the second qubit. The fidelity of qubit to the $|1\rangle$ state can be extracted by averaging the measurement results for each gate pair over $N$ rounds and correcting for readout errors. The eQASM program for one routine of this experiment is shown in Fig. 3. Figure 11 shows the final measurement result of the entire experiment (blue dots), which matches well with the expectation (red line). This demonstrates that the timing control, SOMQ, and VLIW of eQASM work properly in the experiment.

To evaluate the impact of the timing of operations on the error rate, we use single-qubit randomized benchmarking, a technique that can estimate the average error rate for a set of operations under a very general noise model [42, 45]. In this experiment, a sequence of $k$ random Clifford gates are applied on a qubit initialized in the $|0\rangle$ state. Before measurement, a Clifford is chosen that inverts all preceding operations so that the qubit should end up in the $|0\rangle$ state with survival probability $p(k)$. By performing this experiment for different $k$ and averaging over many randomizations, the Clifford fidelity $F_{\mathrm{Cl}}$ can be extracted

from the exponential decay. Because each Clifford gate is decomposed into primitive $x$- and $y$-rotations the gate count is increased by 1.875 on average. The average error rate per gate, $\epsilon$, is then calculated as $\epsilon = 1 - F_{\mathrm{Cl}}^{1/1.875}$.

Single-qubit randomized benchmarking was performed for different intervals between the starting points of consecutive gates (320, 160, 80, 40, and 20 ns). As shown in Fig. 12, the average error per gate decreases by a factor of $\sim 7$, from 0.71% to 0.10% when decreasing the interval from 320 ns to 20 ns. This demonstrates the significant impact of timing on the fidelity of the final computation result, which substantiates the requirement of explicit specification of timing at QISA level to enable platform-specific optimization and especially scheduling by the compiler.

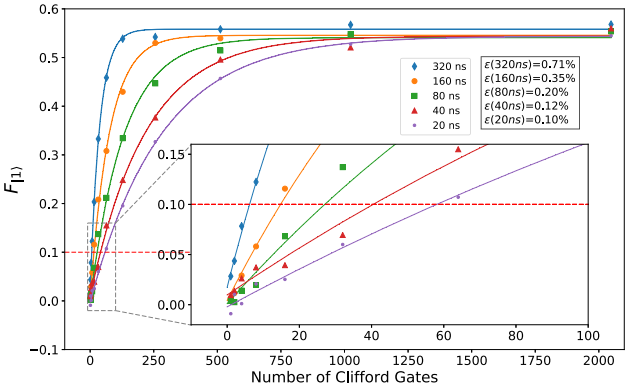


**Fig. 12.** Single-qubit randomized benchmarking results for different intervals between gates. Dashed line indicates a 10% error rate for visual reference.

Fast conditional execution is verified by the active qubit reset experiment with qubit 2 using the code as shown in Fig. 4. We find the probability of measuring the qubit in the $|0\rangle$ state after conditionally applying the `c_x` gate to be 82.7%, limited by the readout fidelity. We verified CFC by connecting the Central Controller and the UHFQC. The eQASM program used is shown in Fig. 5. The UHFQC is programmed to generate alternative mock measurement results for qubit 0. The alternation between $X$ and $Y$ operations is verified by detecting the output digital signals using an oscilloscope. We also measured the feedback latency of fast conditional execution and CFC, which are $\sim 92$ ns and $\sim 316$ ns, respectively. The feedback latency is defined as the time between sending the measurement result into the Central Controller and receiving the digital output based on the feedback from the Central Controller.

As a proof of concept of performing quantum algorithms using eQASM, we executed a two-qubit Grover's search algorithm [46]. The algorithmic fidelity, i.e., correcting for readout infidelity, is found to be 85.6% using quantum tomography with maximum likelihood estimation. This fidelity is limited by the CZ gate.

## VI. Conclusion

In this paper, we have proposed eQASM, a QISA that can be directly executed on a control microarchitecture after instantiation. With runtime feedback, eQASM supports full quantum program flow control at the architecture level. With efficient timing specification, SOMQ, and VLIW, eQASM alleviates the quantum operation issue rate problem and presents better scalability. Quantum operations in eQASM can be configured at compile time instead of QISA design time, which leaves ample space for compiler-based optimization.

As validation, eQASM was instantiated into a 32-bit instruction set targeting a seven-qubit superconducting quantum chip, and implemented using the quantum microarchitecture QuMA_v2. eQASM was verified by several experiments with this microarchitecture performed on a two-qubit chip.

Incorporating the quantum chip topology, timing, and user-defined operations, eQASM tries to provide a comprehensive abstraction of quantum hardware at the cost of portability. The assembly and binary of eQASM can be used separately for different purposes. Although the binary incorporates the quantum chip topology, the assembly code does not. While the timing information binds the code to one specific platform, by removing the timing information in the assembly, the program semantics can be retrieved. This program can be further remapped and scheduled in eQASM and be translated into another executable targeting another hardware platform. A set of standard quantum gates might form the bottleneck for aggressive optimization. In this case, eQASM can be configured with required operations to represent the quantum application to pursue optimal performance. To exploit the potential, eQASM can be also used as an intermediate representation by the compiler to perform platform-specific optimization.

Future work will include performing verifying comprehensive feedback control with qubits and controlling the originally targeted seven-qubit superconducting quantum processor with the implemented microarchitecture. Also, it will be interesting to instantiate eQASM to control other quantum processors, including superconducting quantum processors with a different quantum chip topology, and altogether different quantum hardware, such as spins in quantum dots [47], nitrogen vacancy centers [48].

## Acknowledgment

We thank C. C. Bultink, L. Lao, and M. Zhang for discussions while designing and implementing eQASM, S. Poletto, N. Haider, D. J. Michalak, and A. Bruno for designing and fabricating the two-qubit quantum chip used in experiments, T. van der Laan, and J. Veltin for project management when implementing the Central Controller, Y. Shi for discussions on feedback control. We acknowledge funding from the China Scholarship Council (X. Fu), Intel

Corporation, an ERC Synergy Grant, and the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via the U.S. Army Research Office grant W911NF-16-1-0071. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

## REFERENCES

[1] 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, "An experimental microarchitecture for a superconducting quantum processor," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE/ACM, 2017, pp. 813–825.

[2] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982.

[3] I. Kassal, J. D. Whitfield, A. Perdomo-Ortiz, M.-H. Yung, and A. Aspuru-Guzik, "Simulating chemistry using quantum computers," *Annual Review of Physical Chemistry*, vol. 62, pp. 185–207, 2011.

[4] J. Preskill, "Quantum computing in the NISQ era and beyond," *arXiv:1801.00862*, 2018.

[5] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, vol. 87, p. 307, 2015.

[6] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, pp. 527–586, 2004.

[7] D. Ristè, C. C. Bultink, K. W. Lehnert, and L. DiCarlo, "Feedback control of a solid-state qubit using high-fidelity projective measurement," *Physical Review Letters*, vol. 109, p. 240502, 2012.

[8] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels," *Physical Review Letters*, vol. 70, p. 1895, 1993.

[9] A. Paetznick and K. M. Svore, "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries," *Quantum Information & Computation*, vol. 14, pp. 1277–1301, 2014.

[10] A. Y. Kitaev, "Quantum measurements and the Abelian stabilizer problem," in *Electronic Colloquium on Computational Complexity*, 1996.

[11] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

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

[13] S. Liu, X. Wang, L. Zhou, J. Guan, Y. Li, Y. He, R. Duan, and M. Ying, "$Q|SI\rangle$: A quantum programming environment," in *Symposium on Real-Time and Hybrid Systems*. Springer, 2018, pp. 133–164.

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

[15] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, p. 180, 2017.

[16] M. Ying, *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[17] D. Wecker and K. M. Svore, "LIQUi$|\rangle$: A software design architecture and domain-specific language for quantum computing," *arXiv:1402.4467*, 2014.

[18] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: an open source software framework for quantum computing," *arXiv:1612.08091*, 2016.

[19] IBM Q team, "IBM Quantum Experience," https://www.research.ibm.com/ibm-q/, 2018.

[20] Rigetti, "Rigetti Forest," https://www.rigetti.com/forest, 2018.

[21] Alibaba, "Alibaba Quantum Computing Cloud," http://quantumcomputer.ac.cn, 2018.

[22] C. C. Bultink, M. A. Rol, T. E. O'Brien, X. Fu, B. Dikken, R. Vermeulen, J. C. de Sterke, A. Bruno, R. N. Schouten, and L. DiCarlo, "Active resonator reset in the nonlinear dispersive regime of circuit QED," *Physical Review Applied*, vol. 6, p. 034008, 2016.

[23] N. Ofek, A. Petrenko, R. Heeres, P. Reinhold, Z. Leghtas, B. Vlastakis, Y. Liu, L. Frunzio, S. M. Girvin, L. Jiang, M. Mirrahimi, M. H. Devoret, and R. J. Schoelkopf, "Extending the lifetime of a quantum bit with error correction in superconducting circuits," *Nature*, vol. 536, p. 441, 2016.

[24] L. Hu, Y. Ma, W. Cai, X. Mu, Y. Xu, W. Wang, Y. Wu, H. Wang, Y. Song, C. Zou, S. M. Girvin, L.-M. Duan, and L. Sun, "Demonstration of quantum error correction and universal gate set on a binomial bosonic logical qubit," *arXiv:1805.09072*, 2018.

[25] 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, "A microarchitecture for a superconducting quantum processor," *IEEE Micro*, vol. 38, pp. 40–47, 2018.

[26] S. S. Tannu, Z. A. Myers, P. J. Nair, D. M. Carmean, and M. K. Qureshi, "Taming the instruction bandwidth of quantum computers via hardware-managed error correction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. IEEE/ACM, 2017, pp. 679–691.

[27] C. A. Ryan, B. R. Johnson, D. Ristè, B. Donovan, and T. A. Ohki, "Hardware for dynamic quantum computing," *arXiv:1704.08314*, 2017.

[28] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels, "cQASM v1.0: Towards a common quantum assembly language," *arXiv:1805.09607*, 2018.

[29] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, pp. 66–73, 2010.

[30] D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, J. Gomez, M. Hush, A. Javadi-Abhari, D. Moreda, P. Nation, B. Paulovicks, E. Winston, C. J. Wood, J. Wootton, and J. M. Gambetta, "Qiskit backend specifications for OpenQASM and OpenPulse experiments," *arXiv:1809.03452*, 2018.

[31] J. Werschnik and E. Gross, "Quantum optimal control theory," *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 40, p. R175, 2007.

[32] N. Leung, M. Abdelhafez, J. Koch, and D. Schuster, "Speedup for quantum optimal control from automatic differentiation based on graphics processing units," *Physical Review A*, vol. 95, p. 042318, 2017.

[33] E. A. Lee, "Computing needs time," *Communications of the ACM*, vol. 52, pp. 70–79, 2009.

[34] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *Proceedings of 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 445–456.

[35] Asaad, S. and Bultink, C. C. and Dickel, C. and Dikken, B. and Fu, X. and de Jong, S. and de Lange, G. and Langford, N. and Luthi, F. and Rol, M. A., "DiCarloLab PycQED_py3," https://github.com/DiCarloLab-Delft/PycQED_py3, 2018.

[36] S. Debnath, N. Linke, C. Figgatt, K. Landsman, K. Wright, and C. Monroe, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, pp. 63–66, 2016.

[37] 5-qubit backend: IBM Q team, "IBM Q 5 Yorktown backend specification V1.1.0," Retrieved from https://github.com/QISKit/qiskit-backend-information/tree/master/backends/yorktown/V1, 2018.

[38] S. Balensiefer, L. Kregor-Stickles, and M. Oskin, "An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures," in *Proceedings of 32nd International Symposium on Computer Architecture.* IEEE, 2005, pp. 186–196.

[39] R. Versluis, S. Poletto, N. Khammassi, B. Tarasinski, N. Haider, D. J. Michalak, A. Bruno, K. Bertels, and L. DiCarlo, "Scalable quantum circuit and control for a superconducting surface code," *Physical Review Applied*, vol. 8, p. 034021, 2017.

[40] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, p. 032324, 2012.

[41] L. Riesebos, X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels, "Pauli frames for quantum computer architectures," in *Proceedings of the 54th Annual Design Automation Conference*. ACM, 2017, p. 76.

[42] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical Review Letters*, vol. 106, p. 180504, 2011.

[43] S. Asaad, C. Dickel, N. K. Langford, S. Poletto, A. Bruno, M. A. Rol, D. Deurloo, and L. DiCarlo, "Independent, extensible control of same-frequency superconducting qubits by selective broadcasting," *npj Quantum Information*, vol. 2, p. 16029, 2016.

[44] M. D. Reed, "Entanglement and quantum error correction with superconducting qubits," Ph.D. dissertation, Yale University, 2013.

[45] J. M. Epstein, A. W. Cross, E. Magesan, and J. M. Gambetta, "Investigating the limits of randomized benchmarking protocols," *Physical Review A*, vol. 89, p. 062321, 2014.

[46] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM, 1996, pp. 212–219.

[47] T. F. Watson, S. G. J. Philips, E. Kawakami, D. R. Ward, P. Scarlino, M. Veldhorst, D. E. Savage, M. G. Lagally, M. Friesen, S. N. Coppersmith, M. A. Eriksson, and L. M. K. Vandersypen, "A programmable two-qubit quantum processor in silicon," *Nature*, vol. 555, p. 633, 2018.

[48] J. Cramer, N. Kalb, M. A. Rol, B. Hensen, M. S. Blok, M. Markham, D. J. Twitchen, R. Hanson, and T. H. Taminiau, "Repeated quantum error correction on a continuously encoded qubit by real-time feedback," *Nature Communications*, vol. 7, p. 11526, 2016.