A Brief Qualitative Evaluation of Quantum Instruction Set Architectures

Jack Heavey jch7jm@virginia.edu
Nate Hellmuth nah6qg@virginia.edu
Wei Qi wq4sr@virginia.edu
Matt Walsh mw6es@virginia.edu

***Abstract*—As successful implementations of quantum computing hardware have been introduced, a need has arisen for specialized Instruction Set Architectures (ISAs) to translate code from higher level languages into machine executable code. Initial implementations of these quantum ISAs (QISAs) have varied in the design choices and approaches that they take to low level quantum computing implementation. In this paper, we select three popular QISAs: Quil [9], OpenQASM [2] (pronounced Open-kasm), and eQASM [6] (pronounced ee-kasm) and qualitatively compare the implementations of these languages according to the specifications. We note the differences between QISAs and classical ISAs and how the common ISA evaluation methods are often insufficient for quantum computations. We additionally look at the control flow design of these three QISAs, which differs from how we would normally compare ISAs. We conclude that these three ISAs are similar in many ways and note particularly the lack of specialization that we expected/believe will occur as quantum computing becomes more commonplace.**

I. Introduction

As quantum computers continue to develop towards quantum supremacy [1], the ability to solve complex problems using quantum algorithms becomes more attainable. Current research spends much of its efforts developing and improving low-level hardware [4] or creating high level software (pyQuil, etc) [9] to improve the ease of design of quantum algorithms. However, much of this research abstracts away the translations that must occur between high level software and quantum hardware, leaving this research to be done by someone else. Like in classical computing, quantum instruction set architectures (QISA) abstractly define requirements for both the hardware and the compiler. These requirements can improve hardware design and reduce restrictions on high-level software languages. Code design can be more configurable and allows for better translation and execution of quantum algorithms.

The requirements and use cases of quantum computing are changing and with it so must the hardware, software and QISA. Advanced quantum algorithms, such as the variational quantum eigen solver [7], introduce new challenges in executing the algorithm successfully. One of those challenges is the need to have both classical and quantum computation capabilities. For successful execution, classical code along side quantum operations must execute deterministically [3]. Many current quantum architectures require timing to be defined and determined by the compiler, limiting the types of quantum algorithm that can be executed. In addition to coordinating real-time classical code, providing run time feedback will also be necessary. Without it, there cannot be any variation or control dependencies, limiting even further the types of quantum algorithms that can be performed.

There are a few QISA that have been recently developed. Most fall under one of two categories: high-level abstraction (Quil) [9] or low-level implementations with limited flexibility (OpenQASM) [2]. There are also some that exist in between the two (eQASM) [6], however they are not widely used. A high-level abstracted QISA like Quil provides a framework, but the framework is too far abstracted that it becomes difficult to use in the real world. On the opposite end, OpenQASM, implements a straightforward QISA that allows for easy use and for hand written code while serving as a lower-level compilation target for high level languages. OpenQASM, however, limits the programmer to the most basic quantum gates. eQASM sits between the two. It provides more abstraction than OpenQASM by allowing programmers to define arbitrary one and two-qubit quantum gates at compile time, but is defined well enough such that all available classical instructions and some quantum operations (waiting and target specify instructions) are already implemented [6].

In this paper, we introduce and evaluate three different QISA: Quil [9], OpenQASM (version 2) [2] and eQASM [6]. Though each is defined at different level of abstraction, the decisions made by each QISA design can be qualitatively evaluated and compared. We evaluate the QISA using classical trade offs shared by QISA design, and introduce an additional key design choice: control flow design. We consider the decisions across four main areas: instruction design, compiler and hardware requirements, register design, and finally control flow and feedback design. After the evaluation, we observe a few possibilities of the needs of future QISA designs.

The rest of the paper is laid out as follows. Section II through IV introduce the three QISA being evaluated. Section V is where the design choices of the various QISA will be evaluated. We provide some insights into future QISA design in section VI and conclude in section VII.

II. Quil

Quil [9] is an instruction-based language written for state control of a Quantum Abstract Machines (QAM). Quil is a flexible language that can be written into directly, used as an intermediate step for running classical programs on QAMs, or can be a translation target for compilers or higher level languages. In addition to traditional operations, Quil defines a number of operations that directly affect the quantum state of the QAM that is being operated on. Quil acts on the atomic

objects of Qubits as well as classical bits and classical memory caches in order to provide full Turing functionality. Additional operations include:

- applying arbitrary quantum gates.
- defining quantum gates as (optionally parameterized) complex matrices.
- defining quantum circuits as sequences of other gates and circuits (bit- or qubit-parameterized).
- expanding quantum circuits.
- measuring qubits and recording the measurements into memory.
- synchronizing execution of classical and quantum algorithms.
- branching based on the value of bits in classical memory.

In addition to Quil, Rigetti has developed PyQuil, a Python library for programming in Quil directly [9].

A. Classical Operations

Quil incorporates a number of standard classical unary and binary bitwise instructions such as the operations **True**, **False**, **Negation**, **Conjunction**, **Disjunction**, **Copy**, and **Exchange**, as well as the NOP operation that only affects the program counter and not the state of the machine. These operations define and expand on the Turing completeness of the Quil language. Because the NOP does not have any dependencies on qubits, it can be used to analyze programs in a deterministic execution. Additionally, the authors have used NOPs to break up parallelization of instructions when desired [9]. Additionally, Quil supports conditional and unconditional jumps to different points in the code for branch mechanics. The interesting operations, however, are the operations that interact with the qubits.

B. Gates and Circuits

Quil defines two different types of gates, static gates, which are operators in $U(2^N)$, and parametric gates, which are functions from the complex space $\mathbb{C}^N \rightarrow U(2^N)$ [9]. These two gates create the groundwork on which the rest of the quantum operations are based. Matrices can be written to these gates by building them up using a variety standard mathematical operations, including the standard addition/subtraction/multiplication/division/exponentiation, more advanced mathematical functions such as sin(), cos(), sqrt(), exp(), cis() ($\cos(\theta) + i\sin(\theta)$), and numerical constants e and i. Gates are defined with the DEFGATE instruction followed by the desired matrix entries. Static gates are defined with only mathematical operations, while parametric gates can also be defined using unnamed parameters for functions to be used later. Additionally, circuits can be defined to run a sequence of instructions in a subroutine using the DEFCIRCUIT operation followed by a list of instructions. The standard quantum gates provided by Quil includes one-qubit, two-qubit and three-qubit gates. [9].

C. Measurement

Quil provides two forms of measurement [9]: measurement-for-effect, and measurement-for-record.

Measurement-for-effect is a measurement performed on a single qubit. The only purpose is to change the state of the quantum system.

Measurement-for-record is also a measurement performed on a single qubit. The state of the quantum system is changed and the measurement result is recorded in classical memory.

D. Program Control

The program control in Quil is determined by the state of the program counter. The program counter, which the authors designate as κ, is similar to the program counter in a classical CPU. It contains the address of the next instruction to be fetched and executed. It also shows if the program has halted. Every instruction, except for a few, have the effect of incrementing the program counter. The instructions that do not increment κ are:

- Conditional and unconditional jumps.
- The instruction HALT, which terminates program execution and assigns $\kappa \leftarrow |P|$.
- The last instruction in the program, after which the program execution is terminated as if the HALT instruction was used.

Unconditional jumps are executed by the JUMP instruction. This instruction sets the program counter κ to the address of a particular instruction in the instruction memory.

Conditional jumps are executed by the JUMP-WHEN (resp. JUMP-UNLESS) instruction [9], which set κ to the index of a given jump target if the bit at a particular classical memory address is 1 (resp. 0), and to $\kappa + 1$ otherwise. Additionally, because Quil supports both classical and quantum computations happening simultaneously in different parts of the architecture, sometimes the quantum computation needs to be suspended until some classical states are updated. To accomplish this, Quil implements a WAIT instruction, which stops the execution of quantum computation.

III. OpenQASM

OpenQASM is an imperative programming language for quantum operations [2] with the ability to universally describe the quantum computing circuit model. There are now three versions of OpenQASM. In this paper we will only evaluate OpenQASM 2, which is the most popular version [3].

A. OpenQASM Overview

OpenQASM acts as an interface language to enable experiments on small depth quantum circuits, which can be generated by a composer, hand-written or targeted by higher level languages [2]. It was designed with distinct phases:

1) Compilation
   Designed to take place on a classical computer, this phase compiles a described quantum algorithm into a quantum and classical mixed program. It is not necessary

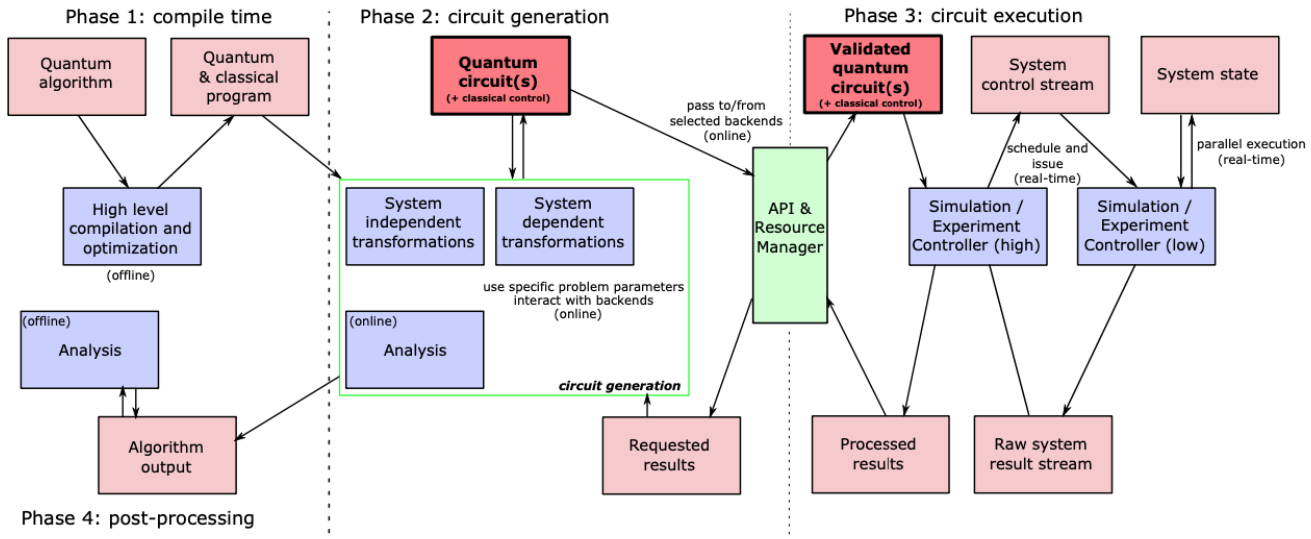

Fig. 1. Diagrams the various phases of OpenQASM. Blue blocks represent processes, red represent abstractions, and green represents the gateway to circuit execution through an API and resource manager. [2]

to know all parameters prior to this phase, but they should be known after compilation.

2) Circuit Generation
Once the quantum algorithm is compiled, circuit generation takes place on the classical computer. It takes the compiled program and converts it into a collection of quantum basic blocks, associated control instructions and classical object code needed. It can have some interaction with the quantum computer so this phase is an ”online” phase [2]
3) Execution
This phase is where quantum code is executed and the only phase that uses the quantum machine. The input from the circuit generation is passed into the quantum processors and instructions are streamed and executed at the appropriate times. The output is a collection of measurements taken during execution.
4) Post-processing
In this phase a classical computer will receive the results from the execution and can do any desired processing after the fact. This can include using the results to compile and run a new quantum algorithm.

More details on possible stages within phases can be seen in Figure 1.

B. OpenQASM Design

OpenQASM is designed at a much lower level than Quil. Like Quil, it does not support general classical instructions within the quantum execution phase. The execution of the classical instruction are limited to execute in between circuit execution of quantum operations in order to ensure proper coherence of the qubits [2]. Due to the limited classical code execution in the quantum machine, users or programs must actively participate between quantum executions. Each time a quantum execution completes the requested results are observed and can them pick the next quantum circuit to execute. The interaction between the classical and quantum hardware occurs through a defined API [2], which allows the user to write their own software that can interact with the hardware. Once the requested results are returned, post-processing can occur offline.

The quantum assembly language can represent the entirety of unrolled quantum algorithms, and unlike other QASM variants, does not use a discrete gate set. OpenQASM is designed to allow for parameterized gate set, which provides some flexibility in it’s implementation. It does, however, define a built-in gate basis of single qubit gates and one single two-qubit gate (CNOT) [2]. This sets the maximum number of qubits that can be operated on in a single gate to two. All other operations can be built up from the single qubit operations and the CNOT using a subroutine-like mechanism. Built-in gates and previously defined subroutines can be bundled into a new subroutine to create a new unitary gates. The new gate is able to take optional parameters allowing for limited code reuse. This allows more complex operations to be performed without defining complex built-in gates explicitly in the QISA design.

OpenQASM defines two types of registers: classical and quantum registers [2]. Quantum registers are designed to hold qubit representations. The quantum registers can then be operated on directly. For example, in a CNOT that compares a qubit to a quantum register of the same size, the CNOT is applied to the qubit and then every index within the quantum register. The classical registers are used for storing any classical data needed, primarily captured measurement information. In addition to data and memory addresses there is a single decision register used for conditional execution. The register holds an integer value that allows for execution of conditional quantum operations when it is a given value. Only quantum

operations are allowed to include an **if** statement prefacing it. This means that complex conditional quantum algorithms can be rewritten to use many conditional **if** statements [2].

IV. eQASM

eQASM is an executable QISA. It supports comprehensive quantum program flow control required by the "quantum data, classical control" paradigm [8].

eQASM adopts a heterogeneous quantum programming model. Namely, the architecture contains both a classical processor (the host CPU) and a quantum processor. The quantum processor is viewed as an accelerator for classically difficult tasks. A quantum algorithm contains a host program and several quantum kernels, which accelerate execution of the algorithm. The compilation is also hybrid. eQASM uses two compilers: a conventional compiler compiles the host program into classical code while a quantum compiler compiles the quantum kernels. The quantum kernels are first compiled into QASM format, which is machine independent. It is then compiled into the analog pulse configuration for physical operations [6], the micro code defining QISA-level operations and the quantum code containing eQASM instructions.

The architecture state of the quantum processor includes:

1) Data Memory.
   A buffer for intermediate result and also serves as communication channel between the host CPU and the quantum processor.
2) Instruction Memory & Program Counter.
   The instruction memory stores all the instructions and the program counter (PC) contains the address of the next instruction.
3) General Purpose Registers.
   A set of 32-bits registers.
4) Comparison Flags.
   The comparison flags store the comparison result of two general purpose registers.
5) Quantum Operation Target Registers.
   The quantum operation target register store the physical address of a set of qubits. Single-qubit target registers are for single qubit operations while two-qubit target registers are for two qubit operations.
6) Timing and Event Queues.
   They are used to buffer timing points and operations generated from quantum instructions.
7) Qubit Measurement Result Registers.
   A set of 1-bit registers. Each of them stores the measurement result of a corresponding qubit.
8) Execution Flag Registers.
   Each qubit is associated with an execution flag register where multiple flags are automatically derived by the last measurement.
9) Quantum Register.
   The collection of all physical qubits.

An eQASM program contains both quantum instructions and auxiliary classical instructions. Since the host CPU provides full classical computation power, the auxiliary classical instructions supported by eQASM contain only: control, data transfer, logical and scalar arithmetic instructions. There are three types of quantum instructions:

1) Waiting instructions, which are used to construct timeline,
2) Quantum operation target register setting instructions, which writes the addresses of a set of qubits into a quantum operation target register, and
3) Quantum bundle instructions, which contains multiple quantum operations. [6]

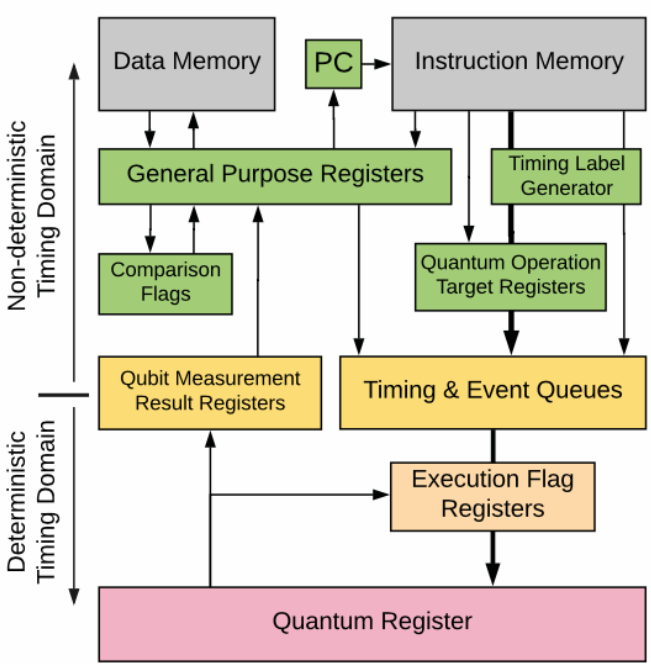

Fig. 2. Architectural state of eQASM with arrows indicating possible data flow [6]

A. Timing Model

The timing model of eQASM is queue based [6]. The execution of quantum instructions are divided into *reserve* phase and *trigger* phase. Based on the execution of instructions in the reserve phase, new time points can be created on the timeline. Instructions are then associated with a time point on the timeline. At a particular time point, all the instructions associated with it are triggered simultaneously.

Timeline can be constructed in different manners. The simplest type is sequential. Quantum instructions fetched from the instruction memory form a stream. Instructions in the stream are executed in order. Thus, consecutive time points are created with the instructions assigned sequentially.

If the fetched instruction is a waiting operation, then a new time point is created on the timeline at a specified interval after the last generated time point.

If the fetched instruction is a quantum bundle instruction, then all operations in the bundle are associated with the last generated time point.

B. Quantum Operation Definition

Instead of defining a fixed set of quantum operations, eQASM allows quantum operations to be be configured at compile time.

The assembler translates a quantum operation to opcode. The micro-code unit translates the quantum opcode into the micro-instruction [6], where each micro-instruction represents one or more micro-operations. They are translated to pulses with pulse generator. Thus, the set of quantum operations can be configured through the configuration of assembler, micro-code unit and pulse generator.

C. Address Mechanism

eQASM uses single-operation-multiple-qubits (SOMQ) execution, which is similar to classical single-instruction-multiple-data (SIMD) execution [5]. SOMQ is based on an indirect qubit addressing mechanism. The addresses of a set of quantum operation target qubits are first written into a quantum operation target register. Then, an operation is performed on all of the qubits specified in the quantum operation target register.

D. Very Long Instruction Word

Apart from SOMQ, eQASM also support parallelism of different quantum operations on different qubits, where parallel quantum operations are defined to be quantum operations triggered at the same timing point. Parallel quantum operations are combined into a quantum bundle in a VLIW format as shown below:

$$[\text{PI,}] < \text{Quantum Operation} > [| < \text{Quantum Operation} >]^*$$

E. Fast Conditional Execution

Fast conditional execution allows executing or cancelling an operation based on the value of a selected flag in the execution flag register corresponding to the target qubit. All the flags in the execution flag register are automatically updated once the measurement result of the corresponding qubit is available.

F. Comprehensive Feedback Control

With comprehensive feedback control, the program control flow can be adjusted based on the measurement result of any qubits. This flexibility costs longer feedback latency. The comprehensive feedback control is 3-step:

1) Apply the measurement instruction on the conditional qubit. Invalidate the corresponding qubit measurement result register. The measurement result is written to the corresponding measurement result register. If there is no pending measurement instruction on this qubit, the measurement result register turns back to valid.
2) Once the corresponding measurement result register turns to valid, its content is fetched into a general purpose register.
3) A branch instruction is issued to select the program follow based on the value of one or more general purpose registers.

V. Evaluation

Classical ISA design is well defined with specific key aspects to consider in the design of the ISA. Those key aspects are as follows:

1) Instruction Design
2) Compiler vs. Hardware Requirements
3) Register Design
4) Specialized Support

In QISA design however, there are a few alterations that need to be made to those key aspects. Code density, compiler vs. hardware requirements, and register design are generally unchanged when evaluating QISA choices, however they do present new considerations due to differing requirements of quantum operations. Specialized support does not translate as easily to QISA evaluations, as the act of quantum/classical hybrid programs is a specific form of specialized support. Since quantum algorithms require special considerations and architectures to enable control flow, we modify this aspect to evaluate a QISA control flow design. Since different designs allow for different types of feedback and conditional execution, it becomes an important aspect in determining use and complexity of QISAs.

For the evaluation of the three QISA, four general design choices are considered. First, instruction design, which includes instruction length, addressing capabilities, and RISC/CISC instruction type choices. Second, requirements of compiler and hardware are analyzed. Register design, such as width, depth and types, is considered third. Finally, we evaluate the QISA specific category, the control flow design of each QISA presented in this paper.

A. Instruction Design

Instruction design is an ambiguous term as it could mean a number of things. For this paper we focus on two main aspects of instruction design: instruction complexity and addressing modes, code density.

1) Addressing Modes and Instruction complexity: While instruction complexity and addressing modes are separate design choices, it is hard to separate the two in the scope of this paper as the abstractness of the QISA leaves little to be discussed for each individually. In the presented QISA, the abstracted nature allows for programmers to define their own gates. This means we cannot evaluate instruction complexity on its own since the available gates and the work they can do are not necessarily known. The two can be evaluated together as there are still base quantum and classical instructions in addition to restricted addressing options.

The addressing configuration for the three QISA are not significantly different, however the operations have different limitations. Both OpenQASM and eQASM allow for quantum gates that operate on one or two qubits. In OpenQASM, there is one built-in two-qubit gate, the CNOT, and a number of built-in arbitrary user defined single qubit gates. The programmer can define a subroutine using built-in gates or other subroutines. The subroutine takes parameters so instructions referencing subroutines can address multiple qubits.

QISA	Quil	OpenQASM	eQASM
Register Types	General Classical Registers General Quantum Registers	General Classical Registers General Quantum Registers Decision Register	General Classical Registers Quantum Operation Target Registers Measurement Result Registers Execution Flag Registers
Maximum Qubits for Single Gate	3	2	2
Uses VLIW	No	No	Yes
Has Conditional Execution	Yes	Yes	Yes

TABLE I
A QUICK VIEW OF THE KEY DIFFERENCES IN THE QISA DESIGNS

However, those subroutines are decoded into one and two qubit built-in gates when compiled. eQASM also compiles complex quantum operations down to micro-operations, but it does not limit the type of two-qubit operations available, leaving all quantum gate implementation up to the user during compilation. The quantum operations are limited to two qubits in order to include no more than the minimum number of operations to implement full quantum algorithms. This keeps base qubit operations as simple as possible while still allowing the programmer to expand the functionality of the ISA using self implemented gates. Quil however allows up to three-qubit operations, meaning in operations can be applied on up to 3 qubits. This allows for more complicated gates.

Operation can be defined by the programmers, therefore the final instruction complexity is decided by the programmer. We can evaluate the instruction complexity at the lowest level of quantum operations available. OpenQASM has the lowest complexity relatively. It only defines a single two-qubit operation, and arbitrary one-qubit operations [2]. Because of this, it is the simplest to implement of the three QISA. eQASM has flexible instruction complexity, but is still limited to one and two-qubit operations [6]. Quil has the highest complexity relative to the other two due to it's ability to operate on 3 qubits in one operation [9].

2) Code Density: Code density is the amount of space that an executable program takes up in memory. It is influenced by the instruction line size and the complexity of the instruction being issued. In a classical context, a higher density of code means there are reduced accesses to memory, since more instruction can be held in the instruction buffer at any given time. This can improve the performance of instruction fetch, but can complicate instruction decode. The dense instructions can be split and decoded into micro operations for the CPU to process and implement or it requires large registers which slows execution time.

A QISA faces the same trade offs when considering higher density architectures. Generally, the number of qubits that a quantum computer has is the number of operations that can technically be done at the same time. In practice this is rarely the case, but it provides a built in capability to run parallel operations. All three of the QISA presented in the paper have a different code density.

eQASM is designed to implement VLIW [6] that can contain multiple quantum operation instructions in a single instruction line. It also implements SOMQ which specifies one operations to be applied on multiple qubits. This allows for more compact code to be stored in memory. In addition, the compiled code will be smaller since more work is done in the hardware. The compiled binary does need as much information, so it will have a smaller size. Therefore, it will generally have the highest code density.

Unlike eQASM, Quil and OpenQASM do not have VILW formatted code and do less work in the hardware. More work is needed by the compiler, producing larger binary. In Quil, however there are more flexible quantum gates that allow operations on up to 3 qubits. With the high complexity of instruction, the density also increases as more work can be accomplished with a single instruction. OpenQASM requires more instructions to accomplish the same algorithms because the available gates are limited to CNOT and single qubit instructions. This means that OpenQASM is expected to have the lowest code density. Increasing the work needed by instruction fetch.

B. Compiler vs. Hardware Requirements

Within the three ISAs examined, they put different pressures on the compiler versus hardware to perform code optimizations and execution. Quil, which is generally being run on QAMs/QVMs (Quantum Virtual Machines) and does not specify any strict hardware dependencies, relies on compiler level optimizations to organize the Quil instructions into their optimal path. In this way, Quil is able to remain mostly hardware agnostic until execution, which only assumes the presence of quantum circuits [9].

Similarly to Quil, OpenQASM leans on the compiler to make all code optimization decisions, serving as a translation target from a compiler. Compilation of a language into OpenQASM is not even required to be on a quantum computer as long as the final executable is run on a physical quantum controller. This allows OpenQASM to be similarly hardware agnostic to Quil despite being a generally lower level expression. Instructions are run sequentially with no assumptions about the underlying quantum circuits save that they are present [2].

eQASM was designed with the principles of "quantum data, classical control" [6] to where it can be executed on real quantum hardware. With this in mind, the designers decided to abstract away "low-level hardware implementation" in order to avoid being limited to specific hardware designs and reducing the risk of hardware updates requiring a rewriting of the language. To this end, eQASM similarly relies on the compiler to correctly order instructions for optimal execution. Execution

simply assumes the presence of quantum circuits and a central controller that operates on three ”slave devices” that maintain and measure the states of the qubits [6].

The three different ISAs all making similar decisions to abstract away instruction optimization to the compiler allows for a much wider executable base of their code. As quantum research is still developing, this paradigm makes sense to ensure that their ISAs are still usable as advances are made within quantum hardware. However, the trade off for these three sets of instruction sets in usability comes in the form of optimization - more hardware specific designs are going to execute code much more quickly for large scale programs. This is not a large issue now as the field still develops, but we hypothesize that the researchers and professionals will quickly begin to develop more specialized ISAs for specific hardware implementations as the quantum computing field matures.

C. Register Design

Quil describes what can be done on quantum abstraction machine. Because of the abstractness of QAM, there is no description of register design. However, in QAM, there are a fixed but arbitrary number of qubits and classical bits which can be viewed as two types of registers.

In OpenQASM, there are three different types of registers: classical registers, quantum registers and decision register. Classical registers store classical bits while quantum registers store qubit. The single decision register is for conditional execution.

In eQASM, there are five different types of registers: general purpose registers, quantum operation target registers, qubit measurement result registers, execution flag registers and quantum register. The first four of them are all classical registers while the last one are registers of qubit. Quantum operation target registers allows eQASM to support SOMQ. Execution flag registers allow eQASM to support fast feedback execution. Qubit measurement result registers allow eQASM to support comprehensive feedback control.

We observe that all three ISA divide registers into classical registers and quantum registers while OpenQASM and eQASM have special-purpose classical registers. Theoretically, it is possible to have just general purpose classical registers and quantum registers while using a specific subset of general purpose classical registers for special purpose. The advantage is that architectures with only two different registers are simpler to implement. However, with more types of classical registers, there are potential trade-offs that can be exploited. Different tasks may induce different patterns of operations on the registers, allowing for task specific optimization of registers.

D. Control Flow Design

The control flow design of any QISA is largely reliant on the measurement system of that architecture and its support of conditional operations. eQASM proposes two techniques for a quantum control flow: fast conditional execution and comprehensive feedback control. Fast conditional execution uses execution flags that are set by the previous measurement of a qubit to determine whether a single qubit operation will be executed. It then measures the qubit and updates the flags accordingly. Comprehensive feedback control supports conditional operations on many qubits by first measuring a qubit and rendering it invalid while the measurement occurs. Once the measurement is complete, the qubit becomes valid and the value is stored in a register. The values in this register are then used to determine control flow in some user-defined manner. This adds flexibility to the control flow that, while adding latency via qubit measurements, allows for programmable feedback [6].

In the cases of Quil and OpenQASM, qubit measurement to classical memory and conditional statements based on classical bits are supported, but control flow design is less well defined when compared to eQASM. Quil has two unique measurement instructions: measurement-for-effect, which is used to measure a qubit solely for the purpose of altering the state of the system, and measurement-for-record, which stores the value of the measurement in classical memory. Quil also defines conditional and unconditional jumps that are executed based on the value of a classical bit. OpenQASM support single qubit measurements to store as single classical bits or entire quantum register measurements to be stored in classical registers. The only conditional statement supported by OpenQASM is **if** statements that execute quantum operations based on the value of classical register.

All other control flow design does not rely on qubit measurements and can therefore generally be handled by classical means in all three systems.

VI. Future Work

As more research is done in QISA design, and the requirements of quantum algorithms change, the decisions of the designed QISA change with it. The evaluation of Quil, OpenQASM, and eQASM presents the key aspects of consideration in the design, which makes the way for future research and designs of QISA. In particular, there has been little research in discovering the optimal implementation of these QISA. Specific decisions such as exact register sizing and depth, number of quantum processors, and other micro-architecture specific design provide optimizations that are currently missing.

Quantum architectures are also in need of the ability to run quantum and classical code in deterministic time [3]. Quantum algorithms are beginning to require classical supported quantum execution. This means that classical code needs to run and be completed in the correct timing use by the quantum operations. This is a difficult problem to solve as there must be a guarantee that the classical code will complete in time for the next quantum operation. This will also allow the processing and evaluation to happen closer to the quantum operations opening the door to increased possibilities in quantum algorithms.

VII. Conclusion

In this paper, we compare the implementations of three different sets of ISAs specifically used as compilation targets for quantum programs. These three instruction set architectures, while different, allow for the execution of both classical and quantum programs on quantum computers through low-level Assembly-like functions. The three languages all make design trade offs through their instruction design and register design, and we additionally develop a new method of comparing these ISAs qualitatively, which we call control flow design, a new design principle that is not used in classical machines. As the field of quantum computing moves forward, we expect to see more specialized ISAs to be developed beyond the three discussed here, and we believe that our evaluation methods can be used to make qualitative judgements on which ISA is the best to use for a given purpose.

Acknowledgements

We would like to thank Dr. Xu Yi for sharing his knowledge on the basics of quantum systems and the mechanisms surrounding quantum computing.

References

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5

[2] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017.

[3] A. W. Cross, A. Javadi-Abhari, T. Alexander, L. Bishop, C. A. Ryan, S. Heidel, N. de Beaudrap, J. Smolin, J. M. Gambetta, and B. R. Johnson, "Open quantum assembly language." Programming Languages Design and Implementation, 2021.

[4] N. P. de Leon, K. M. Itoh, D. Kim, K. K. Mehta, T. E. Northup, H. Paik, B. S. Palmer, N. Samarth, S. Sangtawesin, and D. W. Steuerman, "Materials challenges and opportunities for quantum computing hardware," *Science*, vol. 372, no. 6539, p. eabb2823, 2021.

[5] M. Flynn, "Some computer organizations and their effectiveness. ieee trans comput c-21:948," *Computers, IEEE Transactions on*, vol. C-21, pp. 948 – 960, 10 1972.

[6] X. Fu, L. Riesebos, M. Rol, J. Van Straten, J. Van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, V. Newsum, K. Loh, J. De Sterke, W. Vlothuizen, R. Schouten, C. García Almudever, L. Dicarlo, and K. Bertels, "eqasm: An executable quantum instruction set architecture," in *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, ser. Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, A. Louri and G. Venkataramani, Eds. United States: IEEE, 2019, pp. 224–237, 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019 ; Conference date: 16-02-2019 Through 20-02-2019.

[7] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, vol. 5, no. 1, p. 4213, Jul 2014. [Online]. Available: https://doi.org/10.1038/ncomms5213

[8] P. SELINGER, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, no. 4, p. 527–586, 2004.

[9] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2017.