SAND2020-4650C

Just Another Quantum Assembly Language (Jaqal™)

B. C. A. Morrison*†‡, A. J. Landahl*†‡, D. S. Lobser§,
K. S. Rudinger*, A. E. Russo*, J. W. Van Der Wall§, P. Maunz§
*Center for Computing Research, Sandia National Laboratories, Albuquerque, NM
†Center for Quantum Information and Control, University of New Mexico, Albuquerque, NM
‡Department of Physics and Astronomy, University of New Mexico, Albuquerque, NM
§Center for Microsystems Engineering, Science, and Applications, Sandia National Laboratories, Albuquerque, NM
Corresponding author: benmorr@sandia.gov

***Abstract*—QSCOUT is the Quantum Scientific Computing Open User Testbed, a trapped-ion quantum computer testbed realized at Sandia National Laboratories on behalf of the Department of Energy's Office of Science and its Advanced Scientific Computing (ASCR) program. Jaqal, for Just Another Quantum Assembly Language, is the programming language used to specify programs executed on QSCOUT. We describe the capabilities of the Jaqal language, our approach in designing it, and the reasons for its creation. To learn more about QSCOUT and the Jaqal language developed for it, please visit qscout.sandia.gov or send an e-mail to qscout@sandia.gov.**

***Index Terms*—physics, quantum mechanics, quantum computing**

I. Introduction

QSCOUT is the Quantum Scientific Computing Open User Testbed, a trapped-ion quantum computer testbed realized at Sandia National Laboratories on behalf of the Department of Energy's Office of Science and its Advanced Scientific Computing Research (ASCR) program. As an open user testbed, QSCOUT provides the following to its users:

- **Transparency**: Full implementation specifications of the underlying native trapped-ion quantum gates.
- **Extensibility**: Pulse definitions can be programmed to generate custom trapped-ion gates.
- **Schedulability**: Users have full control of sequential and parallel execution of quantum gates.

In order to provide these features, we must have a quantum assembly language designed around both the flexibility and the detailed control they require. We considered a wide variety of existing languages. However, they either were lacking on one of these points, or were too focused towards a particular other hardware model, which led us to develop our own. Due to the proliferation of such languages in this fledgling field, ours is named **Just Another Quantum Assembly Language**, or **Jaqal**. We attempt to combine key advantages of existing languages to support the needs of our current hardware while providing room for extension to a wide variety of future targets.

This material was funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research Quantum Testbed Program.

A. QSCOUT Hardware 1.0

The first version (1.0) of the QSCOUT hardware realizes a single register of qubits stored in the hyperfine clock states of trapped $^{171}\mathrm{Yb}^{+}$ ions arranged in a one-dimensional chain. Single and multi-qubit gates are realized by tightly focused laser beams that can address individual ions. The native operations available on this hardware include the following:

- Global preparation and measurement of all qubits in the z basis.
- Parallel single-qubit rotations about any axis in the equatorial plane of the Bloch sphere.
- The Mølmer–Sørensen two-qubit gate [4] between any pair of qubits, in parallel with no other gates.
- Single-qubit Z gates executed virtually by adjusting the reference clocks of individual qubits.

Importantly, QSCOUT 1.0 does not support measurement of a subset of the qubits. Consequently, it also does not support classical feedback. This is because, for ions in a single chain, the resonance fluorescence measurement process destroys the quantum states of all qubits in the ion chain, so that there are no quantum states onto which feedback can be applied. Future versions of the QSCOUT hardware will support feedback.

QSCOUT 1.0 uses Jaqal to specify quantum programs executed on the testbed. On QSCOUT 1.0, every quantum computation starts with preparation of the quantum state of the entire qubit register in the z basis. Then it executes a sequence of parallel and serial single and two-qubit gates. After this, it executes a simultaneous measurement of all qubits in the z basis, returning the result as a binary string. This sequence of prepare-all/do-gates/measure-all can be repeated multiple times in a Jaqal program, if desired. However, any adaptive program that uses the results of one such sequence to issue a subsequent sequence must be done with metaprogramming, because Jaqal does not currently support feedback. Once the QSCOUT platform supports classical feedback, Jaqal will be extended to support it as well.

B. Language Goals

To realize our objectives, the Jaqal quantum assembly language (QASM) fulfills the following requirements. While

many of them were inspired by other languages' design choices, this particular set was not available in any mainstream language. Thus, we developed Jaqal to combine the features most relevant to our specific application, and ideally to a variety of future platforms with similar goals.

- Jaqal fully specifies the allocation of qubits within the quantum register, which *cannot* be altered during execution. This explicit specification of which qubits will be used can be found in low-level languages like OpenQASM [3] and Cirq [2], but is lacking from many higher-level languages which often instead have language constructs for allocating "clean" qubits in a known state or "dirty" qubits in an unknown one [8].
- Jaqal requires the scheduling of serial and parallel gate sequencing to be fully and explicitly specified. Many quantum languages leave scheduling either completely unspecified, or constrained only by "barrier" statements, which prevent reordering of gates across them but allow gates between them to be executed in any order and/or simultaneously. Cirq [2] is a notable exception, allowing circuits to be split into "moments" in which all gates are executed in parallel, with each moment executed sequentially after the last. However, Jaqal's scheduling is even more flexible, allowing a sequence of gates to be placed in parallel with a single (longer duration) gate on a different qubit, for example.
- Jaqal can execute any native (built-in or custom) gate specified in any Gate Pulse File it references. There are many standards for pulse-level quantum programming, such as OpenPulse [5], already available. However, these standards do not integrate well with assembly-level languages. You cannot, for example, define a gate at the pulse level in OpenPulse and then call that gate from a OpenQASM program alongside the built-in native gates. Jaqal allows custom gates to be defined, and their implementation encapsulated so their use in a Jaqal program is identical to that of a built-in gate.
- Jaqal can be used to define composite gate 'macros' which are implemented by arbitrary parallel and/or serial combinations of native gates. This is a common feature across both QASM-like languages [3] and other quantum programming languages [7], [8]; Jaqal's syntax is very similar to that used by other QASM-like languages.
- Jaqal's built-in execution flow control is sufficient to concisely express common benchmarking circuit patterns such as repeated gate set tomography germs [1], while also being restricted enough to guarantee halting in bounded time. Some form of flow control is present in most quantum languages, usually conditional execution. Looping constructs like Jaqal's are rarer; Quil does have loops, implemented via jump instructions [7], but—unlike Jaqal—its jump instructions can lead to non-halting programs. Q# similarly has repeat-until and while loops that can run for unbounded time [8]. While this is of course necessary for truly universal computation, in practice we are interested only in programs of bounded (perhaps polynomial) execution time. Jaqal's loops, which run for a fixed number of iterations, guarantee this, but can still significantly reduce the size of programs.

While Jaqal is built upon a lower-level pulse definition in Gate Pulse Files, it is the lowest-level QASM programming language exposed to users in QSCOUT. As an assembly language, it has been designed with metaprogramming (specifically generative programming) in mind from the start. While one can write Jaqal directly—and we do not discourage doing so—we expect most Jaqal code will be machine-generated. This significantly reduces the burden on the relatively limited classical computing hardware attached directly to the QSCOUT testbed: all classical computations are necessarily moved out of the runtime execution of Jaqal and into a metaprogram that runs on a more sophisticated classical computer. This means that we omit several features from Jaqal that are popular in other quantum computing languages, with the expectation that programs (written in a classical programming language) that generate Jaqal code can easily replicate those features, at least from the developer's perspective. For example, we do not include any form of classical arithmetic features; if a gate parameter such as a rotation angle needs to be calculated from known values rather than specified directly, that calculation occurs at code-generation time rather than at runtime. We are currently developing a Python package containing tools for writing metaprograms to generate Jaqal code, which will be available to users of Jaqal. However, we anticipate that users will also develop their own higher-level programming languages that compile down to Jaqal.

II. Gate Pulse File

The laser pulses that implement built-in or custom trapped-ion gates are defined in a ***Gate Pulse File (GPF)***. Eventually, users will be able to write their own GPF files, but that capability will not be available in our initial software release. However, users will be free to specify composite gates by defining them as sub-circuit macros. Additionally, custom native gates can be added in collaboration with Sandia scientists by specifying the pulse sequences to realize the gate. Furthermore, we have have provided a GPF file for the built-in gates of the QSCOUT 1.0 platform, which, other than the Mølmer-Sørensen gate [4], are standard quantum gates as described in Ref. [6]:[1]

- `prepare_all` prepares each qubit in the register in the $|0\rangle$ state in the computational (z) basis.
- `measure_all` measures each qubit in the register in the computational (z) basis.
- `Rx <qubit> <angle>`, `Ry <qubit> <angle>`, and `Rz <qubit> <angle>` rotate the qubit state counterclockwise by an arbitrary angle around the x, y, or z axis respectively.

[1] All angles in Jaqal commands are specified as 64-bit floating point numbers, but are converted by QSCOUT 1.0 hardware to a number between -2π and 2π, inclusive, to 40 bits of precision.

- `Px <qubit>`, `Py <qubit>`, and `Pz <qubit>` rotate the qubit state by π around the respective axes.
- `Sx <qubit>`, `Sy <qubit>`, and `Sz <qubit>` rotate the qubit state by $\pi/2$ counterclockwise around the respective axes.
- `Sxd <qubit>`, `Syd <qubit>`, and `Szd <qubit>` rotate the qubit state by $\pi/2$ clockwise around the respective axes.
- `MS <qubit> <qubit> <phi> <theta>` is the general two-qubit Mølmer-Sørensen gate [4]

$$\exp\left(-i\left(\frac{\theta}{2}\right)(\cos\varphi X + \sin\varphi Y)^{\otimes 2}\right).$$

- `Sxx <qubit> <qubit>` is the XX-type Mølmer-Sørensen gate with $\varphi = 0$ and $\theta = \pi$.

We also include idle gates with the same duration as each single- and two-qubit gate. While it is not necessary to explicitly insert idle on idling qubits in a parallel block, these explicit idle gates give the user even more control of the scheduling of gate execution, and are meant to be used for performance testing and evaluation.

III. Jaqal Syntax

A Jaqal file consists of gates and metadata making those gates easier to read and write. The gates that are run on the machine can be deterministically computed by inspection of the source text. This implies that there are no conditional statements at this level. This section will describe the workings of each statement type.

Whitespace is largely unimportant except as a separator between statements and their elements. If it is desirable to put two statements on the same line, a ';' separator may be used. In a parallel block, the pipe ('|') must be used instead of the ';'. Like the semicolon, however, the pipe is unnecessary to delimit statements on different lines. Both Windows and Linux newline styles will be accepted.

A. Identifiers

Gate names and qubit names have the same character restrictions. Similar to most programming languages, they may contain, but not start with, numerals. They are case sensitive and may contain any non-accented Latin character plus the underscore. Identifiers cannot be any of the keywords of the language.

B. Comments

C/C++ style comments are allowed and treated as whitespace. A comment starting with '//' runs to the end of the current line, while a comment with '/*' runs until a '*/' is encountered. These comments do not nest, which is the same behavior as C/C++.

C. Header Statements

A properly formatted Jaqal file comprises a header and body section. All header statements must precede all body statements. The order of header statements is otherwise arbitrary except that all objects must be defined before their first use.

1) Register Statement: A register statement serves to declare the user's intention to use a certain number of qubits, referred to in the file with a given name. If the machine cannot supply this number of qubits then the entire program is rejected immediately.

The following line declares a register named `q` which holds 7 qubits.

```
register q[7]
```

2) Map Statement: While it is sufficient to refer to qubits by their offset in a single register, it is more convenient to assign names to individual qubits. The map statement effectively provides an alias to a qubit or array of qubits under a different name. The following lines declare the single qubit `q[0]` to have the name `ancilla` and the array `qubits` to be an alias for `q`. Array indices start with 0.

```
register q[3]
map ancilla q[0]
map qubits q
```

The map statement will also support Python-style slicing. In this case, the map statement always declares an array alias. In the following line we relabel every other qubit to be an ancilla qubit, starting with index 1.

```
register q[7]
map ancilla q[1:7:2]
```

After this instruction, `ancilla[0]`, `ancilla[1]`, and `ancilla[2]` correspond to `q[1]`, `q[3]`, and `q[5]`, respectively.

3) Let Statement: We allow identifiers to replace integers or floating point numbers for convenience. There are no restrictions on capitalization. An integer defined in this way may be used in any context where an integer literal is valid and a floating point may similarly be used in any context where a floating point literal is valid. Note that the values are constant, once defined.

Example:

```
let total_count 4
let rotations 1.5
```

D. Body Statements

1) Gate Statement: Gates are listed, one per statement, meaning it is terminated either by a newline or a separator. The first element of the statement is the gate name followed by the gate's arguments which are whitespace-separated numbers or qubits. Elements of quantum registers, mapped aliases, and local variables (see section on macros) may be freely interchanged as qubit arguments to each gate. The names of the gates are fixed but determined in the Gate Pulse File, except for macros. The number of arguments ("arity") must match the expected number. The following is an example of what a 2-qubit gate may look like.

```
register q[3]
map ancilla q[1]
```

```
Sxx q[0] ancilla
```

The invocation of a macro is treated as completely equivalent to a gate statement.

2) Gate Block: Multiple gates and/or macro invocations may be combined into a single block. This is similar, but not completely identical, to how C or related languages handle statement blocks. Macro definitions and header statements are not allowed in gate blocks. Additionally, statements such as macro definitions or loops expect a gate block syntactically and are not satisfied with a single gate, unlike C.

Two different gate blocks exist: sequential and parallel. Sequential gate blocks use the standard C-style '{}' brackets while parallel blocks use angled '<>' brackets, similar to C++ templates. This choice was made to not conflict with '[]' brackets, which are used in arrays, and to reserve '()' for possible future use. In a sequential block, each statement, macro, or gate block waits for the previous to finish before executing. In a parallel gate block, all operations are executed at the same time. It is an error to request parallel operations that the machine is incapable of performing, however it is not syntactically possible to forbid these as they are determined by hardware constraints which may change with time.

The Jaqal language does not have a barrier statement, as many other quantum assembly languages do, that specifies to the execution environment which gates should not be re-ordered to be executed simultaneously. Jaqal gates will be executed simultaneously if and only if the user places them in a parallel block with each other, with no re-ordering at runtime, so no such statement is necessary.

Looping statements are allowed inside sequential blocks, but not inside parallel blocks. Blocks may be arbitrarily nested so long as the hardware can support the resulting sequence of operations. Blocks may not be nested directly within other blocks of the same type.

The following statement declares a parallel block with two gates.

```
< Sx q[0] | Sy q[1] >
```

This does the same but on different lines.

```
<
    Sx q[0]
    Sy q[1]
>
```

Here is a parallel block nested inside a sequential one.

```
{ Sxx q[0] q[1]; < Sx q[0] | Sy q[1] >; }
```

And sequential blocks may be nested inside parallel blocks.

```
< Px q[0] | { Sx q[1] ; Sy q[1] } >
```

3) Timing within a parallel block: If two gates are in a parallel block but have different durations (*e.g.*, two single-qubit gates of different length), the default behavior is to *start* each gate within the parallel block simultaneously. The shorter gate(s) will then be padded with idles until the end of the gate block. For example, the command

```
< Rx q[1] 0.1 | Sx q[2] >
```

results in the `Rx` gate on `q[1]` with angle 0.1 radians and `Sx` gate on `q[2]` both starting at the same time; the `Rx` gate will finish first and `q[1]` will idle while the `Sx` gate finishes. GPF files will allow users to define their own scheduling within parallel blocks (*e.g.*, so that gates all *finish* at the same time instead).

4) Macro Statement: A macro can be used to treat a sequence of gates as a single gate. Gates inside a macro can access the same qubit registers and mapped aliases at the global level as all other gates, and additionally have zero or more arguments which are visible. Arguments allow the same macro to be applied on different combinations of physical qubits, much like a function in a classical programming language.

A macro may use other macros that have already been declared. A macro declaration is complete at the *end* of its code block. This implies that recursion is impossible. It also implies that macros can only reference other macros created earlier in the file. Due to the lack of conditional statements, recursion always creates an infinite loop and is therefore never desirable.

A macro is declared using the `macro` keyword, followed by the name of the macro, zero or more arguments, and a code block. Unlike C, a macro must use a code block, even if it only has a single statement.

The following example declares a macro.

```
macro foo a b {
    Sx a
    Sxx a q[0]
    Sxx b q[0]
}
```

To simplify parsing, a line break is not allowed before the initial '{', unlike C. However, statements may be placed on the same line following the '{'.

5) Loop Statement: A gate block may be executed for a fixed number of repetitions using the loop statement. The loop statement is intentionally restricted to running for a fixed number of iterations. This ensures it is easy to deterministically evaluate the runtime of a program. Consequently, it is impossible to write a program which will not terminate.

The following loop executes a sequence of statements seven times.

```
loop 7 {
    Sx q[0]
    Sz q[1]
    Sxx q[0] q[1]
}
```

The same rules apply as in macro definitions: '{' must appear on the same line as `loop`, but other statements may follow on the same line.

Loops may appear in sequential gate blocks, but not in parallel gate blocks.

IV. Extensibility

As Jaqal, and the QSCOUT project more broadly, have extensibility as stated goals, it is important to clarify what is meant by this term. Primarily, Jaqal offers extensibility in the gates that can be performed. This will occur through the Gate Pulse File and the use of macros to define composite gates that can be used in all contexts a native gate can. Jaqal will be incrementally improved as new hardware capabilities come online and real-world use identifies areas for enhancement. The language itself, however, is not intended to have many forms of user-created extensibility as a software developer might envision the term. Features we do not intend to support include, but are not limited to, pragma statements, user-defined syntax, and a foreign function interface (*i.e.,* using custom C or Verilog code in a Jaqal file).

V. Data Output Format

When successfully executed, a single Jaqal file will generate a single ASCII text file (Linux line endings) in the following way:

1. Each call of `measure_all` at runtime will add a new line of data to the output file. (If `measure_all` occurs within a `loop` (or nested loops), then multiple lines of data will be written to the output file, one for each call of `measure_all` during execution.)
2. Each line of data written to file will be a single bit-string, equal in length to the positive integer passed to `register` at the start of the program.
3. Each bitstring will be written in least-significant bit order (little endian).

For example, consider the program:

```
register q[2]

loop 2 {
    prepare_all
    Px q[0]
    measure_all
}

loop 2 {
    prepare_all
    Px q[1]
    measure_all
}
```

Assuming perfect execution, the output file would read as:

```
10
10
01
01
```

While this output format will be "human-readable", it may nevertheless be unwieldy to work with directly. To help with this, we are developing a Python-based parser in our metaprogramming package to aid users in manipulating output data.

VI. Metaprogramming

A. Compile-Time Classical Computation

Performing classical computations at compile-time, before the program is sent to the quantum computer, can vastly increase the expressiveness of the language. We provide three examples of this here.

1) Calculating Gate Angles: Jaqal does not have built-in arithmetic functions, meaning that gate parameters cannot be calculated by a Jaqal program itself. For example, consider the following program, *which is not currently legal Jaqal code:*

```
register q[1]

let pi 3.1415926536

loop 100 {
    prepare_all; Ry q[0] pi/32; measure_all
    prepare_all; Ry q[0] pi/16; measure_all
    prepare_all; Ry q[0] pi/8; measure_all
}
```

If writing such a Jaqal program by hand, you can define constants as needed to store the computed values:

```
register q[1]

let pi_32   0.09817477042
let pi_16   0.1963495408
let pi_8    0.3926990817

loop 100 {
    prepare_all; Ry q[0] pi_32; measure_all
    prepare_all; Ry q[0] pi_16; measure_all
    prepare_all; Ry q[0] pi_8; measure_all
}
```

However, if you're generating a Jaqal program via a high-level language, then you can include the calculations inline and have the metaprogram automatically substitute the results, as in the following pseudocode block:

```
q = register("q", 1)
loop(100, gate("prepare_all"),
     gate("Ry", q[0], pi/32),
     gate("measure_all"),
     gate(), ...)
```

Assuming suitable definitions of the register, loop, and gate calls in the metalanguage environment, running that metaprogram would then generate a Jaqal program:

```
register q[1]

loop 100 {
    prepare_all;
    Ry q[0] 0.09817477042;
    measure_all;
    ...
}
```

2) Macro Definition: Another example of a case where compile-time classical computation could be useful is in macro definitions. For example, if you wished to define a macro for a controlled z rotation in terms of a (previously-defined) CNOT macro, the following otherwise correct circuit *would not be legal Jaqal:*

```
...
macro CNOT control target { ... }

macro CRz control target angle {
    Rz target angle/2
    CNOT control target
    Rz target -angle/2
    CNOT control target
}
...
CRz q[0] q[1] 0.7853981634;
...
```

This is because both `angle/2` and `-angle/2` are classical computations that are not permitted in Jaqal. Instead of defining a Jaqal macro, users could write the relevant gate sequence manually:

```
...
Rz q[1] 0.3926990817;
CNOT q[0] q[1];
Rz q[1] -0.3926990817;
RNOT q[0] q[1];
...
```

However, this makes for unexpressive code. Instead, a user could define the controlled Z-rotation using a metalanguage, similarly to the following pseudocode:

```
procedure CRz (ctrl, target, angle) {
  gate("Rz", target, angle/2)
  gate("CNOT", ctrl, target)
  gate("Rz", target, -angle/2)
  gate("CNOT", ctrl, target)
}
...
CRz(q[0], q[1], 0.7853981634)
...
```

3) Randomized Algorithms: Another relevant use of classical computation is to generate random numbers to determine the sequence of gates. Applications of randomized quantum programs include hardware benchmarking, error mitigation, and some quantum simulation algorithms. This, too, can be done in this generative programming paradigm, pre-generating all the random values and automatically producing Jaqal code to execute the random circuit selected.

B. Run-Time Classical Computation

Users may also wish to do classical computations while a Jaqal program is running, based on the results of measurements. For example, in hybrid variational algorithms, a classical optimizer may use measurement results from one circuit to choose rotation angles used in the next circuit. In error-correction experiments, a decoder may need to compute which gates are necessary to restore a state based on the results of stabilizer measurements. Adaptive tomography protocols may need to perform statistical analyses on measurement results to determine which measurements will give the most information.

As can be seen from the above examples, run-time classical computation falls into two main categories; determining which circuits to run based on measurement results from former circuits, and determining the gate sequence of a circuit based on intermediate measurements in a circuit. Currently, the measurement operation of the QSCOUT hardware acts on all ions in the trap, destroying their quantum state and taking them out of the computational subspace. Future versions of the QSCOUT hardware will allow for the isolation and measurement of a subset of qubits with a command of the form `measure_subset <qubit> ...`. Similarly, a `prepare_subset <qubit> ...` operation will allow the reuse of measured qubits without destroying the quantum state of the remainder. Once subset measurement is implemented, Jaqal will be extended to support some run-time classical computation.

However, use cases like adaptive tomography and variational algorithms are possible with the current hardware, and can be implemented via metaprogramming techniques. After running a Jaqal program on the QSCOUT hardware, a metaprogram can parse the measurement results, then use that information to generate a new Jaqal program to run. This allows for adaptive and hybrid algorithms to be run without having to execute a numerical optimization routine or similar such tool on the classical control circuitry of QSCOUT.

ACKNOWLEDGMENT

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for DOE's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] R. J. Blume-Kohout, et. al, "Robust, self-consistent, closed-form tomography of quantum logic gates on a trapped ion qubit," 2013, arXiv:1310.4492, unpublished.
[2] Cirq: A Python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits, https://github.com/Cirq.
[3] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017, arXiv:1707.03429, unpublished.
[4] K. Mølmer and A. Sørensen, "Multiparticle entanglement of hot trapped ions." Phys. Rev. Lett., vol. 82, no. 9, p. 1835, 1999.
[5] D. McKay et. al, "Qiskit backend specifications for OpenQASM and OpenPulse experiments," 2018, arXiv:1809.03452, unpublished.
[6] M. A. Nielsen and I. L. Chuang, Quantum information and quantum computation. Cambridge: Cambridge University Press, 2000.
[7] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016, arXiv:1608.03355, unpublished.
[8] The Q# programming language, https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview.