Jaqal™, the Quantum Assembly Language for QSCOUT

A. J. Landahl D. S. Lobser B. C. A. Morrison K. M. Rudinger A. E. Russo
J. W. Van Der Wall P. Maunz
Sandia National Laboratories, Albuquerque, NM 87185

March 4, 2020 (revised May 7, 2024)

To Learn More

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.

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.

Language Versioning

As we continue improving QSCOUT and the software to support it, we add new features to the Jaqal language. At the time of the release of this documentation, the current version of Jaqal is 1.3. Features introduced after Jaqal 1.0 are marked with the first version number in which they are available, notated as **(1.1+)**. Additionally, some planned features for Jaqal 1.4 have preliminary documentation here **(1.4+)**; this is for information only, and actual implementation details and syntax may change before the release of Jaqal 1.4. It is our intention to maintain back-compatibility through the Jaqal 1.X series, except to fix unintended behavior. Any valid, spec-conforming Jaqal 1.X file will remain valid under later Jaqal releases prior to Jaqal 2.0, which may break back-compatibility.

QSCOUT Hardware

The QSCOUT hardware realizes a single register of qubits stored in the hyperfine clock states of trapped ^{171}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 (MS) two-qubit gate 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.
- **(1.2+)** A two-qubit ZZ Mølmer–Sørensen rotation between any pair of qubits, in parallel with no other gates.
- **(1.4+)** A pairwise MS-type rotation between any number of qubits jointly.

Importantly, as of February 2024 the QSCOUT hardware 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 uses ***Just Another Quantum Assembly Language (Jaqal)*** (described below) to specify quantum programs executed on the testbed. As of February 2024, every quantum computation on QSCOUT starts with preparation of the quantum state of the entire qubit register in the z basis. Then it executes a sequence of parallel and sequential 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.

Gate Pulse File

The laser pulses that implement built-in or custom trapped-ion gates are defined in a ***Gate Pulse File (GPF)***. Users can create their own GPF files that add custom native gates, in collaboration with Sandia scientists, by specifying the pulse sequences that have to be applied to the trapped ion qubits to realize the gate. This is done using the JaqalPaw pulse and waveform programming language. Doing so is outside the scope of this manual; instead consult the official JaqalPaw manual, which can be found at qscout.sandia.gov. Additionally, users are free to specify composite gates by defining them as macros.

Special Gate Instructions

The initialization and measurement of qubits are implemented in ways only partially specified by the GPF. The `prepare_all` and `measure_all` gates have additional semantics both at the hardware level and the language level. In particular, Jaqal files are divided into "subcircuits" that are delimited by `prepare_all` and `measure_all` instructions. If none appear in the text, an implicit `prepare_all` is added to the beginning and `measure_all` to the end.

Any Jaqal file that includes other gate statements after a `measure_all` and before the subsequent `prepare_all`, or otherwise outside of an explicit or implicit `measure_all`/`prepare_all` block, is invalid.

Additionally, in Jaqal 1.1 and later, implicit `prepare_all` and `measure_all` instructions can be inserted with the subcircuit block statement.

- `prepare_all`
  Prepares all qubits in the quantum register in the $|0\rangle$ state in the z basis.
- `measure_all`
  Measures all qubits of the quantum register in the z basis. After measurement, ions will be outside the qubit space. Therefore, the qubits have to be prepared again before any other gates can be applied.

QSCOUT Standard Gates

We provide a GPF file for the built-in gates on the QSCOUT platform, which can be loaded as `qscout.v1.std`, and obtained at gitlab.com/jaqal/jaqalpaw. This GPF file contains pulse-level gate definitions for the QSCOUT built-in gates listed below. All angle arguments in this list are in the units of radians, with 40 bits of precision. The chirality of rotations is determined using the right-hand rule. Further information about the contents of `qscout.v1.std` can be found in the official JaqalPaw manual.

- `R <qubit> <axis angle> <rotation angle>`
  Counter-clockwise rotation around an axis in the equatorial plane of the Bloch sphere defined by `<axis angle>`, measured counter-clockwise from the x axis, by the angle defined by `<rotation angle>`.
- `Rx <qubit> <rotation angle>`
  Counter-clockwise rotation around the x axis, by the angle defined by `<rotation angle>`.
- `Ry <qubit> <rotation angle>`
  Counter-clockwise rotation around the y axis, by the angle defined by `<rotation angle>`.
- `Rz <qubit> <rotation angle>`
  Counter-clockwise rotation around the z axis, by the angle defined by `<rotation angle>`.
- `Px <qubit>`
  Counter-clockwise rotation around the x axis, by π. (Pauli X gate.)
- `Py <qubit>`
  Counter-clockwise rotation around the y axis, by π. (Pauli Y gate.)
- `Pz <qubit>`
  Counter-clockwise rotation around the z axis, by π. (Pauli Z gate.)
- `Sx <qubit>`
  Counter-clockwise rotation around the x axis, by $\pi/2$. ($\sqrt{X}$ gate.)
- `Sy <qubit>`
  Counter-clockwise rotation around the y axis, by $\pi/2$. ($\sqrt{Y}$ gate.)
- `Sz <qubit>`
  Counter-clockwise rotation around the z axis, by $\pi/2$. ($\sqrt{Z}$ gate.)
- `Sxd <qubit>`
  Clockwise rotation around the x axis, by $\pi/2$. ($\sqrt{X}^\dagger$ gate.)
- `Syd <qubit>`
  Clockwise rotation around the y axis, by $\pi/2$. ($\sqrt{Y}^\dagger$ gate.)
- `Szd <qubit>`
  Clockwise rotation around the z axis, by $\pi/2$. ($\sqrt{Z}^\dagger$ gate.)

- `MS <qubit> <qubit> <axis angle> <rotation angle>`
  The general two-qubit Mølmer–Sørensen gate in the XX-YY plane. If we let θ represent `<rotation angle>` and φ represent `<axis angle>`, then the gate is
  $$\exp\left(-i\left(\frac{\theta}{2}\right)(\cos\varphi X+\sin\varphi Y)^{\otimes 2}\right).$$
- `XX <qubit> <qubit> <rotation angle>` **(1.3+)** An XX-type two-qubit Mølmer–Sørensen gate. If we let θ represent `<rotation angle>`, then the gate is
  $$\exp\left(-i\left(\frac{\theta}{2}\right)X\otimes X\right).$$
- `YY <qubit> <qubit> <rotation angle>` **(1.3+)** A YY-type two-qubit Mølmer–Sørensen gate. If we let θ represent `<rotation angle>`, then the gate is
  $$\exp\left(-i\left(\frac{\theta}{2}\right)Y\otimes Y\right).$$
- `ZZ <qubit> <qubit> <rotation angle>` **(1.2+)** A ZZ-type two-qubit Mølmer–Sørensen gate. In version 1.2, it must be loaded from `qscout.v1.zz`; in version 1.3 or higher, `qscout.v1.zz` is deprecated and the ZZ gate is included directly in `qscout.v1.std`. If we let θ represent `<rotation angle>`, then the gate is
  $$\exp\left(-i\left(\frac{\theta}{2}\right)Z\otimes Z\right).$$
- `Sxx <qubit> <qubit>`
  The standard, fully entangling ($\theta=\pi/2$) XX-type two-qubit Mølmer–Sørensen gate:
  $$\exp\left(-i\left(\frac{\pi}{4}\right)X\otimes X\right).$$
- `Sxxd <qubit> <qubit>` **(1.3+)** The fully entangling ($\theta=-\pi/2$) XX-type two-qubit Mølmer–Sørensen gate inverse to `Sxx`:
  $$\exp\left(+i\left(\frac{\pi}{4}\right)X\otimes X\right).$$
- `Syy <qubit> <qubit>` **(1.3+)** The standard fully entangling ($\theta=\pi/2$) YY-type two-qubit Mølmer–Sørensen gate:
  $$\exp\left(-i\left(\frac{\pi}{4}\right)Y\otimes Y\right).$$
- `Syyd <qubit> <qubit>` **(1.3+)** The fully entangling ($\theta=-\pi/2$) quarter-turn YY-type two-qubit Mølmer–Sørensen gate inverse to `Syy`:
  $$\exp\left(+i\left(\frac{\pi}{4}\right)Y\otimes Y\right).$$
- `Szz <qubit> <qubit>` **(1.3+)** The standard fully entangling ($\theta=\pi/2$) ZZ-type two-qubit Mølmer–Sørensen gate:
  $$\exp\left(-i\left(\frac{\pi}{4}\right)Z\otimes Z\right).$$
- `Szzd <qubit> <qubit>` **(1.3+)** The fully entangling ($\theta=-\pi/2$) quarter-turn ZZ-type two-qubit Mølmer–Sørensen gate inverse to `Szz`:
  $$\exp\left(+i\left(\frac{\pi}{4}\right)Z\otimes Z\right).$$
- `Rt <qubit> <axis angle> <rotation angle>` **(1.3+)** Counter-clockwise rotation around an axis in the equatorial plane of the Bloch sphere defined by `<axis angle>`, measured counter-clockwise from the x axis, by the angle defined by `<rotation angle>`. The ideal action of this gate is the same as `R`, but it uses a counter-propagating beam implementation on QSCOUT hardware rather than the usual co-propagating implementation. See the QSCOUT Manual ieeexplore.ieee.org/document/9483669 for more details on counter-propagating and co-propagating beams.

- `GMS <qubit> [...] <qubit> <axis angle> <rotation angle>` **(1.4+)** The global Mølmer–Sørensen gate on any number of qubits. Its action is equivalent to `MS <qubit> <qubit> <axis angle> <rotation angle>` simultaneously on every pair of the listed qubits. For example `GMS q[0] q[1] q[2] <axis angle> <rotation angle>` performs the same ideal operation as `MS q[0] q[1] <axis angle> <rotation angle>; MS q[1] q[2] <axis angle> <rotation angle>; MS q[0] q[2] <axis angle> <rotation angle>`, but implemented simultaneously instead of sequentially.

The gate pulse definitions also include idle gates with the same duration as the single- and two-qubit gates. These have a prefix of `I_`. For example an idle gate of the same duration as a `Px` can be obtained by `I_Px <qubit>`. It is important to note that it is not necessary to explicitly insert idle on idling qubits in a parallel block. Explicit idle gates are meant to be used for performance testing and evaluation.

Jaqal Quantum Assembly Language

The open nature of the QSCOUT testbed requires a flexible **Quantum Assembly Language (QASM)** that empowers QSCOUT users to extend the set of native gates and fully control the execution of the quantum program on the QSCOUT testbed. Due to the proliferation of such languages in this fledgling field, ours is named **Just Another Quantum Assembly Language**, or **Jaqal**.

To realize our objectives, the Jaqal QASM language fulfills the following requirements:

- Jaqal fully specifies the allocation of qubits within the quantum register, which *cannot* be altered during execution.
- Jaqal requires the scheduling of sequential and parallel gate sequencing to be fully and explicitly specified.
- Jaqal can execute any native (built-in or custom) gate specified in any GPF file it references.

While Jaqal is built upon a lower-level pulse definition in GPF files, it is the lowest-level QASM programming language exposed to users in QSCOUT. We anticipate that users will develop their own metaprogramming tools and higher-level programming languages that compile down to Jaqal, as indeed many past and present users have. We also provide some metaprogramming tools in the JaqalPaq repository, and recommend their use in constructing readable and concise programs.

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.

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.

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++.

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.

Usepulses Statement

A usepulses statement specifies where to find the intended definitions associated with each gate name used in the Jaqal program. It specifies the location of a Gate Pulse File as a qualified string, similar to a Python import statement. The following line loads every gate from `qscout.v1.std`:

```
from qscout.v1.std usepulses *
```

In Jaqal 1.0–1.3, a usepulses statement must load `*`, signifying every gate in the specified GPF.

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]
```

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]` corresponds to `q[1]`; `ancilla[1]` and `ancilla[2]` correspond to `q[3]`and `q[5]`, respectively.

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
```

Body Statements

Gate Statement

Gates are listed, one per statement, meaning each gate 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.

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.

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] >
}
```

Sequential blocks may also be nested inside parallel blocks.

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

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. Once the Jaqal gate set becomes user-extensible, users may define their own scheduling within parallel blocks (*e.g.*, so that gates all *finish* at the same time instead).

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 ‘{’.

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.

Subcircuit Statement (1.1+)

To aid in the structuring of Jaqal programs, we have added a `subcircuit` block statement that wraps a sequential block in `prepare_all` and `measure_all` gate statements. This block is implemented identically to explicitly written `prepare_all` and `measure_all`. As in a loop statement, the ‘{’ must appear on the same line as `subcircuit` but can be followed by other statements. The following two programs are semantically equivalent.

```
subcircuit {
    Sx q[0]
}
```

```
prepare_all
Sx q[0]
measure_all
```

Subcircuit blocks can be declared only at the top level of a program, and not nested within other blocks.

Jaqal Execution Model

We will also briefly discuss how Jaqal programs are run on the QSCOUT hardware. This is not a standard, and we do not require that any particular runtime implementation of the Jaqal language conform to this model. However, a better understanding of how Jaqal is executed can aid in the writing of better Jaqal files.

A single Jaqal file can contain several subcircuits—sequences of gates starting with state preparation and ending with a measurement of all qubits—whether delimited by `prepare_all` and `measure_all` gates or written as `subcircuit` statements. The whole Jaqal file will be transmitted to the control electronics, parsed, and converted into pulse data as a single unit. However, each subcircuit will be run separately, with the hardware reset and ions cooled between. Accordingly, if you have a large number of subcircuits that all use the same or very similar gates, it is much more efficient to include them all in a single Jaqal file if possible. This minimizes the delay while the QSCOUT hardware is waiting for classical processing of the Jaqal files. On the other hand, if you have subcircuits that use a wide variety of different gates (or gates with different classical parameters), it's better to put each in its own file, to avoid overflowing the QSCOUT control system's gate lookup table.

If you wish to execute an entire subcircuit multiple times, it is not recommended to use a loop statement—indeed, wrapping a subcircuit inside of a loop statement is undefined behavior. Instead, QSCOUT's control systems accept a parameter `__repeats__` that determines how many times to run each subcircuit. If using the batch-execution tools in JaqalPaq to run your Jaqal file, this parameter (and the others discussed below) can be passed to those tools; see the JaqalPaq documentation for further details. If submitting a raw Jaqal file to be run on QSCOUT hardware, provide the QSCOUT experimental team with any such parameters that should be used.

QSCOUT hardware also supports overriding parameters, replacing the value of either a `let` statement or a parameter used by the GPF at runtime. For batch execution of Jaqal files, a list of different parameter values can be provided, and the Jaqal file will be run with each value.

After running all subcircuits the requested number of times, the QSCOUT hardware will return the measurement results in a format that can be parsed by the JaqalPaq software tools to obtain measurement frequencies. For non-JaqalPaq-users, data output is available in any reasonable format; discuss the desired output format with the QSCOUT experimental team when submitting your Jaqal files to run. If measurement data from each individual run is required, this can also be provided.

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, user-defined syntax and a foreign function interface (i.e. using custom C or Verilog code in a Jaqal file).

Examples

Bell state preparation

This example prepares a Bell state using the classic Hadamard and controlled X circuit, then measures it in the computational basis. Up to the limits of gate fidelity, the measurements of the two qubits should always match.

```
macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}

macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control               //
    Sxx control target
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

register q[2]

prepare_all         // Prepare each qubit in the computational basis.
hadamard q[0]
cnot q[1] q[0]
measure_all         // Measure each qubit and read out the results.
```

However, there's a more efficient way of preparing a Bell state that takes full advantage of the native Mølmer-Sørensen interaction of the architecture, rather than using it to replicate a controlled-X gate. The following snippet of code prepares the same Bell state using the same Mølmer-Sørensen gate and a single Z rotation. Because Z rotation gates are implemented on QSCOUT purely as instantaneous software corrections, whereas X and Y rotations are implemented as physical gates, this implementation has only the time and error costs of the `Sxx` gate. That is, we have reduced the time by that of five rotation gates and the error by that of six, as the two `Sxd` gates are run in parallel in the Hadamard+CX implementation.

```
register q[2]

prepare_all
Sxx q[0] q[1]
Sz q[0]
measure_all
```

Single-Qubit Gate Set Tomography

```
register q[1]

// Fiducials
macro F0 qubit { }
macro F1 qubit { Sx qubit }
macro F2 qubit { Sy qubit }
macro F3 qubit { Sx qubit; Sx qubit}
macro F4 qubit { Sx qubit; Sx qubit; Sx qubit }
macro F5 qubit { Sy qubit; Sy qubit; Sy qubit }

// Germs
macro G0 qubit { Sx qubit }
macro G1 qubit { Sy qubit }
macro G2 qubit { I_Sx qubit }
macro G3 qubit { Sx qubit; Sy qubit }
```

```
macro G4 qubit { Sx qubit; Sx qubit; Sy qubit }
macro G5 qubit { Sx qubit; Sy qubit; Sy qubit }
macro G6 qubit { Sx qubit; Sy qubit; I_Sx qubit }
macro G7 qubit { Sx qubit; I_Sx qubit; I_Sx qubit }
macro G8 qubit { Sy qubit; I_Sx qubit; I_Sx qubit }
macro G9 qubit { Sx qubit; Sy qubit; Sy qubit; I_Sx qubit }
macro G10 qubit { Sx qubit; Sx qubit; Sy qubit; Sx qubit; Sy qubit; Sy qubit }

// Length 1
prepare_all
F0 q[0]
measure_all

prepare_all
F1 q[0]
measure_all

prepare_all
F2 q[0]
measure_all

prepare_all
F3 q[0]
measure_all

prepare_all
F4 q[0]
measure_all

prepare_all
F5 q[0]
measure_all

prepare_all
F1 q[0]; F1 q[0]
measure_all

prepare_all
F1 q[0]; F2 q[0]
measure_all

// and many more
// Repeated germs can be realized with the loop

prepare_all
F1 q[0]
loop 8 { G1 q[0] }
F1 q[0]
measure_all
```

Possible Future Capabilities

Jaqal is still under development, and will gain new features as the QSCOUT hardware advances. While the precise feature set of future versions of Jaqal is still undetermined, we discuss some features that may be added or are only partially available, and in some cases identify workarounds for the current lack of those features.

Multiple GPFs (Experimental 1.0+)

Jaqal files with multiple usepulses statements have been supported in the Jaqal emulator since its release, but as of February 2024 are unavailable on the QSCOUT hardware. If you are relying on multiple Gate Pulse Files, we currently recommend working with the QSCOUT experimental team to create a GPF that combines all desired gates before running on hardware—see JaqalPaw gitlab.com/jaqal/jaqalpaw for more details. However, this feature can be useful for testing in the emulator before doing so.

If a Jaqal file's header contains multiple usepulses statements, the last-referenced file that contains a gate with a particular name is the definition used. In Jaqal 1.0–1.3, a usepulses statement must load every gate from the GPF. To avoid overwriting earlier definitions, we recommend using a gate name prefix when writing new GPF files.

Future versions of Jaqal may support loading only specified gates from a Jaqal file. For example, if you had a GPF file named `qscout.v1.sk1` with composite pulse sequences, but wished to use them only for single-qubit X rotations and use the default implementations of all other gates, a future version of Jaqal may allow you to write:

```
from qscout.v1.std usepulses *
from qscout.v1.sk1 usepulses Rx, Px, Sx, Sxd
```

Subset Measurement (Experimental 1.4+)

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, measurement, and reuse of a subset of qubits with a command that may take the form `measure_reprepare <qubit> ...`.

While the QSCOUT hardware does not yet support this feature as of February 2024, we have made it available as an experimental language feature. This implementation is not final API, and is likely to change before being fully released. The `measure_reprepare` gate can take any number of qubit arguments. Measurement is in the z basis, and qubits will be returned to the $|0\rangle$ state after measurement.

```
from qscout.v1.std usepulses *
register q[2]

prepare_all
Rx q[0] 1.5707963267948966
measure_reprepare q[0]
measure_all
```

```
from qscout.v1.std usepulses *
register q[3]

prepare_all
Rx q[0] 1.5707963267948966
measure_reprepare q[0] q[1] q[2]
measure_all
```

Measurement Feedback (Experimental 1.1+)

The QSCOUT hardware does not currently support using measurement outcomes to conditionally execute future gates. We expect this capability will be added in a future version of the QSCOUT hardware, and Jaqal programs will be able to use that capability once it exists.

While the QSCOUT hardware does not yet support this feature at time of writing, we have made it available as an experimental language feature. This implementation is not final API, and is likely to change before being fully released. The `branch` statement can be used to conditionally execute code based on measurement results.

Like other block-type statements, `branch` must be followed by '{' on the same line. The `branch` statement can contain only case statements, which are written as a single-quoted string of output bits in little-endian order, followed by a colon ':' and a sequential block statement. Cases within a `branch` statement can be separated by newlines, semicolons, or both. Each case statement will be compared against the results from the most recent measurement, and the block within executed if they match. If no case statement matches, no gates will be run and execution will immediately continue after the `branch` statement. A case statement with a number of bits that doesn't match the number of bits from the most recent measurement is an error.

```
register q[2]
subcircuit {
    branch {
        '00' : {
            Sx q[1]
        }
        '01' : {
            Sy q[1]
        }
    }
}
```

In combination with the `measure_reprepare` gate, this will allow circuits that change dynamically based on mid-circuit measurement results, as is required in many applications such as teleportation.

Classical Computation

Jaqal does not currently support any form of classical computation. We understand that this is a limitation, and expect future versions of Jaqal to do so. There are two relevant forms of classical computation that we are considering for Jaqal.

Compile-Time Classical Computation

Many potential uses of classical computation do not require any run-time data; for example, calculation of desired angles for rotation gates. Performing these classical computations at compile-time, before the program is sent to the QSCOUT hardware, can vastly increase the expressiveness of the language.

Strictly speaking, it is possible to do these calculations by hand and write Jaqal code that incorporates the results directly. However, this "manual compilation" is a significant inconvenience for writing readable and expressive code in Jaqal. Accordingly, we highly recommend the use of metaprogramming (automated code generation) tools to implement compile-time classical computation. These tools are often implemented using Jupyter notebooks and the Python interfaces provided by JaqalPaq. In particular, JaqalPaq can automatically substitute constants and unroll macros to generate Jaqal with the structure described above.

For example, consider the following experiment, *which is not legal Jaqal code:*

```
register q[1]

let pi 3.1415926536

subcircuit {
    Ry q[0] pi/32
}

subcircuit {
    Ry q[0] pi/16
}

subcircuit {
    Ry q[0] 3*pi/32
}

subcircuit {
    Ry q[0] pi/8
}
```

Jaqal does not support inline parameter calculations like the above. One possible workaround is to define additional constants as needed:

```
register q[1]

let pi_32   0.09817477042
let pi_16   0.1963495408
let pi_3_32 0.2945243113
let pi_8    0.3926990817

subcircuit {
    Ry q[0] pi_32
}

subcircuit {
    Ry q[0] pi_16
}

subcircuit {
    Ry q[0] pi_3_32
}

subcircuit {
    Ry q[0] pi_8
}
```

Another example of a case where compile-time classical computation could be useful is in macro definition. For example, if you wished to define a macro for a controlled z rotation in terms of a (previously-defined) CNOT macro:

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

```
macro CRz control target angle {
    Rz target angle/2
    CNOT control target
    Rz target -angle/2
    CNOT control target
}
...
```

Again, the above example *is not legal Jaqal.* We recommend, in such cases, that you manually unroll macros as needed, then define additional constants as above. That is, rather than using the above macro:

```
...
let phi 0.7853981634;
...
CRz q[0] q[1] phi;
...
```

You should instead call the gates the macro is made up of, substituting the results of the appropriate calculations yourself:

```
...
let phi 0.7853981634;
let phi_2 0.3926990817;
let phi_m_2 -0.3926990817;
...
Rz q[1] phi_2; CNOT q[0] q[1]; Rz q[1] phi_m_2; CNOT q[0] q[1];
...
```

In general, these kinds of programs can and usually should be automatically generated by metaprograms rather than written manually.

Run-Time Classical Computation

Users may also wish to do classical computation 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 is useful only when measurement feedback is possible. Accordingly, we will consider this feature after we have added support for measurement feedback.

However, use cases like adaptive tomography and variational algorithms can be implemented via metaprogramming techniques. After running a Jaqal file on the QSCOUT hardware, a metaprogram can parse the measurement results, then use that information to generate a new Jaqal file to run.

Randomness

Executing quantum programs with gates chosen via classical randomness is desirable for a variety of reasons. Applications of randomized quantum programs include hardware benchmarking, error mitigation, and some quantum simulation algorithms. Jaqal does not currently have built-in support for randomization, although it may in the future, likely in combination with support for run-time classical computation. Our currently recommended workaround is to pre-compute any randomized elements of the algorithm, automatically generating Jaqal code to execute the random circuit selected. For example, the following program isn't currently possible, as there's no means of generating a random angle in Jaqal directly:

```
register q[1]

subcircuit {
    // Do an X rotation on q[0] by a random angle between 0 and 2*pi.
}
// And run the above subcircuit 100 times, with different random angle seach time.
```

However, the same effect can be obtained by a metaprogram (here written in Jaqalpaq's Q-syntax) that generates a Jaqal program:

```
python from random import uniform from math import pi from jaqalpaq.qsyntax import circuit
@circuit def randomness_example(Q):     q = Q.register(1, name='q')     for _ in range(100):
angle = uniform(0.0, 2.0 * pi)         with Q.subcircuit:             Q.Rx(q[0], angle)
```

While the generated Jaqal program is much larger than one that could be written in a potential future version of Jaqal that supported randomized execution, the metaprogram that generates it is quite compact.

Acknowledgements

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.