Open Quantum Assembly Language

Andrew Cross[1], Ali Javadi-Abhari[1], Thomas Alexander[1], Lev Bishop[1], Colm A. Ryan[2], Steven Heidel[2], Niel de Beaudrap[3], John Smolin[4], Jay M. Gambetta[4], and Blake R. Johnson[4]

[1]IBM Quantum, IBM T. J. Watson Research Center, Yorktown Heights, NY
[2]AWS Center for Quantum Computing
[3]University of Sussex
[4]IBM Quantum, IBM T. J. Watson Research Center, Yorktown Heights, NY

March 16, 2021

Contents

1 Introduction

Open quantum assembly language (OpenQASM 2) [1] was proposed as an imperative programming language for quantum computation based on earlier QASM dialects [2–6]. OpenQASM is one of the programming interfaces of the IBM Quantum services [7]. In the period since OpenQASM 2 was introduced, it has become something of a *de facto* standard, allowing a number of independent tools to inter-operate using OpenQASM 2 as the common interchange format. It also exists within the context of significant other activity in exploring and defining quantum assembly languages for practical use [8–13]. These machine-independent languages describe quantum computation in the quantum circuit model [14–16]. Quantum assembly languages are considered low-level programming languages, yet they are not often directly consumed by quantum control hardware. Rather, they are further compiled to lower-level instructions that operate at the level of analog signals for controlling quantum systems. QASM can be hand-written, generated by scripts (meta-programming), or targeted by higher-level software tools.

Our goal is to extend OpenQASM to express classical processing in quantum programs. When OpenQASM was introduced, the language features were essentially limited to a piece of straight line code, i.e. a basic block[1] The extension considered here is intended to be backwards compatible with OpenQASM 2, except in some uncommonly used cases where breaking changes were necessary. Recognizing that OpenQASM is a resource for describing programs that characterize, validate, and debug quantum systems, we also introduce instruction semantics that allow control of gate scheduling in the time domain and the ability to describe the microcoded implementations of gates in a way that is both extensible to future developments in quantum control and is platform agnostic.

In extending OpenQASM to include classical processing, we allow for a richer family of computation that incorporates interactive use of a quantum processor in a program or algorithm. However, we do not wish to reinvent all of classical computing. Rather, we recognize different timescales of quantum-classical interactions including *real-time* classical computations that must be performed within the coherence times of the qubits as well as *near-time* computations with less stringent timing. The real-time computations are critical for error correction and circuits that take advantage of feedback or feedforward. However, the demands of the real-time, low-latency domain might impose considerable restrictions on the available compute resources in that environment, such as limited memory or clock speeds. Whereas with the near-time domain we may assume more generic compute resources, including access to a broad set of libraries and runtimes. Consequently, we choose in OpenQASM 3 to limit our focus to the real-time domain which must be most tightly coupled to the execution of gates and measurement on qubits.

We use a dataflow model [17] where each instruction may be executed by an independent processing unit when its input data becomes available. Concurrency and parallelism are essential features of quantum control hardware. For example, parallelism is necessary for a fault-tolerance accuracy threshold to exist [18, 19], and we expect concurrent quantum and classical processing to be necessary for quantum error-correction [20–22].

Our extended OpenQASM language expresses a program, or any equivalent representa-

[1] The `if` keyword in OpenQASM 2 is a minor exception.

tion of it, as instances of quantum circuits and their associated classical control, i.e. as a collection of (quantum) basic blocks and flow control instructions. This differs from the role of a high-level language. High-level languages may describe programs that generate families of quantum circuits. They make use of external circuit libraries and mathematical software. They may include mechanisms for quantum memory management and garbage collection, or features to specify classical reversible programs to synthesize into quantum oracle implementations. Optimization passes at this high level work with families of quantum operations whose specific parameters might not be known yet. These passes could be applied once at this level and benefit every program instance we execute later. High-level intermediate representations may differ from OpenQASM until we reach the circuit generation phase of program execution, at which point we generate a specific program instance.

Our choice of features to add to OpenQASM is guided by use cases. Although OpenQASM is not a high-level language, many users would like to write simple quantum programs by hand using an expressive domain-specific language. Researchers who study circuit compiling need high-level information recorded in the intermediate representations to inform the optimization and synthesis algorithms. Experimentalists prefer the convenience of writing programs at a relatively high level but often need to manually modify timing or pulse-level gate descriptions at various points in the program. Hardware engineers who design the classical controllers and waveform generators prefer languages that are practical to compile given the hardware constraints. Our choice of language features is intended to acknowledge all of these potential audiences.

2 Philosophy and execution model

In building a program or application for quantum computing it is essential to define abstraction layers that are not trivial. Future quantum software stacks should have three such layers: a framework for developing applications, a language for expressing quantum circuits, and the executables which run on control hardware.

At its heart, OpenQASM is the quantum circuit. The quantum circuit model of computation describes the flow of quantum information through a network of resets, gates, measurements, and feedback [14]. At this level of abstraction, the details of execution on hardware are not captured. That is, the quantum circuit model assumes an abstract quantum machine that manipulates quantum states by applying gates and measurements on them in some order, whereas the actual execution involves an orchestration of code on classical controllers connected to the quantum apparatus.

In OpenQASM 3, we aim to describe a broader set of quantum circuits with concepts beyond simple qubits and gates. Chief among them are arbitrary classical control flow, gate modifiers (e.g. control and inverse), timing, and microcoded pulse implementations. While these extensions are not strictly necessary from a theoretical point of view (any quantum computation could in principle be described using OpenQASM 2), in practice they greatly expand the expressivity of the language.

One key motivation for the new language features is the ability to describe new kinds of computations and experiments. For example, repeat-until-success algorithms [23] or magic state distillation protocols [24] have a non-deterministic component that can be programmed

using the new classical control flow instructions. Dynamical decoupling [25] or characterizations of decoherence and crosstalk [26] are all sensitive to timing, and can be programmed using the new timing features. Calibration of gates can be programmed using embedded pulse descriptions.

Another design goal for the language is to create a suitable intermediate representation (IR) for quantum circuit compilation. Recognizing that quantum circuits have multiple levels of specificity from higher-level theoretical computation to lower-level physical implementation, we envision OpenQASM as a multi-level IR where different levels of abstraction can be used to describe a circuit. The goal is to introduce well-defined semantics that allow the compiler to optimize and re-write the circuit at different levels. For example, the compiler can reason directly on the level of controlled gates or gates raised to some power without decomposing these operations, which may obscure some opportunities for optimization. Furthermore, we aim to introduce semantics that are generic enough that they can capture the intent of the program in a portable manner, without tying it to a particular implementation. For example, the timing semantics in OpenQASM are designed to allow higher-level descriptions of timing constraints without being specific about individual gate latencies.

There are limitations on the scope of OpenQASM. The circuits described in this language are confined to coherent quantum operations on quantum data, i.e. real-time compute. As such, any near-time computation is to be accomplished by purely classical code outside of OpenQASM. This includes pre-processing of problem data (e.g. in Shor's algorithm [27]), post-processing of measurement outcomes (e.g. in tomography [16]) or further generation of circuits based on those (e.g. the outer loop of variational algorithms [28]). A second limitation of scope is that programming of quantum computations outside an explicit circuit model are to be done via higher-level tools that later synthesize into OpenQASM circuits [13, 29–32]. This includes unitary matrices, isometries, quantum channels, classical functions (e.g. oracles or truth tables), Cliffords, Paulis, etc.

Figure 1 illustrates an envisioned compilation and execution flow for quantum programs. We define a model with three levels of classical control (host, global, local) and three corresponding time scales (near-time, real-time, and local):

- The host computer has general purpose computation available with whatever resources are necessary for the application, but it has a "near-time" network connection to the global controller. A near-time connection is one that can be used in an online setting where a user interacts repeatedly with a quantum system with little perceived latency, but the connection is not fast enough to close feedback loops within the coherence time of that system.

- The global controller can do general purpose computation and communicates in real-time to a network of local qubit controllers. The real-time connection incurs some delay, but may be fast enough to close feedback loops well within the qubits' lifetimes, depending on the complexity of the global controller's real-time classical computations.

- Finally, the local controllers have a limited instruction set but make control decisions well within qubit lifetimes. They directly control the qubit control signal generation hardware and may map to only some portion of the system.

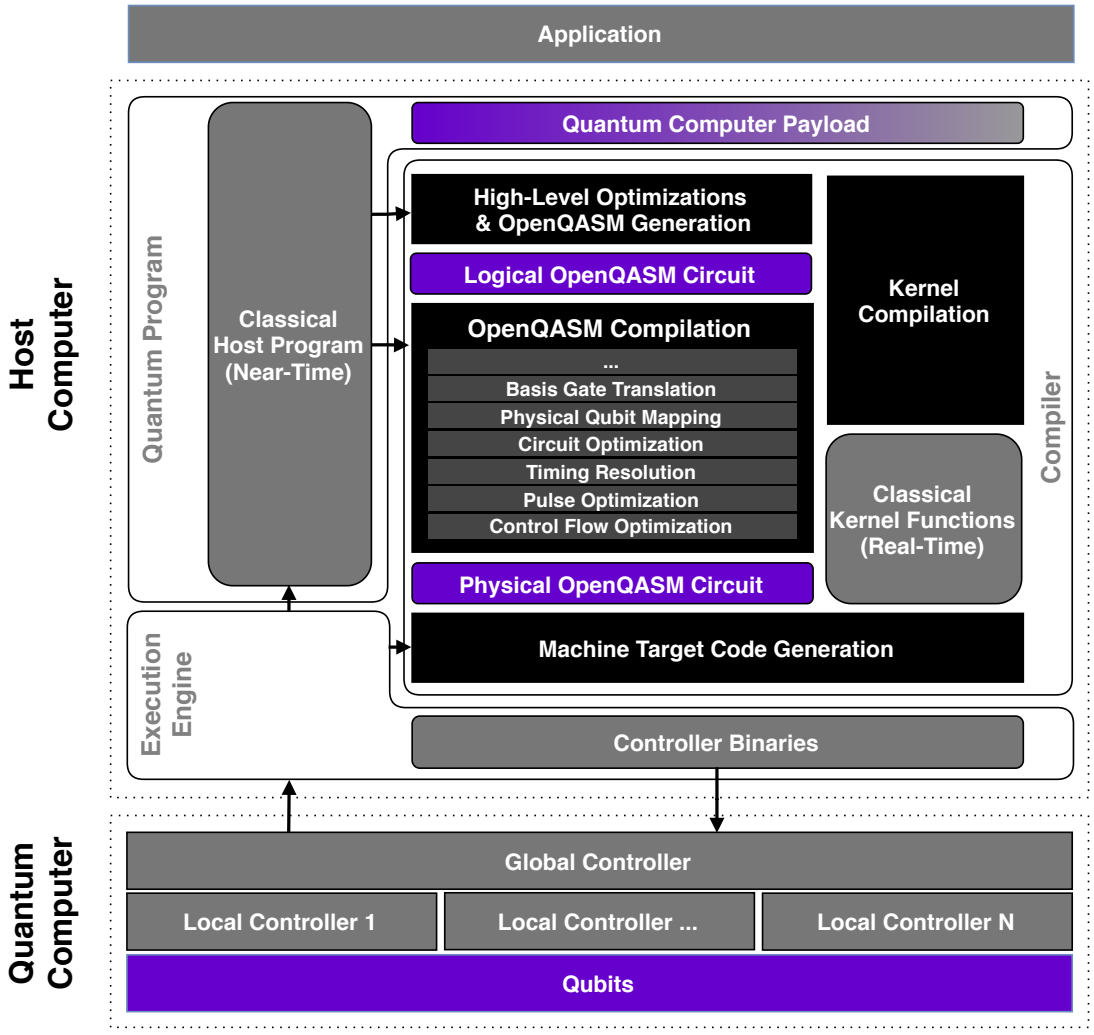


Figure 1: The compilation and execution model of a quantum program, and OpenQASM's place in the flow. An application is written in terms of a quantum program. Certain regions of a quantum program are purely classical and will execute at runtime on a host computer close to the quantum core (near-time processing). The quantum part of the quantum program is the payload that executes on the quantum computer. This payload itself can consist of quantum as well as real-time classical components. OpenQASM is the language to describe the quantum part of the payload, as well as how it interfaces with classical kernel functions. There may be higher-order aspects of the quantum payload which is optimized before OpenQASM is generated. The OpenQASM compiler can transform and optimize all aspects of the circuit described with the IR, including basis gates used, qubit mapping, timing, pulses, and control flow. The final physical circuit plus kernel functions are passed to a target code generator which produces binaries for the global and local controllers of the qubits. The runtime provides an execution environment for the circuit, tracks the circuit execution state and provides an interface for circuit output to be transferred from the quantum computer back to the host program.

The general flow can be broken down into several distinct phases:

1. High-level synthesis: This phase takes place on a host computer and no interaction with the quantum computer is required, i.e. it is offline. The input is high-level source code describing a quantum algorithm and any compile time parameters. The output is a compiled program that describes a family of quantum program instances and accepts problem parameters as input. During this phase, it is possible to compile purely classical procedures into object code and make initial compilation passes on quantum computations that do not require complete knowledge of the problem parameters.

Circuit compilation. This phase takes place on a classical computer in an environment where specific problem parameters are now known, and some interaction with the quantum computer may occur, i.e. this is an online phase. The input is a compiled program expressed using a high level representation, as well as all remaining problem parameters. The output is a collection of quantum circuits, or quantum basic blocks together with associated classical control instructions and classical object code needed at run-time. The quantum circuits may reference the native gate set and connectivity of a particular system and may include lower-level code provided by the user. Since real-time feedback may occur in a circuit, the classical code may include instructions such as run-time parameter computations and measurement-dependent branches. Kernel-level classical object code could include algorithms to process measurement outcomes into control flow conditions or extract results from the local controllers. The output of circuit generation is expressed using a quantum circuit representation such as OpenQASM.

Execution. This takes place on quantum computer controllers in a real-time environment, i.e. the quantum computer is active. The input is a collection of quantum circuits and associated run-time control statements, expressed using a quantum circuit representation. The input is further compiled and scheduled into real-time instructions for the global and local controllers. The program is executed concurrently on those controllers. Measurement data is sent to non-local controllers as needed, where controller run-times may process the input and send back outputs to the local controllers. The output of circuit execution is a collection of processed measurement results returned from the global controller.

3 Concepts of the language

In this section we give a overview of the important properties of OpenQASM. For a detailed overview we recommend reading the live document [33].

3.1 Continuous gates and hierarchical library

We define a mechanism for parameterizing unitary matrices to define new quantum gates. The parameterization uses a set of built-in controlled single-qubit gates that encompasses a universal gate set [15]. Early QASM languages assumed a discrete set of quantum gates, but OpenQASM is flexible enough to describe universal computation with a continuous set of physical gates. This gate set was chosen for the convenience of defining new quantum gates and is not an enforced compilation target. The mechanism allows for code reuse by hierarchically defining more complex operations [6, 34]. For many gates of practical interest,

there is a circuit representation with a polynomial number of one- and two-qubit gates, giving a more compact representation than requiring the programmer to express the full $n \times n$ matrix. In the worst case, a general n-qubit gate can be defined using an exponential number of these gates.

We now describe this built-in gate set. Single-qubit unitary gates are parameterized as

$$U(\theta, \phi, \lambda) := \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix} = e^{i(\phi+\lambda)/2} R_z(\phi) R_y(\theta) R_z(\lambda) \tag{1}$$

where the second equality is an Euler angle decomposition of the rotation. This expression specifies any element of $U(2)$ up to a global phase. The global phase is here chosen so that the upper left matrix element is real. For example, `U(π/2, 0, π) q[0];` applies the Hadamard gate $H = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$ to qubit `q[0]`.

Users should be aware that the definition in Eq. 1 is scaled by a factor of $e^{i(\phi+\lambda)/2}$ when compared to the original definition in [1]. This implies that U is not an element of $SU(2)$ and instead has determinant $e^{i(\phi+\lambda)}$. This change in global phase has no observable consequence for OpenQASM 2 programs.

New gates are associated to a unitary transformation by defining them using a sequence of built-in or previously defined gates. For example, the `gate` block

```
gate h q { U(π/2, 0, π) q; }
```

defines a new gate called "h" and associates it to the unitary matrix of the Hadamard gate. Once we have defined "h", we can use it in later `gate` blocks. The definition does not imply that `h` is implemented by an instruction `U(π/2, 0, π)` on the quantum computer. The implementation is left to the user and/or compiler, given information about the instructions supported by a particular target.

Controlled gates can be constructed by attaching a control modifier to an existing gate. For example, the NOT gate is given by $X = U(\pi, 0, \pi)$ and the block

```
gate cx c, t { ctrl @ U(π, 0, π) c, t; }
cx q[1], q[0];
```

defines the gate

$$\mathrm{cx} := I \oplus X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2}$$

and applies it to `q[1]` and `q[0]`. This gate applies a bit-flip X to `q[0]` if `q[1]` is one and otherwise applies the identity gate. The control modifier is described in more detail in Section 3.2.

Throughout the document we use a tensor order with higher index qubits on the left. This ordering labels states in a way that is consistent with how numbers are usually written with the most significant digit on the left. In this tensor order, `cx q[0], q[1];` is represented

by the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \tag{3}$$

From a physical perspective, the gates $e^{i\gamma}U$ and U are equivalent, although they differ by a global phase $e^{i\gamma}$. When we attach a control to these gates, however, the global phase becomes a relative phase that is applied when the control qubit is one. To capture the programmer's intent, a built-in global phase gate allows the inclusion of arbitrary global phases on circuits. The instruction `gphase(`γ`)` adds a global phase of $e^{i\gamma}$ to the scope containing the instruction. For example,

```
gate rz(tau) q { gphase(-tau/2); U(0, 0, tau) q; }
ctrl @ rz(π/2) q[1], q[0];
```

constructs the gate

$$R_z(\tau) = \exp(-i\tau Z/2) = \begin{pmatrix} e^{-i\tau/2} & 0 \\ 0 & e^{i\tau/2} \end{pmatrix} = e^{-i\tau/2} \begin{pmatrix} 1 & 0 \\ 0 & e^{i\tau} \end{pmatrix} \tag{4}$$

and applies the controlled form of that phase rotation gate

$$I \oplus R_z(\pi/2) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{-i\pi/4} & 0 \\ 0 & 0 & 0 & e^{i\pi/4} \end{pmatrix}. \tag{5}$$

In OpenQASM 2, we defined gates in terms of built-in gates U and CX. In OpenQASM 3, we can define controlled gates using the control modifier, so it is no longer necessary to include a built-in CNOT gate. For backwards compatibility, we include the gate CX in the standard gate library, but it is no longer a keyword of the language.

3.1.1 Standard gate library

We define a standard library of OpenQASM 3 gates in a file we call `stdgates.inc`. Any OpenQASM 3 program that includes the standard library can make use of these gates.

```
// OpenQASM 3 standard gate library

// Gate for OpenQASM 2 backwards compatibility
gate CX c, t { ctrl @ U(pi, 0, pi) c, t; }

// phase gate
gate phase(lambda) q { U(0, 0, lambda) q; }
// controlled-phase
gate cphase(lambda) a, b { ctrl @ phase(lambda) a, b; }
// identity or idle gate
gate id a { U(0, 0, 0) a; }

// Pauli gate: bit-flip
```

```
gate x a { U(pi, 0, pi) a; }
// Pauli gate: bit and phase flip
gate y a { U(pi, pi/2, pi/2) a; }
// Pauli gate: phase flip
gate z a { U(0, 0, pi) a; }

// Clifford gate: Hadamard
gate h a { U(pi/2, 0, pi) a; }
// Clifford gate: sqrt(Z) phase gate
gate s a { U(0, 0, pi/2) a; }
// Clifford gate: inverse of sqrt(Z)
gate sdg a { U(0, 0, -pi/2) a; }

// sqrt(S) phase gate
gate t a { U(0, 0, pi/4) a; }
// inverse of sqrt(S)
gate tdg a { U(0, 0, -pi/4) a; }

// quantum experience gates
gate u1(lambda) q { U(0, 0, lambda) q; }
gate u2(phi, lambda) q { gphase(-(phi+lambda)/2); U(pi/2, phi, lambda) q; }
gate u3(theta, phi, lambda) q { gphase(-(phi+lambda)/2); U(theta, phi, lambda) q; }

// Rotation around X-axis
gate rx(theta) a { U(theta, -pi/2, pi/2) a; }
// rotation around Y-axis
gate ry(theta) a { U(theta, 0, 0) a; }
// rotation around Z axis
gate rz(theta) a { gphase(-theta/2); U(0, 0, theta) a; }
// controlled-rx
gate crx(theta) a, b { ctrl @ rx(theta) a, b; }
// controlled-ry
gate cry(theta) a, b { ctrl @ ry(theta) a, b; }
// controlled-rz
gate crz(theta) a, b { ctrl @ rz(theta) a, b; }

// controlled-NOT
gate cx c, t { CX c, t; }
// controlled-Y
gate cy a, b { ctrl @ y a, b; }
// controlled-Z
gate cz a, b { ctrl @ z a, b; }

// swap
gate swap a, b { cx a, b; cx b, a; cx a, b; }

// controlled-H
gate ch a, b { ctrl @ h a, b; }

// Toffoli
gate ccx a, b, c { ctrl @ ctrl @ x a, b, c; }
// controlled-swap
gate cswap a, b, c { ctrl @ swap a, b, c; }

// four parameter controlled-U gate with relative phase gamma
gate cu(theta, phi, lambda, gamma) c, t { U(0, 0, gamma) c; ctrl @ U(theta, phi, lambda) c, t; }
```

3.2 Gate modifiers

Here we introduce a mechanism for users to modify existing unitary gates to define new ones. A modifier defines a new unitary gate, acting on a space of equal or greater dimension, that can be applied as an instruction. For programming convenience and compiler analysis,

modifers exist for inverting, exponentiating, and controlling gates.

The modifier `inv @ u` represents the inverse $U^\dagger$ of the gate U. The inverse of any gate $U = U_m U_{m-1} \dots U_1$ can be defined recursively by reversing the order of the gates in its definition and replacing each of those with their inverse $U^\dagger = U_1^\dagger U_2^\dagger \dots U_m^\dagger$. The base case is given by replacing `inv @ u(`θ`, `φ`, `λ`)` by `u(-`θ`, -`λ`, -`φ`)` and flipping the sign of any global phase. The inversion modifier commutes with the other modifiers defined below. For example,

```
gate rz(tau) q { gphase(-tau/2); U(0, 0, tau) q; }
inv @ rz(π/2) q[0];
```

applies the gate `inv @ rz(`$\pi/2$`)` defined by `{gphase(`$\pi/4$`); U(0,-`$\pi/2$`,0);}`. A statement such as `inv @ ctrl @ rz(`$\pi/2$`) q[1], q[0];` is equivalent to `ctrl @ inv @ rz(`$\pi/2$`) q[1], q[0];` since $(I \oplus U)^\dagger = I \oplus U^\dagger$.

The modifier `pow(r) @ U` represents the rth power U^r of the gate U where r is a floating point number. Every unitary matrix U has a unique principal logarithm iH whose eigenvalues iE are purely imaginary and satisfy $-\pi < E \leq \pi$. For any real r, we can choose the define the rth power as $U^r = e^{irH}$. For example,

```
gate z q { U(0, 0, π) q; }
gate s q { pow(1/2) @ z q; }
```

defines the phase gate S as the square root of the gate $Z = \left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$. The principal logarithm is $\log Z = \left(\begin{smallmatrix} 0 & 0 \\ 0 & i\pi \end{smallmatrix}\right)$ from which we find $S = e^{\frac{1}{2}\log Z} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & i \end{smallmatrix}\right)$. A compiler pass is responsible for resolving S into gates. In the case where r is an integer, the rth power can be implemented simply, albeit inefficiently. The integral rth power `pow(r) @ U` is equivalent to r repetitions of `u` when $r > 0$ or r repetitions of `inv @ u` when $r < 0$. Continuing our previous example,

```
pow(-2) @ s q[0];
```

defines and applies S^{-2}. This can be compiled to

```
inv @ pow(2) @ s q[0];
```

and further simplified into

```
inv @ z q[0];
```

without solving the problem of synthesizing S. In general, however, compiling `pow(r) @ u` involves solving a circuit synthesis or optimization problem, even in the integral case.

The modifier `ctrl @ u` represents a controlled-U gate with one control qubit. This gate is defined on the tensor product of the control space and target space by the matrix $I \oplus U$ where I has the same dimensions as U. A compilation target may or may not be able to execute the controlled gate without further compilation. The user or compiler may rewrite the controlled gate to use additional ancillary qubits to reduce the size and depth of the resulting circuit.

We can define the Fredkin (controlled-swap) gate in two equivalent ways as follows

```
gate x q { U(π, 0, π) q; }
gate cx c, t { ctrl @ x c, t; }
gate toffoli c0, c1, t { ctrl @ cx c0, c1, t; }
gate fredkin1 c, a, b {
    cx b, a;
    toffoli c, a, b;
    cx b, a;
}
gate swap a, b { cx a, b; cx b, a; cx a, b; }
gate fredkin2 c, a, b { ctrl @ swap c, a, b; }
```

The definitions are equivalent because they result in the same unitary matrix for both Fredkin gates. If we directly apply the definitions, `fredkin1` involves fewer controlled operations than `fredkin2`. A compiler pass may infer this fact or even rewrite `fredkin2` into `fredkin1`. Note that although the `toffoli` gate is well defined here, the compiler or user needs to do further work to synthesize it.

If we go one step further, we can define a controlled Fredkin gate. We require only one controlled `toffoli` gate after we use the $ABA^\dagger$ pattern in the `fredkin1` gate to remove unneeded controls from the `cx` gates. Although the controlled `toffoli` gate is defined on four qubits, it may be beneficial for a compiler or user to synthesize the gate using extra qubits as scratch space to reduce the total gate count or circuit depth. The following example is one way to use an extra qubit of space.

```
// Require: the scratch qubit is 'clean', i.e. initialized to zero
// Ensure: the scratch qubit is returned 'clean'
gate cfredkin2 c0, c1, a, b, scratch {
    cx b, a;  // from fredkin1
    toffoli c0, c1, scratch;  // implement ctrl @ toffoli c0, c1, a, b
    toffoli scratch, a, b;    // using extra scratch space
    toffoli c0, c1, scratch;
    cx b, a;  // from fredkin1
}
```

3.3 Non-unitary operations

The language includes two fundamental non-unitary operations.

The statement `reset qubit` resets a qubit the state $|0\rangle$. Mathematically, this corresponds to a partial trace over those qubits (i.e. discarding them) before replacing them with $|0\rangle\langle 0|$. The `reset` statement also works with an array of qubits, resetting them to the state $|0\rangle \oplus \ldots \oplus |0\rangle$.

The statement `bit = measure qubit;` measures the qubit in the Z-basis and assigns the measurement outcome to the target bit. For backwards compatibility this is equivalent to `measure qubit|qubit[] -> bit|bit[];` which is also supported. Measurement corresponds to a projection onto one of the eigenstates of Z, and qubit(s) are immediately available for further quantum computation. The `measure` statement also works with an array of qubits and an array of bits. The example below initializes, flips, and measures a register of 10 qubits.

```
qubit[10] qubits;
bit[10] bits;
reset qubits;
x qubits;
bits = measure qubits;
```

3.4 Real-time classical computing

Earlier versions of OpenQASM primarily describe *static* circuits. The only mechanism for conditionals was an `if` statement that modified execution of a single `gate`. This constraint was largely imposed by corresponding limitations in control hardware. However, the family of *dynamic* circuits with classical control flow and concurrent classical computation is a richer model of computation. Dynamic circuits are necessary for eventual fault-tolerant quantum computers that must interact with real-time decoding logic. In the near term, these circuits also allow for experimentation with qubit re-use, teleportation, iterative algorithms, and so on. Dynamic circuits have motivated significant advances in control hardware capable of moving and acting upon real-time values [35–37].

To take advantage of these advances in control, we extend OpenQASM with classical data types, arithmetic and logical instructions to manipulate the data, and control flow keywords. These extensions make OpenQASM Turing-complete for classical computations, augmenting the prior capability to describe static circuits composed of unitary gates and measurement. In considering what classical instructions to add, we took a conservative approach, selecting only those elements we thought likely to be used frequently. For instance, we felt it important to be able to describe arithmetic associated with looping constructs and the bit manipulations necessary to shuttle data back and forth between qubit measurements and other classical registers. However, instead of continuing to bulk up the feature set for classical computation — which would require sophisticated classical compiler infrastructure to manage — we instead introduced a *kernel* mechanism to connect OpenQASM3 circuits to arbitrary, opaque classical computations.

These classical *kernel*s are declared like function declarations in a C header file. That is, the programmer provides a signature like:

```
kernel vote(bit a, bit b, bit c) -> bit;
```

and then she can use `vote` like a subroutine acting on classical `bits`. In order to connect to existing classical compiler infrastructure, the definition of `vote` is not embedded within OpenQASM3, but is expected to be provided to the compilation tool chain in some other language format such as python, C, x86 assembly, or LLVM IR. This allows kernels to be compiled with tools like GCC or LLVM with a minimal amount of additional structure to manage execution within the run-time requirements of a global controller. Importantly, this strategy of connecting to existing classical programming infrastructure implies that the programmer can use existing libraries without porting the underlying code into a new programming language.

Unlike other approaches to interfacing with classical computing[13], invoking a kernel function does not imply a synchronization boundary at the call site. A compiler is free to

use dataflow analysis to insert any required target-specific synchronization primitives at the point where outputs from kernel call are used. Said differently, invoking a kernel function *starts* a classical computation but does not wait for that calculation to terminate.

Classical types:

For simple computations, classical operations can be directly embedded within an OpenQASM3 circuit. For this purpose, we introduce classical data types like signed/unsigned integers and floating point values in order to have well-defined semantics for arithmetic operations. The OpenQASM type system was designed with two distinct requirements in mind. When used within high-level logical OpenQASM ciruits the type system must capture the programmer's intent. It must also remain portable to different kinds of quantum computer controllers. When used within low-level physical OpenQASM circuits the type system must reflect the realities of particular controller, such as limited memory and well-defined register lengths.

OpenQASM takes inspiration from type systems within classical programming languages but adds some unique features for dynamic quantum circuits. For high-level logical OpenQASM we introduce some common types which will be familiar to most programmers: `int` for signed integers, `uint` for unsigned integers, `float` for floating point numbers, `bit` for individual bits, and `bool` for boolean true or false values. The precision of these types is not specified within the OpenQASM language itself, rather the precision is presumed to be a feature of a particular controller. Simple mathematical operations are available on these types. More complex classical operations are delegated to kernel functions, as above.

```
int x = 4;
int y = 2 ** x; // 16
```

For low-level physical OpenQASM we introduce a special syntax for defining arbitrary bit widths of classical types. In addition to the hardware-agnostic `int` type OpenQASM3 also allows programmers to specify an integer with exact precision `int[n]` for any positive integer `n`. This kind of flexibility is common in hardware description languages like VHDL and Verilog where manipulations of individual bits within registers are common. We expect similar kinds of manipulations to be common in OpenQASM3, and so we allow for direct access to bit-level updates within classical values. For example,

```
uint[8] x = 0;
// some time later...
bit a = x[2]; // read the 3rd least-significant bit of 'x' and assign it to 'a'
x[7] = 1;     // update the most-significant bit of 'x' to 1
```

shows direct access to the underlying bits in classical values.

Despite this flexibility at the language level, in practice we expect OpenQASM3 implementers to only support particular bit widths of each classical type. For example, some target hardware might only allow for `uint[8]` and `uint[16]` while other targets might only offer `uint[20]`. If the programmer does not require a parcticular bit width she can use the aforementioned type `uint` with the bit width unspecified, although in this case bit-level updates are not permitted.

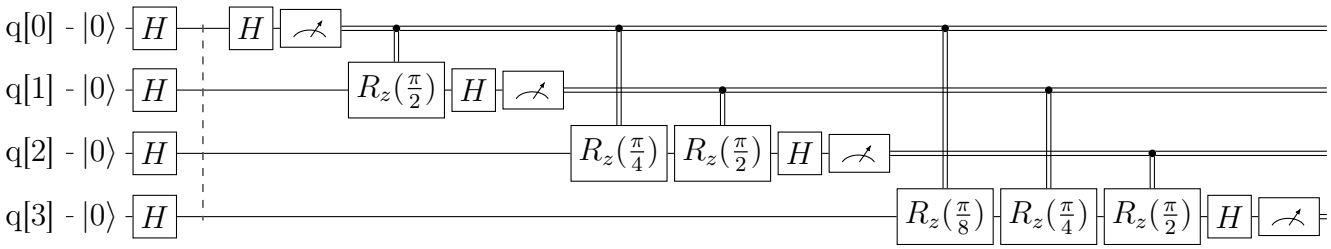


Figure 2: Inverse quantum Fourier transform with the loop unrolled.

OpenQASM3 breaks from the standard set of types with the introduction of a fixed-point `angle[n]` for representing angles or phases. Values stored in `angle` types represent values in the range $[0, 2\pi)$ with a binary decimal expansion of the form $2\pi * 0.c_n c_{n-1} \cdots c_1$. For example, with an `angle[4]`, the binary representation of π would be `1000`, $\pi/2$ would be `0100`, and so forth. This representation of phases is common to both quantum phase estimation[?] as well as numerically-controlled oscillators. The latter case is particularly relevant to real-time control systems that need to track qubit frames at run-time, where they can take advantage of numerical overflow to automatically constrain the phase representation to the domain $[0, 2\pi)$, obviating the need for expensive floating-point modular arithmetic. The confluence of this angle representation in controllers and quantum algorithms influenced us to add first-class support for this type in OpenQASM3.

Classical control flow:

OpenQASM2 programs were effectively limited to straight line code. Quantum algorithms with more involved constructs such as loops were implemented in a higher-level tool such as Python and then synthesized into a long list of instructions. These lists could end up being quite lengthy in some cases and lost all the structure from their original synthesis. Consider an inverse quantum Fourier transform circuit as written in OpenQASM2 (Figure 2):

```
// inverse quantum Fourier transform with the loop unrolled
qreg q[4];
creg c[4];

reset q;
h q;
barrier q;
h q[0];
measure q[0] -> c[0];
if(c[0] == 1) { rz(pi / 2) q[1]; }
h q[1];
measure q[1] -> c[1];
if(c[0] == 1) { rz(pi / 4) q[2]; }
if(c[1] == 1) { rz(pi / 2) q[2]; }
h q[2];
measure q[2] -> c[2];
if(c[0] == 1) { rz(pi / 8) q[3]; }
if(c[1] == 1) { rz(pi / 4) q[3]; }
if(c[2] == 1) { rz(pi / 2) q[3]; }
h q[3];
measure q[3] -> c[3];
```

One can recognize a natural iterative structure to this program, but the lack of a looping

construct has forced us to unroll this structure when writing it out. Furthermore, each iteration has a series of `if` statements that in principle could be combined, as the body of those statements is a Z rotation in all cases.

OpenQASM3 introduces `while` loops and `for` loops. It also extends the `if` statement to allow for multiple instructions within the body of the `if` and allows for an `else` block to accompany it.

Now we can instead write the above program as:

```
// inverse quantum Fourier transform with for loop
qubit q[4];
angle[4] c = 0;
bit b;

reset q;
h q;
barrier q;
for i in [0:3] {
    c >>= 1;
    rz(c) q[i];
    h q[i];
    measure q[i] -> b;
    c[3] = b;
}
```

which preserves the iterative structure of the inverse QFT and uses the `angle[4]` type to directly convert measurement bits into the appropriate Z rotations.

Subroutines:

While classical functions could be defined using the kernel construct, sufficiently simple functions (accompanied by sufficiently capable controllers) allow for subroutines to be fully implemented within the language itself. Consider again the above `vote` function. This could also be implemented using a subroutine:

```
def vote(bit a, bit b, bit c) -> bit {
    int count = 0;
    if (a == 1) count++;
    if (b == 1) count++;
    if (c == 1) count++;

    if (count >= 2) {
        return 1;
    } else {
        return 0;
    }
}
```

3.5 Parameterized Programs

The previous section describes the real-time classical compute constructs of OpenQASM. OpenQASM 3 also introduces a new feature targeted at near-time computation: parameterized programs [38].

For variational quantum algorithms a program is written once with some free parameters. A classical optimizer is used to determine the next set of parameters to use for each

subsequent run of a program. This effectively allows the programmer to communicate many different programs with a single program file. Critically it also means the programmer only has to compile the program once, thus amortizing the cost of compilation across many runs. In this model the compiler would produce an executable that leaves these free parameters unspecified and a program run would take as input both the executable and some choice of the parameters.

Consider a simple example of measuring a program using multiple bases:

```
input int basis;  // 0 = X basis, 1 = Y basis, 2 = Z basis
output bit result;
qubit q;

// Some complicated program...

if (basis == 0) h q;
else if (basis == 1) rx(pi/2) q;
result = measure q;
```

The above code introduces two new constructs for parametric programs: the `input` modifier for specifying that a classical variable will be provided at runtime, and the `output` modifier for specifiying which variables should be returned. It is also possible to specify multiple input variables and multiple output variables by using the modifiers on more than one variable.

OpenQASM 2 did not allow for input variables, nor did it allow the programmer to specific which variables to return. For backwards compatibility, if the compiler sees no `input` or `output` modifiers it will assume that there are no input variables and that all declared classical variables are output variables.

As another example consider the Variable Quantum Eigensolver (VQE) algorithm [39]. In this algorithm the same circuit is repeated many times using different sets of free parameters to minimize an expectation value.

```
input angle param1;
input angle param2;
qubit q;

// Build an ansatz using the above free parameters, eg.
rx(param1) q;
ry(param2) q;

// Estimate the expectation value and store in an output variable
```

The following Python pseudocode illustrates the differences between using and not using parameterized programs:

```
# Example without using parametric programs:

for theta in thetas:
    # Create an OpenQASM program with theta defined
    program = subsitute_theta(read("program.qasm"))

    # The slow compilation step is run on each iteration of the inner loop
    binary = compile_qasm(program)
    result = run_program(binary)

# Example using parametric programs:
```

```
# parametric_program.qasm begins with the line "input angle theta;"
program = read("parametric_program.qasm")
# The slow compilation step only happens once
binary = compile_qasm(program)

for theta in thetas:
    # Each iteration of the inner loop is reduced to only running the program
    result = run_program(binary, theta=theta)
```

Note that in the second example, parametric programs allows us to hoist compilation out of the loop over the parameter theta, which can lead to a substantial performance benefit.

3.6 Timing

A key aspect of expressing code for quantum experiments is the ability to control the timing of gates and pulses. Examples include characterization of decoherence and crosstalk [40], dynamical decoupling [41–43], dynamically corrected gates [44, 45], and gate parallelism or scheduling [46]. We enhance OpenQASM with timing semantics to enable such circuits. We use relative timing, i.e. via `delay` instructions, which allows for more flexibility compared to absolute timing. For example, a qubit's relaxation time (T1) can be measured by an OpenQASM snippet of the following form. Notice the inclusion of a new `length` type, and built-in SI units of time.

```
length stride = 1us;   // a length of time, specified in SI units
p = 3;                 // record the 3rd datapoint in the relaxation curve
reset q[0];
x q[0];
delay[p * stride] q[0];
c0 = measure q[0];       // survival probability of the excited state
```

Stretch types:

While simple experiments can be represented with hardcoded timing, in more complicated circuits this can be challenging given the potential heterogeneity of calibrated gates and their various durations. It is useful to specify gate timing and parallelism in a way that is independent of the precise duration and implementation of gates at the pulse-level description. In other words, we want to provide the ability to capture *design intent* such as "space these gates evenly to implement a higher-order echo decoupling sequence" or "left-align this group of gates" or "apply this gate for the duration of some other sub-circuit". This increases program portability by decoupling circuit timing intent from the underlying pulses, which may change from machine to machine or even from day to day.

To achieve this we introduce a new type `stretch`, which is a sub-type of `length`. "Stretchy" lengths are resolvable to concrete durations at compile time, after the exact duration of calibrated gates are known. They have a natural length of zero, but may stretch to arbitrary lengths in order to fulfill some given constraints. This concept is inspired by how a similar problem is solved in LaTeX [47] using "glues", where the compiler spaces out letters, words, etc. appropriately given a target font. In quantum circuits we have the additional challenge that timings on different qubits are not independent, e.g. if a two-qubit gate connects the qubits. This means that the choice of timing on one qubit (line) can have

a ripple effect on other qubits. Later we will show how the compiler can formulate and solve this problem.

A simple use case for stretchable delays are gate alignments. We can use this feature to gain control on how gates are aligned in a circuit, regardless of the latencies of those gates on the machine. For example a left alignment code snippet is shown below and the circuit is illustrated in Figure 3a. Note the use of spring symbols to denote `stretch`. After the circuit is lowered to a particular machine with known gate calibrations and durations, the compiler resolves every stretchy `delay` instruction into one with concrete timing.

```
// left alignment
stretch g1, g2, g3;
barrier q;
cx q[0], q[1];
delay[g1] q0, q1;
u q[2];
delay[g2] q2;
cx q[3], q[4];
delay[g3] q3, q4;
barrier q;
```

We can further control the exact alignment by giving relative weights to the stretchy delays (Figure 3b): This method is flexible, e.g. to align a gate at the 1/3 point of another gate, we can use a `delay[g]` instruction before, and a `delay[2*g]` after the gate to be aligned.

```
// one-third alignment
stretch g;
barrier q;
cx q[0], q[1];
delay[g];
u q[2];
delay[2*g];
barrier q;
```

Instructions other than delay can also be stretchy, if they are explicitly defined as such. They can be called by passing a valid `length` as their duration. Consider for example a rotation called `rotary` that is applied for the entire duration of some other gate [48]. Notice the special square bracket syntax for duration which makes it distinct from other parameters of the gate, as the compiler will use this information to schedule the circuit. Once resolved to a concrete time, it can be passed to the gate definition (in terms of pulses) to realize the gate.

```
    // stretchy rotation gate
    stretch g;
    cx control, target;
    rotary(amp)[g] spectator;    // rotate for some duration, with some pulse amplitude
    rotary(-amp)[2*g] spectator; // rotate back, for twice that duration
    rotary(amp)[g] spectator;    // rotate forward again, making overall identity
```

Boxed expressions:

We introduce a `box` keyword for scoping a particular part of the circuit. A boxed sub-circuit is different from a `gate` or subroutine (`def`), as it is not a definition but merely a pointer to

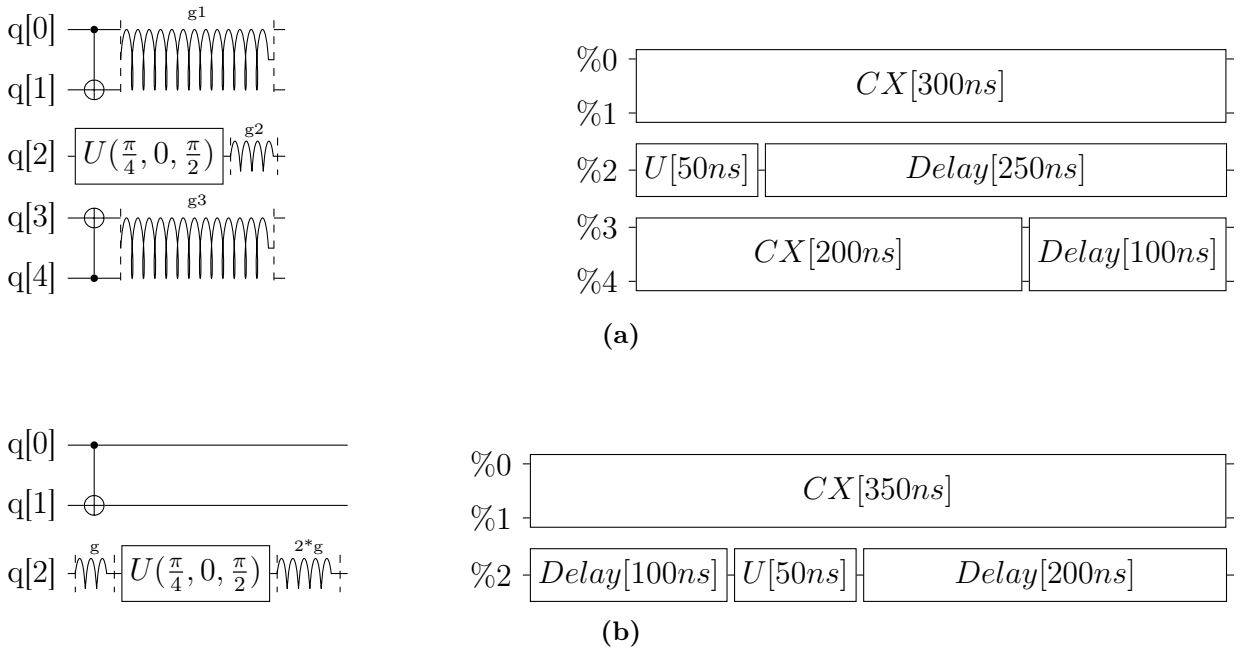


Figure 3: Arbitrary alignment of gates in time using stretchy delays. a) left-justified alignment intent, and timed instructions after stretches are resolved. b) alignment intent of a short gate at the one-third point of a long gate, and after stretch resolution.

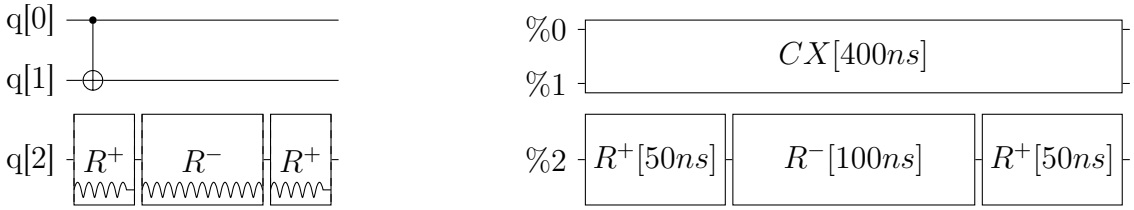


Figure 4: Dynamically corrected CNOT gate where the spectator has a rotary pulse. The rotary gates are stretchy, and the design intent is to interleave a "winding" and "unwinding" that is equal to the total duration of the CNOT. We do this without knowledge of the CNOT duration, and the compiler resolves them to the correct length during lowering to the target backend.

a piece of code within the larger scope it resides in. This enables several features, such as bounding the scope of `stretch`es for the solver as well as being able to refer to the overall duration of the box in other parts of the circuit. The latter is accomplished by labeling the box. The box isolates its instructions, preventing them from being combined with gates outside the box. This is key for expressing timing at the level of sub-circuits: no matter how the contents of the box get optimized, referring to its duration as `lengthof(boxlabel)` has a well-defined meaning.

The benefit of boxed expressions go beyond timing. Boxes can be used to signal permissible optimizations to the compiler. Optimizing within a box is ok, and optimizing across a full box is ok, but optimizing across the box *boundary* is not. For example, consider a dynamical decoupling sequence inserted in a part of the circuit. By boxing the sequence, we create a box that implements the identity. The compiler is now free to commute a gate past the box by knowing the unitary implemented by the box. Delays that are within a box are implementation details of the box; they are invisible to the outside scope and therefore do not prevent commutation. The compiler can thus perform optimizations without interfering with the implementation of the dynamical decoupling sequence.

```
// using box to label a sub-circuit, and later delaying for that entire duration
box mybox {
    cx q[0], q[1];
    delay[200ns] q[0];
}
delay[lengthof(mybox)] q[2], q[3];
cx q[2], q[3];
```

Lastly boxes can be used to put hard deadlines on the execution of a particular sub-circuit. This is especially useful in scenarios where the exact duration of a piece of code is unknown (perhaps it is runtime dependent). Yet we still would like to impose a duration on it in order to be able to schedule the larger circuit. The contents of it will be boxed, and in addition a total duration will be assigned to the box. The natural length of instructions within a box must be smaller than the declared box duration, otherwise a compile-time error will be raised. The `stretch` inside the box will always be set to fill the difference between the declared length and the natural length. If the instructions exceed the deadline at runtime, a runtime error is raised.

```
// define a 1ms box whose content is just a centered CNOT
box 1ms {
    stretch a;
    delay[a] q;
    cx q[0], q[1];
    delay[a] q;
}
```

Barrier instruction:

The `barrier` instruction of OpenQASM2 prevents commutation and gate reordering on a set of qubits across its source line. The syntax is `barrier qregs|qubits;` and can be seen in the following example:

```
// barrier (or its generalized form of delay) provide time synchronization and optimization constraints
cx r[0], r[1];
h q[0];
h s[0];
barrier r, q[0];
h s[0];
cx r[1], r[0];
cx r[0], r[1];
```

This will prevent an attempt to combine the CNOT gates but will not constrain the pair of `h s[0];` gates, which might be executed before or after the barrier, or cancelled by the compiler. The `barrier` in OpenQASM3 can now be seen as a special case of `delay` (with a length of zero). Delays can therefore be seen to provide time synchronization across the qubits on which they are applied.

Solving the stretch problem:

We now expand on the constraints that result from `length`s, `stretch`es and `box`es. We permit `length`s to be variables in a linear system of inequalities that the compiler must later solve to satisfy the constraints. We call this problem the "stretch problem". If we attempt to execute an OpenQASM program, a stretch solving algorithm must be applied to compute explicit durations for all delays.

The instruction `delay[x] q;` means delay for exactly x time on qubit q. The gates on qubit q before and after the delay are understood to occur immediately before and immediately after the delay. If the constraint cannot be satisfied, we would raise an exception at or before the stage where we attempt to build an explicit schedule.

Consider the OpenQASM fragment `x q[0]; x q[1];`. This means that we intend to apply two unitary operations to separate (virtual) qubits `q[0]` and `q[1]`. We have not introduced any timing constraints, and there is no data dependence, so we do not care when we apply the gates. Later compiler passes are free to rewrite the circuit to an equivalent one, provided they respect the semantics of OpenQASM.

Eventually it is necessary to explicitly schedule and execute the program. If we try to explicitly schedule a program without any stretchable delays, the stretch problem may be unsolvable in all but the most trivial cases. By writing `stretch a, b; x q[0]; delay[a] q[0]; x q[1]; delay[b] q[1];` we allow delays to be inserted after the gates if necessary. There is enough information to generate scheduled instructions. In this case we have asked for left alignment, but we could write other programs to request different schedules.

To summarize, we imagine the stretches get resolved to explicit delays in a single late-stage pass that solves the stretch problem, after which all delays are fully specified and the schedule is complete for a given target. It is possible that the delays cannot be resolved as written for a given target and we raise an error. Most OpenQASM programs require additional compilation to run on a given target.

The stretch problem is a lexicographic multi-objective linear programming problem [49] that arises from the need to solve for unknown durations in a quantum circuit. The problem is as follows: given vectors c_i, b, and matrix A, lexicographically maximize $c_1^T x$, $c_2^T x$, ..., $c_m^T x$ subject to $Ax \leq b$ and $x \geq 0$. This linear programming problem can be solved in polynomial time and efficiently in practice; for example, multiobjective optimization is implemented in software such as CPLEX [50]. One way to solve the multiobjective problem is to iteratively

solve a sequence of related linear programming problems. First solve "maximize $c_1^T x$ subject to $Ax \leq b$ and $x \geq 0$" to obtain the cost β_1. Next solve "maximize $c_2^T x$ subject to $Ax \leq b$, $c_1^T x = \beta_1$, and $x \geq 0$" to obtain the cost β_2. Continue in this way until all objectives are met. This is described in [49] and the references therein.

Our goal is that any OpenQASM program with mapped qubits and `defcal`s gives rise to a set of stretch problems that, if bounded and feasible, provides explicit timing for all operations within every basic block. We define the stretch problem such that it is bounded and there is a unique solution if the problem is feasible. If the problem is infeasible, the compilers fails to solve it providing some insight about the unsatisfied constraints.

The stretch problem is formulated independently for each basic block or box B. The block B consists of instructions applied to qubits Q. An instruction is any task with a non-negative duration that involves a subset of qubits. The duration of an instruction is either known or unknown. Unknown durations are affine functions of unknown stretch variables $\{s_i\}$, $i = 0, 1, 2, \ldots$. When there is a possibility to optimize two different stretch variables simultaneously, the earliest objective with the smallest i is optimized first. The block B has total duration T that we wish to minimize. A valid assignment of durations to stretch variables ensures that an instruction occurs on each qubit $q \in Q$ at each time $t \in [0, T]$. In other words, for each qubit there is a linear equation $x_q(\{s_i\})$ that is the sum of all the durations for operations on qubit $q \in Q$. For all q, these durations must equal the total duration $x_q(\{s_i\}) = T$. Multi-qubit blocks add alignment constraints that correspond to each input qubit being available at the same time.

For example, the code below inserts a dynamical decoupling sequence where the *centers* of pulses are equidistant from each other. The circuit is depicted in Figure 5. We specify correct lengths for the delays by using backtracking operations to properly take into account the finite length of each gate.

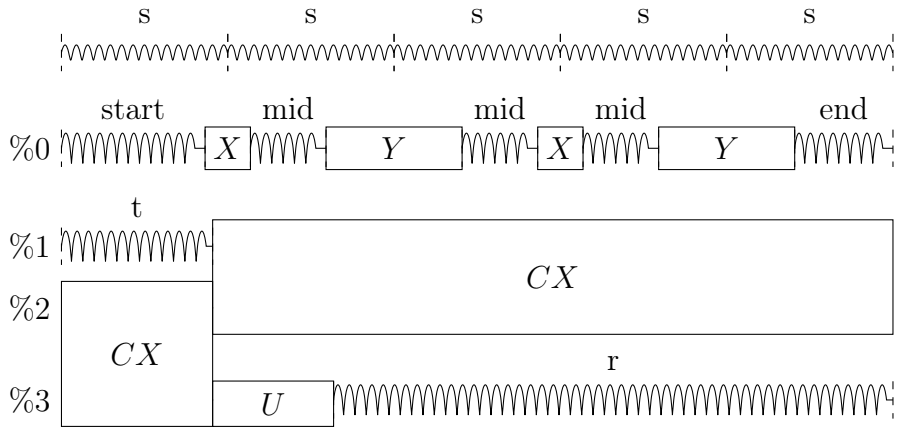


Figure 5: Dynamical decoupling of a spectator qubit using finite-duration pulses. This design intent can be expressed by defining a single stretch variable s that corresponds to the distance between equidistant gate centers. The other lengths which correspond to actual circuit delays are derived by simple arithmetic on lengths. Given a target system with calibrated X and Y gates, the solution to the stretch problem can be found.

```
stretch s, t, r;
length start_stretch = s - .5 * lengthof({x $0;});
length middle_stretch = s - .5 * lengthof({x $0;}) - .5 * lengthof({y $0;});
length end_stretch = s - .5 * lengthof({y $0;});
box {
    delay[start_stretch] $0;
    x $0;
    delay[middle_stretch] $0;
    y $0;
    delay[middle_stretch] $0;
    x $0;
    delay[middle_stretch] $0;
    y $0;
    delay[end_stretch] $0;
    delay[t] $1;
    cx $2, $3;
    cx $1, $2;
    u $3;
    delay[r] $3;
}
```

From the delay statements, $s - d_6/2 \geq 0$, $s - d_6/2 - d_8/2 \geq 0$, $s - d_8/2 \geq 0$, $t \geq 0$ and $r \geq 0$. By definition, $d_6 = d_{10}$ and $d_8 = d_{12}$. We have $t = d_{15}$ and $r = d_{16} - d_{17}$. The qubit durations are $x_{q[0]} = 5s$, $x_{q[1]} = t + d_{16}$, $x_{q[2]} = d_{15} + d_{16}$, and $x_{q[3]} = d_{15} + d_{17} + r$. For all i and j, $x_{q[i]} = x_{q[j]}$. We solve

$$t = d_{15} \tag{6}$$
$$5s = d_{15} + d_{16} \tag{7}$$
$$p = d_{16} - d_{17} \tag{8}$$
$$s \geq d_6/2 + d_8/2 \tag{9}$$
$$t, r \geq 0 \tag{10}$$

which is feasible if $d_{15} + d_{16} \geq 5(d_6 + d_8)/2$ and $d_{16} \geq d_{17}$, in which case $(s, t, r) = ((d_{15} + d_{16})/5, d_{15}, d_{16} - d_{17})$.

The careful reader will notice that the order in which the constraints are introduced and prioritized matters. Consider the simple example below which shows how stretch order resolves ambiguity.

```
stretch a, b, c, d;
delay[a] q[0];
delay[b] q[1];
h q[0];
h q[1];
delay[c] q[0];
delay[d] q[1];
```

We have

$$\text{lexmin } a, b, c, d \text{ s.t.} \tag{11}$$
$$a, b, c, d \geq 0 \tag{12}$$
$$a + d_4 + c = b + d_5 + d \tag{13}$$

and so find $a = b = 0$ and $(c, d) = (d_5 - d_4, 0)$ if $d_5 \geq d_4$ or $(c, d) = (0, d_4 - d_5)$ otherwise. This gives a left aligned schedule. If we solve the problem in reverse order, optimizing first d, then c, and so on, we will find a right aligned schedule.

3.7 Calibrating Quantum Instructions

OpenQASM is a circuit-model language, it is predominantly focused on specifying the application of unitary gates and projective measurements as instructions applied to qubits. Quantum operations are normally implemented with classical time-dependent signals coupled to quantum systems. These signals are emitted from many types of control hardware such as arbitrary waveform generators, flux sources, and lasers. These hardware elements must be orchestrated to synchronously emit the designed control fields to guide the quantum system to implement the desired quantum operation [51]. Consequently, the exact implementation both at the hardware and control engineering level is highly technology-dependent. For example superconducting transmon qubits, encode a qubit in a non-linear oscillator formed by a parallel circuit consisting of a Josephson junction and a capacitor. The state of which is manipulated by applying a series of shaped microwave control pulses [52]. As hardware is prone to drift due to environmental variations, these control pulses must also be regularly re-calibrated and re-linked with the quantum system [**?**]. The design of these control signals is an active area of research, and developments in this area has software updates to improve the performance of existing hardware systems [**?**]. OpenQASM is designed to enable this research by allowing programmers to specify instruction calibrations within their programs in the form of `defcal` declarations. These specify a microcoded [**?**] implementation of a `gate`, `measure`, or `reset` instruction.

For example, the declaration below instructs the compiler that the `x` gate is to be implemented by the instructions enclosed within the block for physical qubit 0.

```
defcalgrammar "openpulse";

defcal x $0 {
    play drive_ch($0), drag(duration, amp, ...);
}
```

The enclosed instructions are implemented in a user-specified *grammar* (otherwise known as a dialect). Support for `defcal` declarations is optional for OpenQASM compilers and target devices. `defcal`s have a similar form to `gate` declarations up to some subtle differences. Where a gate declaration specifies a gate-definition that defines an unknown gate in terms of other known gates, the defcal declares the implementation of a OpenQASM instruction for a target device. `defcal` statements are used to define the implementations of canonical instructions of the target hardware and may not be explicitly called (except possibly from within another `defcal`). Rather, the target system compiler links these definitions to the circuit-level OpenQASM program. This embeds an extendable translation layer between the circuit model and control-hardware implementations of quantum instructions in OpenQASM.

Calibration resolution

The signals required to enact gates on physical hardware vary greatly by the kind of gate, the physical qubits involved, and the gate parameters. Defcal declarations allow specification of gates on both generic and specific physical qubits, and both generic and specific gate parameters. For instance, `defcal rz(angle[20] theta) $q` defines the signals to enact an RZ gate of any angle on any physical qubit `$q`, whereas `defcal rx(pi) $0` specifically targets an X rotation of π radians on physical qubit `$0`. The reference to physical rather than virtual qubits is critical because quantum registers are no longer interchangeable at the pulse level.

At compile-time, specialized `defcal` declarations are greedily matched with qubit specializations matched before instruction arguments. If all parameters are not specialized, the declaration with the largest number of specialized parameters is matched first. In the event of a tie the compiler will choose the `defcal` block that appeared first in the program.

For instance, given:

```
defcal rx(angle[20] theta) $q // (1)
defcal rx(angle[20] theta) $0 // (2)
defcal rx(pi/2) $0 // (3)
```

the operation `rx(pi/2) $0` would match to (3), `rx(pi) $0` would match (2), and `rx(pi/2) $1` would match (1). This strategy is loosely inspired by Haskell's pattern matching for resolving function calls [53].

Calibrations for measurements are specified in a similar way. We distinguish gate and measurement definitions by the presence of a return value type: `defcal measure $0 -> bit`

The final calibration will be selected by the compiler post qubit layout when the OpenQASM program is input to the target system with `defcal` statements included. This enables each hardware system to decide how to compile, store, and reuse microcoded calibration definitions within its hardware. In practice, these definitions may be stored separately from the circuit-level user's program and explicitly included by the user, or linked by the compiler. This enables calibrations to be updated by simply including a new OpenQASM source file, while still giving the freedom to override a single calibration inline.

To integrate with OpenQASM's timing system, at scheduling time the compiler will request from the defcal implementation the duration of each `defcal` call and the physical qubits resources that it utilizes. The scheduler will use this information to resolve the scheduled circuit specified by the programmer. It is required that all resolved `defcal` usages must have deterministic durations. If a `defcal`'s duration depends on its input parameters, the parameters and consequently the definition must be resolvable at compile-time. Deterministic scheduling of the operations at the circuit level is enabled by combining these durations, along with the knowledge of which physical qubits are utilized by the gate application. Calibrations must also be position-independent, so as to allow their substitution anywhere the corresponding circuit instruction is applied. Note that some instructions, such as a `reset` instruction implemented with measurement feedback, might require control flow. A calibration grammar should support such use cases while still guaranteeing deterministic duration of the instruction.

Calibration grammars

In practice, describing the application of control-fields for various quantum platforms may be vastly different. For example, a typical SWAP gate within a superconducting qubit system is performed by applying three CX gates. In turn, each CX gate is implemented with a time-dependent microwave stimulus pulse applied to the target qubits. Comparatively, an ion-trap system may implement a SWAP gate by physically swapping the position of two ions through the application of a series of DC potentials and laser pulses - these may be global in nature [**?**]. The operations and correspondingly, the semantics required to describe these two operations may vary significantly from hardware vendor to vendor.

In OpenQASM we do not attempt to capture all such possibilities, but rather embrace the role of a pluggable multi-level intermediate representation. Vendors are free to define their calibration grammar (otherwise known as a dialect), and accompanying implementation. OpenQASM, in turn, provides the *defcalgrammar <grammar>* declaration which allows the programmer to inform the compiler implementation of which calibration grammar should be selected for the following calibration declarations. Calibration support is optional and OpenQASM compiler implementations that support defining calibrations should provide an interface for new grammars to plugin and extend the compilation infrastructure. It is then the responsibility of the hardware provider to specify a grammar and implement the required hooks into the compiler. This might be as simple as a defcal mapping to custom hardware calibrations on the system, or as complex as a fully-fledged pulse-programming language as explored below in Section 3.7.

By allowing vendors to extend the calibration grammar, OpenQASM provides a standardized pipeline for vendors to lower from a hardware-agnostic representation into one that is hardware-aware. It is the responsibility of the hardware vendor to consume the calibrations along with the rest of the OpenQASM, to compile the program into a final executable for the target hardware. The need for calibration grammars is an acknowledgment of the rapid pace of development within the quantum hardware space and the need for hardware/-software co-design to extract the maximum performance from today's quantum computers. With calibrations, OpenQASM may co-evolve as an intermediate representation alongside hardware as new developments in control technologies are made.

OpenPulse grammar

OpenPulse was initially specified as an API system that was later extended with an implementation for Qiskit [54, 55]. The OpenPulse grammar aims to provide a hardware-independent representation for programming qubits at the level of microwave pulses. The following grammar examples are tentative and are expected to evolve quickly. In the `defcal` declaration above the `x` gate is implemented in the OpenPulse grammar by `play`ing a `drag` pulse on the hardware drive channel for qubit 0 (`$0`). Within the OpenPulse grammar, each qubit is associated with several time-dependent input/output signal channels on which instructions with deterministic durations may be scheduled. All OpenPulse instructions operate on channels that map to qubits. Channels may be accessed by name, - `channel("name")`, or with one of the builtin channel getters which accept a physical qubits as input - `qubit_ch($q0, ..., $qn, "name")`, `drive_ch($q)`, `measure_ch($q)`, and `receive_ch($q)`. There are several

types of channels with such as `transmit` and `receive` channels which respectively enable signaling to and from the quantum device. Instructions may only accept specific channel types and are applied with assembly-style syntax.

The following defcal demonstrates how to define a generic implementation of a virtual Z-gate [**?**] for all qubits in the target system using a qubit wildcard.

```
defcalgrammar "openpulse";

defcal rz(angle[20] theta) $q {
    shift_phase(drive_ch($q), -theta);
}
```

The above example requests the drive channel responsible for stimulating the corresponding qubit, and then shifts the phase of the channel's `frame`. Frames represent a carrier signal $e^{i(2\pi ft+\phi)}$, parameterized by a frequency f and phase ϕ. Frames are tracked throughout a program execution by the hardware runtime and any modifications to a frame within a calibration entry will persist to calls to the same frame elsewhere in the program. This enables calibrations to be defined in a position-independent manner. Every *transmit* channel has a default frame which may be requested and assigned with the `frameof(<transmit>)` instruction. A frame may modified by the `shift_phase`, `set_phase`, `shift_freq` and `set_freq` instructions, which each accept a channel and `angle(float)` as an operand for the respective phase(frequency) operands. For example, `set_freq drive_ch($0), 5e9;` will set the frequency of qubit 0's drive channel to 5GHz.

Calibrations may reference other calibrations within the defcal scope to build up more sophisticated calibrations, as demonstrated with the echoed cross-resonance gate below [**?**].

```
defcal ecr $0 $1 {
    barrier $0, $1, $2;
    frame fd1 = frameof(drive_ch($1));
    playwfr drive_ch($0), flat_top_gaussian(duration, amp, ...), fd1;
    x $0;
    playwfr cr1_0, flat_top_gaussian(duration, -amp, ...), fd1;
    barrier $0, $1, $2;
}
```

This above example demonstrates several additional features of the OpenPulse language. The `barrier $0, $1, $2;` statement allows the OpenPulse grammar extension to communicate to the OpenQASM scheduler that instructions on physical qubit `$2` may not occur during this gate. This is a useful tool to inform the scheduler behaviour of cases where qubits are indirectly coupled and to avoid scheduling them simultaneously, for example, when many qubits are coupled through a resonator bus. The *play with frame* (`playwfr`) is then used to play a pulse with an explicit *frame*. The output pulse will be at the frequency and phase of the target qubit, allowing the implementation of the cross-resonance gate. The standard `play` instruction uses the default frame of the channel operand.

It is also possible to calibrate the `measure` and `reset` implementation, for example, an implementation of `measure` is shown below.

```
defcal measure $0 -> bit {
    play measure_ch($0) flat_top_gaussian(duration, amp, sigma, square_width);
    // Acquire instruction samples an acquisition channel and produces a bit
    return capture receive_ch($0), duration;
}
```

When implementing `defcal` a common pattern is to define a general form of a gate, while at the same time requiring a highly-tuned implementation of the same gate under a fixed parameterization. Consider the case of defining a calibration for a `rx` below,

```
defcalgrammar "openpulse"
length duration = 160μs;

gate rx(angle[20] theta) $0 {
    const angle[20] x_2pi = 0.4 / (2*pi);
    play drive_ch($0), gaussian(duration, fixed[0, 12](theta*x_2pi), ...);
}
```

This is calibrated such that the pulse that implements the `rx` gate has an amplitude proportional to the specified rotation angle `theta`. If the constant `alpha` is properly calibrated this may roughly work in practice, however, the error for a desired rotation angle will be relatively large due to nonlinearities in the control hardware and qubits. This may be a problem if we wish to implement a high-precision `rx(pi)` gate for our CX implementation below. We can take advantage of the defcal resolution order to implement a specialized gate:

```
defcal rx(pi) $0 {
    const fixed[0, 12] x_pi = 0.20011;
    play drive_ch($0), gaussian(duration, x_pi, ...);
}
```

Above we have specialized the `rx` gate calibration for an angle of π, allowing the experimentalist to tune up a highly calibrated gate for reuse, while still having the flexibility to implement a generic implementation of `rx`.

A standard `cx` gate is then constructed using the previously defined `rz`, `sx`, `rc` and `ecr` defcals. The CX gate definition is generic and applies to any pair of qubits so long as they implement the required gates as defcals either directly, or through a path of `gate` declarations that map to a terminating set of `defcals` for the qubit operands such as in the case of the `x` gate call in the definition below and the specialized `rx(pi)` defcal definition above.

```
gate cx q0, q1 {
    rz(pi/2) q0;
    sx q1;
    ecr q0, q1;
    x q0;
}
```

The `cx` gate may then be applied like a normal gate as shown below.

```
// Declare physical qubits 0 and 1.
qubit $0;
qubit $1;
// Apply user-defined gate
cx $0, $1;
measure $0;
```

3.8 Multi-level representation

In previous sections we introduced the language features of OpenQASM, which can range from relatively high-level constructs such as multi-controlled gates to low-level timing and

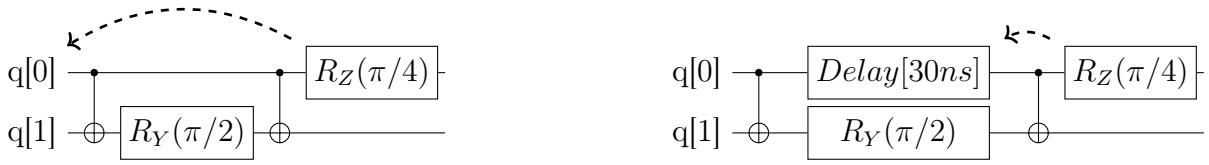


Figure 6: Implicit vs. explicit delay. (a) An implicit delay exists on $q[0]$, but it is not part of the circuit description. Thus this circuit does not care about timing and the RZ gate is free to commute on the top wire. (b) An explicit delay is part of the circuit description. The timing is consistent and can be resolved if and only if this delay is exactly the same length as RY on $q[1]$. The delay is like a barrier in that it prevents commutation on that wire. However RZ can still commute before the $CNOT$ if it has length 0.

microcoded pulses. As such, OpenQASM is designed to be a multi-level intermediate representation, where the focus shifts from target-agnostic computation to concrete implementation as more hardware specificity is introduced.

An OpenQASM circuit can also mix different abstraction levels by introducing constraints where needed, but allowing the compiler to make decisions where there are no constraints. Examples of program constraints are tying a virtual qubit to a particular physical qubit, or left-aligning some parallel gates that have different durations.

We illustrate this point by considering timing constraints in an OpenQASM circuit. When a `delay` instruction is used, even though it implements the identity channel in the ideal case, it is understood to provide explicit timing. Therefore an explicit `delay` instruction will prevent commutation of gates that would otherwise commute. For example in Figure 6a, there will be an implicit delay between the '`cx`' gates on qubit 0. However, the '`rz`' gate is still free to commute on that qubit, because the delay is implicit. Once the delay becomes explicit (perhaps at lower stages of compilation), gate commutation is prohibited (Figure 6b).

Furthermore, even though stretch is used to specify constraints for the solver, its use in the circuit is like any other instruction. This has the benefit that design intents can be specified on a high-level circuit and the intent will be carried with the circuit, until resolved. The following example for simultaneous randomized benchmarking is an instance of this, where alignment is specified on general unitaries of the Clifford group, which may be decomposed in a variety of ways. In addition, the stretchiness can be included as part of the definition of a gate, in which case whenever that gate is expanded those stretch constraints will be automatically inserted.

4 Compilation phases by example

In this section we present a full example using an OpenQASM circuit for an iterative phase estimation algorithm [56]. We show how the circuit may get transformed during multiple phases of compilation, and how OpenQASM can be used as the intermediate representation at each phase. This simple example also serves to highlight many of the features of the language: classical control flow, kernel functions, gate modifiers, virtual and physical qubits, timing and stretches, and pulse-defined calibrations.

Iterative phase estimation is an algorithm for calculating the eigenvalue (phase) of a

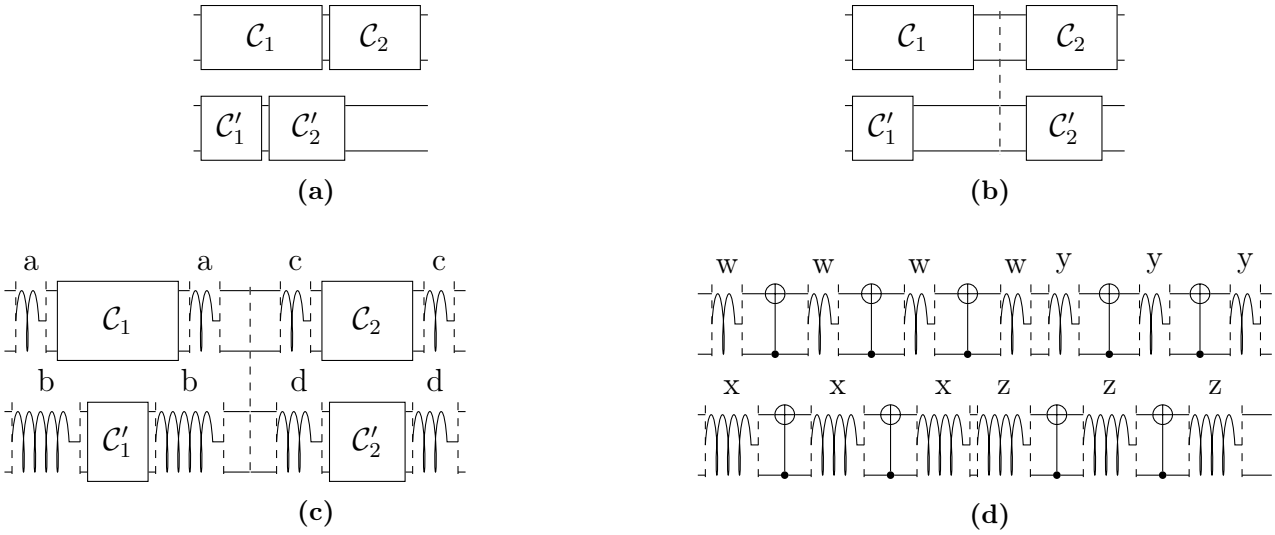


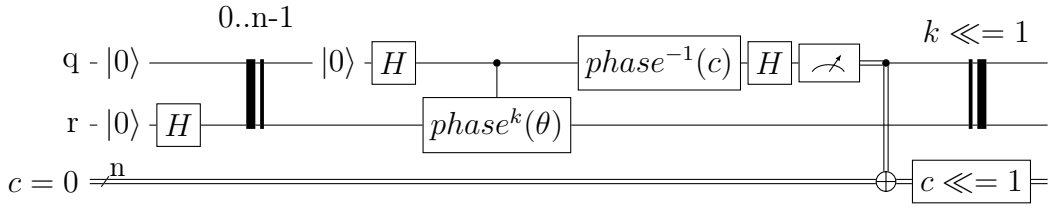
Figure 7: Simultaneous 2-qubit randomized benchmarking using stretchy delays to align gates. (a) Independent sequences of Clifford operations are applied to the top 2 and bottom 2 qubits, but varying decompositions and gate speeds cause misalignment. (b) A barrier can force coarse synchronization points. (c) Stretchy delays inserted before and after every Clifford can ensure better simultaneity. These delays will be carried as part of the circuit, no matter how the Cliffords are decomposed. (d) The decomposition of Cliffords themselves can have stretchy delays, ensuring even finer alignment of gates. The program can be written in a way that these delays override the previous ones in-between Cliffords.

unitary up to some number of bits of precision. In contrast to the textbook phase estimation, it uses fewer qubits (only one control qubit), but at the cost of multiple measurements. Each measurement adds one bit to the estimated phase. Importantly, all iterations must happen in real time during the coherence interval of the qubits, and the result of each measurement feeds forward to the following iteration to influence the angles of rotation.

Initial circuit The initial circuit may have been produced by high level software tools and languages and may have already passed through stages of high level transformations.

```
/*
 * Iterative phase estimation
 */
OPENQASM 3;
include "stdgates.inc";

const n = 3;          // number of iterations
const theta = 3 * pi / 8;  // phase angle on target qubit

qubit q;    // phase estimation qubit
qubit r;    // target qubit for the controlled-unitary gate
angle[n] c = 0; // phase estimation bits

// initialize
reset q;
reset r;

// prepare uniform superposition of eigenvectors of phase
h r;
```

```
// iterative phase estimation loop
for i in [1:n] {  // implicitly cast val to int
  reset q;
  h q;
  ctrl @ pow(2**i) @ phase(theta) q, r;
  inv @ phase(c) q;
  h q;
  measure q -> c[0];
  // newest measurement outcome is associated to a pi/2 phase shift
  // in the next iteration, so shift all bits of c left
  c <<= 1;
}

// Now c contains the n-bit estimate of phi in the
// eigenvalue e^{i*phi} and qreg r is projected to an
// approximate eigenstate of the phase gate.
```



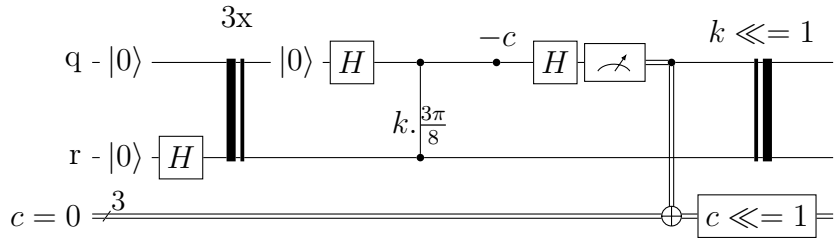
4.1 Target-independent compilation phase

The target-independent phase applies transformations that do not use information about any particular target system. The goal of these transformations is quantum circuit synthesis and optimization.

Constant propagation and folding This transformation substitutes the values of constants as they occur in expressions throughout the circuit. We repeatedly apply this transformation as constants are exposed.

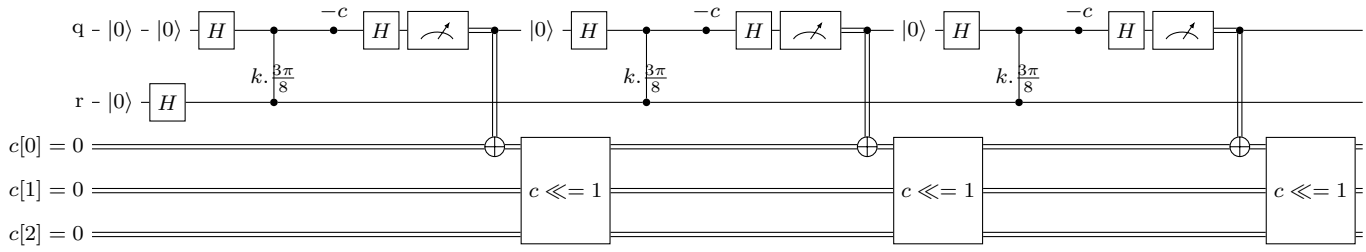
Gate modifier evaluation and synthesis The control, power, and inverse gate modifiers are replaced by gate sequences, and those are simplified when possible based on the structure of the circuit. In this case, the inverse and power modifiers can be simplified by modifying the angle arguments of the phase and controlled-phase gates.

```
// simplify gate modifiers and substitute constants
include "stdgates.inc";
qubit q;
qubit r;
angle[3] c = 0;
reset q;
reset r;
h r;
for i in [1:3] {
  reset q;
  h q;
  cphase(2**i * 3*pi/8) q, r;
  phase(-c) q;
  h q;
  measure q -> c[0];
  c <<= 1;
}
```



Loop unrolling The trip count of this loop is constant and statically known. Therefore the loop can be unrolled. This is not necessary as the machine executing an OpenQASM circuit is assumed to have control flow capabilities. However, doing so in the compiler could expose certain optimization opportunities. For example, the following pass could remove the reset operation in the first iteration, because it is redundant with the initial circuit reset.

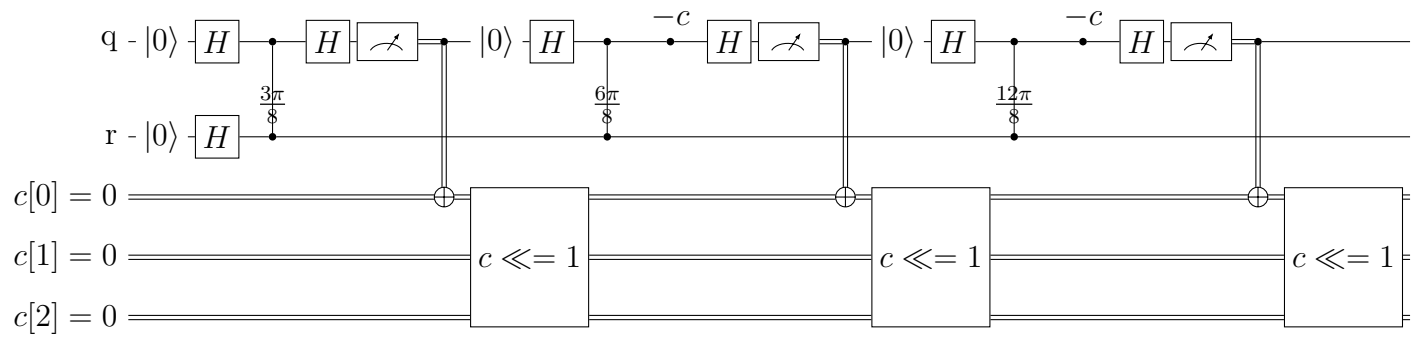
```
// loop unrolling
include "stdgates.inc";
qubit q;
qubit r;
angle[3] c = 0;
reset q;
reset r;
h r;
int i = 1;
reset q;
h q;
cphase(2**i * 3*pi/8) q, r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
i = 2;
reset q;
h q;
cphase(2**i * 3*pi/8) q, r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
i = 3;
reset q;
h q;
cphase(2**i * 3*pi/8) q, r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
```



Constant propagation and folding (repeat) We repeat this step to evaluate and substitute the value of the `power` variable.

Gate simplification Gate simplification rules can be applied to reduce the gate count. In this example, rotations with angle zero are identity gates and can be removed.

```
// constant propagation and simplification
include "stdgates.inc";
qubit q;
qubit r;
angle[3] c = 0;
reset q;
reset r;
h r;
h q;
cphase(3*pi/8) q, r;
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
cphase(6*pi/8) q, r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
cphase(12*pi/8) q, r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
```



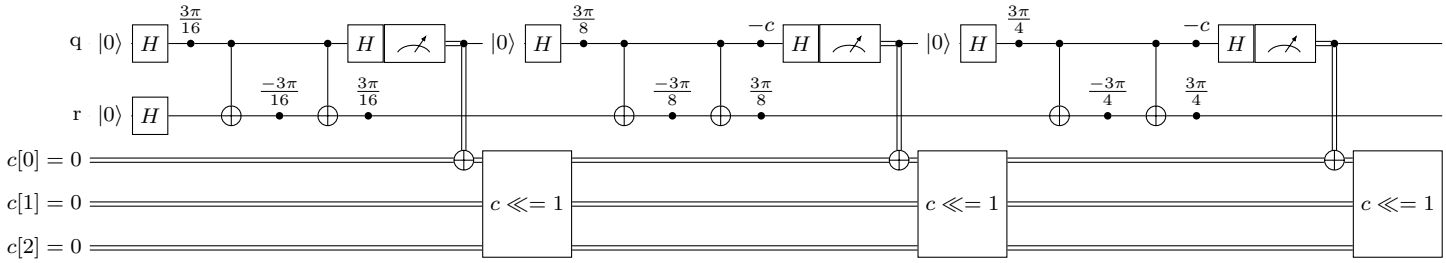
4.2 Target-dependent compilation phase

The target-dependent phase applies transformations that are generically necessary to lower a target-independent OpenQASM 3 program to an executable. Each transformation uses information that is specific to a class of target machines, rather than one specific target, and the class of potential targets is refined with each transformation.

Basis translation As we enter the next phase of compilation, we need to use information about the target system. The basis translation step rewrites quantum gates in terms of a set of gates available on the target. In this example, we assume a quantum computer with a calibrated set of basis gates including `phase`, `h`, and `cx`.

```
// basis translation
gate phase(lambda) q { U(0, 0, lambda) q; }
gate cx c, t { ctrl @ U(pi, 0, pi) c, t; }
gate h a { U(pi/2, 0, pi) a; }
qubit q;
qubit r;
angle[3] c = 0;
```

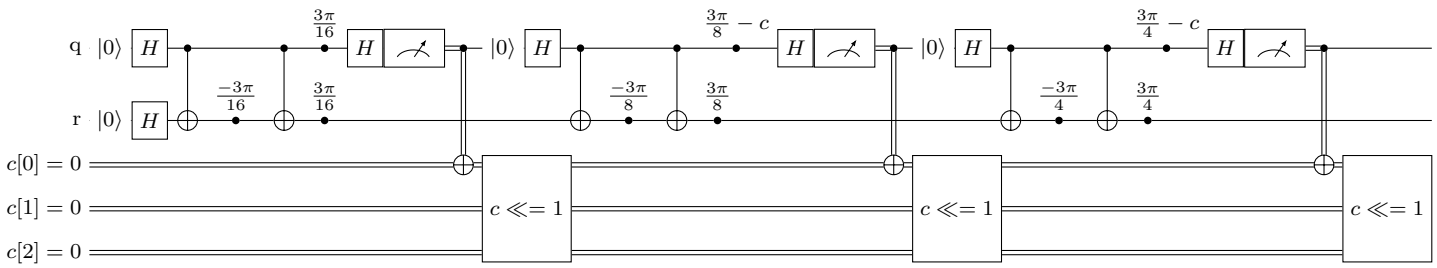
```
reset q;
reset r;
h r;
h q;
phase(3*pi/16) q;
cx q, r;
phase(-3*pi/16) r;
cx q, r;
phase(3*pi/16) r;
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
phase(3*pi/8) q;
cx q, r;
phase(-3*pi/8) r;
cx q, r;
phase(3*pi/8) r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
phase(6*pi/8) q;
cx q, r;
phase(-6*pi/8) r;
cx q, r;
phase(6*pi/8) r;
phase(-c) q;
h q;
measure q -> c[0];
c <<= 1;
```



Gate simplification (repeat) We repeat the gate simplification transformations to reduce the gate count. In this example, phase gates commute through the controls of CNOT gates and can be merged with other phase gates by adding the angle arguments.

```
// gate commutation and merging
gate phase(lambda) q { U(0, 0, lambda) q; }
gate cx c, t { ctrl @ U(pi, 0, pi) c, t; }
gate h a { U(pi/2, 0, pi) a; }
qubit q;
qubit r;
angle[3] c = 0;
reset q;
reset r;
h r;
h q;
cx q, r;
phase(-3*pi/16) r;
cx q, r;
phase(3*pi/16) r;
phase(3*pi/16) q;
```

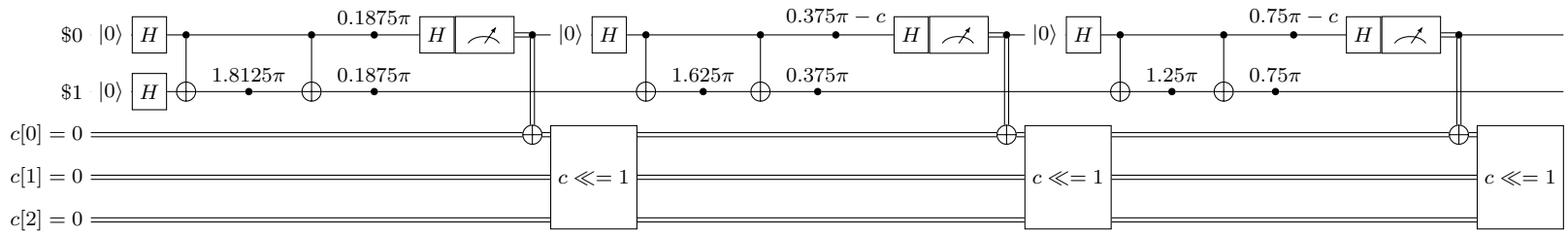
```
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
cx q, r;
phase(-3*pi/8) r;
cx q, r;
phase(3*pi/8) r;
phase(3*pi/8-c) q;
h q;
measure q -> c[0];
c <<= 1;
reset q;
h q;
cx q, r;
phase(-6*pi/8) r;
cx q, r;
phase(6*pi/8) r;
phase(6*pi/8-c) q;
h q;
measure q -> c[0];
c <<= 1;
```



Physical qubit mapping and routing In this step we need to use the physical connectivity of the target device to map virtual qubits of the program to physical qubits of the device. In this example, we assume a CNOT can be applied between physical qubits 0 and 1 in either direction, so no routing is necessary.

```
// layout on physical qubits
gate phase(lambda) q { U(0, 0, lambda) q; }
gate cx c, t { ctrl @ U(pi, 0, pi) c, t; }
gate h a { U(pi/2, 0, pi) a; }
angle[3] c = 0;
reset $0;
reset $1;
h $1;
h $0;
cx $0, $1;
phase(1.8125*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.1875*pi) $1;
phase(0.1875*pi) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
reset $0;
h $0;
cx $0, $1;
phase(1.625*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.375*pi) $1;
angle[32] temp_1 = 0.375*pi;
temp_1 -= c;  // cast and do arithmetic mod 2 pi
```

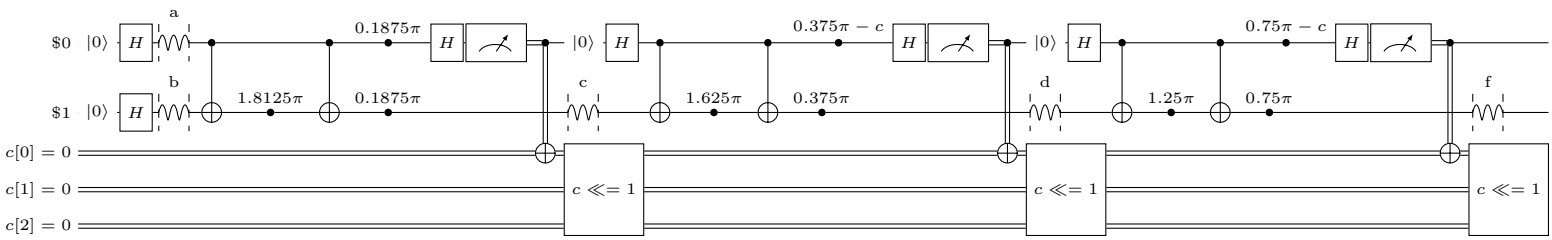
```
phase(temp_1) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
reset $0;
h $0;
cx $0, $1;
phase(1.25*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.75*pi) $1;
angle[32] temp_2 = 0.75*pi;
temp_2 -= c;  // cast and do arithmetic mod 2 pi
phase(temp_2) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
```



Scheduling We can enforce a scheduling policy without knowing or using concrete gate durations. We use stretchy delays for this purpose, which enact a "as late as possible" schedule. The actual gate durations will be available in the next stage where the program is tied to certain gate calibration data, and the stretch problem can be solved.

```
// scheduling intent without hardcoded timing
gate phase(lambda) q { U(0, 0, lambda) q; }
gate cx c, t { ctrl @ U(pi, 0, pi) c, t; }
gate h a { U(pi/2, 0, pi) a; }
angle[3] c = 0;
reset $0;
reset $1;
h $1;
h $0;
stretch a, b;
delay[a] $0;
delay[b] $1;
cx $0, $1;
phase(1.8125*pi) $1;
cx $0, $1;
phase(0.1875*pi) $1;
phase(0.1875*pi) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
reset $0;
h $0;
stretch c;
delay[c] $1;
cx $0, $1;
phase(1.625*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.375*pi) $1;
angle[32] temp_1 = 0.375*pi;
temp_1 -= c;  // cast and do arithmetic mod 2 pi
phase(temp_1) $0;
h $0;
measure $0 -> c[0];
```

```
c <<= 1;
reset $0;
h $0;
stretch d;
delay[d] $1;
cx $0, $1;
phase(1.25*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.75*pi) $1;
angle[32] temp_2 = 0.75*pi;
temp_2 -= c;  // cast and do arithmetic mod 2 pi
phase(temp_2) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
stretch f;
delay[f] $1;
```

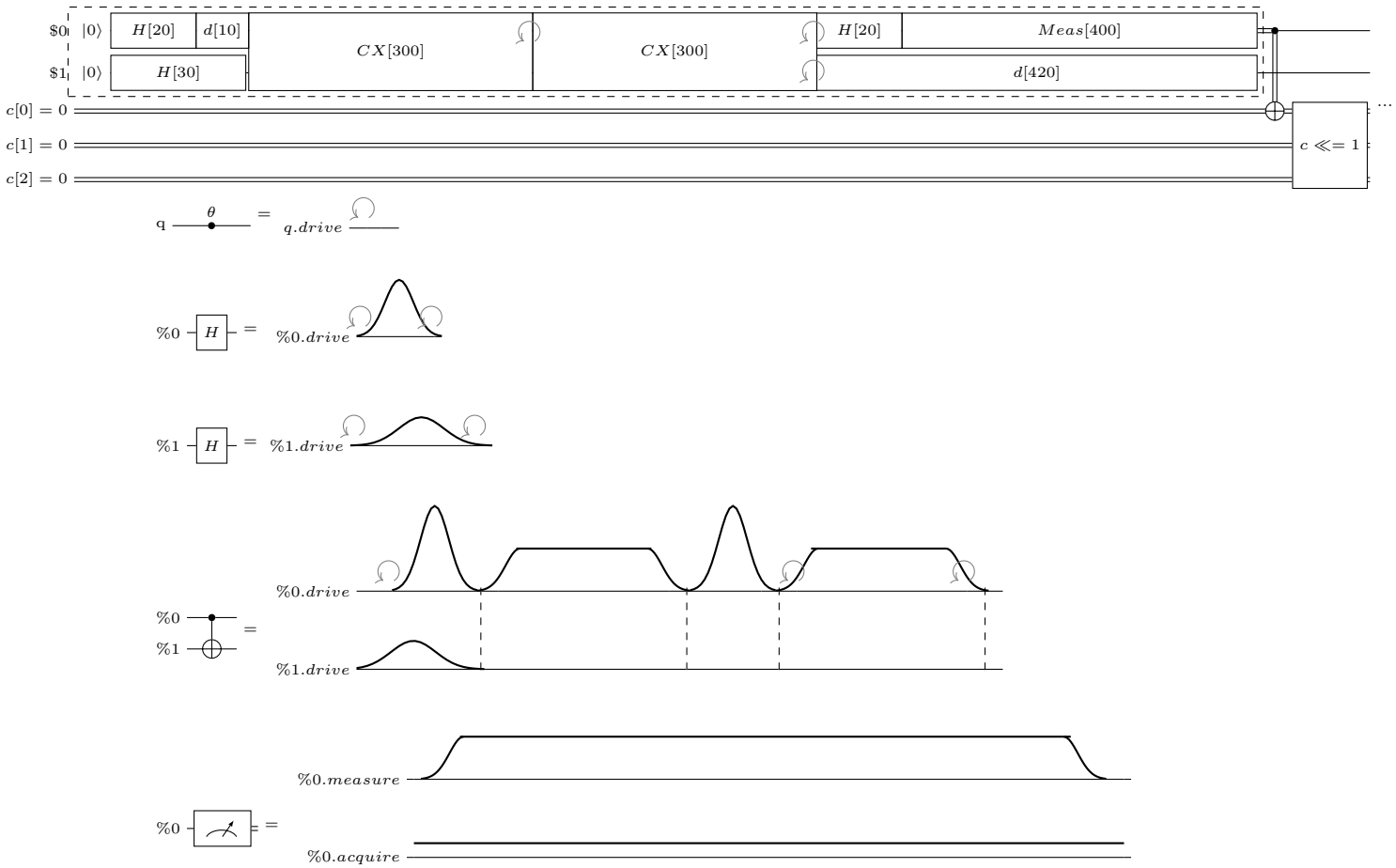


Calibration linking and stretch resolution A pulse sequence is defined for each gate, reset, and measurement. Using their durations, the stretch variables are resolved into delays with concrete timing. Angles are rounded to the precision of the defcal arguments (i.e., to defcal precision) at this step, if they have not already been rounded to the appropriate precision by an earlier transformation.

```
// read calibration and enforce full schedule
include "stdpulses.inc";
defcalgrammar "openpulse";

defcal y90p $0 {
    play drive($0), drag(...);
}
defcal y90p $1 {
    play drive($1), drag(...);
}
defcal cr90p $0, $1 {
    play flat_top_gaussian(...), drive($0), frame(drive($1));
}
defcal phase(angle[20] theta) $q {
    shift_phase drive($q), -theta;
}
defcal cr90m $0, $1 {
    phase(-pi) $1;
    cr90p $0, $1;
    phase(pi) $1;
}
defcal x90p $q {
    phase(pi) $q;
    y90p $q;
    phase(-pi) $q;
}
defcal xp $q {
    x90p $q;
    x90p $q;
}
defcal h $q {
```

```
    phase(pi) $q;
    y90p $q;
}
defcal cx $control $target {
  phase(-pi/2) $control;
  xp $control;
  x90p $target;
  barrier $control, $target;
  cr90p $control, $target;
  barrier $control, $target;
  xp $control;
  barrier $control, $target;
  cr90m $control, $target;
}
defcal measure $0 -> bit {
     complex[int[24]] iq;
     bit state;
     complex[int[12]] k0[1024] = [i0 + q0*j, i1 + q1*j, i2 + q2*j, ...];
     play measure($0), flat_top_gaussian(...);
     iq = capture acquire($0), 2048, kernel(k0);
     return threshold(iq, 1234);
}
angle[3] c = 0;
reset $0;
reset $1;
h $1;
h $0;
stretch a, b;
delay[a] $0;
delay[b] $1;
cx $0, $1;
phase(1.8125*pi) $1;
cx $0, $1;
phase(0.1875*pi) $1;
phase(0.1875*pi) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
reset $0;
h $0;
stretch c;
delay[c] $1;
cx $0, $1;
phase(1.625*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.375*pi) $1;
angle[32] temp_1 = 0.375*pi;
temp_1 -= c;  // cast and do arithmetic mod 2 pi
phase(temp_1) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
reset $0;
h $0;
stretch d;
delay[d] $1;
cx $0, $1;
phase(1.25*pi) $1;  // mod 2*pi
cx $0, $1;
phase(0.75*pi) $1;
angle[32] temp_2 = 0.75*pi;
temp_2 -= c;  // cast and do arithmetic mod 2 pi
phase(temp_2) $0;
h $0;
measure $0 -> c[0];
c <<= 1;
stretch f;
```

```
delay[f] $1;
```



The compiled circuit is now submitted to the target machine code generator to produce binaries for the target control system of the quantum computer. These are then submitted to the execution engine to orchestrate the quantum computation. Additional phases of target-dependent compilation occur that are beyond the scope of OpenQASM 3.

5 Acknowledgements

We thank the community who gave early feedback on the OpenQASM 3 spec [**?**], especially Matthew Amy, Prakash Murali, and Pranav Gokhale. Quantum circuit drawings were typeset using yquant [57].

References

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

[2] I. Chuang. qasm2circ. http://www.media.mit.edu/quanta/qasm2circ/, 2005, accessed November 2020.

[3] A. W. Cross. qasm-tools. http://www.media.mit.edu/quanta/quanta-web/projects/qasm-tools/, 2005, accessed November 2020.

[4] K. M. Svore, A. W. Cross, I. L. Chuang, A. V. Aho, and I. L. Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74—83, 2006.

[5] S. Balensiefer, L. Kreger-Stickles, and M. Oskin. QUALE: quantum architecture layout evaluator. *Proc. SPIE 5815, Quantum information and computation III*, 103, 2005.

[6] M. Dousti, A. Shafaei, and M. Pedram. Squash 2: a hierarchical scalable quantum mapper considering ancilla sharing. *Quant. Inf. Comp.*, 16(4), 2016.

[7] IBM Quantum Experience. https://quantum-computing.ibm.com/. accessed November 2020.

[8] M. Amy. Sized types for low-level quantum metaprogramming. In *Lecture Notes in Computer Science, vol 11497*. Reversible Computation. RC 2019., Springer, 2019.

[9] N. de Beaudrap. ReQASM: a recursive extension to OpenQASM. *private communication*, 2019.

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

[11] A. J. Landahl, D. S. Lobser, B. C. A. Morrison, K. M. Rudinger, A. E. Russo, J. W. Van Der Wall, and P. Maunz. Jaqal, the quantum assembly language for QSCOUT. *arXiv:2003.09382*, 2020.

[12] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. eQASM: An executable quantum instruction set architecture. *Proc. 25th International Symposium on High-Performance Computer Architecture (HPCA19)*, 2019.

[13] PyQuil. https://github.com/rigetticomputing/pyquil, accessed November 2020.

[14] D. E. Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989.

[15] A. Barenco, C. Bennett, R. Cleve, D. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52(3457), 1995.

[16] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[17] W. M. Johnston, J. R. Paul Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1), 2004.

[18] A. M. Steane. Space, time, parallelism and noise requirements for reliable quantum computing. *Fortschritte der Physik*, 46(443), 1999.

[19] Dorit Aharonov and Michael Ben-Or. Fault-tolerant quantum computation with constant error rate. *SIAM Journal on Computing*, 2008.

[20] A. G. Fowler, A. C. Whiteside, and L. C. L. Hollenberg. Towards practical classical processing for the surface code. *Phys. Rev. Lett.*, 108(180501), 2012.

[21] N. Delfosse. Hierarchical decoding to reduce hardware requirements for quantum computing. *arXiv:2001.11427*, 2020.

[22] P. Das, C. A. Pattison, S. Manne, D. Carmean, K. Svore, M. Qureshi, and N. Delfosse. A scalable decoder micro-architecture for fault-tolerant quantum computing. *arXiv:2001.06598*, 2020.

[23] A. Paetznick and K. M. Svore. Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation*, 14(15-16), 2014.

[24] S. Bravyi and A. Kitaev. Universal quantum computation with ideal clifford gates and noisy ancillas. *Physical Review A*, 71(2):022316, 2005.

[25] L. Viola, E. Knill, and S. Lloyd. Dynamical decoupling of open quantum systems. *Physical Review Letters*, 82(12):2417, 1999.

[26] Jay M Gambetta, Antonio D Córcoles, Seth T Merkel, Blake R Johnson, John A Smolin, Jerry M Chow, Colm A Ryan, Chad Rigetti, S Poletto, Thomas A Ohki, et al. Characterization of addressability by simultaneous randomized benchmarking. *Physical review letters*, 109(24):240504, 2012.

[27] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE, 1994.

[28] 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*, 5:4213, 2014.

[29] Héctor Abraham, AduOffei, Rochisha Agarwal, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Eli Arbel, Arijit02, Abraham Asfaw, Artur Avkhadiev, Carlos Azaustre, AzizNgoueya, Abhik Banerjee, Aman Bansal, Panagiotis Barkoutsos, Ashish Barnawal, George Barron, George S. Barron, Luciano Bello, Yael Ben-Haim, Daniel Bevenius, Arjun Bhobe, Lev S. Bishop, Carsten Blank, Sorin Bolos, Samuel Bosch, Brandon, Sergey Bravyi, Bryce-Fuller, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adrian Chen, Chun-Fu Chen, Edward Chen, Jielun (Chris) Chen, Richard Chen, Jerry M. Chow, Spencer Churchill, Christian Claus, Christian Clauss, Romilly Cocking, Filipe Correa, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito,

Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Tareq El Dandachi, Marcus Daniels, Matthieu Dartiailh, DavideFrr, Abdón Rodríguez Davila, Anton Dekusar, Delton Ding, Jun Doi, Eric Drechsler, Drew, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem EL-Safty, Eric Eastman, Grant Eberle, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Axel Hernández Ferrera, Romain Fouilland, FranckChevallier, Albert Frisch, Andreas Fuhrer, Bryce Fuller, Melvin George, Julien Gacon, Borja Godoy Gago, Claudio Gambella, Jay M. Gambetta, Adhisha Gammanpila, Luis Garcia, Tanya Garg, Shelly Garion, Austin Gilliam, Aditya Giridharan, Juan Gomez-Mosquera, Gonzalo, Salvador de la Puente González, Jesse Gorzinski, Ian Gould, Donny Greenberg, Dmitry Grinko, Wen Guan, John A. Gunnels, Mikael Haglund, Isabel Haide, Ikko Hamamura, Omar Costa Hamido, Frank Harkins, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Stefan Hillmich, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Junye Huang, Rolf Huisman, Haruki Imai, Takashi Imamichi, Kazuaki Ishizaki, Raban Iten, Toshinari Itoko, JamesSeaward, Ali Javadi, Ali Javadi-Abhari, Wahaj Javed, Jessica, Madhav Jivrajani, Kiran Johns, Scott Johnstun, Jonathan-Shoemaker, Vismai K, Tal Kachmann, Akshay Kale, Naoki Kanazawa, Kang-Bae, Anton Karazeev, Paul Kassebaum, Josh Kelso, Spencer King, Knabberjoe, Yuri Kobayashi, Arseny Kovyrshin, Rajiv Krishnakumar, Vivek Krishnan, Kevin Krsulich, Prasad Kumkar, Gawel Kus, Ryan LaRose, Enrique Lacal, Raphaël Lambert, John Lapeyre, Joe Latone, Scott Lawrence, Christina Lee, Gushu Li, Dennis Liu, Peng Liu, Yunho Maeng, Kahan Majmudar, Aleksei Malyshev, Joshua Manela, Jakub Marecek, Manoel Marques, Dmitri Maslov, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Dan McPherson, Srujan Meesala, Thomas Metcalfe, Martin Mevissen, Andrew Meyer, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Abby Mitchell, Nikolaj Moll, Jhon Montanez, Gabriel Monteiro, Michael Duane Mooring, Renier Morales, Niall Moran, Mario Motta, MrF, Prakash Murali, Jan Müggenburg, David Nadlinger, Ken Nakanishi, Giacomo Nannicini, Paul Nation, Edwin Navarro, Yehuda Naveh, Scott Wyman Neagle, Patrick Neuweiler, Johan Nicander, Pradeep Niroula, Hassi Norlen, NuoWenLei, Lee James O'Riordan, Oluwatobi Ogunbayo, Pauline Ollitrault, Raul Otaolea, Steven Oud, Dan Padilha, Hanhee Paik, Soham Pal, Yuchen Pang, Vincent R. Pascuzzi, Simone Perriello, Anna Phan, Francesco Piro, Marco Pistoia, Christophe Piveteau, Pierre Pocreau, Alejandro Pozas-iKerstjens, Milos Prokop, Viktor Prutyanov, Daniel Puzzuoli, Jesús Pérez, Quintiii, Rafey Iqbal Rahman, Arun Raja, Nipun Ramagiri, Anirudh Rao, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Julia Rice, Matt Riedemann, Marcello La Rocca, Diego M. Rodríguez, RohithKarur, Max Rossmannek, Mingi Ryu, Tharrmashastha SAPV, SamFerracin, Martin Sandberg, Hirmay Sandesara, Ritvik Sapra, Hayk Sargsyan, Aniruddha Sarkar, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Zachary Schoenfeld, Travis L. Scholten, Eddie Schoute, Joachim Schwarm, Ismael Faro Sertage, Kanav Setia, Nathan Shammah, Yunong Shi, Adenilton Silva, Andrea Simonetto, Nick Singstock, Yukio Siraichi, Iskandar Sitdikov, Seyon Sivarajah, Magnus Berg Sletfjerding, John A. Smolin, Mathias Soeken, Igor Olegovich Sokolov, Igor Sokolov, SooluThomas, Starfish, Dominik Steenken, Matt Stypulkoski, Shaojun Sun, Kevin J. Sung, Hitomi Takahashi, Tanvesh Takawale, Ivano Tavernelli, Charles Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Miroslav Tomasik, Enrique de la Torre, Kenso Trabing,

Matthew Treinish, TrishaPe, Davindra Tulsi, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Almudena Carrera Vazquez, Victor Villar, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Johannes Weidenfeller, Rafal Wieczorek, Jonathan A. Wildstrom, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, Steve Wood, James Wootton, Daniyar Yeralin, David Yonge-Mallo, Richard Young, Jessie Yu, Christopher Zachow, Laura Zdanski, Helena Zhang, Christa Zoufal, Zoufalc, a kapila, a matsuo, bcamorrison, brandhsn, nick bronn, brosand, chlorophyll zz, csseifms, dekel.meirom, dekelmeirom, dekool, dime10, drholmie, dtrenev, ehchen, elfrocampeador, faisaldebouni, fanizzamarco, gabrieleagl, gadial, galeinston, georgios ts, gruu, hhorii, hykavitha, jagunther, jliu45, jscott2, kanejess, klinvill, krutik2966, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sagar pahwa, rmoyard, saswati qiskit, scottkelso, sethmerkel, shaashwat, sternparky, strickroman, sumitpuri, tigerjack, toural, tsura crisaldo, vvilpas, welien, willhbang, yang.luh, yotamvakninibm, and Mantas Čepulkovskis. Qiskit: An open-source framework for quantum computing, 2019.

[30] Cirq. https://github.com/quantumlib/Cirq, accessed November 2020.

[31] ProjectQ. https://projectq.ch, accessed November 2020.

[32] Q#. https://github.com/microsoft/qsharp-language, accessed November 2020.

[33] Openqasm 3.x live specification. https://qiskit.github.io/openqasm/. accessed February 2021.

[34] A. Javadi-Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. Chong, and M. Martonosi. ScaffCC: a framework for compilation and analysis of quantum computing programs. *ACM International Conference on Computing Frontiers (CF 2014)*, 2014.

[35] C. A. Ryan, B. R. Johnson, D. Ristè, B. Donovan, and T. A. Ohki. Hardware for dynamic quantum computing. *Review of Scientific Instruments*, 88(10):104703, 2017.

[36] X. Fu, M. A. Rol, C. C. Bultink, J. Van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, et al. An experimental microarchitecture for a superconducting quantum processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 813–825, 2017.

[37] A. Butko, G. Michelogiannakis, S. Williams, C. Iancu, D. Donofrio, J. Shalf, J. Carter, and I. Siddiqi. Understanding quantum control processor capabilities and limitations through circuit characterization. *arXiv:1909.11719*, 2019.

[38] Peter J Karalekas, Nikolas A Tezak, Eric C Peterson, Colm A Ryan, Marcus P da Silva, and Robert S Smith. A quantum-classical cloud platform optimized for variational hybrid algorithms. *Quantum Science and Technology*, 5(2):024003, Apr 2020.

[39] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), Jul 2014.

[40] J. M. Gambetta, A. D. Córcoles, S. T. Merkel, B. R. Johnson, J. A. Smolin, J. M. Chow, C. A. Ryan, C. Rigetti, S. Poletto, T. A. Ohki, et al. Characterization of addressability by simultaneous randomized benchmarking. *Physical review letters*, 109(24):240504, 2012.

[41] E. L. Hahn. Spin echoes. *Physical review*, 80(4):580, 1950.

[42] S. Meiboom and D. Gill. Modified spin-echo method for measuring nuclear relaxation times. *Review of scientific instruments*, 29(8):688–691, 1958.

[43] G. Uhrig. Keeping a quantum bit alive by optimized π-pulse sequences. *Physical Review Letters*, 98(10):100504, 2007.

[44] K. Khodjasteh and L. Viola. Dynamically error-corrected gates for universal quantum computation. *Physical review letters*, 102(8):080501, 2009.

[45] Amrit De and Leonid P Pryadko. Universal set of scalable dynamically corrected gates for quantum error correction with always-on qubit couplings. *Physical review letters*, 110(7):070503, 2013.

[46] Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1016, 2020.

[47] D. E. Knuth and D. Bibby. *The TeXbook*, volume 3. Addison-Wesley Reading, 1984.

[48] Neereja Sundaresan, Isaac Lauer, Emily Pritchett, Easwar Magesan, Petar Jurcevic, and Jay M Gambetta. Reducing unitary and spectator errors in cross resonance with optimized rotary echoes. *PRX Quantum*, 1(2):020318, 2020.

[49] M. Cococcionia, M. Pappalardo, and Y. D. Sergeyev. Lexicographic multi-objective linear programming using grossone methodology: Theory and algorithm. *Applied Mathematics and Computation*, 318:298–311, 2018.

[50] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[51] S. J. Glaser, U. Boscain, T. Calarco, C. P. Koch, W. Köckenberger, R. Kosloff, I. Kuprov, B. Luy, S. Schirmer, T. Schulte-Herbrüggen, et al. Training Schrödinger's cat: quantum optimal control. *The European Physical Journal D*, 69(12):1–24, 2015.

[52] J. Koch, M. Y. Terri, J. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Charge-insensitive qubit design derived from the Cooper pair box. *Physical Review A*, 76(4):042319, 2007.

[53] Simon Marlow et al. Haskell 2010 language report. *Available online http://www.haskell.org/ (May 2011)*, 2010.

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

[55] T. Alexander, N. Kanazawa, D. J. Egger, L. Capelluto, C. J. Wood, A. Javadi-Abhari, and D. McKay. Qiskit Pulse: Programming quantum computers through the cloud with pulses. *Quantum Science and Technology*, 5(4), 2020.

[56] M. Dobšíček, G. Johansson, V. Shumeiko, and G. Wendin. Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark. *Physical Review A*, 76(3):030306, 2007.

[57] B. Desef. yquant: Typesetting quantum circuits in a human-readable language. *arXiv:2007.12931*, 2020.