2021 IEEE International Conference on Quantum Computing and Engineering (QCE) | 978-1-6654-1691-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/QCE52317.2021.00042

A Quantum Computing Programming Language for Transparent Experiment Descriptions

Virginia Frey[†1,2], Richard Rademacher[†1,2], Elijah Durso-Sabina[1,2], Noah Greenberg[1,2], Nikolay Videnov[1,2], Matthew L. Day[1,2], Rajibul Islam[1,2] and Crystal Senko[1,2]
[1]Department of Physics and Astronomy, University of Waterloo, Waterloo, N2L 3G1, Canada
[2]Institute for Quantum Computing, University of Waterloo, Waterloo N2L 3G1, Canada

***Abstract*—We present a new quantum programming language called 'Qualia' that enables transparent programming of quantum hardware. Qualia allows seamless integration of abstraction layers such as the digital circuit layer and the analog control pulse waveform layer. Additionally, the language supports user-issued low-level hardware instructions like FPGA actions. Mid-circuit measurements and branching decision logic support real-time, adaptive programs. This flexibility allows users to write code for everything from quantum error correction to analog quantum simulation. The combination of a user-facing calibration database and a powerful symbolic algebra framework provides users with an unprecedented level of expressiveness and transparency. We display the salient characteristics of the language structure and describe how the accompanying compiler can translate programs written in any abstraction layer into precisely timed hardware commands. Qualia provides a fully transparent programming interface that allows simultaneous circuit level and hardware level control. There is no "behind-the-scenes" circuit compilation or hardware calibration. Every command is executed exactly as the the user dictates.**

1. Introduction

Over the past decade we have seen an explosion of quantum programming languages (QPLs) and quantum programming paradigms [1,2] that enable users to interact with quantum hardware and implement quantum algorithms. To date, most of the QPL development effort has been focused on circuit-level description languages [3–7] which allow users to express quantum algorithms in the standard quantum circuit picture consisting of a series of time-independent single- and multi-qubit gates [8].

However, the level of abstraction that this circuit layer offers is limited in its usefulness as long as digital quantum computing is dominated by errors. Gate fidelities must be improved by orders of magnitude to reach the thresholds for error correcting codes small enough to be run on current quantum computers [9]. To get us closer to that goal, using quantum computers as the analog machines that they really are may help realize the necessary operational fidelity for the highly anticipated breakthrough in quantum computing [10,11]. Analog control pulse engineering has seen broad adoption across various hardware platforms with encouraging results [12–16], suggesting that it will remain a productive area of research for some time. Apart from general-purpose quantum computing, there are several promising near-term applications such as analog quantum simulation [17,18], digital-analog quantum computation [19] and transport-based quantum control in the trapped-ion QCCD (Quantum Charged Coupled Device) architecture [20], that also require an analog level of control over the quantum hardware that goes far beyond the capabilities of the circuit layer.

For remote-access quantum hardware, languages that are limited to digital circuit descriptions do not allow for user insight into compilation processes or transparency into how abstract gates are implemented on the hardware. Furthermore, analog calibration processes required to tune quantum systems to their peak performance are generally not modifiable or exposed to the user since they require direct access to the experimental hardware and a language that is expressive enough to describe this level of control.

A "full-stack" quantum programming language, that integrates all the required low-level analog hardware controls with digital circuit abstraction layers would be ideal for these applications. Though promising candidates for "pulse-level" control have been proposed [4,22], they do not give users control over hardware beyond the level of control and readout waveforms. At the other end of the spectrum, hardware programming frameworks such as Artiq [23] that offer full hardware control don't integrate natively with higher-level descriptions of quantum operations[1]. Table I visualizes this gap in programmability across a variety of quantum programming languages.

To address these challenges, we have developed a quantum programming language that provides the complete connection between the high-level circuit layer and the low-level timing commands implemented on FPGA (field-programmable gate array) hardware and addresses the calibration challenge in a transparent way. A powerful run-time decision logic allows conditional execution directly in the user's code. A reusable

[†]These two authors contributed equally to this work.
Contact virginia.frey@uwaterloo.ca

[1]Unless when paired with additional (commercial) software tools such as DeltaFlow-On-Artiq

978-1-6654-1691-7/21/$31.00 ©2021 IEEE
DOI 10.1109/QCE52317.2021.00042

	Q#	Qiskit	Cirq	PyQuil	QASM*	Artiq	Qualia
High-level circuit layer							
• Quantum Algorithms • Error correction	✓	✓	✓	✓	✓		✓
Native quantum operations							
• Hardware-tailored parameterization • Pre-defined macros for analog control pulses		(✓)	(✓)	(✓)	✓		✓
Real-time hardware control layer							
• Fully custom control pulses • FPGA instructions with real-time feedback • Auxiliary equipment control						✓	✓

Table I: Existing programming frameworks for different abstraction layers for quantum computing experiments. Supported features of each layer are captured in the first column and the language support is indicated with checkmarks. A grey checkmark in parentheses indicates partial support of the aforementioned features. *The term QASM here refers to lower-level quantum assembly languages such as OpenQASM [4], eQASM [21] that offer gate layer descriptions of quantum programs with limited timing instruction capabilities.

library of standard gates allows algorithm designers to focus on high-level circuits. We designed this language to run on the QuantumION platform, which is a remote-access trapped ion quantum processor built at the Institute for Quantum Computing in Waterloo, Canada[2]. Our programming language fully describes *all* of the operations required to realize a trapped-ion quantum computer. While the accompanying hardware setup for our experiment is tailored to the trapped-ion architecture, the concepts and ideas that went into our language design are generic enough to allow portability to other experimental setups. The language even supports other implementations of quantum computing so long as they are controlled with corresponding FPGA hardware.

This manuscript is structured as follows: we begin by outlining our design philosophy and articulate the core requirements of quantum experiments. Next, we discuss the three key design elements that we have identified to satisfy these requirements. We provide a comprehensive overview of the concrete language elements and how a full quantum program can be constructed and compiled utilizing an error correction experiment and an analog simulation experiment as lead examples. We conclude our work with a summary of the key language elements and make recommendations for future adaptations.

2. The Qualia Programming Language

Qualia (pronounced /ˈkwälēə/) was designed in the context of the QuantumION platform, however the design principles and suggestions we put forth apply to any platform. Qualia is a meta-language defined in XML. Users can also write programs in language bindings such as Python, Matlab, and Julia.

[2]A separate publication about the hardware platform is forthcoming, and we will thus keep its description here brief.

A. Design philosophy and experimental requirements

To design programming interface that is fully transparent to its users, we adhered to the following design principles:

- **Full-stack control.** Users can program at all layers of the stack, from hardware-agnostic, high-level quantum circuits, all the way to precisely timed hardware commands, even within the same program.
- **Transparent Calibration** All calibration data are global to the machine and stored in a historical database. Calibration programs are written in the same language and users can inspect every routine.
- **Open source design.** We go beyond simply publishing the source code, and embrace the spirit of open-source design by exposing the design of gates, timing functions, and the calibration operations within the language itself.

With these design principles in mind, the concrete features of the language are determined by the unique requirements of quantum programs on any hardware platform which we have identified as follows:

- **Support for high-level quantum circuits.** In the quantum circuit picture, the key information is the *order* in which quantum gates are applied, not the time at which they occur. Thus, a QPL must support circuit specifications with no explicit timing information beyond "which gate comes after which". We refer to this programming paradigm as our Gate Layer.
- **Support for precision-timed events.** Quantum hardware is highly dependent on the details of operation timing. Thus, our QPL must have a consistent mechanism to specify the time at which different experimental parameters (such as laser amplitudes, trap voltages, etc) are

changed. Moreover, these specification mechanisms must integrate seamlessly with Gate Layer programming. We refer to this time-dependent programming paradigm as our Timing Layer.

- **Custom waveform control.** Algorithm performance can depend crucially on the shape of the control waveform[24]. Our language can support custom waveforms in multiple ways, including user-specified explicit changes to electrode voltages, parameterized standard waveforms, and realtime-interpolated RF shapes.
- **Real-time decision logic.** Many quantum programs benefit from the ability to measure a subset of qubits and subsequently perform *different* operations based on the outcome of the qubit measurements. Our language provides a generic mechanism to branch between different, reusable segments of code.

With this in mind, we now turn our attention to three key enabling features. These are designed to manage the specification of time sequences for the large number of channels that are used in a complete quantum program.

B. Key technical features

Our language must implement transparent, full-stack control without incorporating hardware specific commands into the integral structure of the language. Users must be able to access all valid control parameters and calibration values necessary to run experiments. To this end, our language framework incorporates the following key technical features:

1) A *Calibration Database*, to store and manage all experimental parameters relevant to user programs
2) A *Symbolic Algebra* framework that allows for flexible integration of calibration parameter into user programs
3) A *Standard Library* that contains all relevant experimental subroutines (including gates) to facilitate the programming process.

While the information stored in the Calibration Database and the Standard Library is hardware-specific, the concepts are hardware-agnostic. In the following we use example parameters applicable to our trapped-ion system to showcase how these features work together, but similar examples could be conceived with parameters suitable to other quantum hardware.

1) Calibration Database

The Calibration Database contains a large collection of machine parameters that users may require to run their experiments. These include constant parameters, such as the ion energy level splittings and the direct digital synthesis (DDS) sample clock frequency, as well as parameters which are actively calibrated, such as laser beam intensities and alignments, pulse durations and amplitudes, and higher-level parameters such as SPAM (state preparation and measurement) errors and gate fidelities.

Regularly calibrated parameters are stored with associated dates and times, and users have access to the entire calibration history of those values. To use a calibration parameter within a program, we provide language constructs such as the `NamedConstant` directive. In the Python binding to our language, users can access parameters as follows:

```
import qualia as ql

ql.NamedConstant("RamanRedSidebandFrequency",
↪  date="most-recent")
```

When used within a program, the language compiler (see Section 3.3) will insert the corresponding value at the time of compilation. Until then, the expression remains symbolic. This is particularly useful in the context of shared-access quantum computers, where several users might submit their programs to a queue: when calibration processes in the background update certain parameters while the user's program is still in the queue, the compiler will automatically insert the most recent calibration value that is available at the time the program is actually run[3].

2) Symbolic Algebra

Parameters can be derived from the Calibration Database through our Symbolic Algebra framework that enables for abstract algebraic expressions with calibration parameters.

In the binding languages, the Symbolic Algebra is enabled through overloading the standard algebraic operators. For example, given the example parameter for a microwave Rabi rate from above, we can calculate the corresponding time it takes to perform a π-pulse via:

```
pi_time = 3.14159 /
↪  ql.NamedConstant("DefaultMicrowaveRabiRate")
```

In the XML language, this would be written as:

```
<qi:DivisionOperator>
  <qi:NumericLiteral>3.14159</qi:NumericLiteral>
  <qi:NamedConstant name="DefaultMicrowaveRabiRate">
</qi:DivisionOperator>
```

Our language supports all standard algebraic operations as well as basic boolean logic.

3) Standard Library

We combine the Calibration Database and the Symbolic Algebra in the Standard Library: a set of pre-defined *calculations*, time-dependent *functions* and time-independent *gates*.

[3]Users may also inspect the values before compiling their program through a dedicated call to the database.

For example, the π-time from above is available through the `NamedCalculation` directive:

```
ql.NamedCalculation("DefaultMicrowavePiTime")
```

Standard experimental routines that are naturally described in the Timing Layer, such as the laser cooling of ions and shuttling routines, are available in the Standard Library through the `FunctionCall` directive:

```
ql.FunctionCall("DopplerCooling",
↪  duration=ql.NumericLiteral(3, "ms"))
```

Circuit-layer routines such as quantum gates are available through the `GateCall` directive:

```
ql.GateCall("CNOT", qubits=[0, 1])
```

All definitions within the Standard Library exist on the user's computer and can be used as-is or as a template for custom user functions and gates.

Standard Library procedures used within a program are automatically included in the program Header (see Section 3.2). There is no further modification or addition to these routines on the compiler side. The user program thus contains the *entire* code required to carry out the experiment.

C. Demonstration of full-stack control

The Calibration Database, the Symbolic Algebra and the Standard Library are the core features of our language. Taken together, they enable users to program our machine at an abstraction level of their choosing and fully customize any of the intermediate operations.

This interplay is illustrated in Figure 1 using the "bottom-up" construction of a CNOT gate [8] as an example. Fundamentally, a CNOT gate (or any "gate", for that matter) is an abstract operation that does not correspond directly to a single experimental action, but instead to a series of precisely timed operations. For laser-based gates in our ion trap qubits, these operations are implemented through laser-atom interactions [25,26]. Our language allows users to specify all relevant parameters for this laser-ion interaction, including pulse amplitude, frequency profiles, phase shifts, beam pointing and polarization control. These techniques have all been shown to greatly increase gate fidelities and reduce the required interaction times [13,24,27,28]. Users wishing to program at this level can create Symbolic Algebra expressions using the relevant Calibration Database values such as the calibrated Rabi rate, the phase and frequency of the individual laser beams, and several more. This powerful combination allows users to create *e.g.* customized Mølmer-Sørensen interactions to implement entangling gates [25].

Users who prefer to program at the level of machine-specific gates instead of time-specific hardware instructions have the option to work in the native Gate Layer. This layer is enabled through a set of pre-defined pulse sequences in the Standard Library that implement standard gates such as individual x, y and z-rotations and the XX entangling gate. Combining these gates, users can construct their own version of a CNOT gate and customize gate parameters like rotation angles, without specifying timing details.

Lastly, the highest level of the programming stack is designed for users focusing purely on high-level quantum algorithms. Here, users can work directly with computational gates as provided by our Standard Library. The implementation of our CNOT gate, for example, can be traced back exactly in the manner described above. This example illustrates how these three key components, the Calibration Database, the Symbolic Algebra and the Standard Library come together to enable fully customizable quantum operations in our system.

3. Technical Details

We will now take a look at the formal definition and actual syntax of the language, discuss the concept of language "bindings", and explain the language compilation process.

A. Formal language definition

Qualia is defined in XML. We chose XML due to its mature schema definition specification, its seamless integration with modern web security protocols, and the compatibility of its functional structure with our Symbolic Algebra needs. Additionally, defining the language elements in XML enables us to create an interface that is easily extensible to various high-level programming languages that have the ability to generate text-based output. These high-level language extensions are called language bindings.

Language bindings allow users to write code in a higher-level language of their choice. This facilitates the process of writing programs with high-level programming paradigms such as loops and functions. We have implemented a Python binding and will release bindings for Julia and Matlab in the future. All bindings use the same naming conventions and object relationships as in the XML language. The Python binding, for example, provides a Python module called `qualia` that contains a 1:1 mapping of XML language elements to Python classes. All these classes are derived from a common base class that implements the translation of Python objects to XML tags. For example, the `NumericLiteral` element, which has the following definition in the language schema:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:qi="https://iqc.uwaterloo.ca/quantumion">

<xs:element name="NumericLiteral">
    <xs:annotation>
```

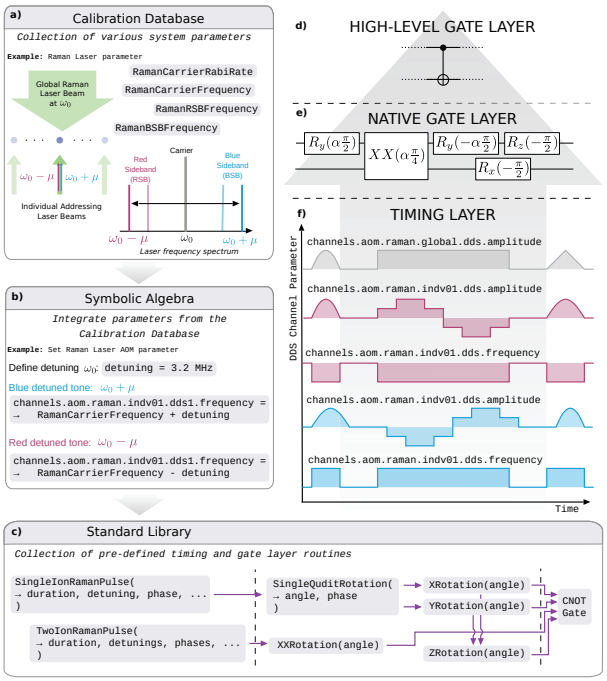

Figure 1: Features of the Qualia language illustrated through the "top-down" reconstruction of a CNOT gate. **a** The Calibration Database stores all parameters relevant for the experiment. Insets show schematics of the Raman laser beam configuration with laser frequency configurations required for single- and two-qubit interactions. **b** The Symbolic Algebra provides the framework through which calibration parameters can be incorporated into programs. Parameters are referred to by name and can be used in standard algebraic expressions. **c** The Standard Library uses the Symbolic Algebra framework to define a set of re-usable experimental routines such as Timing Layer functions and high-level gates. The frequency configurations depicted in **a** and **b** are absorbed into functions for single- and two-ion laser pulses which form the basis for single- and two-qubit gates in the native Gate Layer. The flowchart depicts all functions and native gates required to realize a CNOT gate between two qubits. **d-f** show a schematic breakdown of the CNOT gate into native gates and individual Timing Layer instructions- illustrated through amplitude and frequency profile of selected DDS channels in the setup.

```
      <xs:documentation>A fixed numeric value with units.
      ↪  Can be any real number.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:double">
          <xs:attribute name="units" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

can be used in Python through a class of the same name. Attributes and child tags are passed as arguments[4]:

[4]The `encode_xml` function is a special function that all objects in the binding languages share and which enables the translation between the binding object and the corresponding XML element.

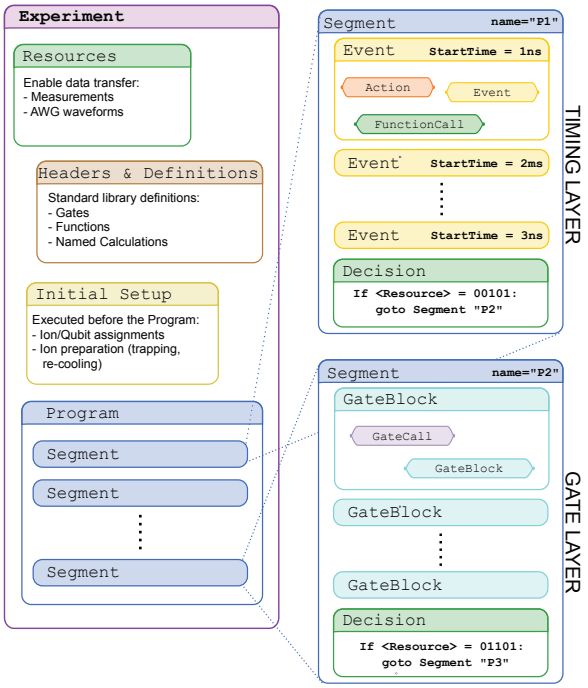

Figure 2: Building blocks of a Qualia program. This schematic illustrates the key elements of the language, represented here as rectangular boxes with background-colored titles, and their relationship to one another. For example, the `Experiment` element contains `Resources`, `Headers & Definitions`, the `Initial Setup` and the `Program` element. The latter in turn is made up of a series of `Segment` elements, which can contain either `Event` or `GateBlock` elements as well as a `Decision` element at the end. For more information on the individual elements, see main text. Two full examples that utilize these elements are shown in Section 4.

```
>>> import qualia as ql
>>> value = ql.NumericLiteral(100, units="MHz")
>>> print(value.encode_xml())
.. <qi:NumericLiteral units="MHz">
     100
   </qi:NumericLiteral>
```

Following the strict definitions set out in the language schema provides a natural way to enable the validation of user programs with linting tools such as `xmllint` and to flag syntax errors before the program reaches the compiler. This provides an additional layer of security as programs that do not abide by the language definition cannot be passed through to the compiler and control computers.

B. Elements of a Qualia program

The fundamental syntactical elements of our language and their relationships to each other are schematically shown in Figure 2. The outermost component of every Qualia program is called `Experiment`, which contains four different containers:

- `Resources` specify containers for data storage such as measurement results or pulse waveform parameters.
- `Headers` and `Definitions` declare and define the Standard Library procedures are used in this experiment, including gates, functions and named calculations.
- The `InitialSetup` contains a set of instructions for the experimental setup appear at the beginning of the experiment, such as desired number of ions/qubits and static oscillator frequencies.
- The `Program` contains the actual experimental instructions which are stored in individual `Segment` objects.

All experiment actions that are carried out during the user's program are defined within `Segment` blocks. There are two interchangeable ways of specifying the content of these elements, depending on whether the user would like to write programs in the Timing Layer with explicit timing specifications, or in the Gate Layer in which timing is implicit.

In the Timing Layer, a `Segment` contains a series of `Event` objects that specify which experimental actions are to take place at a given time. As such, every `Event` must define a `StartTime` that can be interpreted to be *absolute*, with respect to the start of the experiment, or *relative*, with respect to the `StartTime` of the previous `Event`. The other allowed elements within an `Event` are `Action`, `FunctionCall` and finally other `Event` tags. An `Action` specifies a direct and instantaneous experimental action, such as changing a DDS parameter or starting or stopping a photon counter. `FunctionCall` tags allow users to call pre-defined routines from the Standard Library, which are themselves made up of a series of `Event` tags.

In the Gate Layer on the other hand, a `Segment` contains a series of `GateBlock` objects that form a time-independent container for pre-defined gates that are scheduled to start at the same time. Users can access pre-defined gates in the Standard Library though `GateCall` objects. Similar to the `Event` tag, `GateBlock` tags can also be nested. The main difference between a `GateBlock` and an `Event` is that the time-dependence in the former is implicit. The introduction of `GateBlocks` is explicitly for circuit-level programming, although in principle every program can be fully expressed in a series of `Events`. These two containers may also be used interchangeably.

At the end of every `Segment`, users have the option of implementing a `Decision` block that can alter the execution flow of the program by jumping between different `Segments` depending on the outcome of the conditions that are declared in the `Decision` object. Conditions are typically based on measurement outcomes, that are stored in `Resource` elements that represent raw measurements like photon counts. Taking all these elements together, a complete (but largely empty) program in the Python binding may be constructed as follows [5]:

```python
import qualia as ql

# Create a default InitialSetup element
setup = ql.InitialSetup(use_predefined="default")

# Generate a generic resource container
# (A measurement Action would fill this container)
resources = ql.Resource(name="my_measurement")

# Generate Segment with empty Event, GateBlock & Decision
segment_1 = ql.Segment(
  ql.Event(
    # Note: Events must always specify a start time
    start_time=ql.NumericLiteral(0, "ns")
  ),
  ql.GateBlock(),
  ql.Decision(
    resource="my_measurement",
    conditions=[
      # If the outcome is "0", advance to "segment-2"
      ql.Condition("0", destination_segment="segment-2"),
      # If the outcome is "1", advance to "segment-3"
      ql.Condition("1", destination_segment="segment-3")
    ]
  )
)

# Generate two additional, empty Segments
segment_2 = ql.Segment(name="segment-2")
segment_3 = ql.Segment(name="segment-3")

# Combine all Segments within a Program
program = ql.Program(program_segments=[segment_1,
↪  segment_2, segment_3])

# Assemble the full Experiment
experiment = ql.Experiment(
  initial_setup=setup,
  resources=resources,
  program=program)
```

This abstract example highlights the relationships of the language elements to one another without making any assumptions about the actual experimental actions and the underlying physical hardware. To illustrate how our language can be used to implement realistic experiments, we provide two full-code examples in Section 4.

C. Language Compiler

Qualia programs may contain abstract programming concepts such as looping constructs, functions, gates and symbolic calculations. Additionally, the decision logic and the ability to specify `Events` within `Events` allows for programs with several time-lines that are not directly translatable to hardware instructions. To bridge that gap, we have developed a sophisticated language compiler.

Compiling a Qualia program follows three phases: frontend, middle, and backend processing, which is similar to the design philosophy of the GNU Compiler Collection (GCC) [29]. At the first stage, the compiler parses the user's XML program

[5]Note that we have omitted the `Headers & Definitions` element here, as those are inserted automatically when a program contains a call to the Standard Library in either the `Events` or the `GateBlocks`.

into an object-oriented C++ data structure. This includes a validity check of the program and, if errors are detected, the user may be given warnings or errors depending on severity of problems.

The compiler core, or middle-end, performs a series of expansions that re-write the XML in successively less expressive forms. This re-writing compiler process ensures that at all steps, the program is still a valid, equivalent XML program to what the user described. The expressiveness is reduced at each step (meaning progressively simpler commands are used), leading to a simpler, albeit longer, program.

The reduction steps are schematically shown in Figure 3 and can be summarized as follows:

- The structural circuit model is decomposed by expanding `GateCalls` and `GateBlocks` into their definitions.
- The abstract Gate Layer is decomposed by expanding each `GateCalls` into correspondign function definitions. The Standard Library, or user definitions, provide this mapping. This removes both all gate descriptions, yielding a pure Timing Layer program.
- The `FunctionCalls` are expanded into their definitions from the Standard Library, or user functions, resulting only in Timing Layer `Events` and `Actions`. Recursive calls are similarly unwrapped.
- The resulting Timing Layer `Event` and `Action` tags are flattened by solving all relative start times and into a single timeline.
- All symbolic expressions are solved with the latest database values (or specified historical value). The result is simple numeric literal values. Start times are now specified in units of $0.5\,\mathrm{ns}$ 'ticks'.
- The composite actions, like `DDSAction`, are expanded into *channelized* actions. The result is simple, fully qualified names for each execution engine involved.

The resulting output program is viewable by the user in XML form as well. At the last step of compilation, the so-called backend processing, converts the terminal XML form into a series of binary instructions, called 'opcodes' that are used by our FPGA execution engine.

D. FPGA Execution Engine

The language is designed to execute on FPGA hardware via the use of *Execution Engines* and *Action Cores*. These modules reside inside the FPGA itself, and provide the programmability needed to implement language elements. Action cores, as the name suggests, provide core logic for a particular action; examples include the phase accumulator and sine-lookup of a DDS core, or pulse counting of a photon measurement sensor. Each type of Action element of a program controls an independent Action Core.

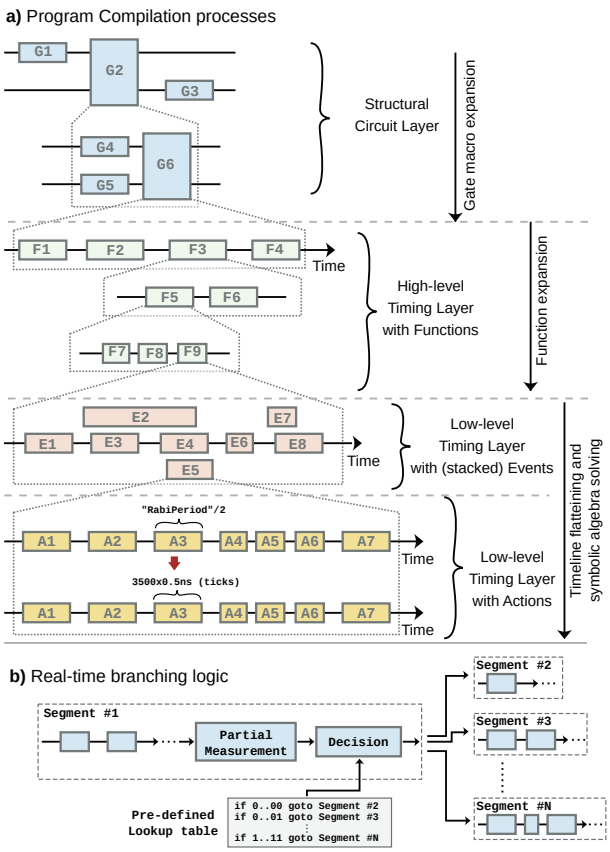

Figure 3: Language compiler and real-time decision logic. **a** illustrates the key steps of the compilation process that successively remove abstraction layers and simplify the program until it can be expressed as a flat time-line with direct experimental actions. **b** shows the schematic flow graph of the real-time branching logic. A pre-defined lookup table dictates the program execution flow.

Execution engines are lightweight, synthetic microprocessors that provide all timing and interface to the user program. Each user-controlled parameter, such as DDS phase, or amplitude, or photon-counter start/stop, is attached to a single, dedicated execution engine. The small footprint of these engines allows many hundreds of units to be instantiated on a single FPGA chip. With only slight changes in data word size, the execution engine module is the same regardless of the type of action core parameter being controlled.

The execution engine interfaces with an action core via processor bus connections. The engines receive instructions in the form of operation codes (opcodes) from the terminal XML channelized output of the compiler. A very simple opcode format allows small logic footprint, as shown in Table II. The primary opcode, `SETVALUE`, engages a change at a precise time. [6]

[6]The first generation execution engine does not contain internal state variables, but does allow for repetition loops.

OPCODE	DESCRIPTION
nn NOP	No Operation
nn SETVALUE x	Set parameter value to x
nn SETLOOP x	Sets the loop counter to x
nn JNZ pc	Jump to instruction pc if loop count is nonzero
nn JZ pc	Jump to instruction pc if loop count is zero
nn DECLOOP	Decrement the loop counter
nn GOTO pc	Unconditionally jump to pc
nn BRANCHLUT m, t	Jump to instruction using look-up table t based on measurement m

Table II: FPGA opcodes. Each corresponds to a fundamental execution engine operation to implement user language features. Each opcode contains a delay prefix of nn ticks

Embedded in each opcode is a field indicating the time-delay that the instruction should be processed. Unlike traditional microprocessors, which execute instructions at the next available cycle, the execution engine embeds a delay counter to every instruction. These delay counters operate at the 2 GHz experiment rate, and define the precision timing of all system changes. To overcome the timing closure demands of modern FPGA chips, the experiment clock rate of 2 GHz is interleaved in four clock phase-lanes, each operating at 500 MHz.

E. Decision Logic and Realtime Communications

The decision logic forms an integral part of our language and allows users to specify programs that execute certain blocks of code *conditioned* on measurement outcomes. This is enabled through the `Decision` element (see Figure 2) placed at the end of each `Segment` to instruct the program which `Segment` to execute next, as illustrated in Figure 3b.

To support the decision logic in our hardware setup, each measurement result m is broadcast over a low-latency Infiniband network. All FPGA modules contain a copy of the decision block's lookup table t, generated by the language compiler. When a BRANCHLUT m, t instruction is executed, the execution engine performs a jump to the opcodes indicated by the appropriate destination segment based on that measurement m[7].

4. Illustrative Examples

Now we turn to two examples that highlight the core features and capabilities of our language both for digital and analog quantum programs. For the sake of brevity and clarity, we are omitting the machine-specific initial setup elements.

[7]The tables can be specified using *don't care* states to reduce table sizes with an ultimate maximum 4096 permutations per decision.

A. 5-Qubit error-correction code

The five qubit code is a distance three error correction code that encodes a single logical qubit using 5 physical qubits [30]. The fault-tolerant syndrome measurement circuit (FT-SMC) we use for our example uses two ancillary qubits, one to measure the syndrome bit and another to flag errors in the measurement process [31].

The code begins by performing a fault-tolerant syndrome measurement circuit (FT-SMC). If a fault is detected, it aborts, performs a non-fault-tolerant syndrome measurement circuit (NFT-SMC) and uses that information to correct the fault. The algorithm can be broken down into the following steps:

1) Prepare all qubits in computational ground state
2) Run first FT-SMC and measure flag qubit
   a) If the flag is raised, interrupt, perform NFT-SMC, correct the fault, and terminate error correction
3) Run second FT-SMC, measure flag qubit, repeat 2a)
4) Run third FT-SMC, measure flag qubit, repeat 2a)
5) Run fourth FT-SMC, measure flag qubit, repeat 2a)
6) If no flag was raised perform error correction based on syndrome measurements

We can express this algorithm through a series of `Segments` linked together through `Decision` blocks. The gates for each SMC and the individual measurements at the end can be packaged into a series of `GateBlocks`. To store the results and make them available for the decision logic, we create an empty `Resources` object at the beginning of our program. With that, the i-th FT-SMC segment can be generated as follows:

```
import qualia as ql

resources = ql.Resources(length=12)

def make_ft_smc(i: int) -> ql.Segment:
    """ Creates the i-th syndrome measurement circuit """
    ft_smc = ql.Segment(
      name=f"FT-SMC-{i}",
      ql.GateBlock(
        ql.GateCall("H", qubit=i, port="Target"),
        ql.GateCall("CX", qubit=(i, i+5), port=("Control",
        ↪  "Target")),
        # [more gates here]
        ql.GateCall("Measure", qubit=i+5,
        ↪  resource=resources[2*i])
        ql.GateCall("Measure", qubit=i+6,
        ↪  resource=resources[2*i+1])
      )
    )
    # Add the Decision block at the end
    decision_block = make_decisions(i)
    ft_smc.add(decision_block)

    return ft_smc
```

The decision logic allows whole code blocks to be run conditioned on the result of a mid-circuit measurement.

To implement the required `Decision` blocks, we generate a series of `Condition` objects that use the measured state (0 or 1) and a "destination" `Segment`, to tell the compiler which `Segment` to execute next.

```
def make_decisions(i: int) -> ql.Decision:
    """ Create the decision block for the i-th SMC segment
    ↪  """
    segment_name = f"FT-SMC-{i}"
    next_segment_name = f"FT-SMC-{i+1}"

    decision = ql.Decision(
      resource=resources[2*i+1],
      conditions=[
        # If the outcome is 1, we want to jump to the
        ↪  non-fault-tolerant
        # SMC segment (not shown)
        ql.Condition(state=1,
        ↪  destination_segment="NFT-SMC"),
        # Otherwise, we want to continue to the next
        ↪  segment (definition shown above)
        ql.Condition(state=0,
        ↪  destination_segment=next_segment_name)
        ]
    )

    return decision
```

Finally, we can create a series of these FT-SMC segments and string them together inside a `Program`.

```
program = ql.Program(
  # Create the FT-SMC segments
  segments=[make_ft_smc(i) for i in range(4)]
  # Add auxiliary correction segments and
  ↪  non-fault-tolerant segments (not shown)
  + aux_segments
)
experiment = ql.Experiment(program, resources)
```

Thus, the QuantumION language can run the whole non-fault-tolerant syndrome measurement circuit with a single decision call from anywhere in the fault-tolerant circuit without any need for code duplication. The segment structure and branching decision logic remove the need for code duplication, and shuttling operations allow mid-circuit measurement without disturbing the rest of the computation.

B. Analog quantum simulation

In this example, we generate an effective Ising interaction between two spins that can be described by the following Hamiltonian [17]

$$H_{\text{Ising}}(t) = \sum_{ij}^{N} J_{ij}\sigma_x^{(i)}\sigma_x^{(j)} + B_y(t)\sum_i \sigma_x^{(i)},$$

Here, N is the number of spin particles, $B_y(t)$ is the effective transverse magnetic field acting on each spin, and the J_{ij} terms are the spin-spin coupling terms that depend on the drive parameters. To implement this interaction, we encode three different waveforms on the laser beams that mediate the ion-ion interactions, two with static amplitudes and frequencies, and one with a time-dependent, decreasing amplitude [32].

We use the `DDSAction` element, that allows for direct control over RF sources in the system and refers to an instantaneous instruction that is integrated into `Event` blocks to specify their start time and duration.

Every `DDSAction` takes as parameter the channel name, and optionally the amplitude, frequency, absolute and relative phases, interpolation parameters and several more. For example, the static waveforms can be defined as follows:

```
import qualia as ql

# Specify waveform frequencies relative to resonance
f0 = ql.NamedConstant("RamanCarrierResonanceFrequency")
f1 = f0 + ql.NumericLiteral(2, "MHz")
f2 = f0 - ql.NumericLiteral(2, "MHz")

# Set default amplitude
a0 =
↪  ql.NamedConstant("DefaultRamanIndividualDDSAmplitude")

ddsaction_1 = ql.DDSAction(
  channel="channels.aom.raman.individual1.dds0",
  amplitude=a0, frequency=f1
)
ddsaction_2 = ql.DDSAction(
  channel="channels.aom.raman.individual1.dds1",
  amplitude=a0, frequency=f2
)
```

We make use of the Calibration Database to import the system-specific Raman laser parameters through the `NamedConstant` element combined with our Symbolic Algebra. To program the time-dependent waveform, we use the interpolation capabilities of our DDSs. For example, we can implement a linear sweep of the form $p(t) = p_1 + p_2 * t$, between two amplitudes `a1` and `a2` over a fixed duration as follows:

```
# Define sweep duration
t_sweep = ql.NumericLiteral(10, "us")
# Set start and stop amplitudes (decreasing)
a1 =
↪  ql.NamedConstant("DefaultRamanIndividualDDSAmplitude")
a2 = a1 - ql.NumericLiteral(50, "mV")
# Set time of the sweep action
ddsaction_3 = ql.DDSAction(
  channel="channels.aom.raman.individual1.dds2",
  frequency=f0,
  interp_type="polynomial",
  interp_p0=a1,
  interp_p1=(a2 - a1)/(t_sweep *
  ↪  ql.NamedConstant("DDSSampleClockFrequency"))
)
```

where we have used the Calibration Database parameter for the DDS clock frequency to calculate the interpolation parameter. These three `DDSActions` form the heart of the analog quantum simulation routine. To integrate them into a full quantum experiment, we wrap `Events` around them to specify timing information. Two `Events` are needed: one for turning these three DDSs on, and another `Event` to turn them off.

```
# Define Ising interaction Events
ising_events = [
  # Event 1: Turn all DDSs on
  ql.Event(
    starttime=ql.StartTime(0, "ns"),
    event_items=[ddsaction_1, ddsaction_2, ddsaction_3]
  ),
  # Event 2: Turn all DDSs off
  ql.Event(
    starttime=ql.StartTime(t_sweep,
    ↪ type="since-last-action"),
    event_items=[
      ql.DDSAction(
        channel="channels.aom.raman.individual1.dds0",
        amplitude=ql.NumericLiteral(0, "V")),
        ql.DDSAction(
        channel="channels.aom.raman.individual1.dds1",
        amplitude=ql.NumericLiteral(0, "V")),
        ql.DDSAction(
        channel="channels.aom.raman.individual1.dds2",
        amplitude=ql.NumericLiteral(0, "V")),
  )
]
```

Note that we have set the `StartTime` of the second `Event` to begin at time `t_sweep`, *i.e.* our desired interaction time. To turn the interactions off, we set all DDS amplitudes to zero. We can now put everything together into a full experiment by adding several preparation routines specific to our ion trap hardware. These are part of the Standard Library, and we envision that users of other hardware platforms can add their hardware-specific routines to that library. In our setup, the full experimental protocol that we need to implement is as follows:

1) Doppler Cooling
2) Optical pumping to the ground state
3) Sideband Cooling
4) Global $\pi/2$ rotation
5) Perform effective Ising interaction
6) Global $\pi/2$ rotation
7) Measurement

To include a measurement in this experimental sequence, we declare a `Resource` that the measurement can be stored in.

```
# Set number of repetitions
num_reps = 100

# Define measurement resource
r0 = ql.APDCounterResourceQueue(size=num_reps)

segment = ql.Segment(
  segment_items=[
      ql.Event( # Step 1
          start_time=ql.NumericLiteral(0, "ns"),
          ql.FunctionCall("DopplerCooling",
          ↪ duration=ql.NumericLiteral(1, "ms"))
      ),
      ql.Event( # Step 2
          start_time=ql.NumericLiteral(0, "ns"),
          ql.FunctionCall("OpticalPumping",
          ↪ duration=ql.NumericLiteral(0.1, "ms"))
      ),
      ql.Event( # Step 3
          start_time=ql.NumericLiteral(0, "ns"),
          ql.FunctionCall("SidebandCooling")
      ),
      ql.GateBlock( # Step 4
        ql.GateCall("XPi/2", ion=0)
      ),
      # Step 5: Insert Ising interaction events (see
      ↪ below)
      *ising_events,
      ql.GateBlock( # Step 6
        ql.GateCall("XPi/2", ion=0)
      ),
      ql.Event( # Step 7
          start_time=ql.NumericLiteral(0, "ns"),
          ql.FunctionCall("GlobalReadout",
          ↪ duration=ql.NumericLiteral(0.5, "ms"),
          ↪ resource=r0)
      ),
  ]
)
```

It is possible to combine both the Timing Layer's `Event` element with the Gate Layer's `GateBlock` in the same program. By default, the `start_time` is taken relative to the last `Action` in the experiment, which in this case would be the last `Action` required to carry out the $\pi/2$ gates. Finally, we construct the full `Experiment` as:

```
experiment = ql.Experiment(
  program=ql.Program(segments=segment),
  resources=r0
)
```

The definitions of the auxiliary routines from the Standard Library are automatically packaged into the `Headers` and `Definition` elements. This example thus implements the full experiment specification of an analog quantum simulation experiment with all required hardware controls.

5. Conclusion and Outlook

We have presented a full-stack quantum programming language that is suitable for both hardware-agnostic circuit-level programming, as well as low-level hardware-specific timing layer programming. These two complementary views of a quantum program are integrated seamlessly in our language which offers an unprecedented degree of transparency into implementation of real quantum computing hardware. This is timely since algorithm performance is still largely dependent on sophisticated analog control design due to the high error budget of current quantum hardware. While this language was designed within the scope of the remote-access QuantumION platform, we believe the design philosophy and principles are applicable to a much broader range of hardware implementations. We hope that our work will provide a meaningful contribution to the quantum computing community, and inspire discussions and collaborations on developing unified software frameworks for quantum programming.

6. Acknowledgements

We acknowledge support from the TQT (Transformative Quantum Technologies) research initiative at the University of Waterloo and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] R. LaRose, "Overview and comparison of gate level quantum software platforms," *Quantum*, vol. 3, p. 130, 2019.

[2] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore, "Quantum programming languages," *Nature Reviews Physics*, pp. 1–14, 2020.

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

[4] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, J. Smolin, J. M. Gambetta, and B. R. Johnson, "OpenQASM 3: A broader and deeper quantum assembly language," *arXiv preprint arXiv:2104.14722*, 2021.

[5] C. Developers, "Cirq," Mar. 2021. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

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

[7] P. J. Karalekas, M. P. Harrigan, S. Heidel, W. J. Zeng, M. S. Alam, M. I. Appleby, L. E. Capelluto, G. E. Crooks, M. J. Curtis, E. J. Davis, K. V. Gulshen, C. B. Osborn, J. S. Otterbach, E. C. Peterson, A. M. Polloreno, G. Prawiroatmodjo, N. C. Rubin, M. G. Skilbeck, N. A. Tezak, and R. S. Smith, "PyQuil: Quantum programming in Python," Jan. 2020.

[8] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," 2002.

[9] E. Knill, R. Laflamme, and W. H. Zurek, "Resilient quantum computation: error models and thresholds," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 365–384, 1998.

[10] Y. Shi, P. Gokhale, P. Murali, J. M. Baker, C. Duckering, Y. Ding, N. C. Brown, C. Chamberland, A. Javadi-Abhari, A. W. Cross, *et al.*, "Resource-efficient quantum computing by breaking abstractions," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1353–1370, 2020.

[11] S. P. Jordan, "Quantum computation beyond the circuit model," *arXiv preprint arXiv:0809.2307*, 2008.

[12] C. F. Roos, "Ion trap quantum gates with amplitude-modulated laser beams," *New Journal of Physics*, vol. 10, no. 1, p. 013002, 2008.

[13] A. R. Milne, C. L. Edmunds, C. Hempel, F. Roy, S. Mavadia, and M. J. Biercuk, "Phase-modulated entangling gates robust to static and time-varying errors," *Physical Review Applied*, vol. 13, no. 2, p. 024022, 2020.

[14] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, and S. J. Glaser, "Optimal control of coupled spin dynamics: design of nmr pulse sequences by gradient ascent algorithms," *Journal of magnetic resonance*, vol. 172, no. 2, pp. 296–305, 2005.

[15] D. Goswami, "Optical pulse shaping approaches to coherent control," *Physics Reports*, vol. 374, no. 6, pp. 385–481, 2003.

[16] Y. Shapira, R. Shaniv, T. Manovitz, N. Akerman, and R. Ozeri, "Robust entanglement gates for trapped-ion qubits," *Physical Review Letters*, vol. 121, no. 18, p. 180502, 2018.

[17] C. Monroe, W. Campbell, L.-M. Duan, Z.-X. Gong, A. Gorshkov, P. Hess, R. Islam, K. Kim, N. Linke, G. Pagano, *et al.*, "Programmable quantum simulations of spin systems with trapped ions," *Reviews of Modern Physics*, vol. 93, no. 2, p. 025001, 2021.

[18] F. Rajabi, S. Motlakunta, C.-Y. Shih, N. Kotibhaskar, Q. Quraishi, A. Ajoy, and R. Islam, "Dynamical Hamiltonian engineering of 2d rectangular lattices in a one-dimensional ion chain," *npj Quantum Information*, vol. 5, no. 1, pp. 1–7, 2019.

[19] A. Parra-Rodriguez, P. Lougovski, L. Lamata, E. Solano, and M. Sanz, "Digital-analog quantum computation," *Physical Review A*, vol. 101, no. 2, p. 022305, 2020.

[20] J. Pino, J. Dreiling, C. Figgatt, J. Gaebler, S. Moses, M. Allman, C. Baldwin, M. Foss-Feig, D. Hayes, K. Mayer, *et al.*, "Demonstration of the trapped-ion quantum CCD computer architecture," *Nature*, vol. 592, no. 7853, pp. 209–213, 2021.

[21] X. Fu, L. Riesebos, M. Rol, J. Van Straten, J. Van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, V. Newsum, K. Loh, *et al.*, "eqasm: An executable quantum instruction set architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 224–237, IEEE, 2019.

[22] D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, *et al.*, "Qiskit backend specifications for OpenQASM and OpenPulse experiments," *arXiv preprint arXiv:1809.03452*, 2018.

[23] G. Kasprowicz, P. Kulik, M. Gaska, T. Przywozki, K. Pozniak, J. Jarosinski, J. W. Britton, T. Harty, C. Balance, W. Zhang, *et al.*, "Artiq and sinara: Open software and hardware stacks for quantum physics," in *Quantum 2.0*, pp. QTu8B–14, Optical Society of America, 2020.

[24] T. Choi, S. Debnath, T. Manning, C. Figgatt, Z.-X. Gong, L.-M. Duan, and C. Monroe, "Optimal quantum control of multimode couplings between trapped ion qubits for scalable entanglement," *Physical Review Letters*, vol. 112, no. 19, p. 190502, 2014.

[25] K. Mølmer and A. Sørensen, "Multiparticle entanglement of hot trapped ions," *Physical Review Letters*, vol. 82, no. 9, p. 1835, 1999.

[26] C. A. Sackett, D. Kielpinski, B. E. King, C. Langer, V. Meyer, C. J. Myatt, M. Rowe, Q. Turchette, W. M. Itano, D. J. Wineland, *et al.*, "Experimental entanglement of four particles," *Nature*, vol. 404, no. 6775, pp. 256–259, 2000.

[27] P. H. Leung, K. A. Landsman, C. Figgatt, N. M. Linke, C. Monroe, and K. R. Brown, "Robust 2-qubit gates in a linear ion crystal using a frequency-modulated driving force," *Physical Review Letters*, vol. 120, no. 2, p. 020501, 2018.

[28] C. D. Bentley, H. Ball, M. J. Biercuk, A. R. Carvalho, M. R. Hush, and H. J. Slatyer, "Numeric optimization for configurable, parallel, error-robust entangling gates in large ion registers," *Advanced Quantum Technologies*, vol. 3, no. 11, p. 2000044, 2020.

[29] Free Software Foundation, "GCC internals." https://gcc.gnu.org/onlinedocs/gccint/index.html, Accessed 6 Jun 2021.

[30] R. Laflamme, C. Miquel, J. P. Paz, and W. H. Zurek, "Perfect quantum error correcting code," *Physical Review Letters*, vol. 77, no. 1, p. 198, 1996.

[31] R. Chao and B. W. Reichardt, "Quantum error correction with only two extra qubits," *Physical Review Letters*, vol. 121, no. 5, p. 050502, 2018.

[32] P. Richerme, C. Senko, J. Smith, A. Lee, S. Korenblit, and C. Monroe, "Experimental performance of a quantum simulator: Optimizing adiabatic evolution and identifying many-body ground states," *Physical Review A*, vol. 88, no. 1, p. 012334, 2013.