Cirq: A Python Framework for Creating, Editing, and Invoking Quantum Circuits

DESIGN DOCUMENT

Team 8

Name	Role
Victory Omole	Project Client
Akhilesh Tyagi	Project Sponsor
Aj Hanus	Organizer, Researcher, Developer
Andrew Hancock	Researcher, Developer
Austin Garcia	Researcher, Developer
Jacob Shedenhelm	Researcher, Developer
Jordan Cowen	Researcher, Developer
Calista Carey	Researcher, Developer

sdmay20-08@iastate.edu
https://sdmay20-08.sd.ece.iastate.edu

Executive Summary

Engineering Standards and Design Practices

The standards required for our project are that all tests in the Cirq repository pass and that all the code that we create has full code coverage with tests. We also signed a contributor agreement license which means that all the code we submit to the Cirq repository is open-source and that Cirq has permission to use and redistribute our contributions as part of the project.

One of the IEEE standards that applied to our project is Software Reviews and Audits. This standard defines five different types of reviews that we can choose to conduct when we start developing the project next semester. We find this standard important as needed to know the best type of review to hold during the duration of our project: management reviews, technical reviews, inspections, walk-throughs, and audits. The standard goes into depth of the procedure required for each, which made it easier for us to determine which to choose.

We also found that the Software Testing, an IEEE Standard, also applied to our project. This standard helped us outline how to set up our tests. As can be found in the documentation for the Cirq repository, current standards have already been set up for the code that has been written. We have identified that the standards outlined in the documentation are in line with the IEEE standard that we read about. The bullet that we found most intriguing was the “Coverage” section. This section explains that the Cirq repository should have 100% code coverage with tests. We took this into heavy consideration when we wrote the code for QUIL had tests that cover it. We also continually made sure that all of the tests continued to pass we combined all of our work.

For this project, we used VSCode to write python code and py tests. We created an Amazon EC2 Linux instance to run the Cirq tests. We will be working in Agile sprints starting in the second half of the semester when we begin designing writing the functions for each of the quantum computing gates.

Summary of Requirements

1. Implement the functionality to output a cirq.Circuit to QUIL
    - https://github.com/quantumlib/Cirq/issues/2386
2. Finish the implementation of the TwoQubitDiagonalGate
    - https://github.com/quantumlib/Cirq/pull/2591
3. Complete smaller issues assigned to group

Applicable Courses from Iowa State University Curriculum

- CS 309 - Agile practices, GitHub experience
- Math 207 - Linear Algebra
- CS 104 - Python course

New Skills/Knowledge acquired that was not taught in courses

- Knowledge of the Quantum Computing
- Knowledge of Google's Cirq Quantum Computing Repository
- Python programming language

Table of Contents

1. Introduction **6**
1.1 Acknowledgement 6
1.2 Problem and Project Statement 6
1.3 Requirements 7
1.4 Intended Users and Uses 7
1.6 Expected End Product and Deliverables 8

2. Specifications and Analysis **10**
2.1 Proposed Design 10
2.2 Development Process 10
2.3 Design Plan 11

3. Statement of Work **13**
3.1 Previous Work And Literature 13
3.2 Technology Considerations 13
3.3 Task: Export as QUIL 13
3.4 Possible Risks And Risk Management 14
3.5 Project Proposed Milestones and Evaluation Criteria 14
3.7 Expected Results and Validation 15

4. Project Timeline, Estimated Resources, and Challenges **16**
4.1 Project Timeline 16
4.2 Feasibility Assessment 17
4.3 Personnel Effort Requirements 17
4.4 Other Resource Requirements 18
4.5 Financial Requirements 18

5. Testing and Implementation **19**
5.1 Interface Specifications 19
5.2 Hardware and software 19
5.3 Functional Testing 19
5.4 Non-Functional Testing 19
5.5 Process 20
5.6 Results 20

6. Closing Material **20**
6.1 Conclusion 20
6.2 References 21
6.3 Appendices 21

1. Introduction

1.1 Acknowledgement

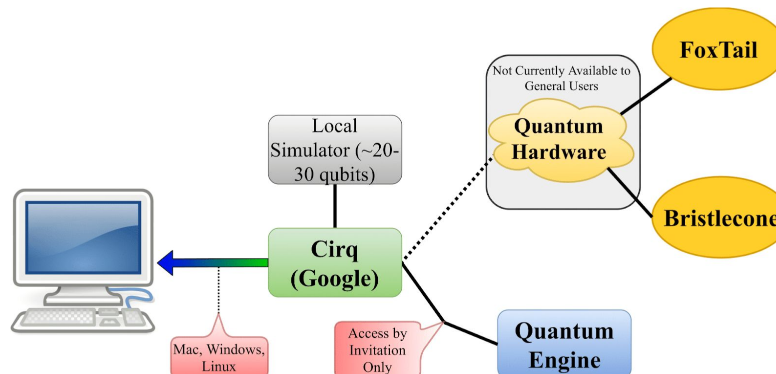
We would like to thank Victory Omole, our client, and the rest of the Cirq developers for providing us with the Cirq repository and a good base to begin our implementation. Our client also provided us with significant assistance for gaining knowledge in Quantum Computing by providing us many lecture notes and other material. He was also very prompt in answering any questions we had for him which we appreciate greatly. We would also like to thank Akhilesh Tyagi, our advisor. He helped coordinate our project when we were having difficulty early in the process and was prompt when we requested meetings to discuss our project.

1.2 Problem and Project Statement

Cirq is a programming language for quantum computing. QUIL is an instruction set architecture being developed by Rigetti Computing for quantum processors. Our main goal is to translate quantum circuits designed in Cirq to QUIL. This would allow people to use Cirq-designed quantum circuits in programs/machines that only run QUIL such as the QVM (Quantum Virtual Machine) designed by Rigetti. Cirq already has the ability to translate to other quantum protocols such as QASM (Quantum Assembly Language).

Based on the QASM translation that was already provided within the Cirq repository, our group implemented the translation for each of the standard, nonstandard, and control flow of QUIL gates. We completed a working implementation with clean code and thorough documentation. During this process we gained a deeper knowledge of Cirq, QUIL, and quantum computing in general. As seen in Figure 1, Cirq was designed to be able to output to both Quantum Engines and Quantum Hardware so the QUIL implementation will increase Cirq's breadth to a larger range of the quantum computing industry.

Figure 1: ***Illustrates how Cirq interacts with the real world***



1.3 Requirements

1. Implement issue #2386 in the Cirq repository (https://github.com/quantumlib/Cirq):

   For this requirement, implemented the file, quil_output.py. Cirq allows for conversions between different Quantum Computing circuit specifications, and they wanted to allow a circuit to be exported as QUIL. This was similar to how Cirq already outputs QASM. We used python and coding in Visual Studio Code.

2. Finish the pull request (https://github.com/quantumlib/Cirq/pull/2891) for TwoQubitDiagonalGate.

   This requirement was assigned later in the semester. For this requirement, we needed to finish the implementation of the TwoQubitDiagonalGate. This was important because three standard gates that needed translations-- CPHASE00, CPHASE01, and CPHASE10-- depend on this Criq gate.

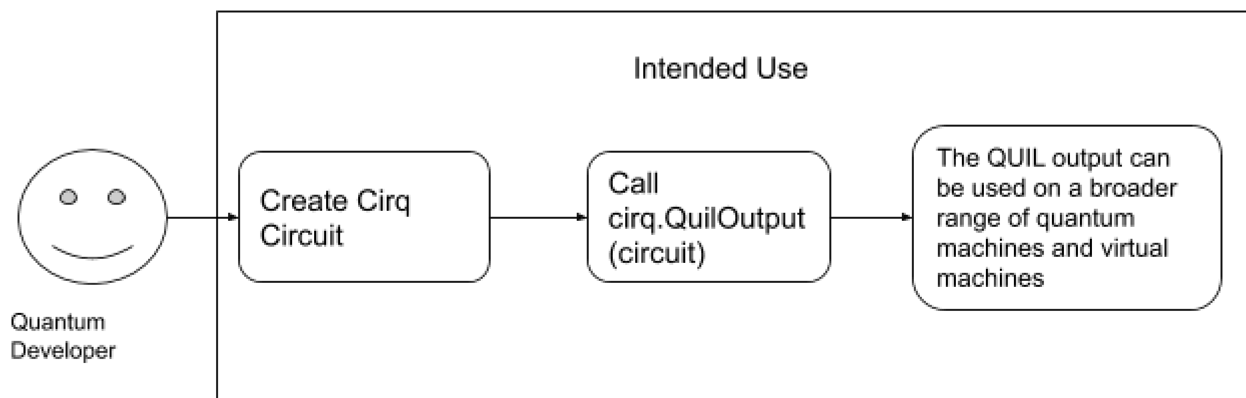
3. Complete smaller issues:

   For the first part of the first semester, our group members completed smaller issues in the Cirq repository in order to become more familiar with the Cirq repository and the code syntax they commonly use.

1.4 Intended Users and Uses

The intended user for our QUIL translation is anyone who uses Cirq to create their quantum circuits and wants to export their circuit to QUIL. Many scientists use quantum computers to conduct virtual experiments, and they may want to translate their circuits into QASM or QUIL [3]. Cirq already has QASM implemented, so it was our job to give the users the option to export to QUIL as well.

Figure 2: ***Diagram to show intended use of QUIL***



1.5 Assumptions and Limitations

Assumptions:

- The current code in the repo does not break.
- Current code can withstand large data tests.
- Our code will be reviewed by professionals that understand Cirq and quantum computing
- The automated testing and format checking system function correctly

Limitations:

- Right now, it is hard to make a quantum computer on a large scale [3].
- Currently have minimal knowledge of Quantum Computing.
- Inexperienced with Cirq codebase
- Not a 1-to-1 relationship as to how circuits in Cirq are used and how QUIL must be exported. We will have to implement new Cirq gates to be compatible with QUIL.

1.6 Expected End Product and Deliverables

We spent the last two semesters working for the Google repository, Cirq, so it is important that we kept Victory up to date with the work our group does throughout the entire process. Because of this, we will tested thoroughly and pushed often, but we worked in monthly sprints. All work for the sprint was planned to be done by the last day of the month (except December).

First Semester (completed in order):

- By the end of September, all group members should have started researching and studying Quantum Computing. The members that have not had previous experience with python need to start learning the language. Cirq works exclusively in python, so it's crucial all group members know the language prior to starting the actual implementation.
- Throughout the month of October, smaller issues will be assigned to group members in order to familiarize themselves with the Cirq repository. All group members should have all smaller repository issues that were assigned completed. Along with this, all group members should research QUIL and gain a thorough understanding of the similarities and differences it shares with Cirq.
- By the end of November, we will have begun designing a basic parser and working on converting the gates to QUIL. We will continue to look into the commonalities between Cirq and QUIL..
- This month is shorter due to the fact that there are finals and winter break beginning this month. In December, all official roles will be assigned to the project. Everyone will know which specific part they are working on in the project. After completing the design, we will update the documentation in the repository to include our design choices and diagrams we created.

Everything we planned to have done by the end of the first semester was completed on time.

Second Semester:

- By the end of January, we will have begun the first implementation of our design. We should have a good base of quil_output.py complete. The initial program should be almost ready to be tested.
- By the end of February, our first implementation should be completed. We will begin testing the file and simulating test cases. We will make improvements to our initial code as needed, and get ready to optimize the solution.
- By the end of March, we should begin optimizing our solution. As well, if our work on quil_output.py is complete, we will work on other issues within the repo. Having the experience that we will gain throughout the previous months while working on Cirq will be very beneficial at this stage.
- By the end of April, we should be ready for the release of the final product. The first part of the month will be spent heavily testing the end program and fine tuning our changes. We will push our final product on April 30th including any changes to the documentation that needs to be made.

We did not finish out first implementation by 2/29 like we had originally hoped. We ended up finishing most of the work by the Friday before Spring Break and continued testing before opening the pull request with all of our work. We quickly got back on track and began optimizing once this pull request was opened.

2. Specifications and Analysis

2.1 Proposed Design

We modeled our QUIL translation off of the preexisting implementation of outputting to QASM format. The general structure is the following:

- Each gate has a private function, _quil_ that returns itself as it would be written in QUIL syntax.
- There is a parser for the Cirq circuit that will iterate through each gate and call the function that should return QUIL syntax.

The parser has its own file with other necessary methods. Each gate's translation function is implemented in that gates class. We determined that there are three separate groups of work required for the translation. We needed to create private functions for both the standard and nonstandard QUIL gates. Lastly, we needed a parser which was implemented in the quil_output.py file.

Table 1: ***Displays Non-Functional and Functional Requirements***

Non-functional Requirements	Functional Requirements
Performance- Our parser should be able to work efficiently with larger circuits.	Correct translation- Our implementation must correctly translate Cirq circuits. They must be usable by programs or machines that accept QUIL
Supports all gates- We should be able to handle any gates that Cirq could use in a circuit and if not gracefully report an error in translation	Be used for virtual experiments- Quantum Computing is often used for simulation of experiments, so our implementation will need to be able to withstand any sort of experimentation that is conducted.

2.2 Development Process

Our team worked with an Agile mindset. Each week we gave each team member the tasks they needed to complete for the week and we communicated through a meeting or group message if we can not find a meeting time for that week. Section 1.6 outlines the deliverables we worked on throughout the semester and our tasks each week were assigned to lead us towards each month's goal.

In Figure 3, we have outlined the quil_output class we needed to implement. This class can be passed the operations, qubits, a header, the precision, and version. The class outputs a string representation of the circuit in QUIL format. This class will also utilize the "__quil__" we added to each of the standard QUIL gates and "_decompose_" functions that already existed.

Figure 3: ***Outline for the QuilOutput class we are implementing***

quil_output.py

Class QuilOutput:
def __init__(self, operations, qubits, header, precision, version) -> None - Initialize a QuilOutput object
def _generate_measurement_ids(self) -> Tuple[Dict[str, str], Dict[str, Optional[str]]] - Generate the measurement keys that contributed to building the circuit
def _generate_qubit_ids(self) -> Dict['cirq.Qid', str] - Give each qubit a unique id
def _is_valid_quil_id(self, id_str) -> bool - Test if id_str is a valid id in QUIL grammar
def _save(self, path) -> None - Write QUIL output to a file specified by the path
def __str__(self) -> str - Return QUIL output as a string
def _write_quil(self, output_func) -> None - Generate QUIL string, calls self._write_operations
def _write_operations(self, op_tree, output, output_line_gap) -> None - Convert operations to QUIL format

2.3 Design Plan

Our design plan was to model the architecture of our code off of the code already present within the Cirq code base. This provided an easier understanding to the current people working on Cirq as well as new people joining the project. The largest constraint within our project was to implement as much cross conversion functionality as possible between Cirq and QUIL. We had to come up with a large number of tests covering a variety of quantum circuit types to be converted. The most important use-case of our project is that we consistently output QUIL specification that can be regenerated in Cirq and can be implemented wherever the Cirq repository could be used. Because Cirq is meant for near-term quantum computing, we wanted to ensure that the parser we choose works well for this technology.

To approach the issue, we created one pull request that was split into three major sections of work. The pull request included the following:

1. Implement cirq output for Standard QUIL Gates
2. Implement cirq output for Non-Standard Gates

3. Implement cirq output for Control flow and Measurements

The design plan is outlined, including different Non-Standard and Standard Gates as well as the control flow and measurements that will be added, in 6.3 Appendices. Each page is a markdown document that we created and added to the Cirq repository issue.

3. Statement of Work

3.1 Previous Work And Literature

Cirq contributors have already done a decent amount of work on a similar task of converting between Cirq circuits and QASM. We modeled our work closely after the QASM conversion code. Obviously the output will be different but we will structure the code similarly but using QUIL.

QUIL is "an abstract machine architecture for classical/quantum computations---including compilation---along with a quantum instruction language called QUIL for explicitly writing these computations. With this formalism, we discuss concrete implementations of the machine and non-trivial algorithms targeting them. The introduction of this machine dovetails with ongoing development of quantum computing technology, and makes possible portable descriptions of recent classical/quantum algorithms." [3].

3.2 Technology Considerations

Since Cirq is built with Python, the only possible option to complete our project involved building our solution with Python. All work towards deliverables was in Python using the current Cirq features. To run and test our work, we used an Amazon EC2 Linux Virtual Machine. We used GitHub and GroupMe for all project management.

3.3 Task: Export as QUIL

Our project was split into the following tasks:

- Learning Quantum Computing and Python
- Complete initial issues to familiarize ourselves with Cirq
- Study the QASM output, and determine how we can model QUIL closely to this sort of export function
- Assign official roles to everyone
- Implement file quil_output.py
- Test and edit the output file
- Optimize work up to this point
- Release final result for implementation into Cirq

3.4 Possible Risks And Risk Management

Skill Training Time: Since our team was unfamiliar with quantum computing concepts and some individuals were unfamiliar with Python, the time before being able to fully jump into development on the main deliverable was a bit significant. We actually had to change our original project goal as the concepts were too hard for members to grasp in a timely manner. This is how the QUIL translation become our project. We still had unfamiliarity with some of the concepts involving QUIL. A way to reduce this risk was to ask Victory questions when needed and help out other group members who may not be picking up the material quickly.

Software Updates: Any updates to the Cirq platform or any third party software being used by our team may trigger setbacks that could halt development. This is an accepted risk with any software project.

Scheduling Conflicts: This project spanned across two semesters with six college seniors attempting to finish their degrees. We definitely had some troubles finding meeting times where all six members could be present. When this occurred, we would meet with as many people as possible then relay all the information back to the members who missed the meeting. We also helped reduce this risk by making sure team members are in constant communication regarding their capability to contribute to the project at any one time.

3.5 Project Proposed Milestones and Evaluation Criteria

Our key milestones are as follows:

- Complete Quantum Computing and Python learning objectives; evaluation in an as needed basis.
- Complete initial issues to familiarize ourselves with Cirq; initial issues have been closed.
- Study qasm_output.py and determine how QUIL can be modeled similarly; pick the gates we need to include in our design
- Complete official role assignment; each team member has an assigned role.
- Complete initial implementation of quil_output.py; new circuit export function has been translated into Cirq but still requires testing and debugging.
- Complete testing and revision of the file; the project is fully functional.
- Complete optimization; code has been cleaned, commented, and performance is optimized for implementation in Cirq.
- Release final result for implementation into Cirq; the project is fully functional and running within the Cirq framework.

3.6 Project Tracking Procedures

Each group member was assigned to a specific section of the pull request to implement during this project (assignments shown below). Cirq has permission to use and redistribute our contributions as part of the project.

Table 2: ***Work Assignments for Group Member's***

Part of the Pull Request	Members
Implement cirq output for Standard QUIL Gates	Calista, Jake
Implement cirq output for Non-Standard Gates	Jordan, Andrew
Implement cirq output for Control flow and Measurements	AJ, Austin

Our group used GitHub to track deliverables and assign issues to team members. Github also allowed team members to view the progress of other members and ease the collaboration process. We also created a spreadsheet that documents whether or not a member has completed a gate. We used this spreadsheet and GitHub contributions to track everyone's process. Also, weekly meetings helped everyone stay accountable for the work they need to do. Meetings between pairs were scheduled individually and as needed, whereas group meetings happened at least once a week-- typically on Sundays.

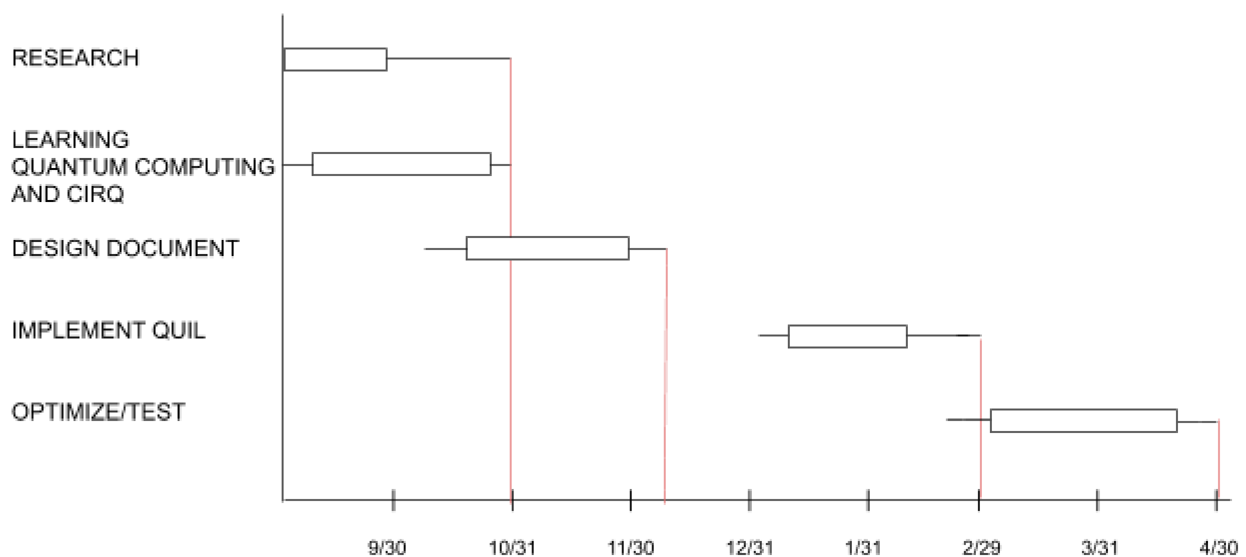
3.7 Expected Results and Validation

The desired outcome of the project is to implement an output function that translates circuits in Cirq to QUIL. The output should model the current possible QASM output, but containing information relevant to QUIL. We validated this result with a set of tests with predetermined results.

4. Project Timeline, Estimated Resources, and Challenges

4.1 Project Timeline

Figure 4: ***Timeline that outlines our project plan. The white boxes identify when the majority of the work for that phase will be completed***



To start the semester this year we began by researching. None of us have any background in the subject material required for our project - quantum mechanics and quantum computing. By September 30 we all read the lecture notes provided to us by our sponsor Victory and watched the accompanying videos. By the end of October we had finished our assigned first issues on GitHub in order to become accustomed to the Cirq framework and understand how to use it and develop within it. By the end of November we had studied circuits in Cirq, QASM, and how QUIL should be exported. At this point we decided upon the best way to implement the parser that allowed for exporting to happen. By December 13th, the end of the semester, we determined official project roles assigned specifically pertaining to implementing the parser, and so that each member can begin their specific work over winter break. By the end of January we had a good base of quil_output.py complete, and it should be almost ready to be tested. By the end of February we planned to have the initial implementation of quil_output.py finished, but we didn't

end up finishing all our code until Spring Break. We will then began thorough testing of the function to verify its proper operation. By the end of March we had all of the testing finished and have a large variety of tests that verify proper implementation of the QUIL parser. These included functional tests, as well as speed tests and stress tests. We have began optimizing our solution and updating our project based on what Victory and other developers have to say about our work. We want to increase performance and potentially pass more speed and stress tests that were not feasible in the first implementation. Optimization will be finished by the end of April and the final product will be delivered.

4.2 Feasibility Assessment

Feasibly, this project will result in the successful python implementation within Cirq's framework of the translation of a circuit to QUIL. One challenge with this was learning the material necessary. Nobody on the team had prior experience with quantum computing or quantum mechanics and therefore needed to do a lot of research.

4.3 Personnel Effort Requirements

Read Lecture Notes: 8/21 - 9/30	Team members were required to read/watch accompanying videos for 5 lectures per week. This would translate to about an hour a day Monday-Friday.
Complete "Good First Issues" in Cirq: 10/1 - 10/31	Varying effort is required depending on the issue assigned and the familiarity with the style of Cirq's development cycle. Team members should at least be expected to spend a few hours a week until they have completed their issues.
Begin the design implementation of QUIL: 11/1 - 11/30	All team members required to study the QASM files, and bring one idea as to how we can write the basic QUIL parser. In order to ensure how our design will be best for Cirq, members should continue to look into the commonalities between Cirq and QUIL. Should spend 1 hour a day writing code for this basic parser
Assign official roles and delimit tasks: 12/1 - 12/13	Since this is before finals not much effort will be required. We will meet as a team and assign roles once we've determined the structure of our parser.
Work on completing quil_output.py:	During this time, we will expect team members to

12/14 - 3/12	push code at least once a week and we will meet every Sunday to verify everyone's code is working together properly. Each member should spend 1-2 hours each day working on creating the parser
Testing Phase: 3/12 - 3/31	Everyone will be required to write at least two tests a week so that we can generate a large number of quality tests.
Optimization Phase: 4/1-4/30	Effort requirements will be similar to the first implementation phase. Members will be required to push code once a week and we will meet to verify optimizations still pass all the tests we designed.

4.4 Other Resource Requirements

All team members had access to the AWS EC2 instance that ran an instance of Linux. This helped with testing purposes between group members.

4.5 Financial Requirements

The Amazon EC2 instance ran at $0.094/hour, so in total, $54 was spent on running this instance

5. Testing and Implementation

5.1 Interface Specifications

Our project did not involve any user interfaces. Rather, added functionality to the backend that processes user designed circuits. Since Cirq is a quantum computer programming language there is no real user interface. It's an API that allows users to write python files that describe the circuit they would like to experiment with.

5.2 Hardware and software

There will be no hardware directly involved in our project. This makes sense since Cirq is used to simulate hardware. There's at least one real quantum computing chip already programmed into Cirq for simulation. It is Google's Bristlecone chip.

Software wise, we added two files, one for testing and one for the actual program. The file for our issue is going to be quil_output.py and the file to test our issue will be quil_output_test.py. We modified many other files that contain previously as well as newly implemented gates in order to add our translation functions and relevant tests.

5.3 Functional Testing

Each file in cirq has a test file in its folder that shares the name with "_test" appended to the end. This will test the full functionality of the file with the goal of getting 100% coverage besides where there are coverage exceptions. In order to run these tests we used the pytest library. We used a variety of circuits, making sure each gate is present, to test correct translation. We will also have a test for each function that will return QUIL syntax.

5.4 Non-Functional Testing

In order to test for performance we will time the simulation of circuits. There will not be any testing in relation to security. We ran format and lint checks to make sure our code follows the Cirq standard. These are already implemented in the Cirq repo.

5.5 Process

This is our process:

1. Each “feature” within the pull request has two members assigned to it.
2. Implement the features in that pull request
3. Implement relevant tests
4. Verify 100% of tests pass
5. Verify 100% of lint and format checks pass

For the parser, we needed more group work and collaboration. All changes were completed on one forked repository owned by a group member.

5.6 Results

After finishing all of our tasks, we made the test file that ran 17 various tests to check code coverage. All of the tests pass. Examples of the output and types of tests written are provided in the markdown file which is included in section 6.3 Appendices. Overall, the output and results of our translation are what is expected.

```
(cirq-py3) ubuntu@ip-172-31-17-21:~/aj/cirq$ pytest /home/ubuntu/aj/cirq/cirq/circuits/quil_output_test.py
================================ test session starts ================================
platform linux -- Python 3.6.9, pytest-3.8.2, py-1.8.1, pluggy-0.13.1
benchmark: 3.2.3 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False wa
rmup_iterations=100000)
rootdir: /home/ubuntu/aj/cirq, inifile:
plugins: asyncio-0.10.0, cov-2.5.1, benchmark-3.2.3
collected 17 items

cirq/circuits/quil_output_test.py .................                                [100%]

================================ 17 passed in 0.08 seconds ================================
```

6. Closing Material

6.1 Conclusion

Thus far, our team has been working hard to gain a fundamental understanding of quantum computing. Our sponsor, Victory, gave us lectures to read and watch. As stated prior, our primary goals/requirements are to implement issue #2386 in the Cirq repository (https://github.com/quantumlib/Cirq), study quantum computing and to complete smaller issues along the way. Our plan of action is spread out over two semesters, the first semester will be spent completing the smaller issues and learning more about quantum computing. Second semester, we will work together to complete our primary requirement, https://github.com/quantumlib/Cirq. Since our team has limited knowledge of quantum

computing, this plan will work very well because it will allow us to gain ground work before attempting a difficult issue.

6.2 References

https://github.com/quantumlib/Cirq

[1] Ambainis, Andris, et al. “What Can We Do with a Quantum Computer?” *Institute for Advanced Study*, www.ias.edu/ideas/2014/ambainis-quantum-computing.

[2] LaRose, R. (2019). Review of the Cirq Quantum Software Framework. [online] Quantumcomputingreport.com. Available at: https://quantumcomputingreport.com/scorecards/review-of-the-cirq-quantum-software-framework/ [Accessed 8 Oct. 2019].

[3] Smith, Robert S., et al. “A Practical Quantum Instruction Set Architecture.” *ArXiv.org*, 17 Feb. 2017, https://arxiv.org/abs/1608.03355.

6.3 Appendices

Lecture Notes:

https://www.scottaaronson.com/blog/?p=3943

Lecture Videos:

https://www.youtube.com/playlist?list=PLm3J0oaFux3YL5qLskC6xQ24JpMwOAeJz

Design for export to QUIL in Cirq

To output a circuit to QUIL format in Cirq, the following call can be made:

Below is an example of printing the QUIL output in Cirq:

```
q0, = _make_qubits(1)
output = cirq.QuilOutput((cirq.X(q0),), (q0,))
print(output)
```

The print statement would output the following:

```
# Created from Cirq
X 0
```

Implementation

The project will be separated into one pull request with three different parts. Each part will have 2 - 3 people working on it. The pull requests will be as follows:

- Part 1: Implement cirq output for Standard QUIL Gates
- Part 2: Implement cirq output for Non-Standard Gates
- Part 3: Implement cirq output for Control flow and Measurements

Additionally, we will be adding the TwoQubitDiagonalGate. This gate will reduce the workload for our non standard gates.

Part 1: Implement Cirq output for Standard QUIL Gates

For each Cirq gate that can be output to a standard QUIL gate, a quil representation was add to the gate.

Below is an example of the quil function for the X gate:

```
def _quil_(self, args, qubits):
    return args.format("X {0}", qubits[0])
```

Cirq Gate	QUIL Gate	Notes
Cirq.H	H	Hadamard gate
Cirq.X	X	Pauli X (PI rotation over X-axis) aka “NOT” gate
Cirq.Y	Y	Pauli Y (PI rotation over Y-axis)
Cirq.Z	Z	Pauli Z (PI rotation over Z-axis)
Cirq.RX	RX	Rotation around the X-axis by given angle
Cirq.RY	RY	Rotation around the Y-axis by given angle
Cirq.RZ	RZ	Rotation around the Z-axis by given angle
Cirq.S	S	PI/2 rotation over Z-axis (synonym for r2)
Cirq.T	T	PI/4 rotation over Z-axis (synonym for r4)
Cirq.CZ (http://Cirq.CZ)	CZ	A gate that applies a phase to the |11⟩ state of two qubits.
Cirq.FREDKIN	CSWAP	A controlled swap gate
Cirq.CSWAP	CSWAP	Controlled swap aka “Fredkin” gate
Cirq.Identity	I	A Gate that perform no operation on qubits.
Cirq.CX (http://Cirq.CX)	CNOT	Controlled Pauli X (PI rotation over X-axis) aka “CNOT” gate
Cirq.CCX	CCNOT	Toffoli aka “CCNOT” gate
Cirq.TOFFOLI	CCNOT	Toffoli aka “CCNOT” gate
Cirq.SWAP	SWAP	Swaps the state of two qubits.
Cirq.ISWAP	ISWAP	Rotates the |01⟩-vs-|10⟩ subspace of two qubits around its Bloch X-axis.
Cirq.SwapPowGate	PSWAP	The SWAP gate, possibly raised to a power, exchanges qubits.
Criq.CZPowGate	CPHASE	A gate that applies a phase to the |11⟩ state of two qubits.
Cirq.TwoQubitDiagnalGate	CPHASE00	

Cirq Gate	QUIL Gate	Notes
Cirq.TwoQubitDiagnalGate	CPHASE01	
Cirq.TwoQubitDiagnalGate	CPHASE10	
Cirq.WaitGate	WAIT	

Part 2: Implement cirq output for Non-Standard Gates

Non-Standard Cirq Gates

For Cirq gates that are not standard gates in QUIL, the `DEFGATE` keyword will be used to output a new gate in QUIL that can be used to represent the Cirq gate. For example:

Cirq.FSimGate

```
DEFGATE FSIMGATE:
    1, 0            , 0            , 0
    0, cos(%theta)   , -i*sin(%theta), 0
    0, -i*sin(%theta), cos(%theta)   , 0
    0, 0            , 0            , exp(-i*%phi)
```

Below is a list of the Cirq gates we intend to add:

- XPowGate
- XXPowGate
- YPowGate
- YYPowGate
- ZPowGate
- ZZPowGate
- QuilOneQubitGate
- QuilTwoQubitGate

Part 3: Implement cirq output for Control flow and Measurements

QUIL keywords to be added

QUIL Keyword	Notes
FORKED	If statement on a specific qubit
DAGGER	Represents the complex-conjugate transpose
CONTROLLED	Takes a gate acting on a qubit and conditions it on a new qubit
WAIT	Suspends quantum computations while classical computations are performed (Cirq.WaitGate)
RESET	Brings the qubit to the zero state
MEASURE	Used to measure the state of a qubit
JUMP	Enables the ability to jump to different parts of the QUIL circuit
PRAGMA	Comments
INCLUDE	Includes another QUIL file (parses in gate and circuit definitions)

Cirq Circuits

For larger gates that can contain a `_decompose_` function, QUIL's `DEFCIRCUIT` command will be used.

Screenshots / Example Images

Pauli CZ gate in Cirq to Quil

Input:

```
q0, q1, = _make_qubits(2)
output = cirq.QuilOutput(PauliInteractionGate.CZ.on(q0, q1), (
    q0,
    q1,
))
```

Output:

```
#Created Using Cirq

CZ 0 1
```

All Gates in Cirq to Quil

Input:

```
            cirq.Z(q0),
            cirq.Z(q0)**.625,
            cirq.Y(q0),
            cirq.Y(q0)**.375,
            cirq.X(q0),
            cirq.X(q0)**.875,
            cirq.H(q1),
            cirq.CZ(q0, q1),
            cirq.CZ(q0, q1)**0.25,  # Requires 2-qubit decomposition
            cirq.CNOT(q0, q1),
            cirq.CNOT(q0, q1)**0.5,  # Requires 2-qubit decomposition
            cirq.SWAP(q0, q1),
            cirq.SWAP(q0, q1)**0.75,  # Requires 2-qubit decomposition
            cirq.CCZ(q0, q1, q2),
            cirq.CCX(q0, q1, q2),
            cirq.CCZ(q0, q1, q2)**0.5,
            cirq.CCX(q0, q1, q2)**0.5,
            cirq.CSWAP(q0, q1, q2),
            cirq.XX(q0, q1),
            cirq.XX(q0, q1)**0.75,
            cirq.YY(q0, q1),
            cirq.YY(q0, q1)**0.75,
            cirq.ZZ(q0, q1),
            cirq.ZZ(q0, q1)**0.75,
            cirq.IdentityGate(1).on(q0),
            cirq.IdentityGate(3).on(q0, q1, q2),
            cirq.ISWAP(q2, q0),  # Requires 2-qubit decomposition
            cirq.PhasedXPowGate(phase_exponent=0.111, exponent=0.25).on(q1),
            cirq.PhasedXPowGate(phase_exponent=0.333, exponent=0.5).on(q1),
            cirq.PhasedXPowGate(phase_exponent=0.777, exponent=-0.5).on(q1),
            cirq.WaitGate(0).on(q0),
        cirq.measure(q0, key='xX'),
        cirq.measure(q2, key='x_a'),
        cirq.measure(q3, key='X'),
        cirq.measure(q2, key='x_a'),
        cirq.measure(q1, q2, q3, key='multi', invert_mask=(False, True))
```

Output:

```
#Created Using Cirq

DECLARE m0 BIT[1]
DECLARE m1 BIT[1]
DECLARE m2 BIT[1]
DECLARE m3 BIT[3]

Z 0
RZ(0.625) 0
Y 0
RY(0.375) 0
X 0
RX(0.875) 0
H 1
CZ 0 1
CPHASE(0.25) 0 1
CNOT 0 1
RY(-0.5) 1
CPHASE(0.5) 0 1
RY(0.5) 1
SWAP 0 1
PSWAP(0.75) 0 1
H 2
CCNOT 0 1 2
H 2
CCNOT 0 1 2
RZ(0.125) 0
RZ(0.125) 1
RZ(0.125) 2
CNOT 0 1
CNOT 1 2
RZ(-0.125) 1
RZ(0.125) 2
CNOT 0 1
CNOT 1 2
```

```
RZ(-0.125) 2
CNOT 0 1
CNOT 1 2
RZ(-0.125) 2
CNOT 0 1
CNOT 1 2
H 2
RZ(0.125) 0
RZ(0.125) 1
RZ(0.125) 2
CNOT 0 1
CNOT 1 2
RZ(-0.125) 1
RZ(0.125) 2
CNOT 0 1
CNOT 1 2
RZ(-0.125) 2
CNOT 0 1
CNOT 1 2
RZ(-0.125) 2
CNOT 0 1
CNOT 1 2
H 2
CSWAP 0 1 2
X 0
X 1
RX(0.75) 0
RX(0.75) 1
Y 0
Y 1
RY(0.75) 0
RY(0.75) 1
Z 0
Z 1
RZ(0.75) 0
```

```
RZ(0.75) 1
I 0
I 0
I 1
I 2
ISWAP 2 0
RZ(-0.111) 1
RX(0.25) 1
RZ(0.111) 1
RZ(-0.333) 1
RX(0.5) 1
RZ(0.333) 1
RZ(-0.777) 1
RX(-0.5) 1
RZ(0.777) 1
WAIT
MEASURE 0 m0[0]
MEASURE 2 m1[0]
MEASURE 3 m2[0]
MEASURE 2 m1[0]
MEASURE 1 m3[0]
X 2 # Inverting for following measurement
MEASURE 2 m3[1]
MEASURE 3 m3[2]
```

Total of 17 tests written for 100% code coverage

```
(cirq-py3) ubuntu@ip-172-31-17-21:~/aj/cirq$ pytest /home/ubuntu/aj/cirq/cirq/circuits/quil_output_test.py
================================================================ test session starts =================================================================
platform linux -- Python 3.6.9, pytest-3.8.2, py-1.8.1, pluggy-0.13.1
benchmark: 3.2.3 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False wa
rmup_iterations=100000)
rootdir: /home/ubuntu/aj/cirq, inifile:
plugins: asyncio-0.10.0, cov-2.5.1, benchmark-3.2.3
collected 17 items

cirq/circuits/quil_output_test.py .................                                                                                            [100%]

============================================================= 17 passed in 0.08 seconds ==============================================================
```