



# SDMAY\_08 – Cirq: A python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits

Team Members: Jacob Shedenhelm, Calista Carey, Aj Hanus, Austin Garcia, Andrew Hancock, Jordan Cowen  
Advisor: Akhilesh Tyagi  
Sponsor: Victory Omole

# Project Plan

---



## High Level Overview



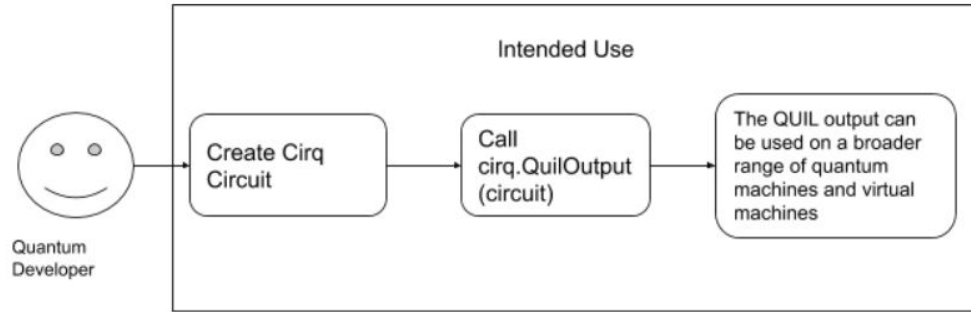
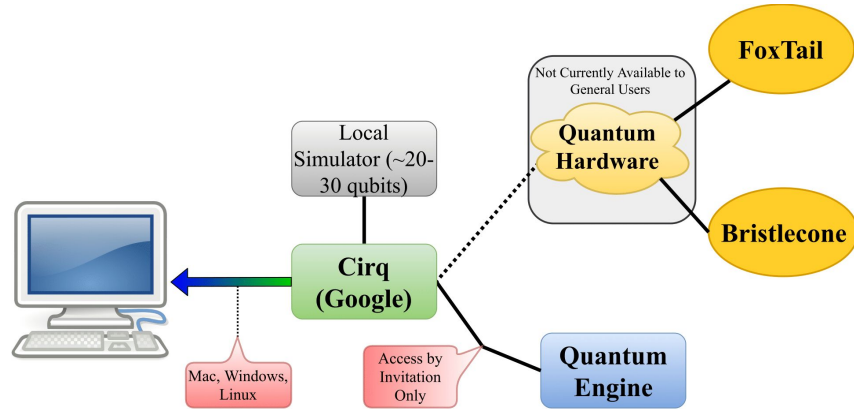
- Cirq is one of Google's '20%' projects
- Open source software that gives users the ability to simulate various quantum circuits
- We want to add in the functionality of translating to another quantum computing language called Quil
- Quil is owned by Rigetti and being developed with the goal of having quantum and classical capabilities



# Problem Statement

- Cirq is a hardware description language for quantum computers
- Quil is an quantum instruction set architecture being developed for interaction of classical and quantum computers
- We want to be able to translate Cirq circuits to Quil
- This would give users the ability to test their Cirq circuit on Rigetti's quantum virtual machine

# Conceptual Sketch





# Functional Requirements

## 1. Implement issue #2386 in the Cirq repository :

For this requirement, we need to implement the file, `quil_output.py`. Cirq allows for conversions between different Quantum Computing circuit specifications, and they want to allow a circuit to be exported as QUIL. This will be similar to how Cirq already outputs QASM. We will be using python and coding in Visual Studio Code.

## 2. Finish the pull request for TwoQubitDiagonalGate:

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 Cirq gate.

## 3. Complete smaller issues:

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



# Technical Constraints/Other Considerations

- Cirq was designed for development on Linux and has been difficult to get up and running for windows systems.
  - After fighting the setup, we decided to use an Amazon EC2 instance running linux through VS Code's Remote SSH extension
- Additionally we all have no prior technical experience with Quantum Mechanics/Computing and have had to learn these subjects
- There is not a 1 to 1 relationship between Quil and Cirq. We will need a full understanding of Quil's functionality and how to derive that from Cirq



# Potential Risks and Mitigation

- Skill Training Time
  - Some of our group members were new to Python and no one was familiar with quantum computing at the beginning. We were able to mitigate this by spending time watching a series of quantum computing lectures from a professor at CMU and having video checkpoints for accountability
- Software Updates
  - Large updates may have affected the implementation of our project, but luckily we never had to deal with this
- Scheduling Conflicts
  - Six seniors with full course loads made it difficult to schedule meetings, but by sharing our schedules and trying our best to be flexible, we were able to mitigate this issue



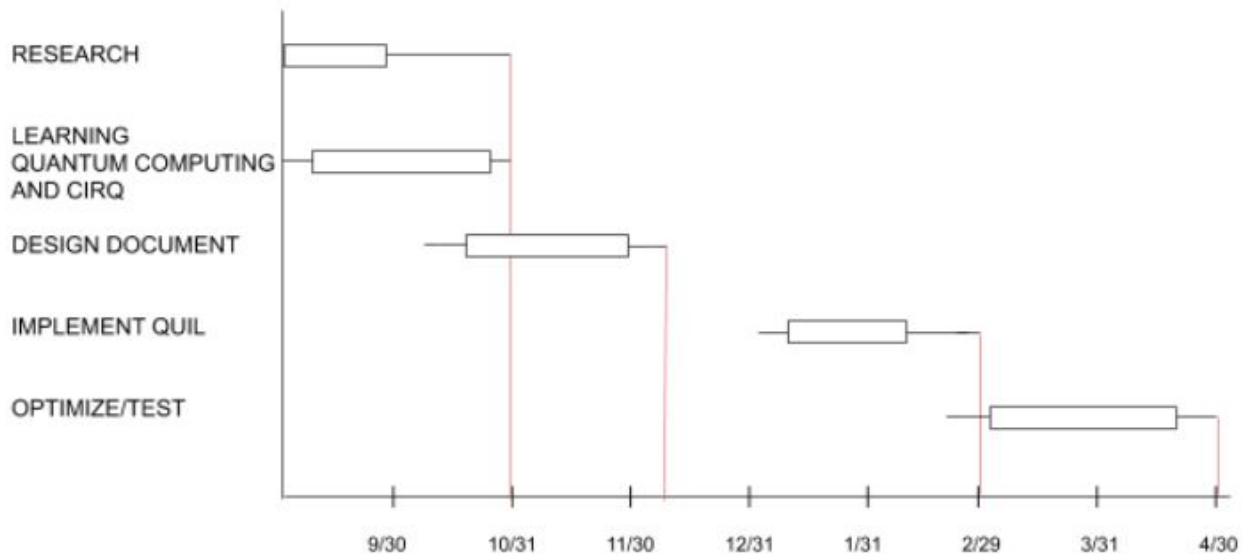


# Resource/Cost Estimate

- Amazon EC2 instance t2.Large runs at a cost of \$0.094/hr
  - We've spent about \$54 running our instance
- Individual time commitment for writing plans/logs/reports and implementing our solution as well as accompanying tests



# Project Milestones and Schedule



*This timeline outlines our project plan. The white boxes identify when the majority of the work for that phase will be completed.*

Austin Garcia

# System Design

---



# Functional Decomposition

- `_write_quil()` is called on a Cirq circuit
- Circuit is broken down into a list of gates that can iterated on
- If no `_quil_()` function in gate, then decompose the gate
- A QUIL translation function is called on each gate
- Various QUIL syntax is included throughout the translation:
  - Measurements where present in Cirq circuit
  - `DEFCIRCUIT` to break down complex circuits
  - `DEFGATE` to implement non standard Cirq gates



# Detailed Design

There are two main parts of our implementation:

- Functionality that processes the translation in `quil_output.py`
  - handles everything from breaking down the circuit to writing the output file
- Individual gate translation present in each class
  - provides the Quil representation of the gate



## Detailed Design cont. - quil\_output.py

- Contains the function `_write_quil()` which does:

- Processes all measurements and outputs Quil representation
- Breaks Cirq circuit into list of operations
- Iterates on list of operations and for each: `for main_op in self.operations:`

- Decomposes if necessary
- Outputs Quil representation

```
decomposed = protocols.decompose(main_op,
                                   keep=keep,
                                   fallback_decomposer=fallback,
                                   on_stuck_raise=on_stuck)

for decomposed_op in decomposed:
    output_func(
        protocols.quil(decomposed_op, formatter=self.formatter))
```



## Detailed Design cont. - Individual Translation

- Each gate either has a function that returns the Quil representation (`_quil_()`) or it decomposes into a series of gates that do
- For example, the `cirq.PhasedXPowGate()` has the following decompose function:

```
z = cirq.Z(q)**self._phase_exponent
x = cirq.X(q)**self._exponent
if protocols.is_parameterized(z):
    return NotImplemented
return z**-1, x, z
```

- Since the Phased XPowGate doesn't contain `_quil_()`, but the gates that it decomposes to do, we get the following Quil Representation (when `phased_exp = 0.777` and `exp = -0.5`)

```
RZ(-0.777) 1
RX(-0.5) 1
RZ(0.777) 1
```

# Detailed Design cont. - DEFGATES

May need the ability to define an arbitrary gate in QUIL by its matrix

- Some Cirq gates are non-standard to quil
- Cirq user defined gates

We made two gates: `QuilOneQubitGate`, `QuilTwoQubitGate`.

- If decompose fails, fallback on defining one of these gates by a unitary matrix representing the operation
- `_quil_` function for these gates prints “DEFGATE USERGATE:  $\backslash n$  {matrix}  $\backslash n$  USERGATE {qubit}”

```
def fallback(op):
    if len(op.qubits) not in [1, 2]:
        return NotImplemented

    mat = protocols.unitary(op, None)
    if mat is None:
        return NotImplemented

    if len(op.qubits) == 1:
        return QuilOneQubitGate(mat).on(*op.qubits)
    return QuilTwoQubitGate(mat).on(*op.qubits)
```

```
def _quil_(self, qubits: Tuple['cirq.Qid', ...],
            formatter: 'cirq.QuilFormatter') -> str:
    return (f'DEFGATE USERGATE:\n\t'
            f'{to_quil_complex_format(self.matrix[0, 0])},\n\t'
            f'{to_quil_complex_format(self.matrix[0, 1])}\n\t'
            f'{to_quil_complex_format(self.matrix[1, 0])},\n\t'
            f'{to_quil_complex_format(self.matrix[1, 1])}\n\t'
            f'{formatter.format("USERGATE {0}", qubits[0])}
```





## Detailed Design cont. - DEFGATE

If there are multiple defgates in one file, each gate must be defined with a different name

Define all as USERGATE initially

Second pass through output adds in 1,2,3 etc.

```
def __str__(self) -> str:  
    output = []  
    self._write_quil(lambda s: output.append(s))  
    return self.rename_defgates(''.join(output))
```

```
DEFGATE USERGATE1:  
    1.0+0.0i, 0.0+0.0i  
    0.0+0.0i, 1.0+0.0i  
USERGATE1 0  
DEFGATE USERGATE2:  
    1.0+0.0i, 0.0+0.0i, 0.0+0.0i, 0.0+0.0i  
    0.0+0.0i, 1.0+0.0i, 0.0+0.0i, 0.0+0.0i  
    0.0+0.0i, 0.0+0.0i, 1.0+0.0i, 0.0+0.0i  
    0.0+0.0i, 0.0+0.0i, 0.0+0.0i, 1.0+0.0i  
USERGATE2 0 1
```



## Detailed Design cont. - QUIL Formatter

- Developed a QUIL Formatter class
  - Requires a qubit\_map and measurement\_map
- Used when returning each of the `_quil_` strings
- Properly formats the qubits to their resulting QUIL output value
  - For example, the string representation of a named qubit in cirq may be ``q0``
  - The QUIL Formatter would output ``q0`` as an integer because all qubits are positive integers, ``0``
- Similar to qubits, measurements have to be defined at the start of the file
- The QUIL Formatter then properly outputs the measurement gates using the `measurement_map`

```
def format_field(self, value: Any, spec: str) -> str:
    if isinstance(value, cirq.ops.Qid):
        value = self.qubit_id_map[value]
    if isinstance(value, str) and spec == 'meas':
        value = self.measurement_id_map[value]
        spec = ''
    return super().format_field(value, spec)
```



## HW/SW/Platforms Used

- Our project doesn't require an hardware. Cirq itself is meant to simulate quantum hardware on classical hardware
- As for software, we are using Python 3.5+ with a long list of packages
- Amazon EC2 instance running a distribution of Linux
- PyTest for testing our implementation
- Git source control
- Pylint for automated formatting checks



git





# Test Plan

- We've created a PyTest Suite that includes a wide variety of tests:
  - A single with and without a exponent
  - Gates with named qubits
  - Saving to a file
  - One and two qubit representation (`_repr_()`) functions
  - Unsupported operations
  - Every possible gate with a variety of exponents

```
def _all_operations(q0, q1, q2, q3, q4, include_measurements=True):  
    return (  
        cirq.Z(q0),  
        cirq.Z(q0)**.625,  
        cirq.Y(q0),  
        cirq.Y(q0)**.375,  
        cirq.X(q0),  
        cirq.X(q0)**.875,  
        cirq.H(q1),
```



## Test Plan cont.

- 100% code coverage is required for this project
  - each line must be executed
- Pytest used as our testing suite
- Initial issues with testing crashing our Amazon instance
  - upgraded to one with more computation power
  - only ran necessary tests

```
quilt_output_test.py .....
```

```
[100%]
```

```
===== 16 passed in 0.11 seconds =====
```



# Prototype implementations

- Our solution is modeled relatively closely to how Cirq translates to QASM
- QASM is another low level quantum computing language
- QASM's parser had characteristics and structure we believed would be crucial to an efficient Quil solution
  - Individual gate translation present in gate's class
  - Nested decomposition down to base gates



# Engineering Standards and Design Practices

- Cirq has standards that we create full code coverage with tests.
  - Signed a CLA, so Cirq has permission to use and redistribute our contributions
- IEEE 12207-2017
  - Software life cycle processes: Develop high quality software with user satisfaction in mind
  - Software Reviews and Audits: determine the type of review needed for our project
  - Software Testing: Goes along with the Cirq standard we have.
- Agile development approach

# Conclusion

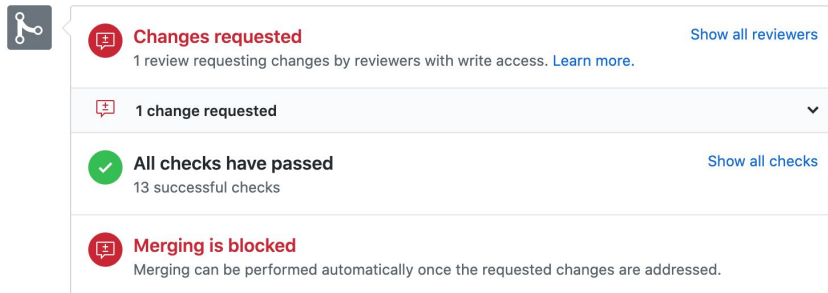
---





# Individual Responsibility

- Implementation broken into 3 major parts under 1 pull request
  - Translation of standard gates - Calista & Jake
  - Translation of non-standard gates - Jordan & Andrew
  - Control flow and measurements - AJ & Austin
- All brought together for a final code review by knowledgeable Cirq developers
- Currently waiting for Cirq developers to accept our change request edits and merge into master



The screenshot shows a pull request status summary with four sections:

- Changes requested** (red icon): 1 review requesting changes by reviewers with write access. [Learn more.](#) [Show all reviewers](#)
- 1 change requested** (red icon): 1 change requested. [Show all changes](#)
- All checks have passed** (green checkmark icon): 13 successful checks. [Show all checks](#)
- Merging is blocked** (red icon): Merging can be performed automatically once the requested changes are addressed.



# Future Prospects

As both Cirq and Quil become more widely used and applicable, we hope that our project can bridge the gap for users to easily move their application between platforms

For us as individuals, we now have a general understanding of quantum computing and can approach future problems with that knowledge