Quantum AI

Cirq-FT: Cirq for Fault-Tolerant Algorithms

Presented by: Tanuj Khattar

IWQC Workshop, Paris

July 22, 2023

Google

Objective - Resource estimation of FT algorithms

Use software to reduce toil and increase correctness when describing and costing out new quantum algorithms

- Fault-Tolerant Quantum Simulations of Chemistry in First Quantization arXiv:2105.12767
  - 100 pages
- Compilation of Fault-Tolerant Quantum Heuristics for Combinatorial Optimization arXiv:2007.07391
  - 80 pages
- Optimal scaling quantum linear systems solver via discrete adiabatic theorem arXiv:2111.08152
  - 77 pages

Next we consider the total number of logical qubits needed for the simulation via this method.

1. The control register for the phase estimation, and the ancillas for the unary iteration, together need $2\lceil \log \mathcal{I} \rceil - 1$ qubits.
2. There are NN_k qubits for the target system.
3. There are $n_{MN} + 2$ qubits for the ℓ register, the qubit rotated in preparing the equal superposition, and the qubit flagging success of preparing the equal superposition.
4. The state preparation on the ℓ register uses $b_{MN} = 2n_k + 2\lceil \log M \rceil + 2\aleph_1 + 2$ qubits. Here $2n_k + 2\lceil \log M \rceil$ is for the ind and alt values of $\mathbf{Q}$ and n, $\aleph_1$ are for keep values, $\aleph_1$ are for the equal superposition state, 1 is for the output of the inequality test, and 2 are for the qubit flagging $\ell \neq 0$ and its alternate value.
5. There are $n_P + 2$ qubits needed for the register preparing $p, q, \mathbf{k}$ values, a qubit that is rotated for the equal superposition, and a qubit flagging success of preparing the equal superposition.
6. The equal superposition state used for the second preparation uses $\aleph_2$ qubits.
7. The phase gradient register uses b_r qubits.
8. The qubits for the spin, controlling the swap of p and q, selection between the real and imaginary parts, and selection between A and B for a total of 4.
9. The QROM needs a number of qubits $b_p k_{p1} k_{p2} + \lceil \log[(MN_k + 1)/k_{p1}] \rceil + \lceil \log[L/k_{p2}] \rceil$.

The QROM for the state preparation on the second register uses a large number of temporary ancillas, which can be reused by later parts of the algorithm, so those later parts of the algorithm do not need the number of qubits counted. The total number of qubits used is then

$$2\lceil \log \mathcal{I} \rceil + NN_k + n_{MN} + n_P + 2n_k + 2\lceil \log M \rceil + 2\aleph_1 + \aleph_2 + b_r + 9 + b_p k_{p1} k_{p2} + \lceil \log[(MN_k+1)/k_{p1}] \rceil + \lceil \log[L/k_{p2}] \rceil \quad \text{(B15)}$$

with $b_p = 2n_k + 2n_N + \aleph_2 + 3$, $n_N = \lceil \log(N/2) \rceil$, $n_P = \lceil \log P \rceil$, $L = N_k N(N+2)/4$. This completes the costing of the low rank factorization method.

Source: Fault-tolerant quantum simulation of materials using Bloch orbitals

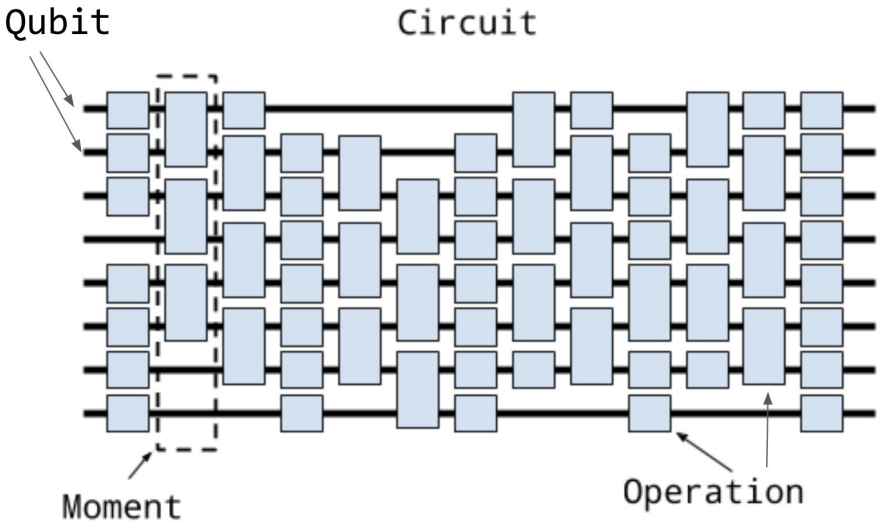
Quantum AI

Related Talk by Yuval Sanders:
Why are Quantum Algorithm Papers so Long?

Cirq - NISQ Assembly Language

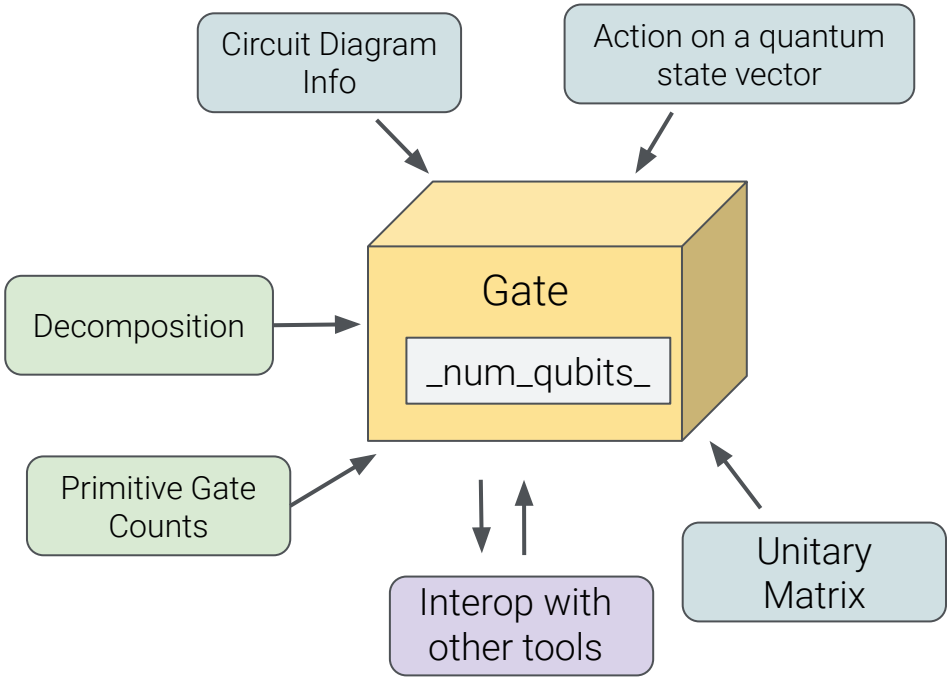
- Cirq is a Python library for NISQ
  - `Circuit` is a collection of Moments
  - `Moment` is a collection of Operations that act on the same time slice
  - `Operation` is an effect that acts on a specific subset of qubits.
  - `Gate` defines a quantum effect, and can be applied on qubits to construct an operation.
- Runs on quantum hardware & simulators
- Circuit Compilation & Optimization
- Device topology, supported gates validation.
- Noise Models and Noisy Simulations

https://github.com/quantumlib/Cirq



Quantum AI

Cirq gates + protocols are flexible



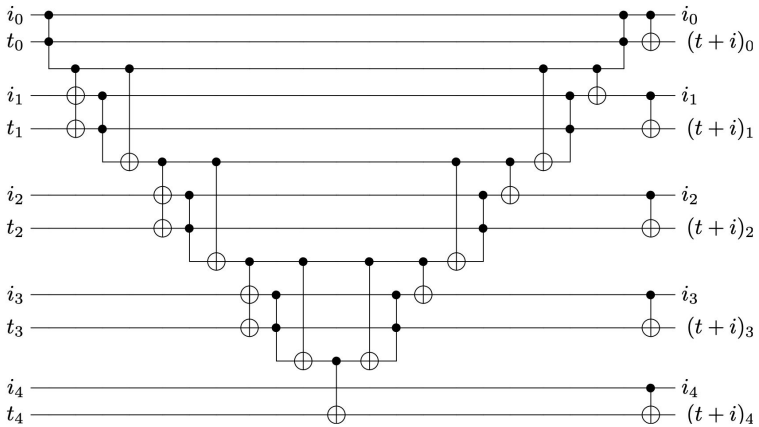

```python
@frozen
class CSWAP(cirq.Gate):
    """$CSWAP = |0><0| I + |1><1| SWAP$."""
    #`_num_qubits_` defines the signature of gate.
    def _num_qubits_(self):
        return 3

    # All other methods are optional annotations.
    # More you provide, more you get.
    def _circuit_diagram_info_(self, args):
        return ['@', 'x', 'x']

    # Gates can be defined in terms of other
    # simpler gates. Decomposition gets you other
    # annotations like unitary etc. for free.
    def _decompose_(self, qubits):
        ctrl, x, y = qubits
        yield cirq.CCNOT(ctrl, x, y)
        yield cirq.CCNOT(ctrl, y, x)
        yield cirq.CCNOT(ctrl, x, y)

qubits = [cirq.q('ctrl'),cirq.q('x'),cirq.q('y')]
SVGCircuit(cirq.Circuit(CSWAP().on(*qubits)))
```

Quantum AI

Cirq isn't ergonomic to express FT algorithms

❌ No way for gates to express their signature in terms of named qubit registers

❌ No way to allocate ancilla qubits only during when specifying the decomposition of a gate.



Quantum AI [Halving the cost of quantum addition]

```python
@frozen
class Add(cirq.Gate):
    """n-bit adder $Add_{n}|x>|y> -> |x>|x+y>$"""
    n: int

    def _num_qubits_(self):
        # Includes ancilla as part of signature.
        return 2 * self.n + (self.n - 1)

    def _decompose_(self, qubits):
        # Manually filtering out the right qubits
        # from a flat list is error prone.
        x = qubits[:self.n]
        y = qubits[self.n : 2*self.n]
        anc = qubits[2*self.n:]
        # Yield decomposition in terms of
        # n-1 AND/AND^† gates and 6n - 9 CNOTs.
        ...

# Need to apply the gate on a flat-list of qubits
# to generate an operation. This is error prone
# and cumbersome.
op = Add(n).on(*flat_qubits)
```

Can we push the limits of Cirq to express and analyze FT algorithms?

Here's an old paper that was a lot of toil

36 pages of compiling primitives

Chemistry paper; best construction for doing a coherent for-loop (Unary Iteration)

"There has to be a better way"

PHYSICAL REVIEW X **8,** 041015 (2018)

Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity

Ryan Babbush,[1,*] Craig Gidney,[2] Dominic W. Berry,[3] Nathan Wiebe,[4] Jarrod McClean,[1] Alexandru Paler,[5] Austin Fowler,[2] and Hartmut Neven[1]

[1]*Google Inc., Venice, California 90291, USA*
[2]*Google Inc., Santa Barbara, California 93117, USA*
[3]*Department of Physics and Astronomy, Macquarie University, Sydney, NSW 2109, Australia*
[4]*Microsoft Research, Redmond, Washington 98052, USA*
[5]*Institute for Integrated Circuits, Linz Institute of Technology, 4040 Linz, Austria*

(Received 9 May 2018; revised manuscript received 1 August 2018; published 23 October 2018)

We construct quantum circuits that exactly encode the spectra of correlated electron models up to errors from rotation synthesis. By invoking these circuits as oracles within the recently introduced "qubitization" framework, one can use quantum phase estimation to sample states in the Hamiltonian eigenbasis with optimal query complexity $\mathcal{O}(\lambda/\epsilon)$, where λ is an absolute sum of Hamiltonian coefficients and ϵ is the target precision. For both the Hubbard model and electronic structure Hamiltonian in a second quantized basis diagonalizing the Coulomb operator, our circuits have T-gate complexity $\mathcal{O}(N + \log(1/\epsilon))$, where N is the number of orbitals in the basis. This scenario enables sampling in the eigenbasis of electronic structure Hamiltonians with T complexity $\mathcal{O}(N^3/\epsilon + N^2 \log(1/\epsilon)/\epsilon)$. Compared to prior approaches, our algorithms are asymptotically more efficient in gate complexity and require fewer T gates near the classically intractable regime. Compiling to surface code fault-tolerant gates and assuming per-gate error rates of one part in a thousand reveals that one can error correct phase estimation on interesting instances of these problems beyond the current capabilities of classical methods using only about a million superconducting qubits in a matter of hours.

DOI: 10.1103/PhysRevX.8.041015

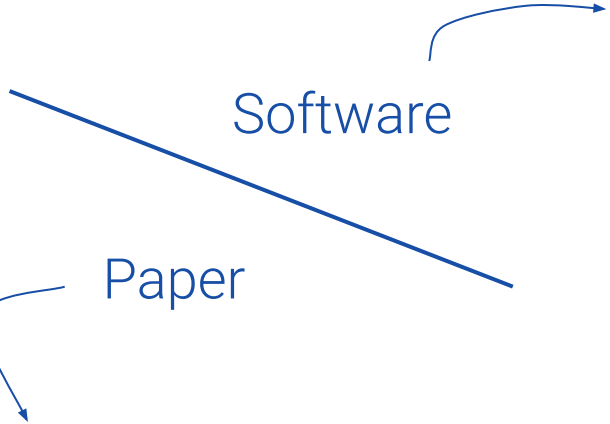
Subject Areas: Chemical Physics,
Quantum Information,
Strongly Correlated Materials

Quantum AI

Source: [Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity (arxiv:1805.03662)]

Phase Estimation of Qubitized Quantum Walk (Hubbard Model)

We coded up the entire Linear T paper!



	dimension	Spin Orbitals	PE Bitsize	T-counts	Rotations	Cliffords
0	6x6	72	16	3.98586e+07	590876	1.73027e+08
1	8x8	128	17	1.1116e+08	656235	5.07546e+08
2	10x10	200	18	3.51817e+08	2.36051e+06	1.63401e+09
3	20x20	800	20	4.62634e+09	9.43856e+06	2.29534e+10
4	30x30	1800	21	1.97888e+10	1.88758e+07	1.00164e+11
5	40x40	3200	22	6.91306e+10	3.77503e+07	3.5189e+11

cirq_ft/algos/phase_estimation_of_quantum_walk.ipynb

Dimension	Spin-Orbitals	Logical Ancilla	Total Logical	T Count
6×6	72	33	105	9.3×10^7
8×8	128	33	161	2.9×10^8
10×10	200	36	236	7.1×10^8
20×20	800	42	842	1.2×10^{10}

More accurate!

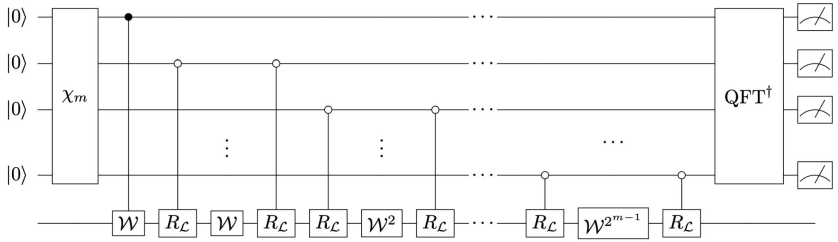
Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity (arxiv:1805.03662)

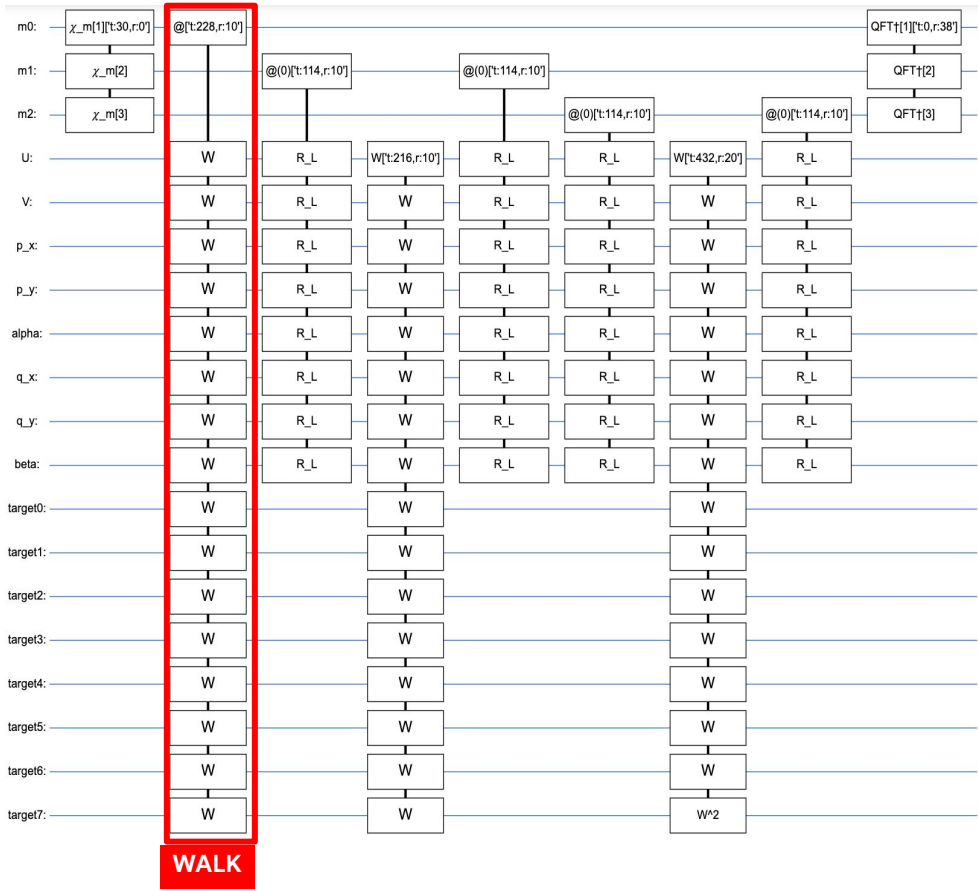
Quantum AI

Linear T Paper coded up

Software

Paper



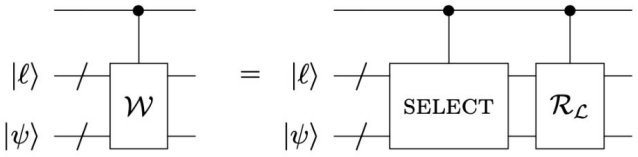


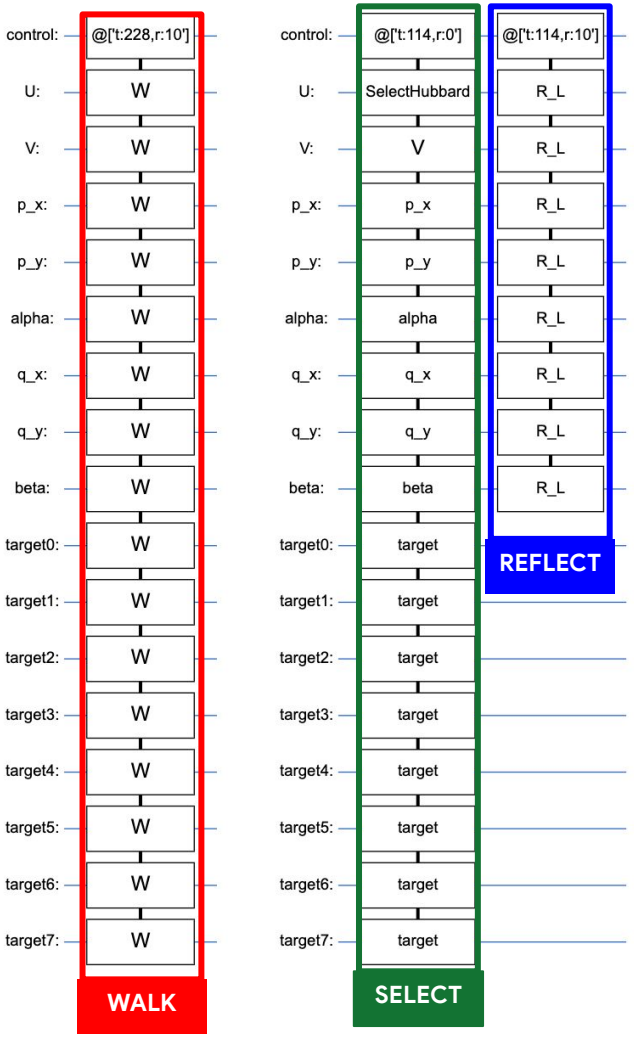
Quantum AI

Linear T Paper coded up

Software

Paper



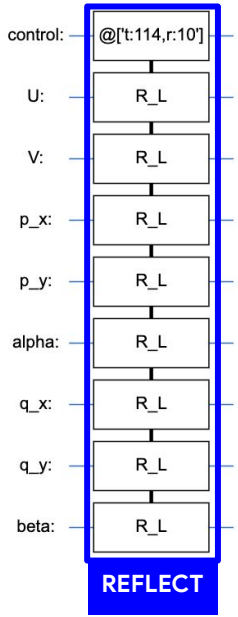


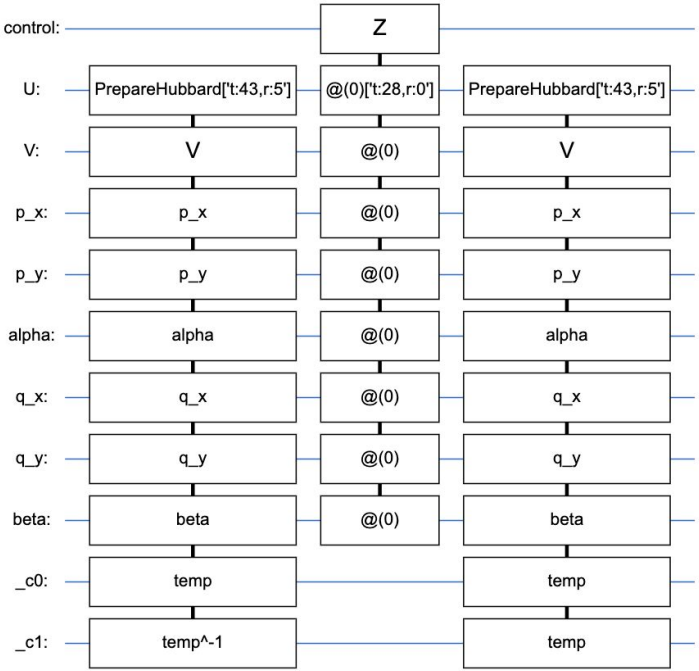
Quantum AI

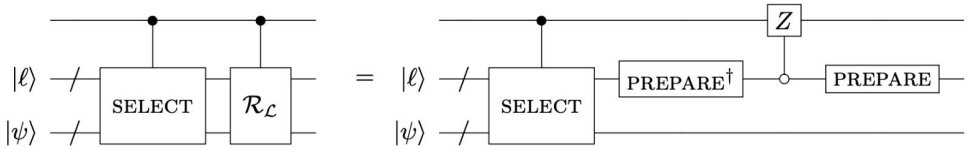
Linear T Paper coded up

Software

Paper







Quantum AI

Et Cetera!

Checkout the entire implementation at Cirq/cirq-ft/cirq_ft/algos

The Software Engineering: Cirq-FT

Cirq-FT: A new sub-package for Cirq that makes it easier to express Fault-Tolerant quantum algorithms.

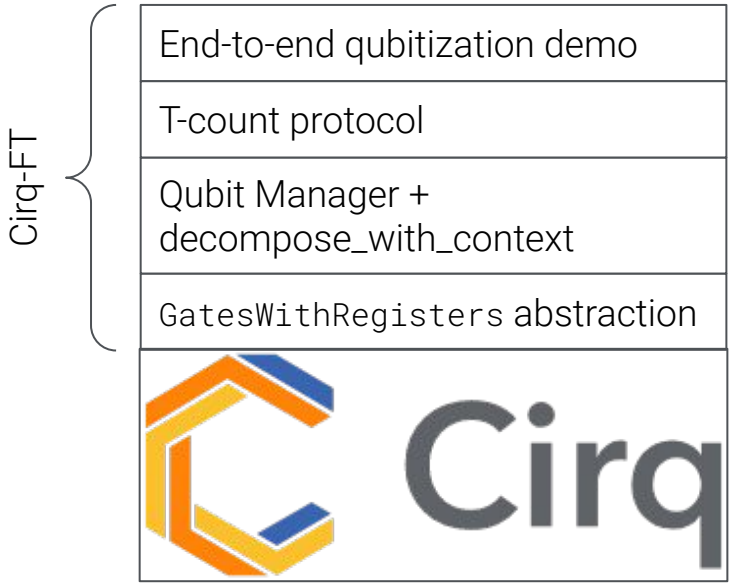
https://github.com/quantumlib/Cirq/tree/master/cirq-ft

```
pip install cirq-ft
```

Design Philosophy

Meet the researchers where they are.

Fill in only what you need: sketch out approx gate counts or provide full implementation.



The Software Engineering: Cirq-FT

Cirq-FT: A new sub-package for Cirq that makes it easier to express Fault-Tolerant quantum algorithms.

https://github.com/quantumlib/Cirq/tree/master/cirq-ft

```
pip install cirq-ft
```

New abstractions

Make it easier to express FT algorithms

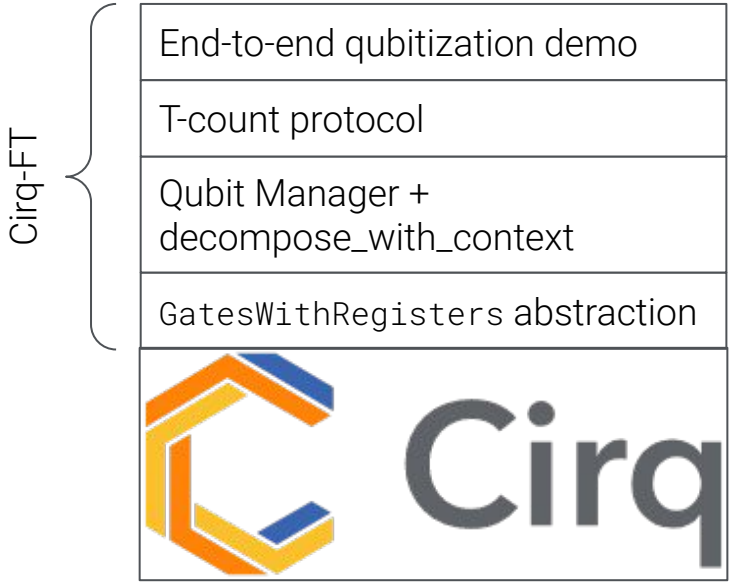
Named qubit registers

Qubit Manager: ancilla allocation and de-allocation

Tools to analyze FT algorithms

Resource estimation protocols

Library of implemented primitives



Cirq-Core → Cirq-FT

```
@frozen
class CZPow(cirq.Gate):
    """$CZPow = |0><0| I + |1><1| ZPow$.

ctrl: ──@──────          ctrl: ──@──────────@──
        │                      │          │
trgt: ──Z^θ── ==>        trgt: ──@──────────@──
        │                      │          │
anc:  ──anc──            anc:  ──X──Z^θ──X──"""

    exponent: float
```

```
def _num_qubits_(self):
    return 3
```

```
def _decompose_(self, qubits):
    ctrl, trgt, anc = qubits
    yield cirq.CCNOT(ctrl, trgt, anc)
    yield cirq.Z(anc)**self.exponent
    yield cirq.CCNOT(ctrl, trgt, anc)
```

```
@frozen
class CZPow(cirq_ft.GateWithRegisters):
    """$CZPow = |0><0| I + |1><1| ZPow$.

ctrl: ──@──────          ctrl: ──@──────────@──
        │                      │          │
trgt: ──Z^θ── ==>        trgt: ──@──────────@──
        │                      │          │
anc:  ──anc──            anc:  ──X──Z^θ──X──"""

    exponent: float
```

```
@cached_property
def registers(self):
    return Registers.build(ctrl=1, trgt=1)
```

```
def decompose_from_registers(self, context,
                             ctrl,trgt):
    anc = context.qubit_manager.qalloc(1)
    yield cirq.CCNOT(*ctrl, *trgt, *anc)
    yield cirq.Z(*anc)**self.exponent
    yield cirq.CCNOT(*ctrl, *trgt, *anc)
    context.qubit_manager.qfree(anc)
```

- ✅ `cirq_ft.GateWithRegisters:` Base class to define FT gadgets as composite gates.
- ✅ `cirq_ft.Registers:` Named Qubit registers for declaring signature of a composite gate.
- ✅ `cirq.QubitManager:` Memory management for ancilla allocations during decomposition.

Quantum AI

Support ancilla allocations within decompose

- **`cirq.unitary`** now supports computing reduced unitary of a gate that allocates ancilla qubits within decompose.
- **`cirq.Simulator`** now supports simulating circuits with gates that can allocate ancilla qubits within decompose.

```
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+1.j]]
```

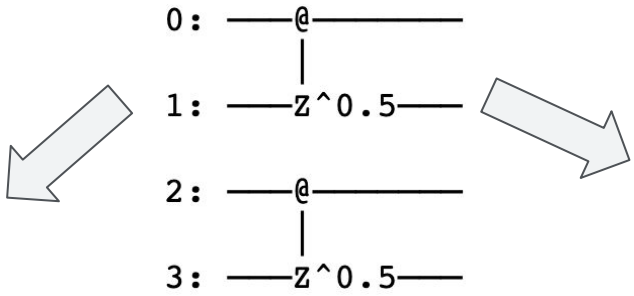
```
print(cirq.unitary(CZPow(0.5)))
```

```
@frozen
class CZPow(GateWithRegisters):
    """$CZPow = |0><0| I + |1><1| ZPow$."""

    exponent: float

    @cached_property
    def registers(self):
        return Registers.build(ctrl=1, trgt=1)

    def decompose_from_registers(self,
        context, ctrl, trgt
    ):
        anc = context.qubit_manager.qalloc(1)
        yield cirq.CCNOT(*ctrl, *trgt, *anc)
        yield cirq.Z(*anc)**self.exponent
        yield cirq.CCNOT(*ctrl, *trgt, *anc)
        context.qubit_manager.qfree(anc)
```

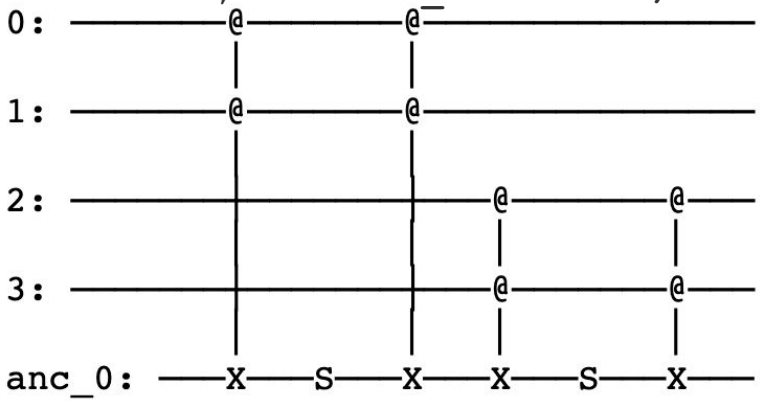
```
print(cirq.unitary(CZPow(0.5)))
assert np.array_equal(cirq.unitary(CZPow(0.01)),
                      cirq.unitary(cirq.CZ**0.01))
```

Qubit allocation strategies during circuit construction

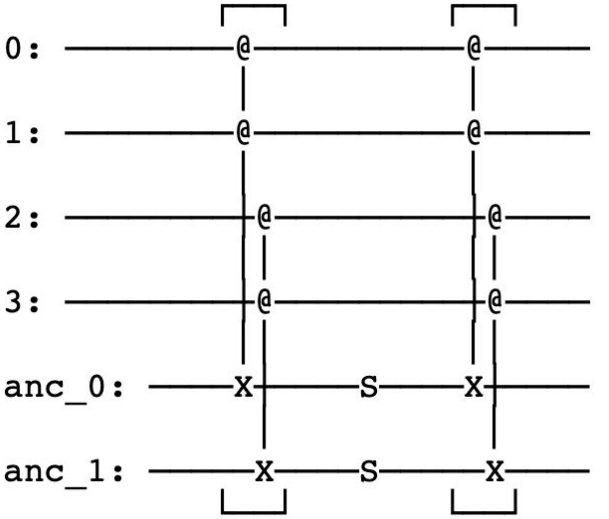
Specify a custom `QubitManager` to `cirq.decompose(op)` at the time of circuit construction.



`GreedyQubitManager('anc', size=1, maximize_reuse=True)`



`GreedyQubitManager('anc', size=2, maximize_reuse=False)`

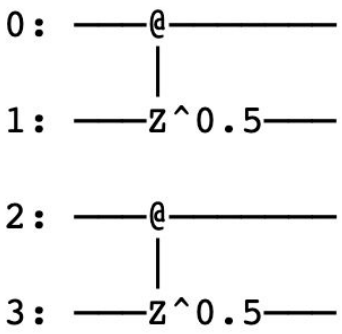


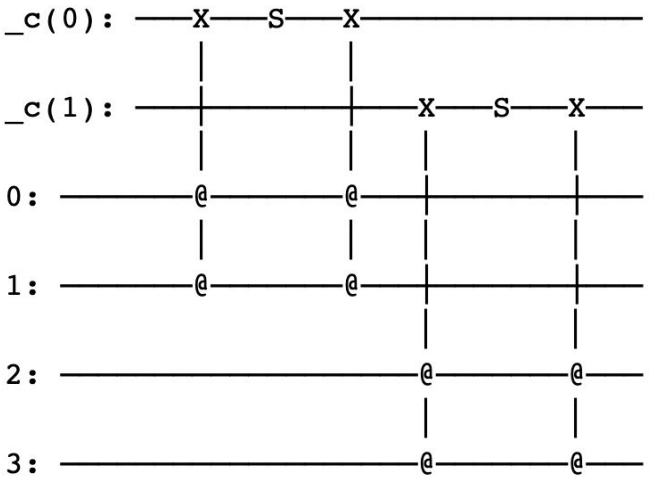
A two-step compilation approach for greater control

Step-1: Construct the underlying compute graph without introducing any additional dependencies – Use a `SimpleQubitManager()` or `QUALTRAN` (Stay tuned!)

- `SimpleQubitManager` always allocates new qubits of an internal type `CleanQubit/BorrowableQubit` to represent clean/dirty allocation requests.

- This is a hack to encode qubit allocation requests as part of the compute graph. We can also obtain the compute graph using `QUALTRAN`, which does this better.



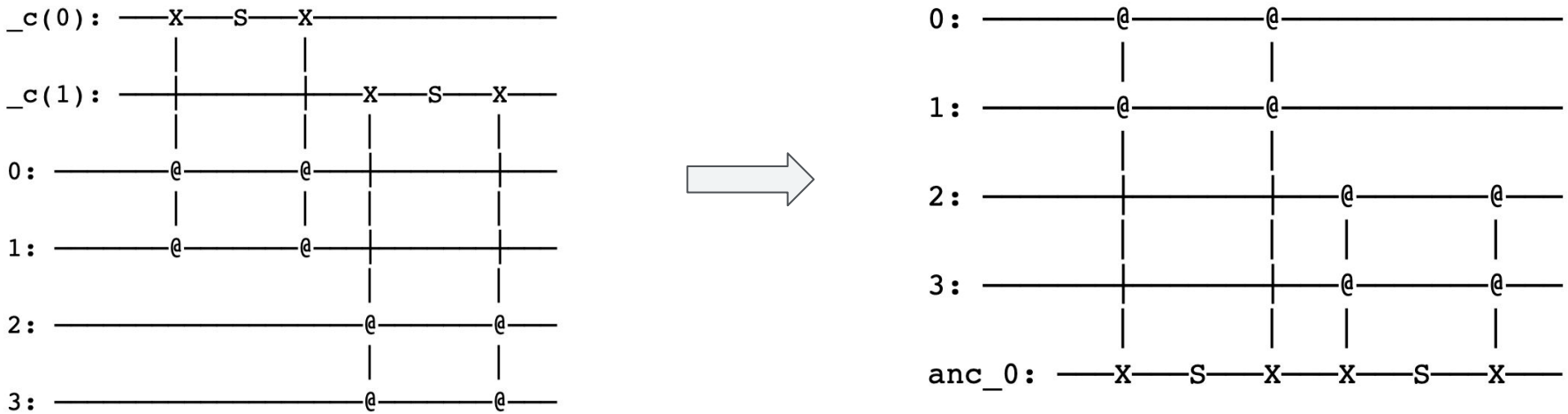


Quantum AI

A two-step compilation approach for greater control

Step-2: Post-Process the compute graph and map qubit allocation / deallocation requests to concrete qubits. Custom mapping strategies can be implemented, which have context of the entire compute graph.

- This two-step approach provides a greater control over different qubit management strategies, compared to previous works like Q# and QParallel.

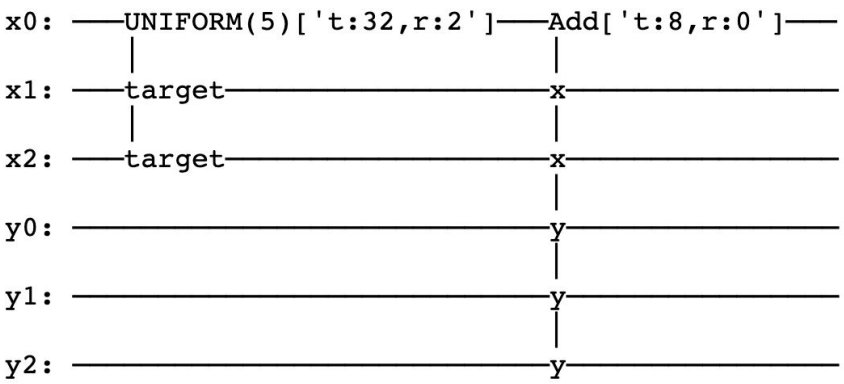


Quantum AI

```
greedy_qm = cirq_ft.GreedyQubitManager('anc', maximize_reuse=True)
cirq_ft.map_clean_and_borrowable_qubits(decomposed_circuit, qm=greedy_qm)
```

Protocols for resource estimation

Way-1: Construct a “bare-bones” gate with only costs specified via the `_t_complexity_` protocol.




```python
@frozen
class Add(cirq_ft.GateWithRegisters):
    """N-bit adder $Add_{n}|x>|y> -> |x>|x+y>$"""
    n: int

    @property
    def registers(self):
        # Specify signature as 2 n-bit registers.
        return Registers.build(x=self.n, y=self.n)

    def decompose_from_registers(self, context, x, y):
        anc = context.qubit_manager.qalloc(self.n-1)
        # Yield decomposition in terms of n-1
        # AND/AND^† gates and 6n - 9 CNOTs.
        raise NotImplementedError("No Decomposition.")
        context.qubit_manager.qfree(anc)

    def _t_complexity_(self):
        # Analytical expression for T-cost as per
        # https://arxiv.org/abs/1709.06648
        c = (self.n - 2) * 19 + 16
        t = 4 * (self.n - 1),
        return cirq_ft.TComplexity(t=t, clifford=c)
```

Protocols for resource estimation

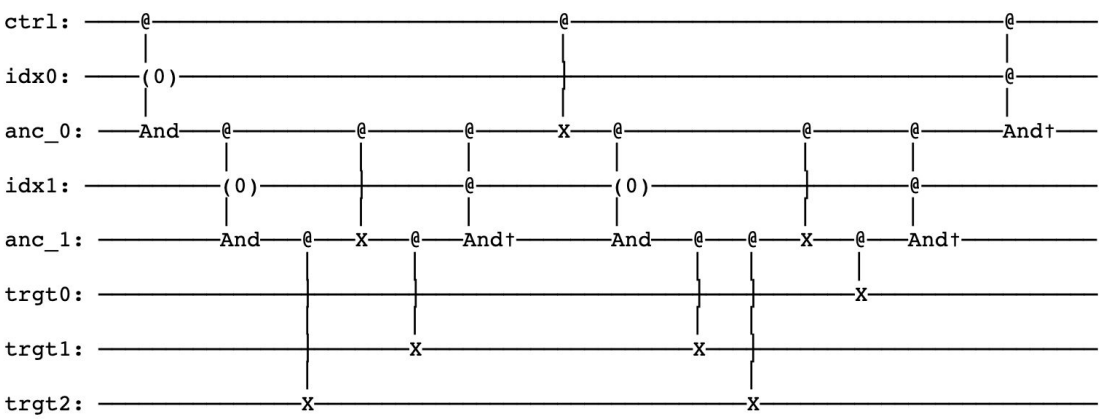
Way-2: Specify decomposition in terms of other "simpler" gates so that T-costs can be inferred automatically.

Note: Caching is implemented at each level of recursion to achieve scalable performance.

```
@frozen
class AddPow(cirq_ft.GateWithRegisters):
    """Decomposes to `power` instances of `Add(n)`."""
    n: int
    power: int

    @property
    def registers(self):
        # Specify the signature as 2 registers of bitsize n.
        return Registers.build(x=self.n, y=self.n)

    def decompose_from_registers(self, context, x, y):
        for _ in range(self.power):
            yield Add(self.n).on_registers(x=x, y=y)

>>> g = cirq_ft.testing.GateHelper(AddPow(200, 5000))
>>> circuit = cirq.Circuit([g.operation] * 5000)
>>> %time result = cirq_ft.t_complexity(circuit)
>>> print(result)
CPU times: user 899 ms, sys: 16.5 ms, total: 915 ms
Wall time: 927 ms
T-count:   1.99e+10
Rotations: 0
Cliffords: 9.445e+10
```

Library of Primitives: UnaryIteration

- Unary Iteration is a commonly used construction which allows us to apply $|n\rangle|\psi\rangle \rightarrow |n\rangle \, \mathrm{ops}_n \cdot |\psi\rangle$ using 4n T-gates.
- We provide an easy-to-use abstraction for Unary Iteration which also supports multi dimensional selection registers!




```
@frozen
class QROM(cirq_ft.UnaryIterationGate):
    data: Tuple[int, ...]
    bitsize: int

    @property
    def control_registers(self):
        return Registers.build(ctrl=1)
    @property
    def selection_registers(self):
        N = len(self.data)
        lgN = (N - 1).bit_length()
        sreg = SelectionRegister('idx', lgN, N)
        return SelectionRegisters([sreg])
    @property
    def target_registers(self):
        return Registers.build(trgt=self.bitsize)

    def nth_operation(self, context,
                      control: cirq.Qid,
                      idx: Tuple[int, ...],
                      trgt: Sequence[cirq.Qid]):
        for i, q in enumerate(trgt):
            if self.data[idx] & (1 << i):
                yield cirq.CNOT(control, q)
```

```
35 QROM(data=(1, 2, 3, 4), bitsize=3)
```

Source: arxiv:1805.03662

Library of Primitives

- Arithmetic Gates
  - AdditionaGate, AddMod, ContiguousRegisterGate, LessThanGate etc.
- State Preparation
  - PrepareUniformSuperposition: using a single round of amplitude amplification.
  - StatePreparationAliasSampling: QROM based state prep using classical alias sampling.
- Data Loading
  - QROM: Unary iteration based data loading using O(iteration_length) T-gates.
  - SelectSwapQROM: “advanced” QROM using O(sqrt(iteration_length)) T-gates.
- Qubitization
  - QubitizationWalkOperator, ReflectionUsingPrepare, SelectOracle, PrepareOracle
- Robin’s Mean Estimation Algorithm
  - MeanEstimationOperator, ComplexPhaseOracle, ArcTan etc.
- Others
  - UnaryIteration base class to enable expressing nested coherent for-loops using multi-dimensional selection registers.
  - ProgrammableRotationGateArray: QROM based rotation synthesis introduced in Guang Hao’s double factorization paper

Conclusion

- Cirq-FT pushes the boundaries of Cirq to make it more amenable to use for doing resource estimation of FT algorithms.
  - New abstractions like Named Qubit Registers and Qubit Manager makes it easier to express FT algorithms as a hierarchical composition of Cirq Gates.
  - Protocols based approach gives users the flexibility to define bare-bone gates and only provide necessary annotations based on use case.
  - Mature Cirq infrastructure gives access to a well annotated library of gates, simulators and protocols.

- QUALTRAN is a new library specifically designed for Resource Estimation and with bi-directional interop with Cirq-FT.
  - Stay tuned for a follow-up talk by my colleague Matt Harrigan later in the day!

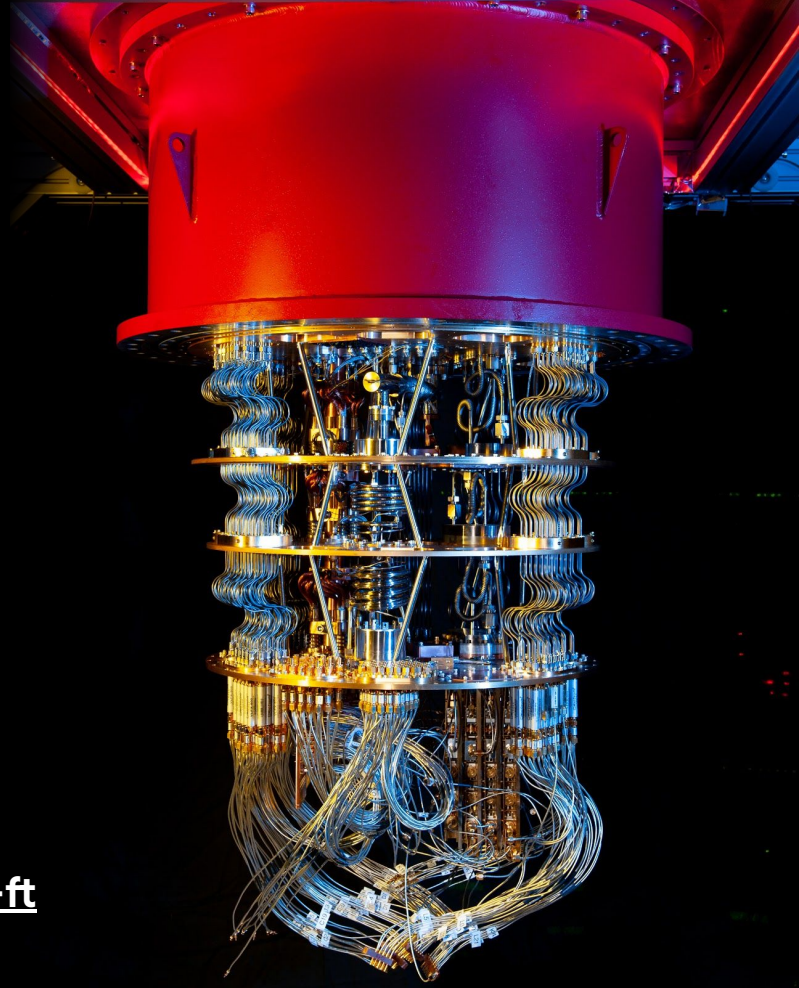
Thank you!

Quantum AI

Matt Harrigan Nick Rubin
Fionn Malone Nour Yosri

tanujkhattar@google.com

github.com/quantumlib/Cirq/tree/master/cirq-ft
https://groups.google.com/g/cirq-dev/about



Google